"Engineering scalable propagation in constraint programming"

Van Cauwelaert, Sascha

Abstract

Combinatorial Optimization is intrinsically hard, including for computers because of the exponential increase of the search space when larger problems are considered. Several technologies have been developed in the previous decades in order to provide computer systems with intelligent reasoning. One of them is Constraint Programming, a declarative paradigm to solve/optimize discrete constrained problems. At the core of this technology lies Propagation, which is responsible for eliminating provable wrong (partial) combinations. This thesis focuses on this part of the Constraint Programming technology. It is well known that the amount of data generated and stored is augmenting. Not surprisingly the size of the problems to be solved by optimization also tends to increase. Therefore all our algorithmic design choices are motivated by scalability. The first part of this work is dedicated to the evaluation of the procedures involved in constraint programming propagation, so-called propagat...

Document type : *Thèse (Dissertation)*

ENGINEERING SCALABLE PROPAGATION IN CONSTRAINT PROGRAMMING

SASCHA VAN CAUWELAERT

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

January 2018

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

Thesis Committee

Pierre Schaus (director)	UCLouvain, Belgium
Michele Lombardi	UniBo, Italy
Philippe Laborie	IBM France, France
Yves Deville	UCLouvain, Belgium
Peter Van Roy	UCLouvain, Belgium
Charles Pecheur (president)	UCLouvain, Belgium

ABSTRACT

Combinatorial Optimization is intrinsically hard, including for computers because of the exponential increase of the search space when larger problems are considered. Several technologies have been developed in the previous decades in order to provide computer systems with intelligent reasoning. One of them is Constraint Programming, a declarative paradigm to solve/optimize discrete constrained problems. At the core of this technology lies Propagation, which is responsible for eliminating provable wrong (partial) combinations. This thesis focuses on this part of the Constraint Programming technology. It is well known that the amount of data generated and stored is augmenting. Not surprisingly the size of the problems to be solved by optimization also tends to increase. Therefore all our algorithmic design choices are motivated by scalability.

The first part of this work is dedicated to the evaluation of the procedures involved in constraint programming propagation, so-called propagators. This is a non-trivial task, since propagation is only a component of the technology and the other parts may influence the solving performances. Specifically, the way search is performed can have a strong repercussion on the performance associated with a given propagator. We therefore introduce a fair manner to evaluate propagators, in the sense that this effect is diminished as much as possible. We also depict an easy technique in order to probe the impact of accelerating a propagator. Despite its simplicity, one can learn if the improvement of a filtering procedure has a chance to be fruitful in practice. On that regard, we give some negative results which are useful for the research community, so that no endeavor is invested on unpromising directions from a practical perspective. Finally, a tool to visualize solver performances is described.

The second part introduces two new scalable propagators that can be used in a broad range of problems. The first one is a unified version of the *Unary Resource with Transition Times* global constraint. This constraint can be found in scheduling problems with unary resources in which sequence-dependent transition times between activities are involved. The filtering procedure is able to reason with family-based transition times and optional tasks. We then describe our second propagator, called *Resource-Cost AllDifferent*. It can be used in optimization problems where a set of items, each requiring a possibly different amount of resource, must be assigned to different slots for which the price of the resource can vary.

Both introduced propagators are experimentally evaluated on industrial and academic problems with our evaluation framework. The results show that they generally outperform the state-of-the-art once the size of the problems gets larger. Moreover, they are robust, in the sense that if our approach is slower than the best existing approach on a given problem instance, it is by a small factor.

ACKNOWLEDGEMENTS

I first warmly thank my advisor, Pierre Schaus, for his guidance and his rigorous remarks. He constantly pushed me to strengthen and justify ideas/opinions. I believe this significantly improved the value of this thesis. One of his great qualities is his availability and readiness for help, even when if he is super busy. Thank you for always having your office door open and for answering my numerous questions. I learned considerably from Pierre, especially to correctly write papers and to program efficiently. Apart from work, I also had much fun discussing complex subjects and laughing at the many *trolls* you made (occasionally at my expense).

I am also grateful to the members of the jury for their comments on this document. Michele, thanks a lot for inviting me in your lab, I think it really helped me to progress in the right directions, in particular at the beginning. Thank you for motivating me when I was not very confident about specific ideas.

A lot of thanks go to my different office mates. Renaud, thank you for the many discussions we had about work and life in general. Your pedagogical explanations on the white board regarding papers you read were always very interesting. It was also nice to drink too much coffee and to play *Corridor* (though I lost very often). Steven, even if you are slightly crazy, it was a pleasure to have you as a research fellow. You were frequently ready to give advice and comments about my work, this helped me a lot. Thanks as well for your incomparable metal styled singing. A special thanks goes to Cyrille, with whom I wrote several papers. We had an uncountable number of technical discussions on the white board using the *activity magnets*. It was sometimes difficult for me to do not get crazy, but one of your awesome qualities is that you are consistently calm and in a good mood. It was a real pleasure working with you. Thank you for all the laughs we had in the office and for making me discover wonderful music (and in particular *Radio Paradise*). Thank you to Guillaume, who knows a lot about data structures and algorithms (and politics, and much more). It is always enjoyable to get your opinions regarding work and the world in general. You are unfailingly willing to help to solve some problems (by altruism or because you find it amusing, I am not quite sure). Our discussions allowed me to fix tiny details and to get fresh ideas for my research.

Thank you to all the *BeCool* members. We had awesome moments and conversations, especially at the different conferences where we went together. I am also grateful to all fellows of the *INGI* department. The work atmosphere is really nice thanks to you guys (in particular Vanessa who does everything she can to make this environment as enjoyable as possible).

I wish to thank my parents for their support in my life and for giving me the chance to study. I also thank my parents and my sister for all the happiness we have in our family. Moreover, even if you are not aware of that, talking with you about this work has helped me a lot, especially in the difficult moments. Danke für die schönen Momente in der Familie. I am also thankful to my grandfather, Frans Van Cauwelaert, who gave me the taste for mathematics and computer science when I was still a child. Thank you as well to my extended family (from Belgium, Germany and Canada), for the uncountable great moments we live together.

I also thank all my friends for the pleasant times and parties we had. I will not enter into any details here, there are too many stories. I apologize again for my absence to some events, I will catch up in the future.

It is clear now that I am not the only one that allowed this thesis to be completed. But I wish to thank two more people in particular. Without you, Emilie, I would never have been able to start this work. You have been present for me every day for several years, included in the most difficult times (thank you for your understanding and patience, you still impress me). You make my life shine and fill it with joy every day. You also remind me what really matters in our existence. I cherish the moments of happiness we have together and the ones to come in the future. Thank you for everything, Emilie. Finally, I am grateful to my daughter, Violette. She (unwillingly) forces me to remember at each instant that although this work is important to me, it is negligible as compared to life.

CONTENTS

INTRODUCTION

The amount of processed data has increased drastically in recent years. This has led to fascinating improvements in Artificial Intelligence, in turn causing a gain of interest by the public in general. This growth goes hand in hand with an augmentation of the problems sizes a computer has to solve. Artificial Intelligence, Machine Learning and Operation Research often require solving discrete (optimization) problems. Unfortunately, problem size is a major issue for NP-Complete problems due to the combinatorial explosion of the search space. Several technologies (e.g., Local Search and Mixed-Integer Programming) have been developed and extensively used in order to solve this kind of problem efficiently.

In particular, this thesis lies in the context of the Constraint Programming (CP) paradigm, where problems are described as *constraint programs*. Basically, a constraint program defines a search space by declaring variables and their associated finite domains, as well as a set of constraints that must all be respected by an assignment in order to be a solution. An important asset of this paradigm is that it is declarative, meaning intuitively that the programmer can focus on describing what the solution must be instead of how to compute it. The responsibility of finding an (optimal) solution is left to a *constraint solver*, that usually performs a Depth-First Search. Filtering algorithms (also named propagators) are called at each search node to verify that constraints can still be respected and to remove inconsistent values from the variable domains.

Filtering (also known as Propagation) is a key ingredient of CP: it makes a constraint solver capable of pruning large portions of the search space, possibly saving significant exploration times. In practice, the strongest filtering algorithms are not always the winners on every problem. As explained in [Smi05], maintaining a higher level of consistency takes more time; on the other hand, if more values can be removed from the domains of the variables, the search effort will be reduced and this will save time. Whether or not the time saved outweighs the time spent depends on the problem, the algorithm, its implementation, the search heuristics, and the propagation queue strategy used in the solver.

The CP community has invested a lot of effort to improve this trade-off by researching the most efficient filtering algorithms. As an example, the SEQUENCE constraint was introduced in 1994 [BC94], but no poly-time Global Arc Consistency (GAC) algorithm was available until 2006 [Hoe+06]. Then, the original GAC run time of $O(n^3)$ was not low enough to consistently beat weaker (but cheaper) propagators. This motivated improvement efforts that are still ongoing [Bra+07, CY10, BCH14]. Other people suggested guarding techniques to reduce the number of times a heavy GAC algorithm is triggered. In the case of guarding propagation, [DB+13] proposes a probabilistic model to estimate if ALLDIFFERENT (bound consistency) will be able to reduce a domain. In [SS05], the authors determine situations where domain propagators can simply be replaced by lighter bound propagators without increasing the search space.

The trade-off between computation time and pruning power is even more critical for NP-hard constraints. For example, Energetic Reasoning (ER) was introduced as a (powerful) filtering technique for CUMULATIVE in the nineties (see [EL90, BLPN01]): however, the method has never been widely employed due to its large run time. Improving the original $O(n^3)$ algorithm took in this case around 20 years [DP14], while an approach to reduce the overhead by guarding the ER activation with a necessary condition was presented only in 2011 [BHS11].

SCALABLE PROPAGATION While computer hardware advances are constantly ongoing, larger and larger problems must be tackled in our ever-evolving society. Size has always been an intrinsic issue for NP-complete problems as the search space increases exponentially, but in the context of CP, it is also problematic because propagators (time and size) complexities generally depend on the number of variables/values they consider. When the problem size grows, the number of search nodes can augment exponentially *and* the running time increases inside each search node, possibly leading to an additional exponential processing time if the supplemental filtering is not sufficiently strong (the worst case being when no additional filtering is performed). Moreover, most constraint inference is done deep in the search tree. For these reasons, while strong but slow algorithms might reduce the number of search nodes importantly on some problem instances, they rarely allow a solving time improvement in general when large-scale instances are considered. These observations imply that such heavy algorithms are seldom used in practice, since only an accidental and drastic inference provides a speed-up.

This is a strong motivation for this thesis: we wish to work with propagators that can be used on large-scale problems, since we assume the sizes of the problems we will have to solve in the future will grow. The aim of these propagators is therefore not to perform the strongest propagation, but to provide beneficial filtering while being able to keep up once the problem size gets larger. We will denote those propagators as *scalable propagators*. In the context of this thesis, scalable propagators are therefore *competitive or better than existing approaches when the problem size is large*.

There is a close link between the scalability and the computational complexity of propagators. However, complexities generally provide bounds on the resources required by algorithms (time is usually the important resource in the case of propagators) but they do not consider/measure the gain provided by the filtering. Complexities are consequently not sufficient to describe the efficacy of a propagator in practice[1]. Moreover, it happens that significant constants are hidden in the complexities formulation. Finally, incremental computation of data structures[2] used by the propagators can also provide a substantial speed-up. It can therefore lead to more scalable propagation, even if the time complexity of the propagator is a bit larger. Hence, low time complexity and incremental computation, while being desirable properties, are only correlated with the scalability of a propagator. They are not sufficient, as we wish to ensure the propagator offers a powerful filtering. Scalability has thus to be measured in practice by means of benchmarking.

In order to assess a propagator is scalable (scales well), we benchmark on many large-scale instances: if it provides a significant time gain for a non-negligible percentage of the instances while not deteriorating too much the performances in general, we denote the propagator as being scalable. We describe the complexities of the propagators in this document, yet we concentrate on their effectiveness in practice.

We focus on a last important point when we evaluate a given propagator. We want to guarantee that the solving time difference that we compute is only due to the filtering of the studied propagator. To do so, we set up the *Replay Framework*, that ensures by design that only the gain from filtering is measured, whichever the search strategy we use. This framework is described in great details in Chapter 3. We think it should be used in the general case[3], in order to avoid bias in conclusions from propagator evaluations.

CONTRIBUTIONS The contributions of this thesis are listed below. All the involved development was integrated in the open-source OscaR solver [Osc12], except from the web tool[4].

Performance Profile Web Tool This tool allows one to build so-called performance profiles and to share them easily with the scientific community. They are used to visually compare solving techniques (e.g., solvers, filtering algorithms, search heuristics) with each other on a given benchmark. Our tool also enables what-if analysis of computation time improvements of a given algorithm involved in the search process. We made an extensive usage of this tool during the thesis and hope it will be helpful to others.

Filtering Experimental Evaluation Framework We introduce an experimental framework to evaluate filtering algorithms in Constraint Programming. We call this the *Replay Framework*. In brief, the search heuristic is decoupled from the

1 Unless we compare two propagators with different complexities that compute the exact same filtering for any input.
2 Along a branch of the search tree.
3 Or at least as a complementary method.
4 Available at the URL `http://performance-profile.info.ucl.ac.be/`.

filtering, i.e., filtering has no effect on the search decisions. This is done by 1) generating a search tree 2) saving it in memory and 3) traversing the saved tree with the different models we wish to compare. The goal of this framework is to ensure that the measurements (e.g., time gain) can only be attributed to the evaluated filtering algorithms. That is, only the effects of filtering are evaluated even if dynamic search strategies are used. Moreover, it allows evaluating with large instances even if the search cannot terminate. We also introduce an easy way to probe the impact of being able to make a heavy propagator more scalable. This evaluation framework has been used for all experiments made in this thesis.

Generalized Unary Resource with Transition Times We propose a scalable filtering algorithm for a generalized version of the Unary Resource constraint when transition times are involved. This constraint can be used to solve scheduling problems where a resource can only be used by one activity at a time. In addition, the constraint imposes sequence-dependent transition times between the activities. It often happens that activities are grouped into families with zero transition times within a family. Moreover, some of the activities might be optional from the resource viewpoint (typically in the case of alternative resources). The global constraint we describe can deal with both optional activities and families of activities and strengthen the pruning of domains as compared with existing approaches.

Resource-Cost AllDifferent We describe an efficient and scalable filtering algorithm for the Resource-Cost AllDifferent constraint, which is a special case of the Global Cardinality Constraint with Costs. It is useful for problems where a set of machines require different amount of resource while the price of this resource varies. A typical use case is the scheduling of electricity-consuming activities under electricity prices fluctuations.

THESIS OUTLINE

In Part I, we discuss the necessary background to read the thesis. Part II describes how we propose to evaluate propagators in CP. We begin by explaining in Chapter 2 how our web tool can be used to build performance profiles. Chapter 3 provides all details about our experimental framework to evaluate propagators. It is used to evaluate well-known existing propagators. Our new propagators are then described and evaluated in Part III. Chapter 4 and Chapter 5 respectively discuss the Generalized Unary Resource with Transition Times and the Resource-Cost AllDifferent constraints. We evaluate them using our replay framework on real-life problems. Since we focus on scalability, we challenged our propagators on large instances. Results illustrate that for the instances we consider, the algorithms are often the fastest and are robust, in the sense that when they are not the fastest, it is by a small factor as compared with the best approach.

PUBLICATIONS The different papers that were published during this thesis are given below. Notice the work on the *Resource-Cost AllDifferent* constraint has been extended for a journal publication shortly after its acceptance for publication in the conference proceedings (Fast Track publication).

JOURNAL PUBLICATIONS

[VCLS18] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. "How efficient is a global constraint in practice ?" In: *Constraints* 23.1 (2018), pp. 87–122.

[VCS17b] Sascha Van Cauwelaert and Pierre Schaus. "Efficient filtering for the Resource-Cost AllDifferent constraint." In: *Constraints* 22.4 (2017), pp. 493–511.

CONFERENCE PUBLICATIONS

[VCLS15] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. "Understanding the potential of propagators." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Springer. 2015, pp. 427–436.

[VCS17a] Sascha Van Cauwelaert and Pierre Schaus. "Efficient Filtering for the Resource-Cost AllDifferent Constraint." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Springer. 2017.

[VC+16] Sascha Van Cauwelaert, Cyrille Dejemeppe, Jean-Noël Monette, and Pierre Schaus. "Efficient Filtering for the Unary Resource with Family-Based Transition Times." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 520–535.

WORKSHOP

[VCLP14] Sascha Van Cauwelaert, Michele Lombardi, and Schaus Pierre. "Supervised Learning to Control Energetic Reasoning : Feasibility Study." In: *Proceedings of the Doctoral Program of CP2014*. 2014.

TECHNICAL REPORTS

[VCDS18] Sascha Van Cauwelaert, Cyrille Dejemeppe, and Pierre Schaus. *The Unary Resource with Transition Times and Optional Activities*. Tech. rep. UCLouvain, 2018.

[VCLS16] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. *A Visual Web Tool to Perform What-If Analysis of Optimization Approaches*. Tech. rep. UCLouvain, 2016.

We allow ourselves to mention a last publication as it is the starting point of one of our main contributions, even if this publication is not strictly speaking a part of this thesis:

[DVCS15] Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. "The Unary Resource with Transition Times." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 89–104.

Part I

BACKGROUND

1 CONSTRAINT PROGRAMMING

Constraint Programming (CP) is a *declarative*[1] programming paradigm, that is, a paradigm where programs describe the result of the computation, rather than how to obtain it. This is of great interest, since the programmer can focus only on the result instead of on the complexities of the computation. In particular, CP is mainly used to solve hard combinatorial problems. This paradigm has been successfully used in a broad range of domains, including data mining [AGS16, AGS17], traffic routing [Har+15], scheduling and planning [Lab03, LM12]. In CP, constraints can be used in a modular fashion, as building blocks used to describe the solution of the problem. One of the reasons for the success of the paradigm is the efficient algorithms underlying the constraints, sometimes enabling important reduction of the search space.

1.1 MODELING

Solving a given problem in CP first requires a formal *model* to be found such that finding a solution to the model is equivalent to solving the initial problem. For a given problem, several models may exist. A model is expressed as a Constraint Satisfaction Problem (CSP).

CONSTRAINT SATISFACTION PROBLEMS Formally, a constraint program implements a Constraint Satisfaction Problem (CSP)[2]. A CSP consists of a set of c constraints C to be satisfied altogether and that are imposed on a set of n variables X. For a given variable $x \in X$, its *domain* $D(x)$ is the set of values the variable can take. The minimum and maximum of the domain of a variable x are written $\underline{x}$ and $\overline{x}$, respectively. For a given constraint $c \in C$, we write $X(c)$ the *scope* of the

1 In a weaker sense than the definition from [VRH04].
2 Also referred to as a *Constraint Network*.

constraint c, i.e., the set of variables constrained by c. A CSP can then be seen as the triplet $\langle X, D, C\rangle$. In this thesis, we will always consider the domains to be finite and discrete. The search space is therefore defined as:

$$D_1 \times \ldots \times D_i \times \ldots \times D_n$$

Solving a CSP amounts to finding a feasible assignment for the variables $x \in X$, such that:

$$\bigwedge_{i=1}^{c} C_i$$

Example 1. *Let us consider the 0-1 Knapsack Decision Problem: given a knapsack with a discrete maximum capacity W and a set of n items, each item i having a weight w_i and a value v_i, one has to decide if a minimum total value V can be obtained by taking a subset of the items in the knapsack without exceeding the capacity W. A CSP to model this problem is:*

$$\sum_{i=1}^{n} v_i \cdot X_i \geq V$$
$$\sum_{i=1}^{n} w_i \cdot X_i \leq W$$

where $\forall i \in \{1, \ldots, n\} : D(X_i) \in \{0, 1\}$. This problem is NP-complete.

CONSTRAINT OPTIMIZATION PROBLEMS A CSP can be extended with an objective function O to optimize. A Constraint Optimization Problem (COP) is then a quadruplet $\langle X, D, C, O\rangle$ and solving it consists in finding (one of) the feasible assignments of $x \in X$ such that O is optimum.

Example 2. *The knapsack optimization problem is the COP of Example 1 with the objective:*

$$O(X) = \max_{X} \sum_{i=1}^{n} v_i \cdot X_i$$

This problem is then NP-hard.

1.2 CONSTRAINT SOLVER

Once the initial problem is represented as a CSP, it must be transmitted to an implemented system that will actually perform the computation of the solution. This system is often called a *constraint solver* and it is made of two major complementary components:

- *Search*, that traverses the search space in order to find an assignment of the variables that respects all the constraints, and that possibly optimizes a given objective function. To do so, it narrows the search space by adding arbitrary constraints to the CSP and backtracks if the CSP becomes infeasible (typically a domain gets empty).

- *Propagation*, that infers more valid constraints of a given CSP in order to further narrow the search space. To this end, the solver uses so-called *propagators*, i.e., procedures associated with constraints whose aim is to verify the constraints can be respected and to infer more constraints. Often, the inferred constraints amount to update some variable domains. It is important to remark that several propagators may exist for the same constraint. A propagator is (one of) the concrete procedure(s) that implement(s) the abstract, logical, constraint.

In this thesis, we will often use the term *model* to designate a set of propagators/ filtering procedures that allow solving a given problem in a sound and complete manner using search. One can think of a model as the actual implementation used by the solver to represent the CSP. Moreover, the notation $M \cup \phi_1 \ldots \cup \phi_n$ indicates that the set of filtering procedures $\{\phi_1, \ldots, \phi_n\}$ is added to the model M. This notation will be used in several parts of this document.

1.3 SEARCH

NP-complete problems are usually solved with backtracking search procedures. In particular, Depth-First Search is typically used in CP, because it is a good time-memory trade-off and efficient trail-based state restorations exist[War83, AK99]. As described in [VB06], in a Depth-First Search backtracking algorithm, a node $p = \{b_1, \ldots, b_j\}$ in the search tree is identified by a set of branching constraints where b_i, $1 \leq i \leq j$ is the branching constraint posted at the level i of the search tree. A node p is extended by adding the k nodes $p \cup \{b^1_{j+1}\}, \ldots, p \cup \{b^k_{j+1}\}$ for some branching constraints b^i_{j+1}, $1 \leq i \leq k$.

The branches are often dynamically ordered using a heuristic, with the left-most branch being the most promising. To ensure completeness, the constraints posted on all the branches from a node must be exhaustive (for efficiency reasons, they are typically also mutually exclusive). Usually, branching strategies consist in posting unary constraints (e.g., $X \leq a$ and $X > a$) or binary constraints (e.g., $X \leq Y$ and $X > Y$). In this case, a variable ordering heuristic is used to select the next variable to branch on and the ordering of the branches is determined by a value ordering heuristic.

Definition 1. *A **branching procedure** is a function that, given a search node $p = \{b_1, \ldots, b_j\}$ at level j of the search tree, computes the branching constraints at the next level: $\beta(p) = \langle b^1_{j+1}, \ldots, b^k_{j+1} \rangle$. The branching constraints are contracting [Ben96], i.e., domains can only be reduced. Moreover, we assume that after imposing a constraint, at least one domain is reduced. Finally, the procedure has implicitly access to the current state of the constraint store (not written explicitly for notation brevity).*

Example 1. *β_{ff} is the first-fail binary branching procedure. On the left branch, it assigns the variable with the smallest domain cardinality to its smallest value, and it removes this value from the variable domain on the right branch. Formally, it returns* *two branching constraints c and $\neg c$, with $c \equiv \underset{x \in X}{\operatorname{argmin}}(|D(x)|) = \min(D(x))$, where X is the set of current unbound decision variables and $|D(x)|$ is the cardinality of the domain of x.*

We call *Constraint Branching Tree* the structure that results of a (Depth-First) Search with a model M and a branching procedure. This notion will be used extensively in Chapter 3.

Definition 2. *A Constraint Branching Tree (CBT) t rooted at node p is a (possibly empty) ordered finite sequence of pairs $(\langle b^1, t^1\rangle, \dots, \langle b^i, t^i\rangle, \dots, \langle b^k, t^k\rangle)$, where $\beta(p) = \{b^1, \dots, b^k\}$ are the branching constraints (returned by a branching procedure) and t^i is a CBT for the node $p \cup b^i$. Intuitively, it can be thought of as a tree with branches labeled with branching constraints, to be traversed with a Depth-First Search. The definition of branching procedure ensures the structure is finite. An empty CBT is written $()$. $\mathbb{T}$ is the set of all CBTs.*

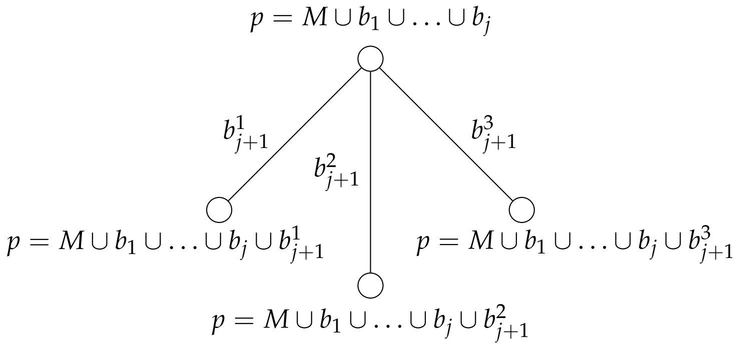

$p = M \cup b_1 \cup \dots \cup b_j$
b^1_{j+1}
b^2_{j+1}
b^3_{j+1}
$p = M \cup b_1 \cup \dots \cup b_j \cup b^1_{j+1}$
$p = M \cup b_1 \cup \dots \cup b_j \cup b^3_{j+1}$
$p = M \cup b_1 \cup \dots \cup b_j \cup b^2_{j+1}$

Figure 1.1: Example of a CBT.

Finally, let us define the function $isSolved(M)$ whose purpose is to verify if a given model is solved in a given state of the search.

Definition 3. *The function $isSolved(M) \rightarrow \{True, False\}$ returns the status of the model M, solved or not. This corresponds to the fact that a solution to the modeled problem is found. That is, for the set of constraints C imposed on X, we have: $\bigwedge_i C_i \wedge \forall x \in X : |D(x)| = 1$, i.e., all variables of the model have a valid assignment (all the constraints are satisfied by the assignment).*

1.4 PROPAGATION

Exhaustively searching through the space of all possible variable instantiations is of course very inefficient. Constraint propagation aims to search less often in inconsistent regions of the search space. Its goal is to infer more valid constraints from (the current state of) a CSP. Each propagator is involved in this inference and a chain reaction may occur, until a fix point is reached, i.e., no inference can be performed by any propagator anymore. Definition 4 formalizes this computation.

Definition 4. *If X is a set of variables, and M is a model that constrains elements of X, the function $fixPoint(M) \rightarrow \{\bot, ?\}$ reduces domains of the variables in X. It returns $\bot$ if a domain is wiped out (i.e., a constraint cannot be satisfied, hence no solution can be found), or $?$ if all domains are still non-empty (infeasibility cannot be proven).*

GLOBAL CONSTRAINT The notion of *global constraint* was formally defined a decade ago [BVH03] through a comparison with its decomposition into simpler constraints. Several definitions are provided depending on the considered perspective. Intuitive descriptions for each perspective are:

- A constraint is global if no such decomposition exists (*expressiveness*).
- A constraint is global if its filtering algorithm is strictly stronger than the decomposition, at least in some cases (*quality of filtering*).
- A constraint is global if its filtering algorithm has a strictly better time and/or space complexity, compared to the filtering done by the decomposition (*computational efficiency*).

In this thesis, we will interchangeably use the term propagator and global constraint to refer to a filtering procedure ϕ that maps a set of domains $D_1, \dots D_n$ to a second set of domains $D'_1, \dots D'_n$ such that $D'_i \subseteq D_i$. The reason is that this work focus more on the implementation perspective of CP than on the modeling one. We will often succinctly describe the CSP of a problem, and then study the details of changing the filtering procedures used to implement the same constraint. We allow ourselves to use the term global constraint as we will study propagation of n-ary constraints.

CONSISTENCY Propagation derives local inconsistencies, or said differently, ensures a certain level of (local) consistency. Many levels of consistencies of a CSP have been defined in the literature. We refer the reader to [Bes06, VB06] for a detailed description of the different consistencies. In this thesis, we only consider a few *constraint-based* consistencies. We provide their definition (largely inspired by the ones given in [Bes06]) hereafter, beginning with the well-known Generalized Arc Consistent (GAC):

Definition 5. *Given a CSP $\langle X, D, C \rangle$, a constraint $c \in C$, and a variable $x \in X(c)$,*

- *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 generalized arc consistent on c for x* iff *all the values in $D(x)$ are consistent with c in D.*

A weaker level of consistency is (generalized)*Bound(Z) Consistency*, that we simply refer as Generalized Bound Consistent (GBC) in this thesis:

Definition 6. *Given a CSP $\langle X, D, C\rangle$, a constraint $c \in C$, a bound support τ on c is a tuple that satisfies c and such that for all $x \in X(c)$, $\underline{x} \leq \tau[x] \leq \overline{x}$. A constraint c is Bound(Z) Consistent* iff *forall $x \in X(c)$, $(x, \underline{x})$ and $(x, \overline{x})$ belong to a bound support on c.*

A weaker consistency is Forward Checking consistency, for which several definitions exist in the literature. A simple constraint-based definition (adapted from [VH89]) is:

Definition 7. *Given a CSP $\langle X, D, C\rangle$, a constraint $c \in C$ is forward checking consistent according to a partial instantiation I on a set $Y \subseteq X(c)$,* iff *I is locally consistent, $|X(c)| \setminus Y = \{x\}$ and for all $y \in Y$, x is arc consistent on c.*

Part II

OPTIMIZATION EVALUATION

2 A VISUAL WEB TOOL TO PERFORM WHAT-IF ANALYSIS OF OPTIMIZATION APPROACHES

In Operation Research (OR), evaluation is of great importance in order to validate a given solution method with respect to existing ones: for example one may be interested in assessing the effect of an improved neighborhood in Local Search, a faster cut for Mixed Integer Programming, or a global constraint in Constraint Programming. When reporting research results, it is critical to have the possibility to provide a meaningful analysis and interpretation of tests performed over representative benchmarks. However, some communities (e.g., Constraint Programming) tend to limit the presentation of the results to tables, sometimes with only a few instances. This can drastically reduce the significance of the derived conclusions for the general case, which should instead be the primary target when an evaluation is performed. Finding a meaningful and effective way to aggregate the results is not trivial and it has a direct impact on the conclusions. Some indicators such as the arithmetic average of normalized measures are well known to bias the results [FW86]. An additional difficulty is the timeout given to the experiments: some methods indeed become better if they are given more time while others are superior at the early stage of the execution.

A *performance profile* [DM02] is a tool that is more and more widely employed to grasp a lot of conclusive information out of evaluated benchmarks. Performance profiles are cumulative distributions for a performance metric that do not suffer from the aforementioned problems. Among other advantages, they directly provide an approximate cumulative (probability) distribution function[1] that a certain

1 Under the assumption the studied benchmarks are representative enough.

method can solve an arbitrary instance. Equipped with such a tool, one is clearly able to make better decisions regarding a solver, based on data from quantitative evaluations.

While performance profiles are useful in practice, there exists no tool to generate and analyze them easily (e.g., via what-if analysis on heterogeneous benchmarks), hampering their broader usage by researchers. The OR community would therefore greatly benefit from an easy-to-use tool to build and export such profiles, with an easy and well-defined input format. The current chapter proposes a free Web tool to fill in this gap.

It is well known that *premature optimization is the root of all evil* [Knu74]. It happens that many OR researchers spend time and energy trying to improve an algorithm that is not the bottleneck of the whole problem. The tool we introduce permits what-if analysis on the performance profiles. We can for instance simulate the effect on the whole computation time of reducing the time complexity of a sub-algorithm (e.g., cut generation, global constraint, local search move).

We first describe the tool, and then illustrate how it can be used for the Mixed Integer Linear Programming technology. It is also extensively used to describe our results in CP in the subsequent chapters of this thesis.

2.1 WEB TOOL DESCRIPTION

In order to facilitate the use of performance profiles and hopefully to spread their usage among the community as a standard evaluation tool, we have built a Web tool to construct them easily. It is publicly available at `http://performance-profile.info.ucl.ac.be/`. It allows generating profiles from a simple and well-defined JSON format, and to visualize the approximate effect of improving the solvers considered in an experimentation.

2.1.1 *Introductory Example*

To describe how to use the tool, we will use a simple running example. While the tool is used to evaluate the performance of solvers, we will use here an analogy with cars. Let us assume that we desire to compare performances of 3 cars, c_A, c_B and c_C. Those cars will be evaluated according to the time that they need to complete tracks (i.e., from our analogy, the problem instances). The benchmark is made of 6 tracks that have different non-exclusive characteristics that can be used as *labels*. In this example, some tracks can have parts with roads and woods. Finally, the time required by a car to complete a track can be split in several *components*. That is, the total time to complete a track for a car is the sum of the times of all components of the car. For example, the car c_A has a time associated with its wheels and its motor (the rest is considered as negligible). The sum of both time components will be considered as the total time for the car. Table 2.1 reports the evaluation results of this fictive experiment: each row corresponds to a given component of a car (e.g., c_{A_w} stands for the wheels of the car c_A) and each column corresponds to a track with labels (e.g., first column has the label *Road*).

As an example, the total time required by the car c_A to finish the first track is $c_{A_w} + c_{A_m} = 100 + 20 = 120$. Notice not all cars must have all components: c_B only has a time associated with its wheels (said differently, the time imputed to the motor is negligible for c_B).

Labels	Road	Road	Road, Wood	Road, Wood	Wood	Wood
c_{A_w}	100	30	40	50	10	20
c_{A_m}	20	3	4	5	1	2
c_{B_w}	10	7	45	55	30	50
c_{C_w}	15	12	35	40	15	25
c_{C_m}	10	3	4	5	1	2

Table 2.1: Results for the introductory car example. c_{A_w} and c_{A_m} stand for the wheels and motor component of c_A, respectively.

A *performance profile* is a cumulative distribution of a performance metric for a solver. Let $\mathcal{S}$ be the set of all considered solvers ($\{c_A, c_B, c_C\}$ in our introductory example) and let $\mathcal{I}$ be the set of instances (e.g., the set of tracks). We also refer to the metric value for the solver s on the instance i as $metric(s, i)$ (e.g., the time required to complete a track in our example). The profile of the solver $s \in \mathcal{S}$ is then given by:

$$F_s(\tau) = \frac{1}{|\mathcal{I}|} \left| \left\{ i \in \mathcal{I} : \frac{metric(s,i)}{\min\limits_{b \in \mathcal{B}}(metric(b,i))} \leq \tau \right\} \right| \tag{2.1}$$

where $\mathcal{B} \subseteq \mathcal{S}$ is the set of baselines, that is, the solvers against which the ratio is computed. If $\mathcal{B}$ contains all the approaches (i.e., $\mathcal{B} = \mathcal{S}$), the definition is the one of the original work introducing performance profiles [DM02]. If $\mathcal{B}$ contains only one solver ($|\mathcal{B}| = 1$), it is the definition used in [VCLS15]. The latter definition is extensively used in this thesis and is more discussed in Chapter 3. Some intermediary settings ($|\mathcal{B}| > 1 \wedge \mathcal{B} \subset \mathcal{S}$) are also possible, in an attempt to make the tool as generic as possible.

When the data from Table 2.1 is given to the tool with $\mathcal{B} = \{c_A, c_B, c_C\}$, the profile given in Figure 2.1 is generated. In this figure, one can observe that the x-axis is divided in 2 linear parts. The first one goes from 0 to 2, while the second one goes from 2 to 12. The bounds of the first part can be set by the user (see section 2.1.2), in order to select the region of τ values to be studied. The end of the second part (from 2 to 12 in the figure) cannot be configured and corresponds to the largest metric ratio computed over all the data. This second region provides a shrunk and long-term view of the profiles.

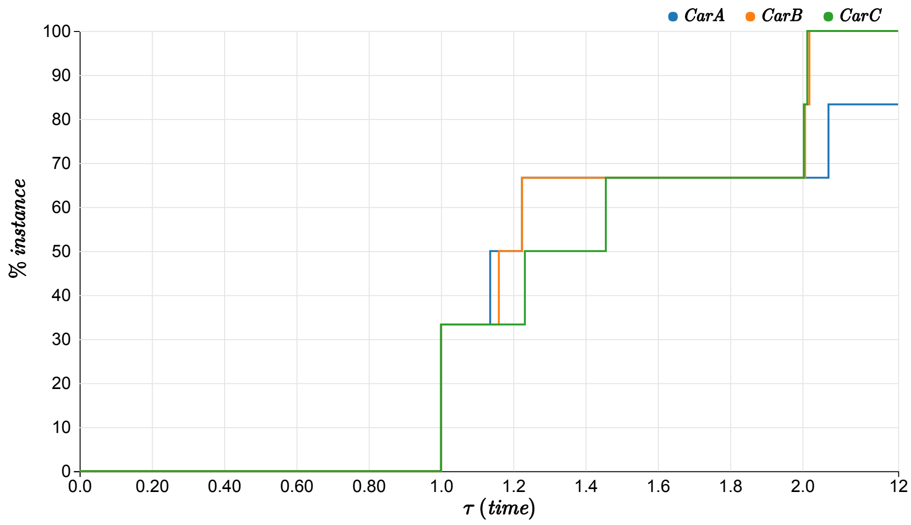


Figure 2.1: Performance profile generated from the data of Table 2.1 with $\mathcal{B} = \{c_A, c_B, c_C\}$.

2.1.2 *Usage*

In this section we describe the input and output formats for the Web tool, using our running example. We then describe the different controls that are part of the graphical user interface, and their utility.

INPUT/OUTPUT: The input consists of a JSON file that must be passed to the interface (see Figure 2.3). It has to be valid according to a JSON schema, available from the Web page. A JSON Schema allows ensuring the input JSON file is correct according to the expected format. As an example, Figure 2.2 shows the JSON file corresponding to the data from Table 2.1:

- Line 2 defines the *metric* name to be used in the plot legend.
- Line 3 defines the *labels*, an array of strings that can be associated with each instance separately.
- Line 4 defines the *instance labels*, an array of arrays. Each element of this array contains the indices of the labels to be associated with the given instance. For instance, the first instance only has the *Road* label, while the third one has *Road* and *Wood* labels.
- Lines 5-17 defines the *data* for the different cars, that is, the solvers employed in the experimentation. Each car/solver is defined as a set of components. For a given instance and solver, the total metric is the sum of the metric value for all components.

```
{
  "metric" :"time",
  "labels" :["Road","Wood"],
  "instances" :[[0],[0],[0,1],[0,1],[1],[1]],
  "data": {
     "Car A": {
       "wheels" :[100,30,40,50,10,20],
       "motor" :[20,3,4,5,1,2]
     },
     "Car B": {
       "wheels" :[10,7,45,55,30,50]
     },
     "Car C" :{
       "wheels" :[15,12,35,40,15,25],
       "motor" :[10,3,4,5,1,2]
     }
  }
}
```

Figure 2.2: An example of input JSON file for the Web Tool. This file must be correct according to the JSON Schema.

To export the graph, we offer two possibilities (see Figure 2.3). First, a button allows downloading the generated profiles as SVG files. SVG is a vector graphics format, similarly to PDF, but that can still be modified if required (using vector graphics tools, or even via a text editor). Moreover, several tools to convert SVG files to PDF exist. A second button provides the possibility to export the plot as a minimal Web page. This page can then be included by the user in another Web page of his choice.

Figure 2.3: Input as a JSON file and outputs as an SVG file or a Web page.

CONTROLS: Several controls are available for configuring the plot (see Figure 2.4). First, the user can define the non-empty set of solvers to be used as baselines. Moreover, a set of optional labels can be assigned to each instance in order to characterize them (e.g., number of variables, number and type of constraints, authors of the instance, . . .). Some instances might have no label. Using the tool controls, it is possible to specify which labels should be considered in the profile: removing a label from the set will filter out any instance characterized by this label before generating the profiles. This allows investigating easily how the performance profiles are affected by specific groups of instances in the benchmark.

For each *component* in the input data (to which values for the metric are associated), a slider is generated in order to allow the user to study the impact on the performance of being able to reduce the metric for the component by a given ratio. For example, the user may use the slider to quickly assess the impact on the profiles of making wheels that are 20% more efficient. This feature is useful for assessing the potential benefits of a certain algorithm improvement, *before actually starting to research how the improvement can actually be obtained*. Alternatively, this technique allows one to estimate the amount of improvement that would be necessary to achieve a given goal.

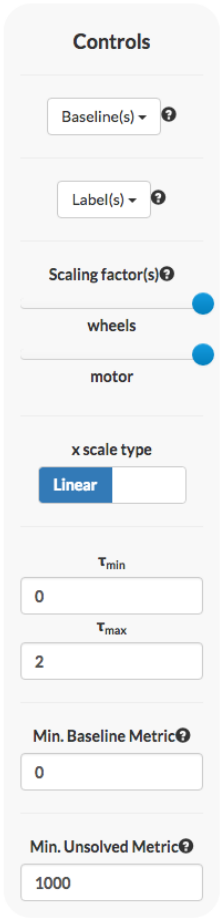


Figure 2.4: The different available controls.

The minimum and maximum τ values can be specified via an input box, thus allowing the user to focus on a specific τ region, or to have a general viewpoint. The button *x scale type* can be used to switch between a linear and a logarithmic scale. If the scale is linear, it is split in two parts: the first one goes from τ_{min} to τ_{max} and the second one goes from τ_{max} to the maximum τ value computed out of the data. This allows focusing on the desired τ region (i.e., τ_{min} to τ_{max}) while also having a "long-term" view.

Finally, a threshold for the minimum metric value of the baselines can be specified. That is, if one of the approaches considered as a baseline has a value smaller than this threshold for a given instance, this instance is filtered out. This can be used to remove noisy measurements (e.g., imprecision of time measurements when the solution time is very small) or non-significant measures. A second threshold for the *minimum unsolved metric* can be specified. That is, if the value of the metric for a solver on a certain instance is larger than this second threshold, the instance will be considered as unsolved. Formally, the metric value for this solver-instance pair is considered infinite. This approach can for example be used to define the time-out value for an experimentation.

2.2 USE CASES

We showcase how our tool can be used for Mixed Integer Linear Programming and Constraint-Based Local Search[2].

2 We also use the tool extensively in Chapter 3 in the case of CP.

2.2.1 *Mixed Integer Linear Programming*

A well-known framework to solve Mixed Integer Linear Programs is the Branch and Cut method, a sophisticated version of Branch-and-Bound. At each node of the search tree, cutting plane algorithms can be used to find additional linear constraints satisfied by all feasible solutions. Those *cuts* sometimes allow discarding important parts of the search space, leading to non-negligible improvements. However, their computation time can be large, so it would be of great interest to know the gain provided by a more efficient computation. Yet if the gain is shown to be negligible, research should focus on other aspects of the solving process.

Let us study the *Concorde* solver [App+06], that has been shown to be efficient at solving the Travelling Salesman Problem. In particular, its authors introduced *local cuts* [App+01, App+11], that were crucial to solve some instances. We are interested in knowing the potential of being able to compute those cuts more efficiently.

For that purpose, we define the hypothetical solver $concorde_{hypothetical}$ as the Concorde solver where the local cuts are computed instantly in $\mathcal{O}(1)$. The performance of $concorde_{hypothetical}$ is an upper bound of any performance that could be obtained by improving local cuts in the Concorde solver.

We use the performance profile definition from [VCLS15], that is, exactly one of the approaches is used as a baseline in Equation 2.1. We use that definition because it allows reading the gain of an approach as compared with a given baseline approach (in this case the Concorde solver, written $concorde$). For our case, Equation 2.1 becomes:

$$F_s(\tau) = \frac{1}{|\mathcal{I}|} \left| \left\{ i \in \mathcal{I} : \frac{time(s,i)}{time(concorde,i))} \leq \tau \right\} \right|$$

We considered the TSPLib [Rei91] set of instances, augmented with the VLSI[3] instances. For each instance, we set a time limit of 900 seconds, and we measured the total time required to compute the local cuts. Instances that could not be solved to optimality or lasted less than 1 sec. were filtered out.

Figure 2.5 compares the Concorde solver with $concorde_{hypothetical}$. The profile $F_{concorde_{hypothetical}}(\tau)$ provides an upper bound of the performance that can be obtained by improving the computation of local cuts. One can read
$F_{concorde_{hypothetical}}(0.9) \simeq 0.2$, which means that for $\sim 20\%$ of the instances,
$concorde_{hypothetical}$ is (at least) faster by a factor 0.9 as compared with the baseline solver $concorde$. This means that by improving local cuts computation, we would be able, *at the very most*, to save $\simeq 10\%$ of the solving time for only $\simeq 20\%$ of the instances. Or said differently, for $\sim 80\%$ of the instances, the speed-up factor cannot be larger than $\simeq 0.9$. Similarly, one can see that $F_{concorde_{hypothetical}}(0.7) \simeq 0.05$, so for $\sim 95\%$ of the instances, the speed-up factor cannot be larger than $\simeq 0.7$. This is an indicator that working on this time reduction has a low chance to provide substantial gains in practice.

3 http://www.math.uwaterloo.ca/tsp/vlsi/index.html

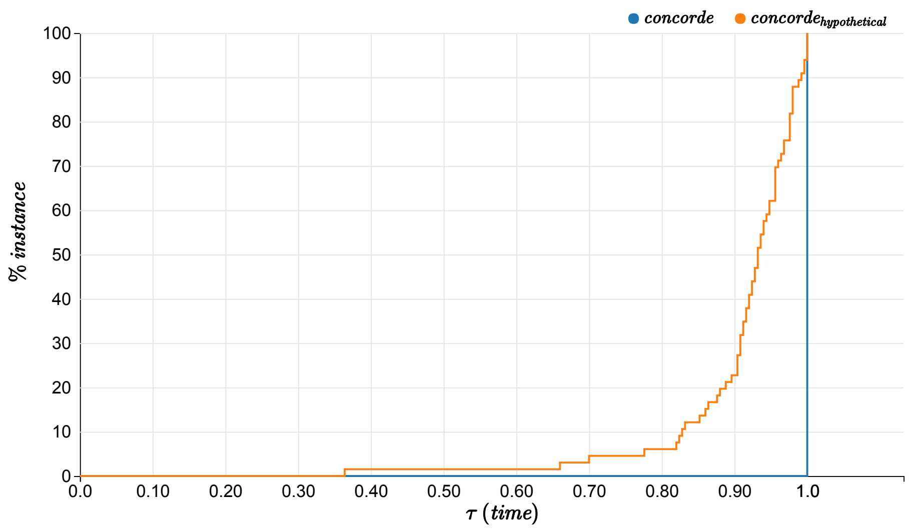


Figure 2.5: Performance profiles for the Concorde Solver on the TSPLib and the VLSI instances.

2.2.2 *Constraint-Based Local Search*

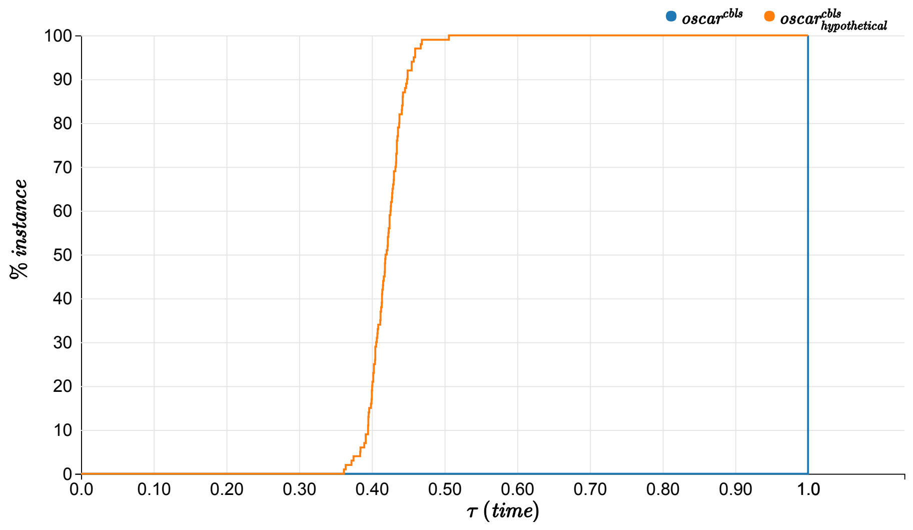


Figure 2.6: Perf. profiles for OscaR (CBLS) on the Pick-up and Delivery Problem.

Let us now consider the solution of a Pick-Up and Delivery Problem using Constraint-Based Local Search [HM09, MVH17]. We generated 100 instances with 500 cities, 10 vehicles, random distances and random mandatory precedences between the cities. We used the OscaR (CBLS) solver [Osc12]. Again, we measured the time taken by the capacity constraint invariant, as it can be time consuming. We are interested in knowing if reducing the time complexity of this invariant can be beneficial. The results are given in Figure 2.6. An hypothetical $\mathcal{O}(1)$ capacity constraint invariant would provide a gain factor between ~ 0.4 and ~ 0.5, *for all instances*. This research direction seems thus more promising.

3 A FAIR EXPERIMENTAL FRAMEWORK FOR GLOBAL CONSTRAINT EVALUATION

Not surprisingly filtering is still an important research topic in the CP community. Unfortunately, rigorous tools and methodologies to analyze the performance of filtering algorithms for global constraints are missing. This chapter introduces generic tools and a methodology to probe the potential of filtering techniques and to assess the likely impact of specific improvements (e.g., time complexity, better implementation). Such tools would allow researchers to focus their efforts in the most promising directions. For example, a researcher may be interested in finding a more efficient way to enforce Generalized Arc Consistent (GAC) for a specific constraint: with the current methodologies, knowing if this line of research is worth investigating remains an open question until a new algorithm is actually devised and evaluated. With the approach we propose, instead, it becomes possible to estimate and bound a-priori the potential effectiveness of a propagator improvement.

Tools to analyze the solving process of CP are not new. Some interesting visual tools have already been introduced. For instance, the Oz/Gecode explorer allows visualizing the search tree [Explorer97] and interacting with it through a graphical user interface. CP-Viz is a generic visualization platform for CP [Sim+10] allowing an advanced post-mortem analysis of the solving process. In CP-Viz the user can visualize each constraint and its filtering, which is very useful for teaching CP or debugging models and constraints. The visual search tree profiling tool introduced in [Shi+15] allows comparing search trees visually with convenient navigation techniques, letting the user compare and understand the differences in terms of search space exploration between different configurations and models.

Unfortunately those visualization tools do not allow a fine-grained analysis of the time benefits of adding a specific filtering to an existing model for a large set of

instances. These tools do also not allow evaluating what would be the benefit of reducing the computation time of a specific filtering procedure.
The definitions of *global constraint* given in Section 1.4 provide an elegant framework to theoretically compare a global constraint with a decomposition, but do not explain how to assess the practical benefits provided by a global constraint on a set of representative instances. One of the contributions of this chapter is a practical and rigorous framework as an attempt to fill in this gap.

For preliminary analysis, standard profiling tools already allow discovering the fraction of time spent in each propagator, making it possible to estimate roughly potential speedups. This chapter goes one step further and tries to answer those questions by proposing a methodology and visual analysis tools inspired by performance profiles [DM02].

A typical approach to compare different filtering algorithms consists in measuring time and the number of backtracks with respect to a baseline approach, on a set of benchmark instances that are solved to completeness. This allows assessing the propagator performance, but provides little or no information on the consequences of its speed-up. It is also common to use static search strategies (e.g., fixed variable heuristic, minimum value) to make the evaluation fair and rigorous since a stronger filtering has a guarantee to explore a reduced search tree. The rationale behind static strategies usage is that the search nodes order is known a priori and is not influenced by the current solver state, e.g., by current domains filtered by the evaluated algorithms. A first drawback of this approach is the risk to bias the analysis, since dynamic strategies are often preferred in practice. Second, instances that can be solved to completeness (required to ensure that the same search space is visited) are generally small, which may not be the case for real applications. Third, differences in the complexity of filtering algorithms become more relevant as the instance size grows: therefore, being forced to focus on small instances may lead to misjudging the performance gap between different propagators.

CONTRIBUTIONS We propose to extend the traditional evaluation approach with two main contributions:

1. A method to compare propagators in a principled fashion, by storing and *replaying* search trees (see [VCLS15, VCLS18]), in order to enable fair comparisons with arbitrary search strategies and instance sizes. Its main asset is that it enables to measure the exact impact of a propagator on the solving of a given problem. Shishmarev et al. already noticed the importance of replaying, in the context of search tree visualisation [Shi+15, Shi+16b] and better understanding of learning solvers behavior [Shi+16a].

2. A simple model to evaluate the potential for improvement that a propagator has. This is achieved by instrumenting the solver to collect information about the constraint whose potential is to be evaluated.

CHAPTER OUTLINE This chapter first motivates the need for our framework. It then introduces our replay technique used to make a fair evaluation of filtering procedures, and describes how to implement it. We then propose our simple approach to assess the impact of a propagator improvement. For ease of access, our method has been made an integral part of the OscaR solver [Osc12], and Section 3.4 explains how the method can be used. We finally give a case study about different propagators for several constraints using our evaluation approach: ALLDIFFERENT, CUMULATIVE, BINPACKING and UNARY with transition times. As for the impact of propagator improvements, we focused on CUMULATIVE and the Revisited Cardinality Reasoning for BINPACKING. Applying our method allowed obtaining valuable insights: for example, we found that (somehow counter-intuitively) Energetic Reasoning (ER) cannot provide improvements on a number of typical scheduling instances, *even if the run-time of the propagator is reduced to zero*. Conversely, investigating different, complementary forms of filtering for CUMULATIVE has a much greater potential. We also observed that changing the search strategy may have a significant impact on the effectiveness of some propagators.

3.1 MOTIVATION

This section provides the motivation that impelled us to propose our approach. We wish to design methods that allow a thorough analysis of the behavior of propagators in CP and to understand their potential. More precisely, we want to characterize exactly how much search space reduction and time gain is provided by the additional use of a given propagator. This is usually done by comparing the execution with and without the evaluated propagator. Currently, the comparison is made using dynamic and static search strategies. Both methods have their merits, but also substantial limitations:

- Dynamic strategies have a significant impact on which part of the search space is visited first, sometimes providing tremendous solving speed-up as compared to static strategies. They therefore intuitively make for more realistic evaluations since they are the ones programmers use in practice. Nevertheless, they allow poor control and limited insights in the behavior and potential of the propagator itself, since it is impossible to quantify how much additional filtering and search decisions were influenced by each other.
- On the other hand, static strategies allow measuring exactly the search space reduction provided by a propagator. However, they are rarely used in practice because they generally provide poor solving time performance. At the same time, dynamic and static search strategies perform differently, so an evaluation of a propagator with a static strategy can hardly be generalized to the usage with a dynamic strategy. This makes the comparison with static strategies somewhat artificial, since it is not representative of practical usage.

We argue that the designers of global constraints are currently missing an additional methodology that enables the same degree of control and ease of analysis of static strategies, in an experimental setup that is almost as realistic as that of dynamic strategies. In our opinion, as a key feature, such an approach should retain the ability of static strategies to distinguish clearly the benefits that are provided by inference and those that come from the search strategy. One of the main contributions of this chapter is a framework to perform global constraints evaluation that owns these important characteristics.

SECTION OUTLINE The motivation for our approach is presented as follows: we start by formally discuss evaluation of propagators in CP, and we identify the conditions that enable one to measure precisely the impact of inference on the search performance. We use the formalism to discuss the current evaluation methods and to point out their limitations. Finally, we show how the approach we propose fills in the gap between the traditional evaluation methods, and allows the designer of a global constraint to obtain more insights in the algorithm behavior and its potential.

3.1.1 *Evaluation of Global Constraints*

Formally, we consider the problem of evaluating a filtering function ϕ that maps a set of domains $D_1, \dots D_n$ to a second set of domains $D'_1, \dots D'_n$ such that $D'_i \subseteq D_i$.[1] In practice, ϕ may represent a propagator for enforcing GAC or a domain-specific consistency level (e.g., Energetic Reasoning), or it can be some kind of meta-propagation scheme such as Singleton Arc Consistency [BD05].

Like many other approaches, we measure the performance of the algorithm to compute ϕ by comparing the time needed to solve a target CSP using a *baseline* model $M \cup \phi_M$ and an extended model $M \cup \phi$ (see Chapter 1 regarding this notation). We call ϕ_M the *baseline filtering function* (e.g., a decomposition of a global constraint) that is to be replaced in the extended model by the algorithm ϕ that we want to evaluate. We also require the property that ϕ_M is subsumed by ϕ, i.e., ϕ performs *at least* the same deductions as ϕ_M (for instance, in the case of an AllDifferent constraint, ϕ_M can be a binary decomposition while ϕ would be the GAC filtering).

Notice that it often happens that we wish to evaluate the use of a stronger propagator while keeping weaker algorithms in the propagation queue with a higher priority, as stronger propagation can come at the cost of a higher time complexity. This also has to be done if we want to compare ϕ with a baseline filtering function ϕ_M that ϕ does not subsume. In this case, we compare the model $M \cup \phi_M$ with $M \cup \phi_M \cup \phi$, i.e., we construct the extended model by *adding* ϕ to the baseline model instead of replacing ϕ_M by ϕ. To reduce the notation, when ϕ is added to the baseline model instead of replacing ϕ_M, one can denote the baseline model by M and the extended model by $M \cup \phi$.

1 Some particular CP approaches, such as the ones using Decision Diagrams [Ber+16], perform inference on internal data structures. But in the end, potential partial assignments are removed by propagation (e.g., by removing an arc of a Decision Diagram) and the inference made on Decision Diagrams can always be projected to variable domains.

To ensure the measured difference between the two models is only due to propagation alone, two conditions must be respected:

C.1 The two runs must explore the same search space;

C.2 All search nodes (and therefore the solution nodes) that are visited by both runs are visited in the same order.

The first requirement is always met as long as $M \cup \phi_M$ and $M \cup \phi$ are semantically equivalent (i.e., they have the same solutions) and the problem is solved to completeness (feasibility or optimality). Without the second requirement, $M \cup \phi_M$ or $M \cup \phi$ could get an unfair advantage if the search strategy quickly allows hitting a feasible solution (and stops, for feasibility problems), or a high-quality solution (and gets a good bound, for optimality problems). The next section discusses existing evaluation methods from the perspective of those two requirements.

3.1.2 *Current Evaluation Methods and their Limitations*

There exist two families of search strategies in CP: *static* and *dynamic* strategies. For static strategies, also known as lexicographic-order search strategies, the order in which the branching constraints are posted is (implicitly) known prior to the search process, meaning that the pruning happening at a search node has no effect on this order. An example is a branching procedure that always returns two constraints C and $\neg C$, such that C assigns, according to a fixed variable order that is known a priori, the first unassigned variable to its smallest domain value. On the other hand, dynamic strategies do take the search state into account when creating the branching constraints. They are an essential asset of CP to get good performances. An example of dynamic strategy is the branching procedure described in Example 1 (Chapter 1).

While static and dynamic strategies are important, they both have strong limitations from an evaluation perspective, as we argue hereafter.

DYNAMIC STRATEGIES can be used for evaluation while respecting condition **C.1** under the condition that instances are solved to completeness. This is already a strong limitation, since it prevents evaluation on large-scale instances. Moreover, dynamic strategies cannot guarantee that condition **C.2** is satisfied in general. Let us illustrate with a first hypothetical example how this can lead to unfair conclusions (see Figure 3.1). On the left tree, a model $M \cup \phi_M$ is used, and the branching procedure returns the branching constraints $\langle C, \neg C \rangle$ at the root node. This leads the search towards a region where the only solution of the problem (found in the green node) lies. Let us now consider the right tree: the same branching procedure is used but the model is extended to $M \cup \phi$, such that more domain pruning occurs at the root node. It is possible, due to the dynamic nature of the search strategy, that the branching constraints are swapped so that the constraint $\neg C$ is imposed on the left branch. The whole left subtree will have to be traversed before the correct search region can even be considered. It would be unfair to impute this bad performance to ϕ solely, since the propagator was in fact able to prune (perhaps significantly) more, and could help in practice with a slightly different context (e.g., with a different search strategy). When using dynamic strategies, the unpredictable effects of combined search and propagation prevent measurement of benefits/harm induced by the additional pruning[2]. The effects of ϕ should be isolated to really quantify the amount of pruning that a new propagator can perform.

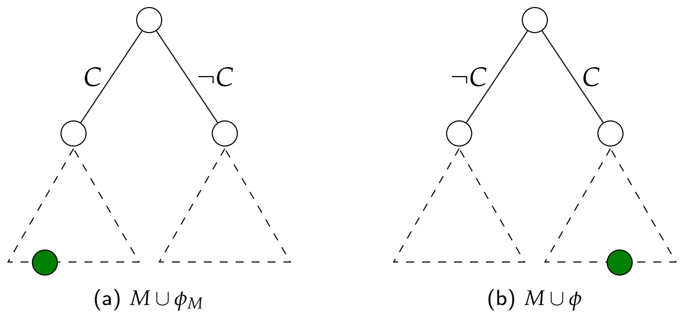


Figure 3.1: Comparison of searches by two different models $M \cup \phi_M$ and $M \cup \phi$ with the same dynamic search strategy.

Let us insist on this point with a real example. Example 2 illustrates how using dynamic strategies to evaluate a global constraint can lead to unfair conclusions.

Example 2. *Consider the first* BL *instance [BLP00] with 20 activities for the Resource Constrained Project Scheduling Problem (RCPSP). The Time-Tabling propagator and Energetic Reasoning Checker (ERC) [BLPN01] are used for the* CUMULATIVE *constraint in the baseline model M. If the branching procedure β_{ff} of Example 1 is used, 100 nodes are required for finding the optimal solution. However, if the ER Propagator ϕ is used* ***additionally*** *(i.e., together with the ERC and Time-Tabling propagator, and hence additional pruning is possible in $M \cup \phi$), then 124 nodes are required. While the model $M \cup \phi$ has been able to prune more, more nodes were required to find the optimal solution. The blame for this counterintuitive behavior is on the interleaving of propagation and branching, rather than on the ER propagator itself. Indeed, in this case, more propagation occurred at a certain node, and the branching procedure generated a different sequence of branching constraints such that a behavior similar to the one illustrated in Figure 3.1 happened. So, if the solution time is used as a metric for the evaluation, in such a case a clear non-beneficial bias[3] would apply against ER. Notice an opposite bias is also possible.*

2 Notice that a similar reasoning could be done for another example where the left and right tree are attributed to $M \cup \phi$ and $M \cup \phi_M$, respectively.

3 In principle, one might remove most of the bias by using statistics, i.e., by running experiments with many instances and many search strategies. However, this is an expensive process and it is not always viable.

STATIC STRATEGIES allow ensuring that condition **C.2** is fulfilled, by definition. Condition **C.1** is easily respected, by solving instances to completeness, possibly after having reduced the search space by adding constraints at the root node. This explains why researchers sometimes use those strategies to evaluate a global constraint. However, a major issue with that approach is that static search strategies are rarely used in practice, since they are often outperformed by dynamic ones. So the obtained conclusions may be biased, as the propagator will rarely be used in that context.

With Example 3, let us briefly showcase that static strategies do not line up with reality.

Example 3. *We consider the same comparison made in Example 2. Let us use a lexicographic branching strategy on the starts of the activities: on the left branch, the activity start is assigned to its minimum value, on the right branch this value is removed from its domain. Without ER, one obtains 20081 nodes to prove optimality, while with the additional propagation one only requires 8789 nodes. Since the strategy is static, we need less nodes with more pruning, as expected. However, in Example 2, we made the opposite conclusion with a dynamic strategy. More importantly, the results reported here are quite different from those a user would get in practice: both approaches require much more nodes and the ratio of the number of nodes required by* $M \cup \phi$ *by the number of nodes required by* M *goes from* 1.24 *to* ~ 0.44*.*

Finally, Example 4 should convince the reader that when evaluating a filtering algorithm, one may get opposite outcomes depending on whether a static or a dynamic search strategy is used.

Example 4. *Let us consider the 15th* BL *instance [BLP00] with 25 activities for the RCPSP. We wish to evaluate if adding ER to solve the instance is beneficial. Let us first use the static strategy of Example 3. The required number of nodes to solve the instance without ER is 193038 while only 24140 nodes are necessary if it is used additionally. Moreover, it takes* ~ 16 *and* ~ 13 *seconds*[4] *to solve the instance without and with the additional propagator, respectively. One could therefore conclude that ER should be used. However, if we make the same comparison using the* Set-Times *dynamic strategy from [LP+94], the results differ: only 51754 (respectively 18064) nodes are required without (respectively with) ER. The ratio is therefore quite different (*$24140/193038 \simeq 0.125$ *as compared to* $18064/51754 \simeq 0.349$*), but more importantly, the solution time is* ~ 1.5 *seconds in the first case and* ~ 4 *seconds in the latter case. We would therefore consider in this case that Energetic should* not *be used to solve the instance. This illustrates that there is a need to line up with dynamic strategies that are actually used in practice when we perform the evaluation of a global constraint, since static branchings can lead to opposite conclusions. Although illustrated on a single instance, this phenomenon is not rare on a complete benchmark suite.*

4 With a 2.2 GHz Intel Core i7 processor.

3.1.3 *Aim of this work*

In this chapter, we suggest a framework to evaluate a global constraint ϕ while respecting conditions **C.1** and **C.2** so that any difference in execution can only be attributed to additional propagation provided by ϕ. This point was already raised in [VCLS15] and then in [Shi+15, Shi+16b]. At the same time, we wish for the ability to use search strategies that are as close as possible to those actually employed in practice, and we want to keep the possibility of using large instances, so we wish to avoid forcing completeness to ensure condition **C.1**.

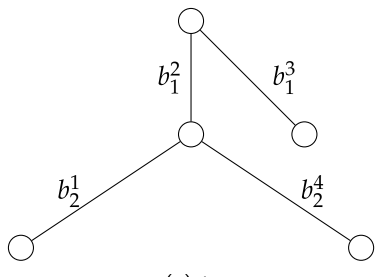
In addition, to better understand the potential of improving propagators from a computation time point of view, we suggest the simple yet very informative concept of *fictional propagator*. In brief, they allow estimating the impact on the solving time of a problem, assuming that a propagation time improvement has been found (e.g., reduction of the time complexity of a given propagator).

Notice that the approach we propose is not necessarily meant to replace existing ones. We simply think that our framework can strengthen the conclusions of an evaluation. But one could use our framework together with traditional evaluations: for instance, by comparing the results we obtain with those of a traditional evaluation based on dynamic strategies, it is possible to measure the effect of the interplay between the stronger inference and the search.

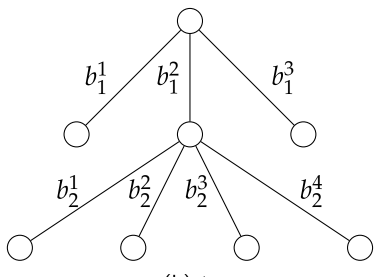
In order to ensure conditions **C.1** and **C.2** are satisfied during our evaluation, the compared models must traverse Constraint Branching Tree (CBT)s (see Chapter 1) linked by a well-defined relation that we call *CBT inclusion* (Figure 3.2 provides an illustration of CBT inclusion). It is formally defined with :

Definition 8. *A CBT* $t_1 = (\langle b_1^1, t_1^1\rangle, \dots, \langle b_1^i, t_1^i\rangle, \dots, \langle b_1^n, t_1^n\rangle)$ *is* included *in a CBT* $t_2 = (\langle b_2^1, t_2^1\rangle, \dots, \langle b_2^j, t_2^j\rangle, \dots, \langle b_2^m, t_2^m\rangle)$*, denoted as* $t_1 \subseteq t_2$ *iff :*

- $\exists (1 \leq l_1 < \dots < l_n \leq m)$ *s.t.* $(b_1^i = b_2^{l_i} \wedge t_1^i \subseteq t_2^{l_i})\ \forall 1 \leq i \leq n$
- *or* $t_1 = ()$ *(empty sequence)*



(a) t_a



(b) t_b

Figure 3.2: CBT inclusion : $t_a \subseteq t_b$.

In the next section, we propose an approach based on *replaying* CBTs that ensures the CBT traversed by the extended model $M \cup \phi$ (in a depth-first manner) is included in the CBT traversed by the baseline model $M \cup \phi_M$.

3.2 FAIR EVALUATION THROUGH THE *replay* TECHNIQUE

This section first describes our *replay* technique [VCLS15, VCLS18] from a high-level perspective. It then discusses its limitations and provides details on how to implement the approach.

3.2.1 *Replaying (High-Level Description)*

The goal behind the replaying technique is to be able to evaluate only the effect of additional propagation, while using a search heuristic that is as similar as possible to the ones used in practice. To do so, we first generate a CBT, and replay it using the models to be evaluated. Replaying is therefore a two-step procedure:

1. Generation of a CBT to be replayed.
2. CBT replay with one (or several) models to be evaluated.

CBT GENERATION is done using a function $generate(M, \beta) \rightarrow \mathbb{T}$ that returns a CBT from a model M and a branching procedure β. It relies on the functions $fixPoint(M)$ and $isSolved(M)$, defined in Chapter 1. Notice these definitions are provided for the sake of characterizing the semantic of our approach, yet they do not necessarily correspond to the implementation.

The function $generate(M, \beta)$ is then:

$$generate(M,\beta) \rightarrow \mathbb{T} = \begin{cases} (\langle \beta(M)^1, & \\ generate(M \cup \beta(M)^1, \beta)\rangle, \ldots, & \\ \langle \beta(M)^k, & \\ generate(M \cup \beta(M)^k, \beta)\rangle, \ldots, & \\ \langle \beta(M)^n, & \\ generate(M \cup \beta(M)^n, \beta)\rangle) & \text{if } fixPoint(M) \neq \bot \\ & \wedge \neg isSolved(M) \\ () & \text{otherwise} \end{cases}$$

It can be computed with a classic CP Depth-First Search. Example 5 illustrates a CBT generation, and Figure 3.3a provides a visual example of a generated CBT that will be replayed further in this chapter.

Example 5. *Consider the branching procedure β_{ff} of Example 1 (Chapter 1) and the model $M = \{x > 0 \implies y > 2\}$, where $x \in \{0,1,2,3\}$ and $y \in \{0,1,2,3,4\}$.* $generate(M, \beta_{ff}) = (\langle x = 0, (\ldots)\rangle, \langle x \neq 0, (\langle y = 3, (\ldots)\rangle, \langle y \neq 3, (\ldots)\rangle)\rangle)$

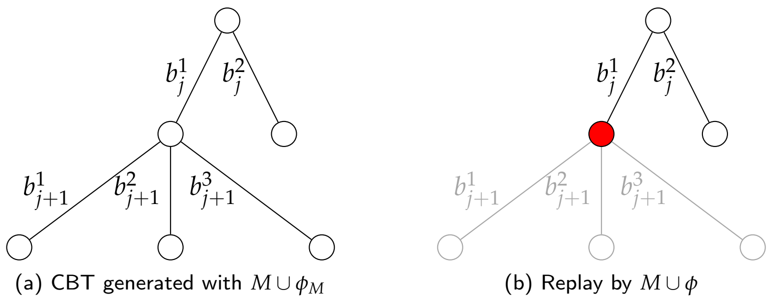


Figure 3.3: A CBT generated by a model $M \cup \phi_M$ and replayed by an extended model $M \cup \phi$.

CBT REPLAY: Let $t = (\langle b^1, t^1 \rangle, \dots, \langle b^i, t^i \rangle, \dots, \langle b^n, t^n \rangle)$. Let us define the function $replay(t, M)$ whose purpose is intuitively to re-traverse a CBT t using a model M and to return a new CBT t_{incl} such that $t_{incl} \subseteq t$:

$$replay(t, M) \to \mathbb{T} = \begin{cases} (\langle b^1, replay(t^1, M \cup b^1) \rangle, \dots, \\ \langle b^k, replay(t^k, M \cup b^k) \rangle, \dots, \\ \langle b^n, replay(t^n, M \cup b^n) \rangle) & \text{if } fixPoint(M) \neq \bot \\ & \wedge \neg isSolved(M) \\ () & \text{otherwise} \end{cases}$$

The rationale behind this function is to make sure that exactly the same search space is visited. Moreover, we want to guarantee that the modifications made to the constraint store (i.e., adding or removing constraints to the store, and modifying the variable domains accordingly) are done in the exact same order. From the definitions, we can see that, as long as β is a deterministic function and ϕ_M is subsumed by ϕ, we have that (Property 1 and 2):

Property 1. $\forall \beta : replay(generate(M, \beta), M) = generate(M, \beta)$

Property 2. $replay(t, M \cup \phi) \subseteq replay(t, M \cup \phi_M)$

In other words, replaying with the original model leads to the original CBT, and extending the model leads to a CBT that is included in the original one.

An illustration of a replay of the CBT of Figure 3.3a is given in Figure 3.3b. In this figure, the extended model $M \cup \phi$ is able to prove infeasibility at the red node, i.e., before the baseline model $M \cup \phi_M$. The time required to visit the 3 gray nodes is therefore saved.

PROPAGATOR EVALUATION: the evaluation of a propagator ϕ is simply done by computing in sequence:

$$t \leftarrow generate(M \cup \phi_M, \beta) \tag{3.1}$$

$$replay(t, M \cup \phi_M) \tag{3.2}$$

$$replay(t, M \cup \phi) \tag{3.3}$$

Then we compare the results of the two latter runs, which both use *replay* and hence incur the same search overhead. It is important that the generate run is done with the baseline problem $M \cup \phi_M$, because, thanks to the additional propagation performed by ϕ, the run with $M \cup \phi$ may skip some parts of t. However, all of the runs will always explore the same search space and visit the shared nodes in the same order.

This approach offers two significant advantages: 1) it allows tackling arbitrarily large instances, since a limit (e.g., time or number of nodes) can be enforced on the first run and the replays will still be guaranteed to explore the same search space. 2) It allows using any search strategy, including dynamic ones, making the evaluation more realistic.

Interestingly, if a limit to the generation is imposed, $M \cup \phi$ might actually find one additional solution. This occurs if the generation is stopped at an internal node with a partial assignment that is part of a solution, not found by $M \cup \phi_M$ since the generation was stopped due to the imposed limit. In this case, $M \cup \phi$ can remove more domain values inconsistent with the partial assignment, and by doing so, it might reduce the domains up the point where a total assignment is found, i.e., an additional solution is discovered.

Finally, let us designate as $metric(t, M)$ the metric quantity[5] required to replay the CBT t with the model M. In particular, we respectively write $time(t, M)$ and $backtracks(t, M)$, the time and the number of backtracks needed to traverse t.

3.2.2 *Limitations*

There are a few limitations to our approach that we must acknowledge. First, one can only use *monotonic* [Tac09, ST09] propagators in the baseline model M. A monotonic propagator is a propagator ϕ such that $D_i^1 \subseteq D_i^2 \implies \phi(D_i^1) \subseteq \phi(D_i^2)$, for any variable i considered by the propagator. Intuitively, this means that the more the domains are reduced, the more the propagator can infer inconsistent values. This property is required because once the model is extended with ϕ, more pruning might happen because of ϕ, implying reduced domains. If some propagators of M are not monotonic, they might prune less than when the CBT was generated. This has undesirable implications. For instance, one could reach a leaf solution node while still having unbound solution variables.

Another limitation is that we only allow the evaluation of a propagator ϕ by comparing a model $M \cup \phi_M$ with an *extended* model $M \cup \phi$. This requirement is due to ensure that $replay(t, M \cup \phi) \subseteq replay(t, M \cup \phi_M)$. This means that if we

5 Similarly to the definition given in Chapter 2.

have two propagators ϕ_1 and ϕ_2 for a constraint that do not subsume each other, one cannot use the model $M \cup \phi_1$ as a baseline and replay with only $M \cup \phi_2$. One would have to replay with $M \cup \phi_1 \cup \phi_2$, or generate with M only (if it is sufficient to solve the problem in a sound manner) and replay with $M \cup \phi_1$ and with $M \cup \phi_2$.[6]

Finally, one could argue that when using the replay, we are comparing the stronger model $M \cup \phi$ in a search tree that would never be visited in practice, since it has been generated using the *baseline* model. We argue that in practice the replayed search tree is often similar to what it would be using the real dynamic search. We conducted a few small experiments to illustrate this. The results are given in Table 3.1. We first experimented with the Golomb ruler (length 11). The CBT was generated with the branching procedure of Example 1 using forward checking as the filtering technique for the AllDifferent constraints. The replay uses GAC AllDifferent constraints. The number of nodes was decreased from 7386480 to 2929035 using the stronger filtering. Even though the search tree is drastically reduced (more than divided by two), we measured that there are $\sim 99\%$ of (local) *matching decisions*, i.e., decisions imposed during the replay that are exactly the same as the ones that would locally be returned by the real dynamic branching procedure if it was called at each search node of the replayed tree. This demonstrates that although being a static strategy, decisions of the replay are very similar to what would happen in practice with the dynamic strategy. We then considered an instance of the Job Shop Scheduling problem (36 activities) with a Conflict Ordering Search strategy [Gay+15], a state-of-the-art dynamic search strategy. The baseline model $M \cup \phi_M$ only uses binary constraints for the Disjunctive constraints, and the algorithms of [Vil07] are used in $M \cup \phi$. In this case, the node ratio is ~ 0.44 and the percentage of matching decisions is $\sim 84\%$, which is still very large. Finally, we experimented with the Traveling Salesman Problem with the branching procedure of Example 1. $M \cup \phi_M$ uses a sum of Element constraints, while $M \cup \phi$ uses the MinimumAssignment constraint [Foc+99] with exact reduced costs, as proposed in [DCP16]. We considered the instance *gr21* from the TSPLib [Rei91]: the node ratio is ~ 0.09, and $\sim 22\%$ of the decisions are matching. This is less than for the other two problems, but the search space reduction is more drastic.

Clearly, the comparison between $M \cup \phi_M$ and $M \cup \phi$ remains artificial to some degree, because an actual dynamic strategy may behave differently for the run using $M \cup \phi$. Still, the ability to ensure conditions **C.1** and **C.2** and therefore isolate the contributions of ϕ, while using an arbitrary strategy, is a significant asset: one can exactly measure how fruitful/detrimental a filtering algorithm is in a realistic practical context.

6 A last solution is to use as a baseline the model that makes use of the *constructive disjunction* of ϕ_1 and ϕ_2 [WM96, MW95, Jef+10], that only prunes when both algorithm prune. However, in the general case, the time overhead for performing deductions only made by ϕ_1 or ϕ_2 cannot easily be deducted from the propagation time. This penalizes the baseline from a time perspective when it is replayed.

Problem	Golomb ruler	Job Shop	Traveling Salesman
# Nodes with $M \cup \phi_M$	7386480	898	82194
# Nodes with $M \cup \phi$	2929035	399	7177
% Same Decisions	99.86	84.42	21.77

Table 3.1: Number of nodes of replays by $M \cup \phi_M$ and $M \cup \phi$, and percentages of locally matching decisions with the dynamic strategy.

3.2.3 *Implementation of the Replay Technique*

To implement the replay technique, we first generate a flat *linearized* version of the CBT by doing a *preorder taversal* using the baseline model $M \cup \phi_M$. This linearized CBT is simply a sequence of triples of the form $\langle b, c, d \rangle$ meant to represent a node of the original CBT. For a given triple $\langle b, c, d \rangle$:

- b is the branching constraint on the branch between the node it represents and its parent.
- c is the number of children of the node.
- d is the number of descendants of the node.

As we shall see, those triples are required to re-traverse the CBT with a given extended model $M \cup \phi$.

LINEARIZING THE CBT must be done so that when the sequence is traversed, the behavior is the same as a CP Depth-First Search. The sequence must therefore represent the *preorder* traversal of the CBT. As an example, the sequence for the CBT given in Figure 3.3a is:

$$\langle \top, 2, 5 \rangle, \langle b_j^1, 3, 3 \rangle, \langle b_{j+1}^1, 0, 0 \rangle, \langle b_{j+1}^1, 0, 0 \rangle, \langle b_{j+1}^3, 0, 0 \rangle, \langle b_j^2, 0, 0 \rangle$$

where $\top$ is the True clause.

Recording the sequence is done with Algorithm 3.2.1. This procedure defines in a recursive manner a classic CP Depth-First Search. More specifically, each time a branching constraint b is added to the model M, a triple $\langle b, c, d \rangle$ is added to a sequence S, where c is the number of branching constraints generated by the branching procedure β. However, unless the search proves infeasibility or finds a solution, the number of descendants d is only known after the recursive call. The triple is therefore updated at that moment. The calls to $RECORD\_STATE(M)$ and $RESTORE\_STATE(M)$ allow backtracking the state of the constraint store. We do not enter into details on how backtracking is performed, so that trail-based (inherited from [War83, AK99]) and copy-based [Sch99] solvers fit into our proposed framework.

Algorithm 3.2.1 : Linearized CBT Generation Algorithm

```
Algorithm generate($M,\beta$)
    Input  : A model $M$ and a branching procedure $\beta$
    Output : A sequence $S$ that represents a generated CBT
    $S \longleftarrow ()$ /* Empty Sequence */
    $nVisitedNodes \longleftarrow 0$
    visit($M,\beta,S,\top$) /* $\top$ is the True clause */
    return $S$

Algorithm visit($M,\beta, S, b$)
    Input  : A model $M$, a branching procedure $\beta$, a sequence $S$ and a
             branching constraint $b$
    Output : A CBT $t$
    $nVisitedNodes \longleftarrow nVisitedNodes + 1$
    if $fixPoint(M) = \bot$ then
        $S \longleftarrow S|\langle b,0,0\rangle$ /* Infeasibility */
    else if $isSolved(M)$ then
        $S \longleftarrow S|\langle b,0,0\rangle$ /* Solution */
    else
        $(b^1,\dots,b^i,\dots,b^c) \longleftarrow \beta(M)$
        $nodeIndex \longleftarrow nVisitedNodes$ /* Store the current node index. */
        $S \longleftarrow S|\langle b,c,?\rangle$ /* Number of descendants not known yet. */
        for $i \leftarrow 1$ to $c$ do
            $RECORD\_STATE(M)$
            $M \leftarrow M \cup b^i$ /* Add $b^i$ to the constraint store */
            visit($M,\beta,S,b^i$)
            $RESTORE\_STATE(M)$
        end
        $S_{nodeIndex} \longleftarrow \langle b,c,nVisitedNodes - nodeIndex\rangle$ /* Store the
            number of descendants after the recursive call. */
    end
```

REPLAY ALGORITHM: A sequence generated with a baseline model $M \cup \phi_M$ can be replayed using any extended model $M \cup \phi$ with Algorithm 3.2.2. Notice that if the baseline model $M \cup \phi_M$ is replayed, this "linearized" process will behave exactly as the traditional search. Conversely, if the baseline model is extended with ϕ, some additional pruning might occur, implying that the store is either in a failed or in a solution state at an internal node n of the CBT (i.e., we may have $\mathit{fixPoint}(M) = \perp$ or $\mathit{isSolved}(M)$, see Definition 4 in Chapter 1). This event is illustrated in Figure 3.3b, where the red node is failed due to additional pruning. When this happens, the replay process is able to directly skip all the descendants of the current node. This is illustrated by the light gray branching constraints in Figure 3.3b. The sequence replayed by $M \cup \phi$ becomes:

$$\langle \top, 2, 5\rangle, \langle b_j^1, 3, 3\rangle, \underbrace{\langle b_{j+1}^1, 0, 0\rangle, \langle b_{j+1}^1, 0, 0\rangle, \langle b_{j+1}^3, 0, 0\rangle,}_{\text{Skip}} \langle b_j^2, 0, 0\rangle$$

Algorithm 3.2.2 : Replay Algorithm for a linearized CBT.

```
1  Algorithm replay($M, S$)
       Input  : A model $M$ and a sequence of nodes obtained by linearizing the
                CBT $t$
       Output : $time(t, M)$, $backtracks(t, M)$
2      $nBacktracks \longleftarrow 0$
3      replayNode(1)
4      return $nBacktracks$

5  Algorithm replayNode($index$)
6      $RECORD\_STATE(M)$
7      $\langle b, nChildren, nDescendants\rangle \longleftarrow S_{index}$
8      $newIndex \longleftarrow index + 1$
9      $M \leftarrow M \cup b$
10     if $\mathit{fixPoint}(M) = \perp \vee \mathit{isSolved}(M)$ then
11         $nBacktracks \longleftarrow nBacktracks + 1$
12         $RESTORE\_STATE(M)$
13         return $newIndex + nDescendants$ /* Skip the subtree */
14     end
15     for $i \leftarrow 1$ to $nChildren$ do
16         $newIndex \longleftarrow$ replayNode($newIndex$)
17     end
18     $RESTORE\_STATE(M)$
19     return $newIndex$
```

SHARING SEQUENCES An interesting property of this approach is that the sequence can be serialized into files (as proposed in [DHM00, LDD03]). We can therefore replay CBTs that do not fit in RAM (if a sequence is too large to fit in one file, it can simply be divided into chunks and put in several files). Additionally, those files can be shared among the community. Provided a well-defined format exists, all solvers implementing Algorithm 3.2.2 will be able to replay CBTs obtained using another solver. This opens the doors to common evaluation of propagators in the community.

3.3 ASSESSING THE POTENTIAL OF A PROPAGATOR

We assume we are interested in reducing the time for computing the output of a filtering procedure ϕ, without changing the function definition, i.e., without changing its input-output behavior. In particular, our goal is to assess the potential of two improvement directions: 1) increasing the efficiency of the current implementation/algorithm and 2) guarding the activation of ϕ with a necessary condition. Notice that the content of this section is actually orthogonal to the replay technique presented in Section 3.2. Yet, both can be combined, as exemplified in Section 4.5.

In order to assess the potential of improving the efficiency of ϕ or controlling its activation, we instrument the solver to collect detailed information about the propagator. Specifically, we store the total time for running ϕ, making a distinction between activations that actually lead to some pruning and fruitless activations. The two time statistics are respectively referred to as t^+_ϕ and t^-_ϕ. Making those measurements only requires to write a procedure w_ϕ that wraps ϕ and is used instead. Every time w_ϕ is called during search, it registers the current domain size of the decision variables and calls ϕ. If the size of any domain has changed, the CPU time required to execute ϕ is added to the t^+_ϕ counter. If not, it is added to the other counter t^-_ϕ. The complexity of using w_ϕ is $\theta(n)$ where n is the number of decision variables. This approach is lightweight and easy to implement on most solvers.

It is now easy to get a rough, but valuable, estimate of the impact of specific measures on the solution time. First, we can estimate the impact of reducing the run time of ϕ by a factor $\mu \in [0, 1]$ by computing:

$$time(t, M \cup w_\phi) - \mu \cdot (t^+_\phi + t^-_\phi) \tag{3.4}$$

i.e., by subtracting a fraction of the total computation time of ϕ. Similarly, we can assess the impact of guarding ϕ with a necessary condition that stops a fraction $\mu \in [0, 1]$ of the fruitless propagator activations. This is done by computing:

$$time(t, M \cup w_\phi) - \mu \cdot (t^-_\phi) \tag{3.5}$$

This simple, linear, approach allows us to compare *fictional* implementations of ϕ with real ones. By doing so, we get a chance to explore which values of μ would be necessary for beating the baseline, and we get a better understanding of the effort required to achieve such a goal. In particular, we can approximately evaluate the impact of having an hypothetical time complexity for a fictional propagator. For instance, if the current implementation for ϕ is in $O(n^3)$ (where n is the number of variables), then we can estimate roughly what would be its cost for an $O(n^2)$ algorithm by choosing $\mu = (n-1)/n$ in Equation 3.4.

3.3.1 *Representative Propagator Evaluation with Performance Profiles*

While it is interesting to make a quantified evaluation of filtering procedures for a given CBT, deeper and more general insights can be obtained by making use of benchmark suites. In order to aggregate the information and derive general conclusions, we rely on *performance profiles* [DM02]. A performance profile is a cumulative distribution function $F(\tau)$ of a given performance metric τ. In our case, the τ value is the ratio between the solving metric (typically, time or number of backtracks) of a target approach and that of the baseline $M \cup \phi_M$.

Formally, let $\phi_0, \phi_1, \ldots$ be the set of all considered implementations (possibly fictional, see Section 3.3.2) of ϕ, and let $\mathscr{T}$ be the set of all CBTs generated from the benchmark instances. Then the performance profile of ϕ_i is given by:

$$F_{M\cup\phi_i}(\tau) = \frac{1}{|\mathscr{T}|}\left|\left\{t \in \mathscr{T} : \frac{metric(t, M \cup \phi_i)}{metric(t, M \cup \phi_M)} \leq \tau\right\}\right| \tag{3.6}$$

For the sake of clarity, let us provide an introductory visual example in Figure 3.4. In this plot[7], one can see that the profile $F_{M\cup\phi_M}$ for the baseline model is a step function such that $F_{M\cup\phi_M}(\tau < 1) = 0$ and $F_{M\cup\phi_M}(\tau \geq 1) = 1$ (by definition, it will always be the case). Moreover, one can read that $F_{M\cup\phi}(2) = 0.75$. This means that the performance of $M \cup \phi$ is within a factor of 2 from the baseline in 75% of the benchmark problems. Assuming the benchmark is representative enough, the value of $F(\tau)$ can be interpreted as a probability.

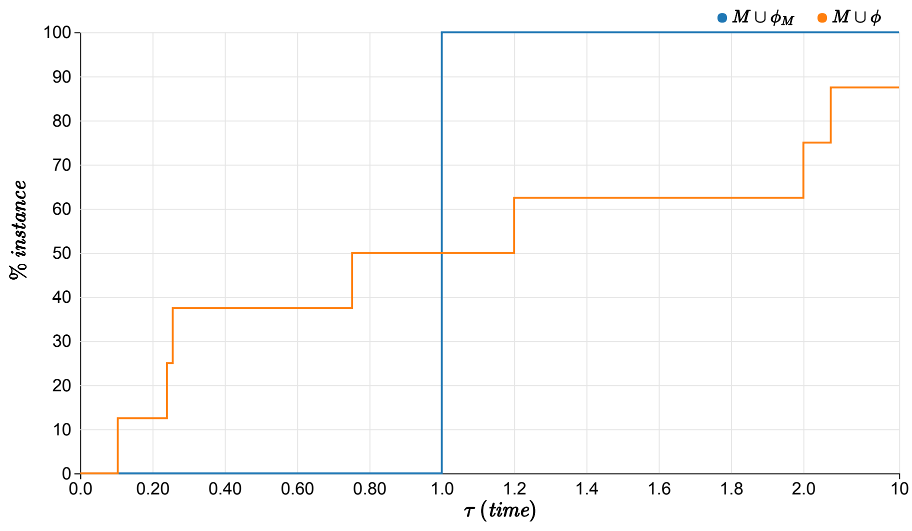


Figure 3.4: Example of a Performance Profile to compare a baseline model $M \cup \phi_M$ with an extended model $M \cup \phi$.

7 The reader might be surprised by the x-axis of the plots, as there is a change of scale on the right-most side. This helps to have a long-term view of the profiles while keeping the focus on the τ region of interest.

An important value of a performance profile $F_{M\cup\phi_i}(\tau)$ is in $\tau = 1$. For a given ϕ_i, $F_{M\cup\phi_i}(\tau = 1)$ gives the percentage of instances that can be solved using $M \cup \phi_i$ with a value for the target metric that is less than (or equal to) the one of the baseline model $M \cup \phi_M$. For instance, in Figure 3.4, 50% of the instances are solved by the extended model $M \cup \phi$ in a time smaller or equal to the one of the baseline. The space of τ is therefore divided in two important regions, $\tau < 1$ and $\tau \geq 1$. If $F_{M\cup\phi_i}(\tau) = 1$ for some $\tau < 1$, then using the model $M \cup \phi_i$ is always better than using the baseline, i.e., $M \cup \phi_i$ provides a speed-up for every instance. Unfortunately, this situation rarely happens in practice and it is thus interesting to read more carefully the performance profile. For a given pair ϕ_i, ϕ_j it is interesting to observe $F_{M\cup\phi_i}(\tau)$ - $F_{M\cup\phi_j}(\tau)$, which indicates the *gain* of ϕ_i over ϕ_j. That is, $F_{M\cup\phi_i}(\tau)$ - $F_{M\cup\phi_j}(\tau)$ reflects how many more (or less) instances can be solved by using $M \cup \phi_i$ instead of $M \cup \phi_j$ within a factor τ of the baseline metric value. Finally, the region above $F_{M\cup\phi}(\tau)$ for $\tau < 1$ is very informative, as it exhibits the gain of a given ϕ_i compared to the baseline $M \cup \phi_M$ **and** to $M \cup \phi$. Finally, instances with similar performance give rise to step-like changes in $F_{M\cup\phi}(\tau)$, while a linearly growing $F_{M\cup\phi}(\tau)$ is symptomatic of a diversified performance across the benchmark suite.

3.3.2 *Fictional Propagators*

In order to assess the potential for improvements, we considered the following classes of fictional implementations:

- ϕ_{μ}^{cost}, i.e., an implementation for which the time is reduced by a factor μ.
- $\phi_{\mathcal{O}(f(n))}^{cost}$, i.e., an implementation for which the time complexity is $\mathcal{O}(f(n))$. It is a particular case of ϕ_{μ}^{cost} for which μ is well selected based on the actual complexity of ϕ and on the value of the parameter n.
- ϕ_{p}^{oracle}, i.e., an implementation that guards ϕ with a necessary condition causing useless activations with a probability p.

We then use performance profiles as described in Subsection 3.3.1 to derive general conclusions about the fictional propagators. For fictional implementations of ϕ, $time(t, M \cup \phi_i)$ is computed using Equation (3.4) or (3.5).

Assuming the studied benchmark suite is representative enough, the joint use of performance profiles and fictional propagators allows us to provide quantitative and representative potential for improvements. The μ parameter in equations (3.4) or (3.5) plays an important role, as it allows quantifying how much reduction should be targeted to obtain the corresponding performance profile. In particular, the profile of ϕ_0^{oracle} (perfect necessary condition) bounds the gain that can be obtained by any necessary condition. The profile of $\phi_{\mathcal{O}(1)}^{cost}(\tau)$ (zero-cost implementation[8]) bounds the performance of any possible implementation. Against common intuition,

8 Another way to think of $\phi_{\mathcal{O}(1)}^{cost}(\tau)$ is to consider additional inference made by ϕ to be integrated into the baseline model.

$\phi^{cost}_{\mathcal{O}(1)}$ is not guaranteed to beat the baseline, since a weak filtering done by ϕ may trigger other (possibly expensive) propagators during fix point iteration.

3.3.3 *Characterization of Time Efficiency and Potential*

The following definitions allow quantifying the gain obtained thanks to the extension of the baseline model $M \cup \phi_M$ with ϕ.

Definition 9. *The* actual gain $\mathcal{G}^{\phi}_{M\cup\phi_M}$ *of a filtering procedure ϕ compared to a baseline model $M \cup \phi_M$ is the probability that $time(t, M \cup \phi) \leq time(t, M \cup \phi_M)$ for any CBT $t \in \mathbb{T}$. It can be estimated with $F_{M\cup\phi}(1)$.*

The actual gain quantity represents the proportion over *all* existing CBTs for which $M \cup \phi$ is faster to traverse than $M \cup \phi_M$. While it is of course impossible to compute this value, we can estimate it with $F_{M\cup\phi}(1)$. The next two definitions provide the same quantity while considering the best fictional propagators that can be obtained out of ϕ.

Definition 10. *The* upper bound gain $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ *of a filtering procedure ϕ compared to a baseline model $M \cup \phi_M$ is the probability that $time(t, M \cup \phi^{cost}_{\mathcal{O}(1)}) \leq time(t, M \cup \phi_M)$ for any $t \in \mathbb{T}$. It can be estimated with $F_{M\cup\phi^{cost}_{\mathcal{O}(1)}}(1)$.*

Definition 11. *The* activation-control upper bound gain $\mathring{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ *of a filtering procedure ϕ compared to a baseline model $M \cup \phi_M$ is the probability that $time(t, M \cup \phi^{oracle}_0) \leq time(t, M \cup \phi_M)$ for any $t \in \mathbb{T}$. It can be estimated with $F_{M\cup\phi^{oracle}_0}(1)$.*

The quantity $\mathcal{G}^{\phi}_{M\cup\phi_M}$ provides the probability that ϕ will actually be beneficial to solve an instance, if it is used to extend the model $M \cup \phi_M$. As long as it is non-zero, it means some gain could be obtained. Of course, the higher the value, the more ϕ is actually useful in practice in general. Quantities $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ and $\mathring{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ are of great interest when compared to $\mathcal{G}^{\phi}_{M\cup\phi_M}$, as they allow quantifying the gap between the current gain, and the one that could be obtained by working on more efficient algorithms/implementations or finding necessary conditions for the algorithm to prune. Clearly, if $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M} - \mathcal{G}^{\phi}_{M\cup\phi_M} \approxeq 0^+$, devising a more efficient algorithm will not be very fruitful in terms of practical efficiency.

POTENTIAL OF INFERENCE RULES: It is sometimes easier to find inference rules for a constraint than to directly propose an efficient algorithm to apply those rules. Instead of directly investing energy in order to find an efficient algorithm, one could postpone this work until the potential benefit of its discovery is known: an inefficient but easy algorithm to compute ϕ might be written in order to apply the inference rules. The value $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ can then be used to quantify how fruitful in practice it would be to actually construct an efficient algorithm performing the inference rules. Again, if $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ is very small, investing some time to find such an algorithm would not be beneficial in practice.

GLOBAL CONSTRAINT MAXIMAL PROPAGATION: In the same direction, another aspect that is useful is the gain of the maximum propagation that can be performed by a global constraint with respect to a given consistency (e.g., GAC or Generalized Bound Consistent (GBC)). In particular, studying constraints for which reaching a given consistency is NP-hard provides a lot of insight. An inefficient propagator to get the given consistency is straightforward: it is sufficient to embed a search process in the propagator. In the case of GAC, any instantiation of the *GAC-Schema* [BR97] can be used. From that, we can compute $\bar{\mathcal{G}}^{\phi^{max}}_{M \cup \phi_M}$, where ϕ^{max} is the inefficient procedure allowing to reach the level of consistency. $\bar{\mathcal{G}}^{\phi^{max}}_{M \cup \phi_M}$ gives the maximum gain that could ever be reached using the given level of consistency. Again, this can be compared with existing approaches in order to quantify how fruitful it would be to be able to prune more. The gain might again be negligible, meaning that research time should be better spent looking for new search strategies or models, rather than improving the consistency level.

3.4 IMPLEMENTATION IN THE OSCAR SOLVER

This section explains how the proposed framework is used in the OscaR solver [Osc12]. It also gives some implementation details and design choices. The design of the replay framework was guided by the motivation of making it orthogonal to the existing OscaR search and without requiring any modification of existing default search heuristics (such as [Gay+15]). The existing search of OscaR was kept unmodified and agnostic to the replay framework. A search observer linearizes the search by capturing branching decisions into closures.

As an illustrative example, we use the well-known *n-Queens* problem. The OscaR model is provided in Figure 3.5. It has been extended to integrate the replay technique. The additional instructions specific to the replay framework are highlighted in bold.

INITIAL MODEL: The model without replay is quite straightforward. The position variables of the queens are defined in lines 3-8. The constraints imposing that the queens cannot attack each other are declared in lines 10-12. They are, however, only added to the constraint store in the `startSubjectTo` bloc (line 20) where the search is effectively started (under some additional constraints that will eventually be removed from the model on search completion). In our case, this allows us to impose the ALLDIFFERENT constraints before starting the search. We finally define the search heuristic in line 15 (the heuristic is here β_{ff}, from Example 1 in Chapter 1).

CBT GENERATION: The generation of the CBT is simply done by passing an additional search-listener parameter to the standard search. This listener stores the required sequence of triples used to replay (see Algorithm 3.2.1). The replaying searches will then re-use this exact same sequence. Notice that Depth First Limited

```
 1 object Queens extends CPModel with App {
 2
 3   val nQueens = 10
 4   // Number of queens
 5   val Queens = 0 until nQueens
 6
 7   // Variables
 8   val queens = Array.fill(nQueens)(CPIntVar.sparse(0, nQueens − 1))
 9
10   val allDiffs = Seq(allDifferent(queens),
11     allDifferent(Queens.map(i => queens(i) + i)),
12     allDifferent(Queens.map(i => queens(i) − i)))
13
14   // Search heuristic
15   search(binaryFirstFail(queens))
16
17   val linearizer = new DFSLinearizer()
18
19   // Execution with FC allDifferent
20   val statsInit = startSubjectTo(){
21     add(allDiffs,Weak)
22   }(cp, linearizer)
23
24   // Replay with AC allDifferent
25   val statsReplayAC = cp.replaySubjectTo(linearizer, queens) {
26     add(allDiffs,Strong)
27   }
28 }
```

Figure 3.5: Model for the n-Queens problem. The additional required instructions to replay and track a constraint are in bold. The rest of the model remains unchanged.

Discrepancy Search [HG95] and Large Neighborhood Search [PR10] could also be used to perform the generation, as they are based on a regular OscaR search.

The listener interface defined in OscaR is given in Figure 3.6. This interface allows defining the expected behavior when a node is expanded in Algorithm 3.2.1, i.e., after the branching procedure has been called. The only method (*onExpand*) takes an *Alternative* as an argument, which is basically a closure. This closure is to be applied when the branching is performed. The alternative therefore encapsulates any branching constraint addition to the model (i.e., $M \cup b_i$). The implementation of the linearizer (given in Figure 3.7) is then direct: an internal buffer is filled with *preorder elements*, i.e., pairs with a branching constraint and a number of children, each time a node is expanded. The number of descendants of each node is then computed from this sequence after the generation is completed (without going into details, the sequence of triples is computed in two passes because the search in OscaR is slightly different from Algorithm 3.2.1).

From a modeling point of view, to be able to generate the sequence during a search, the only requirement is to set up a *linearizer listener* (line 17 in Figure 3.5) and pass it as a parameter, as in line 22. After this search is finished, the sequence is stored and we can replay it as many times as we want, potentially with constraints having a stronger pruning added to the model (as in line 26).

```
trait DFSearchListener {
  /** Called on expand events */
  def onExpand(currentBranchAlternative: Alternative, nChildren: Int): Unit
}
```

Figure 3.6: Depth-First Search Listener interface

```
class DFSLinearizer extends DFSearchListener {
val nodes : ArrayBuffer[PreorderElement] = ArrayBuffer[PreorderElement]()
val nDescendantsOf : ArrayBuffer[Int] = ArrayBuffer[Int]()
def onExpand(currentBranchAlternative: Alternative, nChildren: Int) = nodes += new
      PreorderElement(currentBranchAlternative, nChildren)
}
class PreorderElement(val alternative: Alternative, val nChildren: Int) extends
      Tuple2[Alternative, Int](alternative, nChildren){}
```

Figure 3.7: Depth-First Search Linearizer. This class implements the interface of Figure 3.6 to linearize the Depth-First Search of OscaR .

CBT REPLAY: In order to replay the model, we call the *replaySubjectTo* procedure (line 25 in Figure 3.5) that implements Algorithm 3.2.2. This procedure must know what variables must be assigned in order to detect solutions. OscaR indeed has no explicit store status when the problem is solved. During a replay, we consider that a solution is found once all constraints are satisfied (i.e., no failure during the fix point and no domain wipe-out) and all the variables passed to the replay primitive are assigned.

Once completed, the replay primitive returns the solution time, the number of found solutions, the number of backtracks and the number of nodes. Those statistics can be used to compare the performance of the baseline and that of the extended models.

TRACKING A CONSTRAINT: It is optionally possible to track the activation of a propagator/constraint. This is useful to perform a study as described in Section 3.3. Basically, the modeler must just use the *track* function that returns the constraint passed as a parameter, augmented with code that implements the tracking behavior. This allows one to measure the time for each propagation call of the constraint. In order to separate pruning propagation time (t_ϕ^+) from non-pruning propagation time (t_ϕ^-), we must specify the variables for which we want to make this distinction. This is the second argument of the track function. The tracking behavior then verifies that some pruning happened for those variables by checking the size of their domains before and after propagation. The complexity for tracking is therefore $\mathcal{O}(n)$.

3.5 EXPERIMENTATION

We applied our approach to several constraints and ran tests on AMD Opteron processors (2.7 GHz) using the Java Runtime Environment 8 and the constraint solver OscaR [Osc12]. For each solved instance, we limited the run-time of $generate(M, \beta)$ to 600 seconds. Instances for which $generate(M, \beta)$ took less than 1 second were filtered out. The additional filtering put on top of the baseline model was executed with a lower priority by the constraint scheduler.

3.5.1 *AllDifferent*

We analyzed the well-known ALLDIFFERENT constraint, since it is ubiquitous in Contraint Satisfaction Problems. The ALLDIFFERENT forward checking algorithm [DSVH90, VHD87] (written $allDiff_{FWC}$) is used in the baseline model, and we considered the following additional filtering methods:

- the GBC ALLDIFFERENT, written $allDiff_{BC}$ [LO+03].
- the counting-based ALLDIFFERENT, written $allDiff_{CB}$, and described in [MP15].
- the GAC ALLDIFFERENT [Rég94], written $allDiff_{AC}$.

We used the 291 instances from the XCSP 2.1 benchmarks that contain ALLDIFFERENT constraints, namely *bqwh-18-141_glb*, *medium*, *bqwh-15-106_glb*, *QG3*, *ortholatin*, *small*, *latinSquare*, *pigeons_glb*, *compet02* and *compet08*.

To assess the benefits of $allDiff_{BC}$, $allDiff_{CB}$ and $allDiff_{AC}$, we replayed with all the combinations of additional filtering procedures such that the replayed CBT is included into the generated one. We also considered models without $allDiff_{FWC}$ when possible ($allDiff_{CB}$ and $allDiff_{AC}$ subsume $allDiff_{FWC}$). Finally, the priority of $allDiff_{AC}$ in the propagation queue was the lowest, $allDiff_{BC}$ and $allDiff_{CB}$ had the same priority, and $allDiff_{FWC}$ had the highest priority. The branching procedure used to generate the CBTs is β_{ff}, as defined in Example 1.

Figure 3.8 provides the time performance profiles. Notice that we do not report the thirteen propagator combinations but only the profiles of the most different approaches in order to make the plots easier to read. From a time perspective, the approaches that are not shown have a profile with a shape that is generally in-between the curve of $fwc \cup ac$ and that of $cb \cup bc \cup ac$.
Our first observation is that even if $fwc \cup bc$ has an actual gain $\mathcal{G}^{\phi}_{M \cup \phi_M} \simeq 0.3$ (see Section 3.3.3) compared to the baseline (see the orange line in $\tau = 1$), it is clearly outperformed by the other approaches. $cb \cup bc$ (red curve) and $fwc \cup cb \cup bc$ (purple curve) require a bit more time to approximately catch up with the other models. More importantly, we can see that while $fwc \cup cb$ has not the highest gain for τ values close to 0, it actually has the highest actual gain $\mathcal{G}^{\phi}_{M \cup \phi_M} \simeq 0.93$ (see the green line in $\tau = 1$) and stays better for larger τ values. This is of great interest as the counting-based ALLDIFFERENT algorithm is actually very simple.

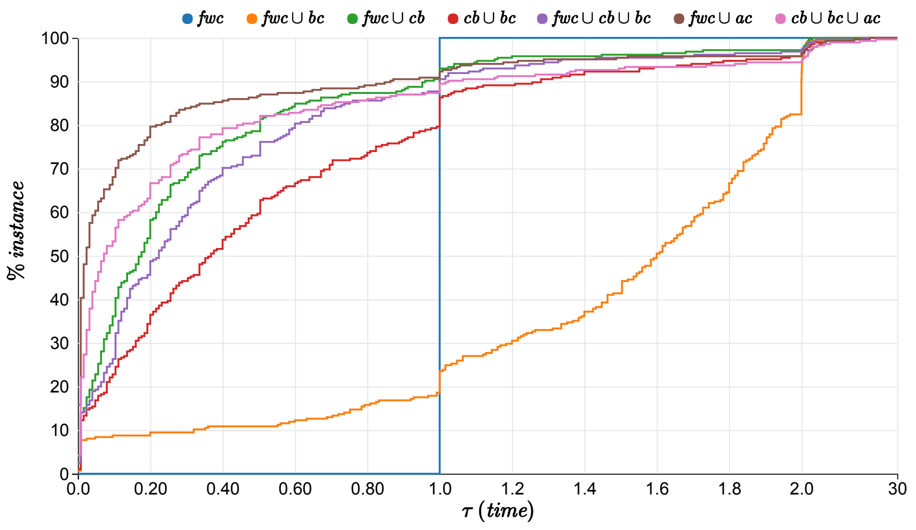


Figure 3.8: Time performance profiles for combinations of AllDifferent propagators for XCSP instances.

If we now look at the backtrack profiles of Figure 3.9, we better understand why the *fwc* ∪ *bc* model is outperformed by the others, its gain in backtracks being way smaller than the other ones, especially for small τ values. As expected from the time profiles, the gains of the other models all have the same shape. Still, we can notice that *fwc* ∪ *bc* is one of the "worst" approaches in terms of backtrack gain, while it has the highest time gain, as just mentioned.

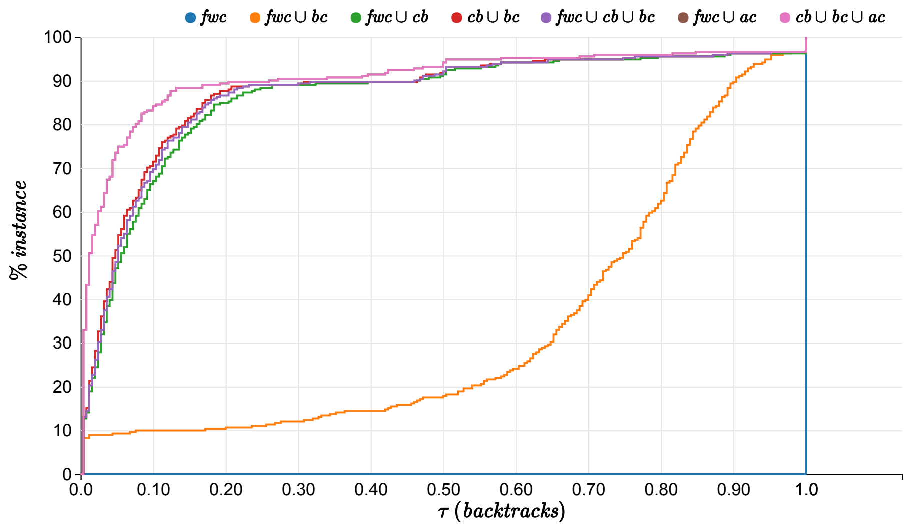


Figure 3.9: Backtrack performance profiles for combinations of AllDifferent propagators for XCSP instances.

As a brief conclusion, we learned that Bound Consistency is not a sufficiently strong level of consistency for the ALLDIFFERENT constraint, from the point of view of both time and number of backtracks. On the contrary, the counting-based ALLDIFFERENT infers almost as much as the Arc Consistency algorithm, allowing it to get similar time performances. Still, these conclusions must be taken with some care, as the problems we consider are quite structured.

3.5.2 *Energetic Reasoning for the Cumulative Constraint*

We analyzed the Energetic Reasoning (ER) propagator for the CUMULATIVE constraint [AB93, BLP00] on RCPSPs. The baseline model M employs the Time-Tabling algorithm from [LBC12] and the ER Checker [BLPN01], which both run in $\mathcal{O}(n^2)$ [BLPN01, DP14]. We did not use the improvements proposed in [DP14]. We use a dynamic search strategy, i.e., the classic *SetTimes* approach from [LP+94]. We consider two benchmarks: the BL instances [BLP00] (20-25 activities) and the PSPLIB (j30 and j90, with 30 and 90 activities) [KSS99]. We focus on investigating, for the chosen benchmarks: 1) the potential benefit of having an ER algorithm running in $O(n^2)$ rather than in $O(n^3)$; 2) the potential benefit of a perfect necessary condition (see [VCLP14] and [BHS11] for related works).

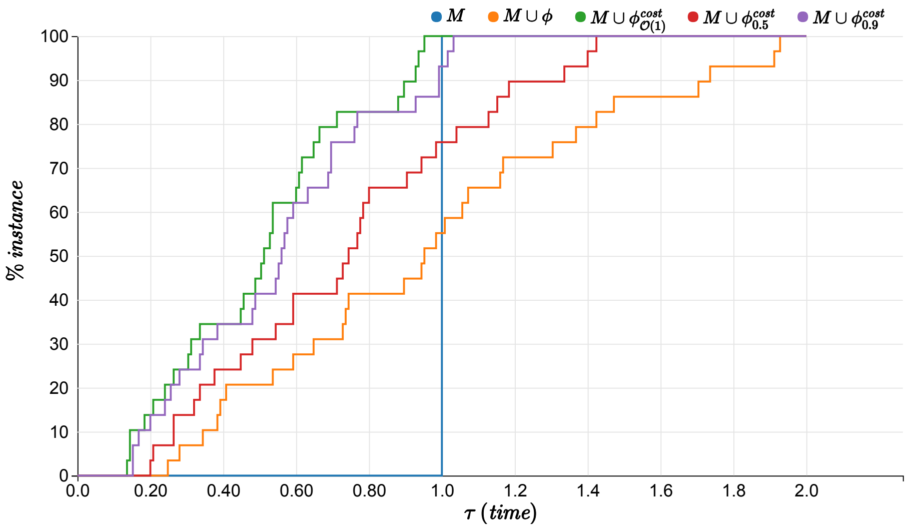


Figure 3.10: Performance profiles for real and fictional (ϕ^{cost}_{μ}) ER propagators on the BL instances.

Figures 3.10/3.11 and Figures 3.12/3.13 report profiles respectively for the BL and j90 instances. The real ER propagator has an actual gain $\mathcal{G}^{\phi}_{M \cup \phi_M} \simeq 0.5$ when BL instances are considered, but of only ~ 0.05 for the j90 instances (see the orange curves in $\tau = 1$ in Figures 3.10/3.11 and Figures 3.12/3.13). The larger problem size is a likely reason for the performance drop, so it is interesting to analyze the fictional, reduced-cost implementations (Figures 3.10 and 3.12). In the

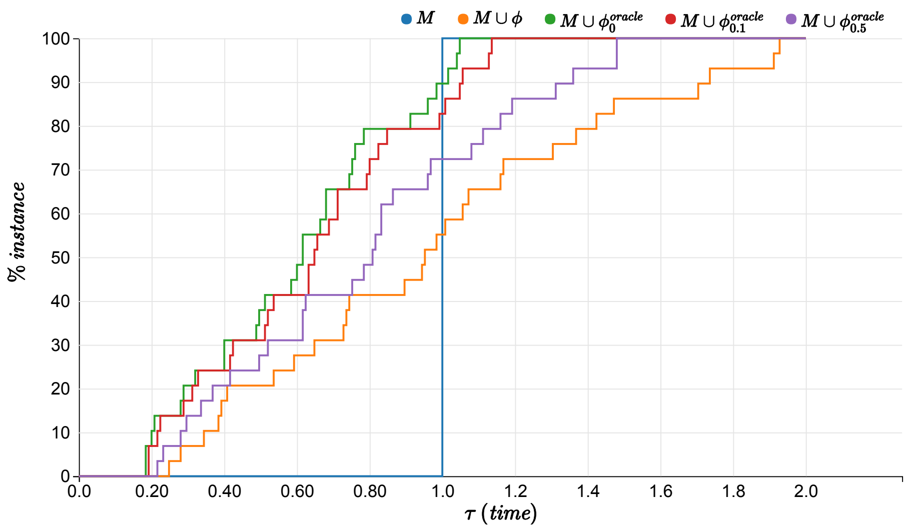


Figure 3.11: Performance profiles for real and fictional (ϕ_p^{oracle}) ER propagators on the BL instances.

BL benchmark a cost reduction translates to roughly proportional benefits. On j90, an $O(n^2)$ ER would lead to dramatic performance improvement, but it would beat the baseline in only 40% of the cases (see the pink curve in Figure 3.12 in $\tau = 1$).

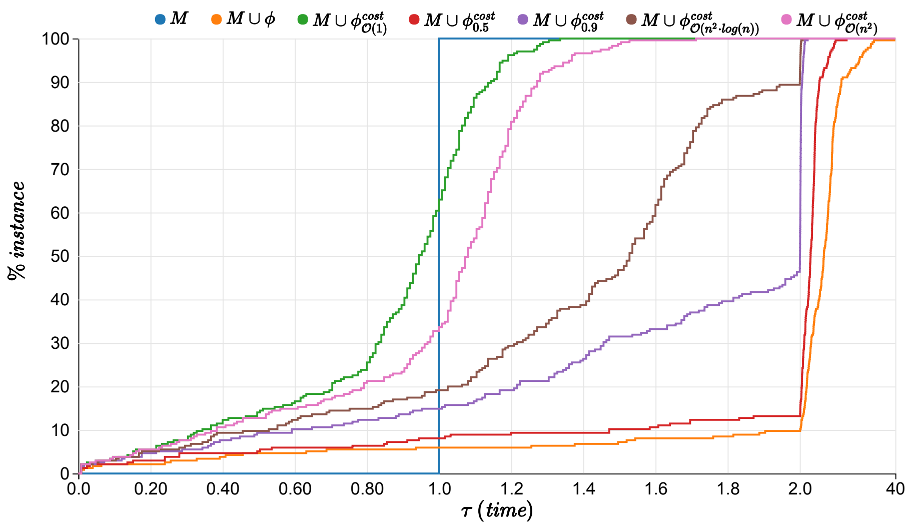


Figure 3.12: Performance profiles for real and fictional (ϕ_μ^{cost} and $\phi_{\mathcal{O}(f(n))}^{cost}$) ER propagators on the j90 instances.

More interestingly, for the upper bound gain we have $\bar{\mathcal{G}}_{M \cup \phi_M}^{\phi} \simeq 0.65$ (see the green curve Figure 3.12, in $\tau = 1$), meaning there is about a 35% portion of instances where the baseline would win *no matter what the efficiency of ER is*, i.e., where the additional pruning of ER is sometimes detrimental rather than beneficial: despite an $\mathcal{O}(1)$ hypothetical ER, the additional ER filtering causes a larger number of iterations to reach the fix point. On such instances, ER cannot lead to benefits unless we find a way to activate it only when it provides an actual advantage. As for using a necessary condition, a perfect approach would enable the same gain as that of a $O(n^2)$ ER (see the green curve in Figure 3.13, in $\tau = 1$), but even a small mistake probability would cancel most of the benefits (in the same plot, compare the green curve with the red and purple curves).

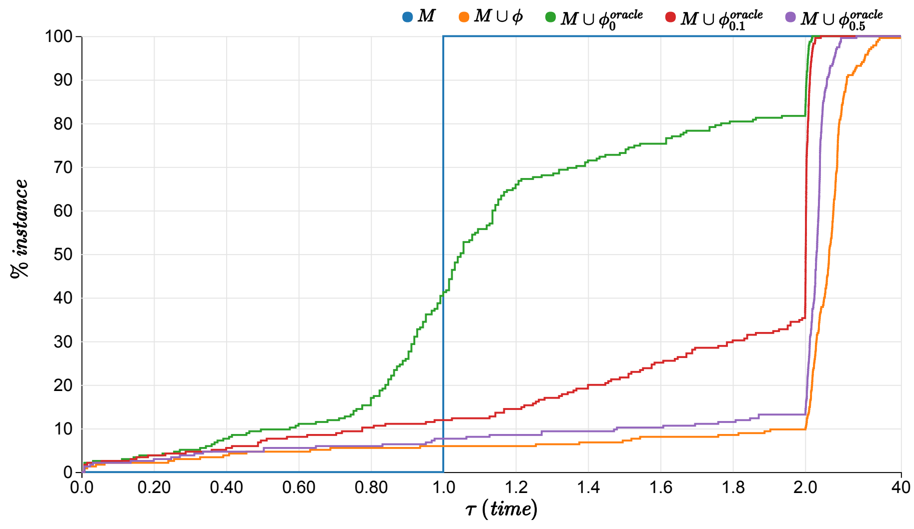


Figure 3.13: Performance profiles for real and fictional (ϕ_p^{oracle}) ER propagators on the j90 instances.

Figures 3.14 and 3.15 compare profiles for different search strategies on the j30 instances (*SetTimes* and β_{ff}, as defined in Example 1): the potential gain of reducing the cost (e.g., the green curves) is very different for the two strategies, even if the performance of the real propagator (orange curves) is roughly identical.

From this experiment, one can realize that although ER is one of the strongest filtering algorithms for the CUMULATIVE constraint, it does not provide much improvement for PSPLIB instances, even if we were able to perform its computation more efficiently. This illustrates that ER has two drawbacks when used in addition to Time-Tabling and the ER checker on those instances: 1) a heavier computation time, 2) a rather weak additional filtering in practice. Our simple method allows discovering that information before investing time in the research of a more efficient algorithm.

In addition, one can see that the possible performance improvements between the extended and the baseline models differ substantially depending on the kind of search strategies (static or dynamic) that we use. This points out the importance

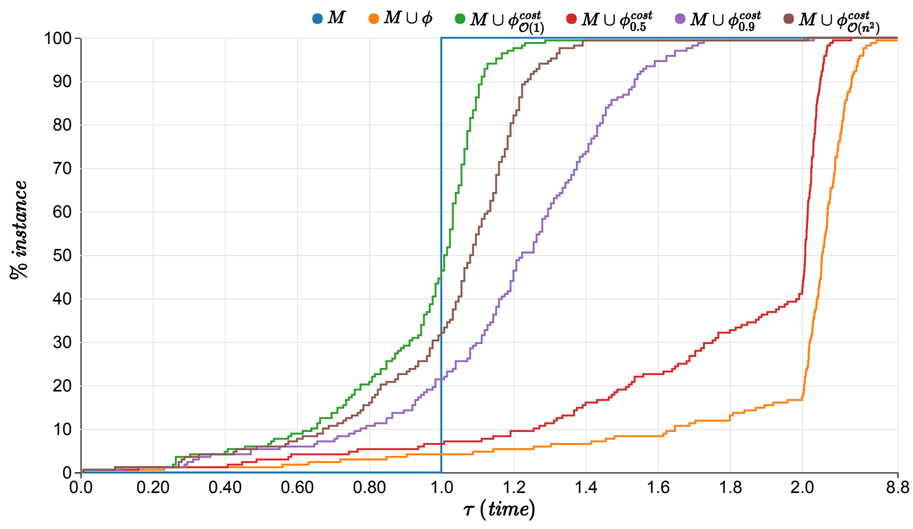


Figure 3.14: Performance profiles for the SetTimes dynamic strategy.

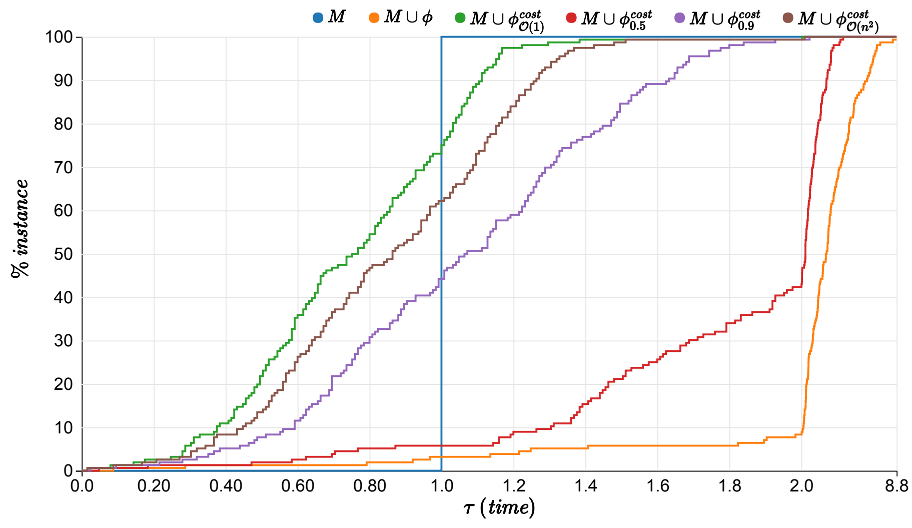


Figure 3.15: Performance profiles for the binary static strategies.

of having an approach for the rigorous comparison of propagators using practical search strategies.

3.5.3 *Revisited Cardinality Reasoning for BinPacking*

In our analysis of the RCRB propagator, we use as a benchmark the instances of the Balanced Academic Curriculum Problem (BACP) from [Sch09, PSR13]. The baseline model M employs the BinPacking propagator from [Sha04] and a GCC constraint (model **A** in [PSR13]). The branching procedure is again β_{ff}, as defined in Example 1.

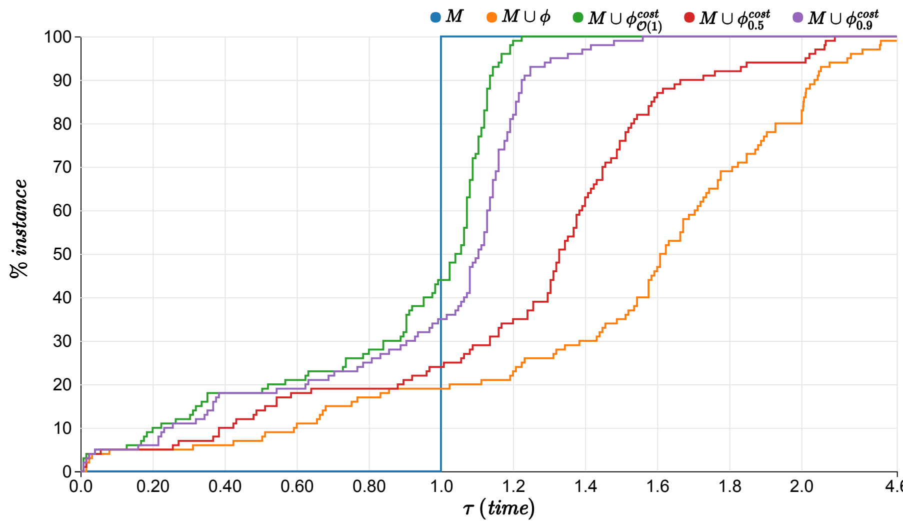


Figure 3.16: Performance profiles with fictionally cost-reduced RCRB propagators.

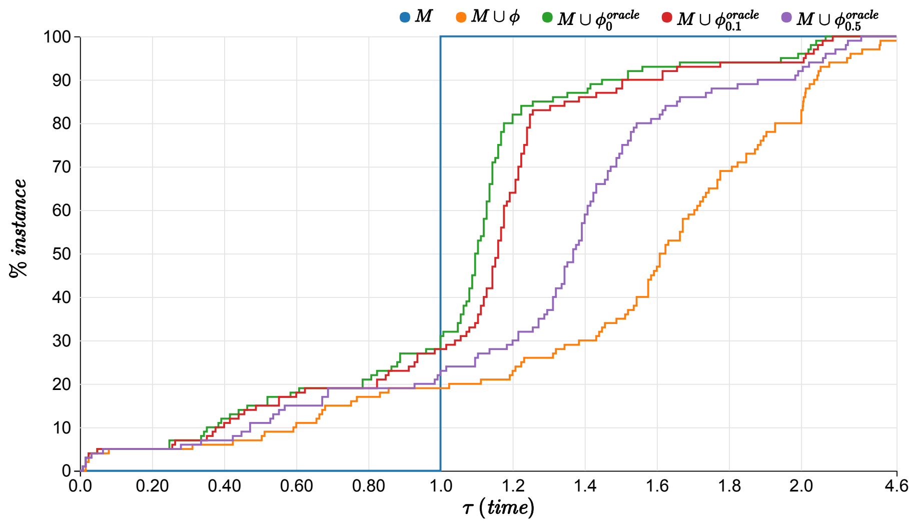


Figure 3.17: Performance profiles with fictionally reduced RCRB propagators (necessary condition).

Figure 3.16 is very informative about the cost of RCRB. We can see for the actual gain that $\mathcal{G}^{\phi}_{M \cup \phi_M} \simeq 0.2$ (see the orange curve in $\tau = 1$ in Figure 3.16), i.e., $\sim 20\%$ of the instances are solved faster than the baseline model. However, reducing the propagator cost down to 0 provides only a small gain before $\tau = 1.1$

(see the green curve in Figure 3.16): similarly to the ER case on j90, even a zero-cost version of the propagator would not be able to beat the baseline in $\sim 55\%$ of the instances (see the green curve in $\tau = 1$). For $\tau > 1.1$, reducing the propagator cost has a stronger effect, but a factor 0.9 reduction is required to solve a lot more of the instances (see the purple curve in Figure 3.16). Hence, reducing the cost would improve the RCRB, but not that much compared to the baseline model as the benefits come "too late" in terms of τ. A similar analysis can be done for Figure 3.17 for the potential gain of introducing a necessary condition.

3.5.4 *Unary Constraint with Transition Times*

We here report results obtained with a recent approach [DVCS15]. This work extends the classic unary/disjunctive resource propagation algorithms to include propagation over *sequence-dependent* transition times between activities. In brief,[9] the unary resource with transition times imposes the following relation:

$$\forall i, j : (\mathsf{end}_i + tt_{i,j} \leq \mathsf{start}_j) \vee (\mathsf{end}_j + tt_{j,i} \leq \mathsf{start}_i) \tag{3.7}$$

where start_j, end_i and $tt_{j,i}$ are the start and the end of an activity i, and the minimum transition time between activities i and j, respectively.

For each considered instance, the three following filterings for the unary constraint with transition times were used :

1. Binary constraints[10] (ϕ_b) given in Equation 3.7. The baseline model M employs this constraint.
2. Binary constraints given in Equation 3.7 with the Unary global constraint of [Vil07] (ϕ_{b+u}).
3. The constraint of [DVCS15] (ϕ_{uTT}).

The search strategy used to generate the CBTs was *Conflict Ordering Search* [Gay+15]. Figures 3.18 and 3.19 respectively provide the profiles for time and number of backtracks for all the 960 instances. Figure 3.20 provides a "long-term" view of Figure 3.18.

From Figure 3.18, we can first conclude that ϕ_{b+u} (orange curve) is clearly worse than ϕ_{uTT} (green curve) and ϕ_b (blue curve) from a time perspective. Moreover, Figure 3.19 shows that ϕ_{b+u} rarely offers more pruning than ϕ_b.

In comparison, we can see from Figure 3.18 that for $\sim 25\%$ of the instances, ϕ_{uTT} is about 5 times faster than ϕ_b (see $F_{\phi_{uTT}}(0.2)$), and that $\sim 65\%$ of the instances are solved faster (see $F_{\phi_{uTT}}(1)$). Moreover, it offers more pruning for $\sim 100\%$ of the instances, meaning that the actual gain in terms of the number of backtracks $\mathcal{G}_M^{\phi_{uTT}} \simeq 1$ (see $F_{\phi_{uTT}}(1)$, in Figure 3.19).

From Figure 3.20, we can see that the constraint does not have too much overhead, as ϕ_{uTT} is at worst 2 times slower than ϕ_b for $\sim 20\%$ percent of the

9 A more detailed and general description will be given in Chapter 4.

10 For efficiency reasons, dedicated propagators have been implemented instead of posting reified constraints.

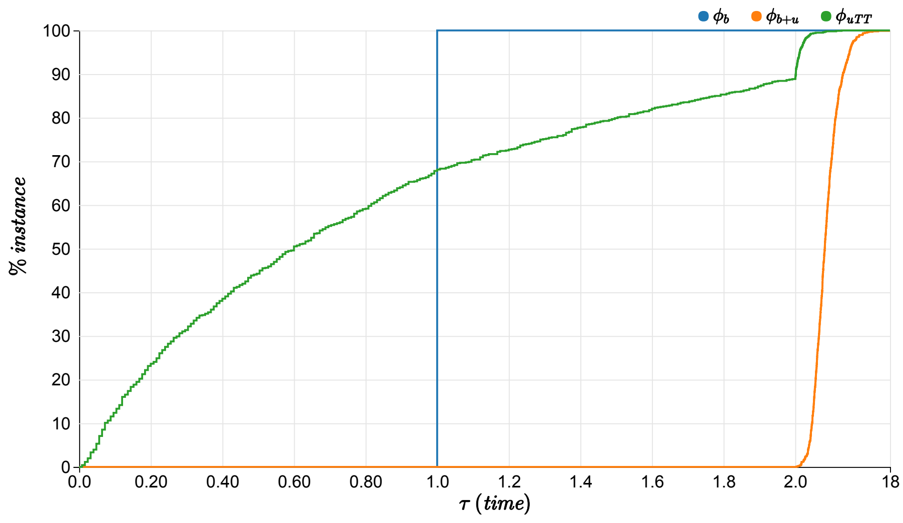


Figure 3.18: Short-term time performance profiles for the Unary Resource with Transition Times

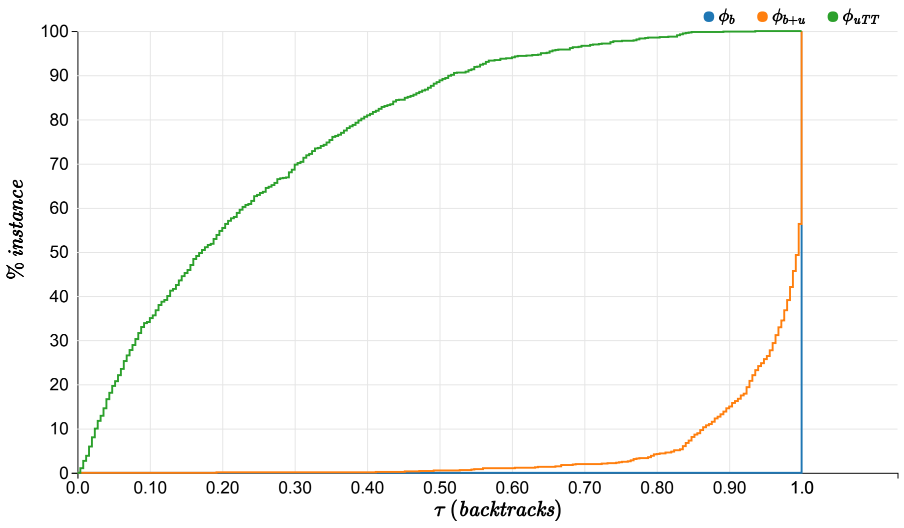


Figure 3.19: Backtrack performance profiles for the Unary Resource with Transition Times

instances $(F_{\phi_{uTT}}(2) - F_{\phi_{uTT}}(1))$. It is a bit slower for the remaining $\sim 10\%$, but almost all instances are solved in a time at most 5 times slower than the baseline (since $F_{\phi_{uTT}}(5) = 1$).

The conclusion is clear: when transition times are involved, the unary resource algorithms that do not consider them provide almost no additional filtering and therefore only incur overhead. On the contrary, the unary resource with transition times [DVCS15] prunes much more and is therefore often beneficial.

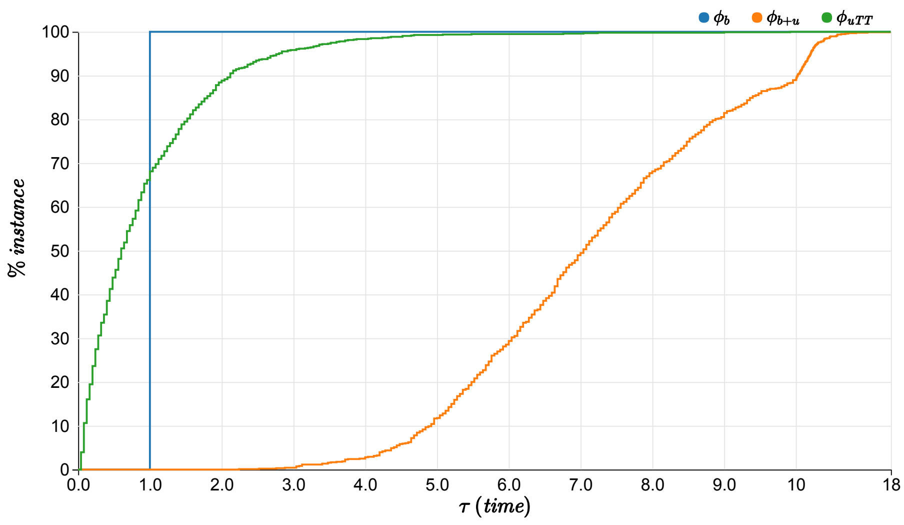


Figure 3.20: Long-term time performance profiles for the Unary Resource with Transition Times

3.5.5 *Bound Consistent Cumulative*

As proposed in Section 3.3.3, we studied the gain that would be provided by a Bound Consistent CUMULATIVE constraint, in order to estimate how far the current propagators are from the maximal pruning. The baseline model M uses poly-time propagators that suffice to achieve the strongest propagation we can get so far in the OscaR solver, namely *Energetic Reasoning* [BLPN01], *Not-First Not-Last* [BLPN01], and *Time-Table Disjunctive Reasoning* [GHS15b]. The Bound Consistent Cumulative Propagator was constructed as an exponential algorithm into which we basically embedded a search. A checker and a propagator were constructed and they were used to replay the generated CBTs. We only used the BL instances [BLP00] as they remain quite small in terms of the number of activities (20-25). We measured the backtracks and none of the instances were filtered out in this case. Figure 3.21 gives the performance profiles and Figure 3.22 gives a zoomed version near $\tau = 0$. One can notice that there is still a lot to gain (it might not be possible as Bound Consistency for the Cumulative constraint is NP-hard), and this kind of measurement allows quantifying an upper bound on this gain. It is especially impressive to look near 0. For instance, for almost 30% of the instances, there is a potential gain factor of 100 (see the curves in $\tau = 0.01$ in Figure 3.22) in terms of the number of backtracks, even if we are already using the strongest poly-time pruning we know so far. This illustrates that, while working on efficient practical algorithms (e.g., [GHS15a]) is important, finding complementary poly-time and efficient algorithms to the ones used so far would clearly provide improvement. Another interesting point to notice is that the Bound-Consistent propagator almost provides no improvement compared to the checker. Devising new efficient checkers might actually suffice.

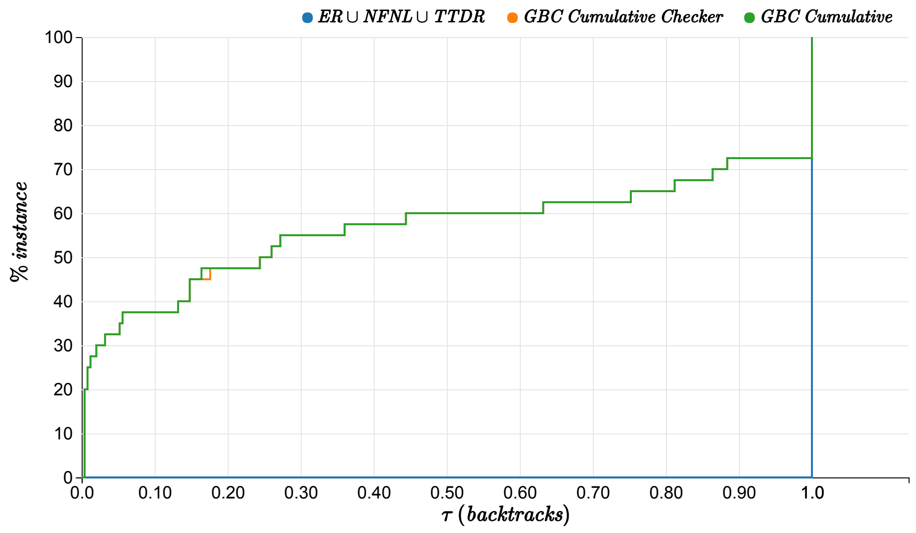


Figure 3.21: Backtrack performance profiles for the Bound Consistent Cumulative Constraint

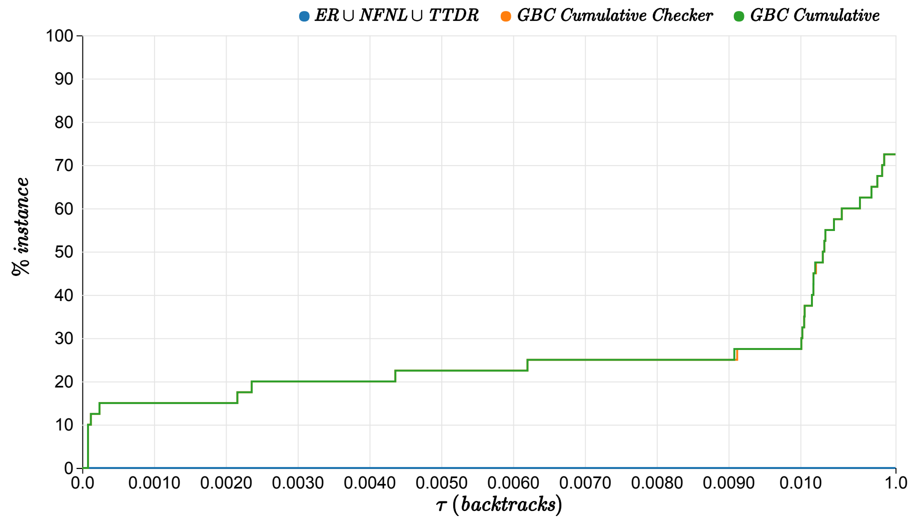


Figure 3.22: Backtrack performance profiles for the Bound Consistent Cumulative Constraint (zoomed in)

Part III

NEW SCALABLE PROPAGATORS

4

GENERALIZED UNARY RESOURCE WITH TRANSITION TIMES

Unary resources with sequence-dependent transition times (also called set-up times) for non-preemptive activities are very frequent in real-life scheduling problems. A first example is the quay crane scheduling in container terminals [Zam+13], where the crane is modeled as a unary resource and transition times represent the moves of the crane on the rail between positions where it needs to load or unload containers. A second example is the continuous casting scheduling problem [GSDS14], where a set-up time is required between production programs.

Although efficient propagators have been designed for the standard unary resource constraint (UR) [Vil07], transition time constraints between activities generally make the problem harder to solve because the existing propagators do not take them into account. A propagator for the unary resource constraint with transition times (URTT) was recently introduced [DVCS15] as an extension to Vilím's algorithms, in order to strengthen the filtering in the presence of transition times.

Unfortunately, the additional filtering quickly drops in the case of a sparse transition time matrix, which typically occurs when activities are grouped into families with zero transition times within a family. The reason for a weak filtering with sparse matrices is that it is based on a shortest path problem with free starting and ending nodes and a fixed number of edges. The length of this shortest path drops in the case of zero transition times. In addition, while Vilím algorithms allow to reason with *optional* activities, the approach from [DVCS15] does not support them.

The main contribution of the present chapter is to introduce a generalized unary resource with transition times that unifies filtering rules and algorithms such that they consider family-based transition times and optional activities. The main asset of our approach is its scalability: we obtain a strong filtering while keeping a low time complexity of $\mathcal{O}(n.\log(n).\log(f))$, for n activities and f families. In general $f \ll n$, hence the theoretical complexity is very close to the one of the propagators in [Vil07] and [DVCS15]. The filtering is experimentally tested on instances of the Job-Shop Problem with Sequence Dependent Transition Times (JSPSDTT), although it can be used for any type of problems, e.g., with other kinds of objective function than the makespan minimization. We first consider the case where it is known prior to search on which machine the activities must be executed. We then experiment with the more general case where activities must be executed by one of several alternative machines. The results show that our propagator improves the resolution time over existing approaches and is more scalable.

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

Over the years, many CP approaches have been developed to solve such problems [FLN00, ABF04, Wol09, GH10, DVCS15]. For instance, in [ABF04], a Traveling Salesman Problem with Time Window (TSPTW) relaxation is associated with each resource. The activities used by a resource are represented as vertices in a graph, and edges between vertices are weighted with the corresponding transition times. The TSPTW obtained by adding time windows to vertices from bounds of corresponding activities is then resolved. If one of the TSPTW is found unsatisfiable, then the corresponding node of the search tree is pruned. A similar technique is used in [AF08] with additional propagators, which are, to the best of our knowledge, the state of the art propagators when families of activities are present.

CHAPTER OUTLINE The chapter starts by providing the background for the considered problems in Section 4.1. The content is then described in a top-down fashion: Section 4.2 describes the filtering rules for the unary resource with transition times and the different algorithms to apply those rules. Then, we explain in Section 4.3 the data structures required by the filtering algorithms. Section 4.4 gives lower bounds for the minimum total transition time that must hold in a given set of cardinalities. This is required to strengthen the filtering. Finally, Section 4.5 compares the results of the different existing approaches for the unary resource with transition times.

4.1 BACKGROUND

Non-preemptive scheduling problems are usually modeled in CP by associating three variables to each activity i: s_i, c_i, and p_i representing respectively the starting time, completion time, and processing time of i. These variables are linked together by the following relation: $s_i + p_i = c_i$. Depending on the problem, the scheduling of the activities can be restricted by the availability of different kinds of resources required by the activities. In this chapter, we are interested in the unary resource (sometimes referred to as machine or disjunctive resource) and the propagators associated with one unary resource. Let T be the set of activities requiring the unary resource. The unary resource constraint prevents any two activities in T to overlap in time:

$$\forall i, j \in T : i \neq j \implies (c_i \leq s_j) \vee (c_j \leq s_i)$$

TRANSITION TIMES The unary resource can be generalized by requiring transition times between activities. They are described by a square *transition matrix tt* in which $tt_{i,j}$, the entry at line i and column j, represents the minimum amount of time that must elapse between the activities i and j when i directly precedes j. We assume that transition times respect the triangular inequality. That is, inserting any activity between two activities never decreases the transition time between these two activities: $\forall i, j, k \in T : tt_{i,j} \leq tt_{i,k} + tt_{k,j}$.

The unary resource with transition times constraint imposes the following relation:

$$\forall i, j \in T : i \neq j \implies (c_i + tt_{i,j} \leq s_j) \vee (c_j + tt_{j,i} \leq s_i) \tag{URTT}$$

An example of a transition matrix is given in Figure 4.1, where we can notice that it is not symmetric (e.g., $tt_{1,2} = a \neq c = tt_{2,1}$ in Figure 4.1). As exemplified, it induces a *transition graph*, that will be used in the forthcoming sections.

$$tt = \begin{pmatrix} 0 & a & b \\ c & 0 & d \\ e & f & 0 \end{pmatrix}$$

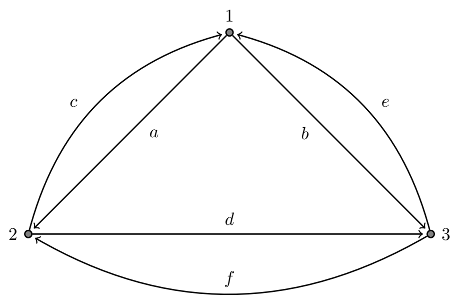


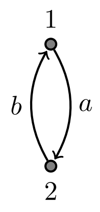
Figure 4.1: Example of a transition matrix tt and its induced Transition Graph.

FAMILY-BASED TRANSITION TIMES When transition times are present, it is often the case that activities are grouped into families on which the transition times are expressed. Formally, we denote by F_i the family of activity i and by $\mathcal{F}$ the set of all families. Moreover, for a given set of activities Ω, we write $F_\Omega = \{F_i \mid i \in \Omega\}$. In a family-based setting, the transition times are described as a square *family transition matrix* $tt^{\mathcal{F}}$ of size $|\mathcal{F}|$. The transition time between two activities i and j is the transition time between their respective families F_i and F_j, and it is zero if $F_i = F_j$:

$$\forall i, j \in T : tt_{i,j} = tt^{\mathcal{F}}_{F_i,F_j} \wedge \left(F_i = F_j \implies tt^{\mathcal{F}}_{F_i,F_j} = 0\right) \tag{4.1}$$

Given a set of activities, their families and a transition matrix between families, $tt^{\mathcal{F}}$ one can expand $tt^{\mathcal{F}}$ into a transition matrix between activities tt. tt is then larger and sparser than $tt^{\mathcal{F}}$. An example of this expansion is given in Figure 4.2, where the *family transition graph* induced by $tt^{\mathcal{F}}$ is also illustrated. Notice that $tt = tt^{\mathcal{F}}$ is the special case occurring when each activity is in its own family.

$$tt^{\mathcal{F}} = \begin{pmatrix} 0 & a \\ b & 0 \end{pmatrix}$$



$$tt = \begin{pmatrix} 0 & 0 & a & a & a \\ 0 & 0 & a & a & a \\ b & b & 0 & 0 & 0 \\ b & b & 0 & 0 & 0 \\ b & b & 0 & 0 & 0 \end{pmatrix}$$

Figure 4.2: Example of a family transition matrix $tt^{\mathcal{F}}$, its induced Family Transition Graph, and the expanded transition matrix tt if $F_1 = F_2 = 1$ and $F_3 = F_4 = F_5 = 2$.

OPTIONAL ACTIVITIES Some activities can optionally be used by the resource, i.e., it is unknown a priori if a given optional activity must be processed by the resource in the final schedule. This case typically occurs when an activity must run on one of several alternative resources [FLN00]. Following Vilím's notation, we call R the set of regular activities (known to be running on the resource, i.e., $R = \{i : v_i\}$) and O the set of optional activities, with $R \cup O = T$ and $R \cap O = \emptyset$.

To model optional activities, an additional boolean variable v_i is used to represent the fact that the activity i is used by the machine. The unary resource with transition times constraint involving optional activities imposes the following relation:

$$\forall i, j \in T : i \neq j \wedge v_i \wedge v_j \implies (c_i + tt_{i,j} \leq s_j) \vee (c_j + tt_{j,i} \leq s_i) \tag{URTTO}$$

PRECEDENCE GRAPH The precedence graph $G = \langle T, E\rangle$ is a data structure [Bru99, FLN00] used to maintain the precedences between activities of a given resource. In this graph, each vertex represents a given activity, and there is a directed edge from a vertex i to vertex j if and only if the activity i precedes the activity j, i.e., $c_i + tt_{i,j} \leq s_j$. In [BČ10], the authors describe propagation rules for the precedence graph while taking optional activities into account.

One can use the graph in a branching procedure (see Definition 1 in Chapter 1) where precedences between activities are added by adding edges. A recent CP approach [GH10] demonstrated experimentally that branching on the precedences can be effective[1], using smart search techniques rather than sophisticated propagators.

Finally, from a filtering perspective, additional precedences can be detected by computing the transitive closure of the graph.

BOUNDS OF A SET OF ACTIVITIES Ω The earliest starting time of an activity i is denoted est_i and its latest starting time is denoted lst_i. The domain of s_i is hence the interval $[est_i; lst_i]$. Similarly the earliest completion time of i is denoted ect_i and its latest completion time is denoted lct_i. The domain of c_i is thus the interval $[ect_i; lct_i]$. These definitions can be extended to a set of activities Ω. For instance, est_Ω is the earliest time when any activity in Ω can start and ect_Ω is the earliest time when all activities in Ω can be completed. We also define $p_\Omega = \sum_{j\in\Omega} p_j$ to be the sum of the processing times of the activities in Ω. While one can directly compute $est_\Omega = \min\{est_j | j \in \Omega\}$ and $lct_\Omega = \max\{lct_j | j \in \Omega\}$, it is NP-hard to compute the exact values of ect_Ω and lst_Ω [Vil07]. Instead, one usually computes a lower bound for ect_Ω and an upper bound for lst_Ω, as we will see in this chapter.

4.2 GLOBAL FILTERING RULES AND PROPAGATION ALGORITHMS

This section first describes the inference rules of the unary resource without transition times. Those rules are then extended in order to handle transition times. We also provide the different algorithms in order to compute them efficiently. The data structures required by the algorithms are described in Section 4.3.

4.2.1 *Filtering Rules for the Unary Resource*

The filtering rules presented in [Vil07] for the UR constraint fall in several categories known as Overload Checking (OC), Detectable Precedences (DP), Not-First/Not-Last (NF/NL), and Edge Finding (EF). They are valid for the general definition of ect_Ω of the earliest completion time of a set of activities $\Omega \subseteq T$. However, since the computation of its exact value is NP-hard, their implementation relies on an efficient computation of a lower bound ect_Ω^{LB0}, defined as:

$$ect_\Omega^{LB0} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + p_{\Omega'}\} \tag{4.2}$$

To define the different rules, we use the notation ect_Ω even if ect_Ω^{LB0} is used in practice, as we will use a stronger lower bound under the presence of transition

1 In their case, they do not actually use a precedence graph structure explicitly, but reify the precedence constraints and branch on the associated boolean variables.

times later in this chapter. Each rule has a symmetric counterpart that can easily be retrieved from the given definitions.

OVERLOAD CHECKING This rule tries to detect an inconsistency given the current domains. Intuitively, for a set of regular activities $\Omega \subseteq R$, if the earliest completion time is found to be larger than the latest completion time, an infeasibility is detected. Additionally, if Ω is extended with an optional activity i such that there would be an inconsistency, we know that the activity cannot be executed by the machine. Formally, we have:

$$\forall \Omega \subseteq R, \forall i \in (T \setminus \Omega) : ect_{\Omega \cup \{i\}} > lct_{\Omega \cup \{i\}} \implies \neg v_i \tag{OC}$$

Notice that if we have $i \in R \wedge \neg v_i$, the constraint is infeasible.

DETECTABLE PRECEDENCES This rule detects new precedences between pairs of activities. The reasoning uses the set of activities $DPrec(R, i)$ that can be detected as preceding a given activity i based on the current domains. It is defined as:

$$DPrec(R, i) = \{j \neq i \in R : ect_i > lst_j\} \tag{DPrec}$$

The inference rule states that the earliest start time of an activity i must at least be the earliest completion time of the set of activities that are detected as preceding i, that is $DPrec(R, i)$. Formally:

$$\forall i \in T : v_i \implies est_i = \max(est_i, ect_{DPrec(R,i)}) \tag{DP}$$

Notice that only the activities known to be running on the resource can be used to update other activities, hence the use of $DPrec(R, i)$ and not $DPrec(T, i)$. On the contrary, all activities (including optionals) can be updated. However, the domain of an optional activity should be updated only once it is known to be running on the resource (i.e., $v_i = \mathbf{true}$). At the same time, the inference about the domain of this activity *if it runs on the resource* can be used by other inference rules. Therefore, the domain is not updated until $v_i = \mathbf{true}$, but the inference on the domain *if the activity runs on the resource* is saved internally until $v_i = \mathbf{true}$. This is also done for the next inference rules.

NOT-LAST When a given activity i has a latest starting time that is strictly smaller than the earliest completion time of a set of regular activities Ω, this activity cannot be scheduled as the last one of the set $\Omega \cup \{i\}$. Its latest completion time can therefore be reduced to the maximum latest start time of the activities in Ω:

$$\forall \Omega \subseteq R, \forall i \in (T \setminus \Omega) : v_i \wedge ect_\Omega > lst_i \implies lct_i = \min(lct_i, \max_{j \in \Omega} lst_j) \tag{NL}$$

EDGE FINDING The rule detects new edges in the precedence graph: if adding an activity i to a set of activities Ω leads to an earliest completion larger than the latest completion of the set, then the activity i must succeed the activities in Ω:

$$\forall \Omega \subseteq R, \forall i \in (T \setminus \Omega) : v_i \wedge ect_{\Omega \cup \{i\}} > lct_\Omega \implies est_i = \max(est_i, ect_\Omega)) \tag{EF}$$

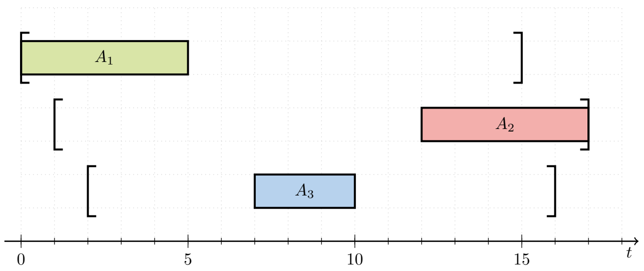


Figure 4.3: Example illustrating the missed failure detection of OC when not considering transition times.

LIMITATION Under the presence of transition times, the rules can be improved if they were considered, as illustrated in Example 1. In the next section, we strengthen the lower bound of ect_Ω so that it takes the transition times into account.

Example 1. *Consider a set of 3 regular activities* $\Omega = \{1, 2, 3\}$ *as shown in Figure 4.3. Consider also, for simplicity, that all pairs of activities from* Ω *have the same transition time* $tt_{i,j} = 3\,\forall i, j \in \{1, 2, 3\}$. *The OC rule detects a failure when* $ect_\Omega^{LB0} > lct_\Omega$. *The lower bound is:*

$$ect_\Omega^{LB0} = est_\Omega + \sum_{i\in\Omega} p_i = 0 + 5 + 5 + 3 = 13$$

As we have $lct_\Omega = \max_{i\in\Omega}\ lct_i = lct_2 = 17$, *the OC rule from [Vil07], combined with the transition times binary decomposition (Equation (URTTO)), does not detect a failure. However, as there are 3 activities in* Ω, *at least two transitions occur between these activities and it is actually not possible to find a feasible schedule. Indeed, taking these transition times into account, one could compute* $ect_\Omega = 13 + 2 \cdot tt_{i,j} = 13 + 2 \cdot 3 = 19 > 17 = lct_\Omega$, *and thus detect the failure.*

4.2.2 *Extending the Filtering Rules with Transition Times*

Let Π_Ω be the set of all possible permutations of activities in Ω. For a given permutation $\pi \in \Pi_\Omega$, where $\pi(i)$ is the activity taking place at position i, we can define the total time spent by transition times, tt_π, as follows:

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

A lower bound for ect_Ω that considers transition times can then be defined as:

$$ect_\Omega^{LB1} = \max_{\Omega'\subseteq\Omega} \left\{ est_{\Omega'} + p_{\Omega'} + \min_{\pi\in\Pi_{\Omega'}} tt_\pi \right\} \tag{4.3}$$

Unfortunately, computing this value is NP-hard as computing the optimal permutation $\pi \in \Pi$ minimizing tt_π amounts to solving a Traveling Salesman Problem.

Since embedding an exponential algorithm in a propagator is generally impractical, a looser lower bound should be used instead.

For each possible subset of cardinality $k \in \{0, \dots, n\}$, we compute the smallest transition time permutation on the set T of all activities requiring the resource:

$$\underline{tt}(k) = \min_{\{\Omega' \subseteq T:\ |\Omega'|=k\}} \left\{ \min_{\pi \in \Pi_{\Omega'}} tt_\pi \right\} \tag{4.4}$$

For each k, the lower bound computation thus requires one to find the shortest node-distinct $(k-1)$-edge path between any two nodes of the transition graph (see Section 4.1), which is also NP-hard as it can be cast into a resource-constrained shortest path problem. We proposed in [DVCS15] various lower bounds to achieve the pre-computation in polynomial time. They are described in Section 4.4. Our final lower bound formula for the earliest completion time of a set of activities, making use of pre-computed lower-bounds on transition times, is:

$$ect_\Omega^{LB2} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + p_{\Omega'} + \underline{tt}(|\Omega'|)\} \tag{4.5}$$

The different lower bounds of ect_Ω can be ordered as follows:

$$ect_\Omega^{LB0} \leq ect_\Omega^{LB2} \leq ect_\Omega^{LB1} \leq ect_\Omega$$

LIMITATION An important limitation of this approach arises in the context of *sparse* transition matrices, which typically occurs when activities are grouped in families (see Section 4.1). Indeed, when there exists a node-distinct path with K zero-transition edges, we have: $\underline{tt}(k) = 0 \ \forall k \in \{0, \dots, K+1\}$. The pruning achieved by the propagator is then equivalent to the one of the original algorithms from Vilím [Vil07], which has been shown to perform poorly when transition times are involved (see [DVCS15]). This is illustrated in Example 2.

Example 2. *Consider again the three activities $\Omega = \{1, 2, 3\}$ shown in Figure 4.3 with activity 1 belonging to family F_1, activity 2 to family F_2, and activity 3 to family F_3. The transition times are equal to 3 between activities from different families and equal to 0 between activities of the same family. Assume that 3 additional activities (not represented) also belong to family F_1. Since the transition times between any pair of activity from a same family is 0, we have that $\underline{tt}(2) = \underline{tt}(3) = 0$ and $ect_\Omega^{LB2} = 13 = ect_\Omega^{LB0}$, hence the OC of [DVCS15] is unable to detect the failure.*

To cope with this limitation, we will use a stronger lower bound by counting the number of different families present in a set Ω of activities instead of the cardinality of Ω. This amounts to find the shortest node-distinct $(k-1)$-edge path in the family transition graph (see Section 4.1) instead of the transition graph. Counting the number of families results in non-zero lower bounds even for small sets, assuming that there are no zero transition times between families. Formally, Equation (4.5) is replaced by:

$$ect_\Omega^{LB3} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + p_{\Omega'} + \underline{tt}(|F_{\Omega'}|)\} \tag{4.6}$$

where $F_\Omega = \{F_i \mid i \in \Omega\}$. The term $\underline{tt}(|F_{\Omega'}|)$ in Equation (4.6) is pre-computed using the same lower bounds as before, but using $tt^{\mathcal{F}}$ instead of tt. Notice that if $tt = tt^{\mathcal{F}}$, we have $ect_\Omega^{LB2} = ect_\Omega^{LB3}$.

Lemma 1. *In the presence of families, $ect_\Omega^{LB2} \leq ect_\Omega^{LB3}$.*

Proof. $tt^{\mathcal{F}}$ induces a graph that is isomorphic to a subgraph of the graph induced by tt and any (shortest) path induced by $tt^{\mathcal{F}}$ has a corresponding valid path induced by tt. Moreover, a shortest path of exactly k edges induced by tt has a length that is at most equal to a shortest path of exactly k edges induced by $tt^{\mathcal{F}}$. □

4.2.3 *Adapting the Algorithms*

We adapt the original algorithms of [Vil07] in order to consider transition times. Most of the modifications actually impact the underlying Θ-tree and Θ-Λ-tree data structures (described in Section 4.3), hence the algorithms are similar to the original ones. We think this is a strength of our approach, since adapting the algorithms can be done without too much work. Every algorithm has a counterpart that considers the activities in the opposite order. Those counterparts are not described for brevity.

NOTATION We denote by ect_Θ^* a lower bound of ect_Θ^{LB3} that will be used by the different algorithms. We describe in Section 4.3.1 the Θ-tree data structure that is used to compute this value. Moreover, following Vilím's notation, we will use a specific set of *gray activities* $\Lambda \subseteq T$ such that $\Lambda \cap \Theta = \emptyset$. For a given set Θ, this set is used to evaluate how ect_Θ would evolve *if* one of the gray activities of Λ were to be added to the set Θ. Formally, we are interested in computing

$$\overline{ect}_{(\Theta,\Lambda)} = \max(ect_\Theta, ect_{\Theta \cup \{i\}}, i \in \Lambda)$$

Again, we will compute a lower bound of this value, written $\overline{ect}^*_{(\Theta,\Lambda)}$. Section 4.3.2 describes the Θ-Λ-tree data structure, used to compute this value efficiently.

OVERLOAD CHECKING The checker (see Algorithm 4.2.1) goes over each activity in non-decreasing order of lct_i. For each activity, if it is not yet known if it will be executed by the resource, it is added to the set Λ (line 4) and the next activity is considered. If the activity has to run on the resource, it is added to the set Θ. The OC rule is then applied: if the earliest completion time of the current set Θ is larger than the latest completion time of the activity i we just added to Θ, the activity i cannot be executed on the machine. Since i is not optional, a feasible schedule cannot be found (see lines 7-9). The current optional activities in Λ are then possibly updated in lines 10-14: as long as it is possible to find an optional activity o such that adding it to Θ would lead to an overload, it is inferred that o cannot be executed by the machine, and o is removed from Λ.

DETECTABLE PRECEDENCES Algorithm 4.2.2 describes how the DP inference rule can be applied. It first sorts the regular activities by non-decreasing order

Algorithm 4.2.1 : Overload Checker

```
1  $(\Theta,\Lambda) \leftarrow (\emptyset,\emptyset)$
2  for $i \in T$ in non-decreasing order of $lct_i$ do
3      if $|D(v_i)| > 1$ then
4          $\Lambda \leftarrow \Lambda \cup \{i\}$                    /* $i$ is still optional. */
5      else if $v_i$ then
6          $\Theta \leftarrow \Theta \cup \{i\}$     /* $i$ is known to be used by the machine. */
7          if $ect^*_\Theta > lct_i$ then
8              return $\perp$                           /* Infeasibility detected. */
9          end
10         while $\overline{ect}^*_{(\Theta,\Lambda)} > lct_i$ do
11             $o \leftarrow$ optional (gray) activity responsible for $\overline{ect}^*_{(\Theta,\Lambda)}$
12             $v_o \leftarrow$ false                 /* $o$ cannot run on the machine. */
13             $\Lambda \leftarrow \Lambda \setminus \{o\}$
14         end
15     end
16 end
```

of latest start time and put them into a queue Q (line 2). Then, it traverses all the activities (including optional ones as they can be updated): for each activity i, as long as its earliest completion time is strictly larger than the latest start time of the first activity j in Q, j is removed from the queue and added to the set Θ. Once this is done, Θ is the set $DPrec(R, i)$ (see DPrec), and we can apply the DP rule (line 9). Moreover, as transition times are involved, the minimal transition from any family $F_j \in F_\Theta$ to the family F_i can also be added as it was not taken into account in the computation of ect^*_Θ. This transition is the minimal one from any family $F_j \in F_\Theta$ to F_i, because we do not know which activity will be just before i in the final schedule. The update rule becomes:

$$est_i \leftarrow \max \left\{ est_i, ect^*_\Theta + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i} \right\}$$

Notice that the value $\min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}$ can only be available in $\mathcal{O}(1)$ if it was pre-computed for any subset of families, which is exponential in $|\mathcal{F}|$ and therefore problematic if there are many families. It can also be computed in linear time, but it would increase the time complexity of the overall algorithm. In practice, the implementation can make use of the minimum transition from *any* family $f \in \mathcal{F} \setminus F_i$ *if* $F_i \notin F_\Theta$, and 0 otherwise.

When no transition times are involved, Vilím proved that *detected* precedences are all eventually *propagated* (see [Vil07]). In our case, this is not guaranteed: if a precedence is detected for a given pair of activities i and j, it is not ensured that after propagation we will have $est_j \geq ect_i + tt_{i,j}$ and $lct_i \leq est_j - tt_{i,j}$. The reason is that ect^*_Θ uses a lower bound on the transition times in Θ.

Algorithm 4.2.2 : Detectable Precedences

```
1  $\Theta \leftarrow \emptyset$
2  $Q \leftarrow$ queue of all regular activities $r \in R$ in non-decreasing order of $lst_r$
3  $j \leftarrow Q.pop()$
4  for $i \in T$ in non-decreasing order of $ect_i$ do
5      while $ect_i > lst_j$ do
6          $\Theta \leftarrow \Theta \cup \{j\}$
7          $j \leftarrow Q.pop()$
8      end
9      $est'_i \leftarrow \max\left\{est'_i, ect^*_{\Theta\setminus\{i\}} + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}\right\}$
10 end
```

NOT LAST The NL inference rule can be applied with Algorithm 4.2.3, similarly to Algorithm 4.2.2: a queue Q is filled with regular activities, and all activities (regular and optional) are then traversed in non-decreasing order of latest completion time. For each activity i, activities from Q having a larger latest starting time than the latest completion time of i, are removed from the queue and added to the set Θ (line 5-8). Θ is then the set of activities with a latest starting time strictly smaller than the latest completion time of i. The NL rule can then be applied (lines 9-11). An analogous reinforcement to the DP rule due to transition times can be applied when updating lct_i (see line 10).

Algorithm 4.2.3 : Not-Last

```
1  $\Theta \leftarrow \emptyset$
2  $Q \leftarrow$ queue of all regular activities $r \in R$ in non-decreasing order of $lst_r$
3  $j \leftarrow Q.peek()$
4  for $i \in T$ in non-decreasing order of $lct_i$ do
5      while $lct_i > lst_j$ do
6          $\Theta \leftarrow \Theta \cup \{j\}$
7          $j \leftarrow Q.pop()$
8      end
9      if $ect^*_{\Theta\setminus\{i\}} > lst_i$ then
10         $lct'_i \leftarrow \min\left\{lct'_i, lst_j - \min_{f \in F_\Theta} tt^{\mathcal{F}}_{F_i,f}\right\}$
11     end
12 end
```

EDGE FINDING Unlike the previous algorithms, Algorithm 4.2.4 starts with a set Θ filled with all regular activities. We also directly fill the set Λ with the optional activities[2] so that their domain can be updated but they can never be used to update other activities (since they will not be in the set Ω in the EF rule). A queue Q of regular activities sorted in non-increasing order of latest completion time is also initialized. The algorithm traverses this queue and the activities in Θ will progressively be removed from Θ and added to the set Λ of gray activities. For each activity j popped out the queue Q, the algorithm first checks for an overload, before j is removed from Θ (lines 5-7). This is equivalent to what is done in Algorithm 4.2.1 for regular activities, so it is actually facultative. The activity j is then grayed : it is transferred from Θ to Λ. This means it is no more in the set Θ we consider, but it will be part of the activities used to infer what would happen if one of them was added to Θ. Lines 10-14 try to apply the EF rule for some of the current gray activities: as long as adding one of the gray activities would imply an overload (i.e., condition in line 10 is verified), we identify which gray activity i is responsible for this potential overload, we update its earliest start time, and remove it from Λ. The EF rule is strengthened using transition times similarly to the DP and NL rules.

Algorithm 4.2.4 : Edge Finding

```
1  $(\Theta, \Lambda) \leftarrow (R, O)$
2  $Q \leftarrow$ queue of all regular activities $r \in R$ in non-increasing order of $lct_r$
3  $j \leftarrow Q.peek()$
4  while $|Q| > 1$ do
5      if $ect^*_\Theta > lct_j$ then
6          return $\perp$
7      end
8      $(\Theta, \Lambda) \leftarrow (\Theta \setminus \{j\}, \Lambda \cup \{j\})$
9      $j \leftarrow Q.pop()$
10     while $\overline{ect}^*_{(\Theta,\Lambda)} > lct_j$ do
11         $i \leftarrow$ gray activity responsible for $\overline{ect}^*_{(\Theta,\Lambda)}$
12         $est'_i \leftarrow \max\left\{est_i, ect^*_\Theta + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}\right\}$
13         $\Lambda \leftarrow \Lambda \setminus \{i\}$
14     end
15 end
```

PRECEDENCE GRAPH PROPAGATOR The last propagator uses the precedence graph data structure (see Section 4.1). The topological order of all known precedences (i.e., edges in the digraph) is first constructed so that one can update the variable domains efficiently. Indeed, the earliest start time of an activity i that is before an other activity j in this order cannot be influenced by the domain of

2 Notice that this is equivalent to what is proposed in [Vil09b, Vil09a]. The author suggests handling optional activities by modifying the input data rather than the algorithm: lct_o is assumed to be ∞ for all optional activities $o \in O$.

s_j and c_j. Algorithm 4.2.5 therefore builds a queue Q of activities in topological order of the precedence graph. It then simply traverses Q and for each activity i, it applies the pairwise rule URTTO for all its successors in the precedence graph. In addition, if a successor j of an activity i is known to be running on the resource (i.e., v_j is true), then one can use j to update the latest completion time of the activity i (see lines 6-8).

Important note Notice that when transition times are involved, this algorithm is mandatory in order to ensure the pruning is complete: because we use a lower bound of the earliest completion time of a set of activities Θ in the other algorithms (ect^*_Θ), they are are not sufficient to ensure correctness of a given (partial) assignment of all s_i, $\forall i \in T$.

Algorithm 4.2.5 : Precedence Graph Propagation

```
1  $Q \leftarrow$ queue of all regular activities $r \in R$ in topological order in the
   precedence graph $G$
2  while $|Q| > 1$ do
3      $i \leftarrow Q.pop()$
4      foreach successor $s$ of $i$ do
5          $est_s \leftarrow \max\{est_s, ect_i + tt_{i,s}\}$
6          if $v_s$ then
7              $lct_i \leftarrow \min\{lct_i, lst_i - tt_{i,s}\}$
8          end
9      end
10 end
```

COMPLEXITIES Section 4.3 describes data structures that allow to retrieve ect^*_Θ in $\mathcal{O}(1)$ while addition/removal of an activity to/from Θ are performed in $\mathcal{O}(\log(n) \cdot \log(f))$, for n activities and f families. All algorithms but the Precedence Graph have therefore a time complexity of $\mathcal{O}(n \cdot \log(n) \cdot \log(f))$. The precedence graph propagator runs in $\mathcal{O}(n^2)$.

4.3 EXTENDING THE Θ-TREE AND Θ-Λ-TREE DATA STRUCTURES

To efficiently use the sets Θ and Λ, the algorithms described in Section 4.2.3 rely on the so-called Θ-tree and Θ-Λ-tree data structures, introduced by Vilím. Those structures are used to compute efficiently and incrementally ect^*_Θ and $\overline{ect}^*_\Theta$ for sets of activities Θ and Λ. We describe in this section how we extended them to handle (family-based) transition times.

4.3.1 *Extended Θ-tree*

A Θ-tree is a balanced binary tree in which each leaf represents an activity from a set Θ and internal nodes gather information about the set of activities represented by the leaves under this node, denoted $Leaves(v)$. We write $l(v)$ for the left child of v and $r(v)$ for the right one. Leaves are ordered in non-decreasing order of the earliest start time of the activities: for two activities i and j, if $est_i < est_j$, then the leaf representing i is at the left of the leaf representing j.

The main value stored in a node v is the lower bound of $ect_{Leaves(v)}$, denoted ect^*_v. To be able to compute this value incrementally upon insertion or deletion of an activity in the Θ-tree, one needs to maintain additional values.

Without any transition times involved, Vilím has shown [Vil07] that by defining $ect^*_v = ect^{LB0}_{Leaves(v)}$, it suffices to store additionally $p_v = p_{Leaves(v)}$. In a leaf v representing an activity i, one can compute $p_v = p_i$ and $ect^*_v = ect_i$. In an internal node v, one can compute:

$$\begin{aligned} p_v &= p_{l(v)} + p_{r(v)} \\ ect^*_v &= \max\left\{ect^*_{r(v)}, ect^*_{l(v)} + p_{r(v)}\right\} \end{aligned}$$

Hence, the values only depend on the values stored in the two children.

In our case, we would like instead to define $ect^*_v = ect^{LB3}_{Leaves(v)}$ in order to take (family-based) transition times into account. However, this value cannot easily be computed incrementally, so we compute a lower bound, i.e., $ect^*_v \leq ect^{LB3}_{Leaves(v)}$. In addition to ect^*_v, one needs to store not only p_v, but also $F_v = F_{Leaves(v)}$, the set of the families of the activities in $Leaves(v)$. In a leaf v representing an activity i, one can compute $p_v = p_i$, $ect^*_v = ect_i$, and $F_v = \{F_i\}$. In an internal node v, one can compute:

$$\begin{aligned} p_v &= p_{l(v)} + p_{r(v)} \\ F_v &= F_{l(v)} \cup F_{r(v)} \\ ect^*_v &= \max\begin{cases} ect^*_{r(v)} \\ ect^*_{l(v)} + p_{r(v)} + \underline{tt}\left(\left|F_{r(v)} \setminus F_{l(v)}\right| + 1\right) \end{cases} \end{aligned}$$

Intuitively, ect^*_v is maximized either by only considering activities in $r(v)$, or by adding to $ect^*_{l(v)}$ the processing times and (a lower bound of) the transition times due to activities in $r(v)$. In the latter case, only *additional* families are counted to compute the lower bound on transition times, that is, the families that are present in the right child but not in the left one. Hence, the cardinality of the set $F_{r(v)} \setminus F_{l(v)}$ is considered. Notice we always add 1 family to the count because of the definition of $\underline{tt}(k)$.

Before we prove this lower bound is correct, let us prove in Lemma 2 a property of the function $\underline{tt}(k)$.

Lemma 2. $\forall n \in [0,T], k \in [0,n] : \underline{tt}(n) \geq \underline{tt}(k) + \underline{tt}(n-k+1)$

Proof. The optimal path p^{opt} in the transition graph leading to the value $\underline{tt}(n)$ can be split in two subpaths:

- $p^{opt}_{[1..k]}$ with $k-1$ edges. Its total length is greater than or equal to $\underline{tt}(k)$ (as $\underline{tt}(k)$ is the minimum), the length of the optimal path with $k-1$ edges.
- $p^{opt}_{[k..n]}$ with $n-k$ edges. Its total length is greater than or equal to $\underline{tt}(n-k+1)$, the length of the optimal path with $n-k$ edges.

Therefore $\underline{tt}(n) = p^{opt}_{[1..k]} + p^{opt}_{[k..n]} \geq \underline{tt}(k) + \underline{tt}(n-k+1)$. □

Lemma 3. $ect^*_v \leq ect^{LB3}_{Leaves(v)}$

Proof. By induction. If v is a leaf representing activity i, then $ect^*_v = ect_i = ect^{LB3}_{\{i\}}$. Otherwise, our induction hypothesis is that $ect^*_{l(v)} \leq ect^{LB3}_{Leaves(l(v))}$ and $ect^*_{r(v)} \leq ect^{LB3}_{Leaves(r(v))}$. Let us call $\Omega^{LB3} \subseteq Leaves(v)$ the optimal set to compute $ect^{LB3}_{Leaves(v)}$. For space reasons, we write $L(\Omega)$ to denote $Leaves(\Omega)$.

One can consider two cases:

a) $ect^*_v = ect^*_{r(v)}$. We have $ect^*_{r(v)} \leq ect^{LB3}_{L(r(v))}$ (by induction) and $ect^{LB3}_{L(r(v))} \leq ect^{LB3}_{L(v)}$ (by definition). Therefore, $ect^*_v \leq ect^{LB3}_{L(v)}$.

b) $ect^*_v = ect^*_{l(v)} + p_{r(v)} + \underline{tt}\Big(|F_{r(v)} \setminus F_{l(v)}| + 1\Big)$. Then, we have:

$$\begin{aligned}
ect^*_v &\leq ect^{LB3}_{L(l(v))} + p_{r(v)} + \underline{tt}\Big(|F_{r(v)} \setminus F_{l(v)}| + 1\Big) && \text{(by induction)}\\
&= \max_{\Omega_l \subseteq L(l(v))} \{est_{\Omega_l} + p_{\Omega_l} + \underline{tt}(|F_{\Omega_l}|)\} + p_{r(v)} + \underline{tt}\Big(|F_{r(v)} \setminus F_{l(v)}| + 1\Big)\\
&= \max_{\Omega_l \subseteq L(l(v))} \{est_{\Omega_l} + p_{\Omega_l} + p_{r(v)} + \underline{tt}(|F_{\Omega_l}|) + \underline{tt}\Big(|F_{r(v)} \setminus F_{l(v)}| + 1\Big)\}\\
&= \max_{\Omega_l \subseteq L(l(v))} \{est_{\Omega_l} + p_{\Omega_l \cup L(r(v))} + \underline{tt}(|F_{\Omega_l}|) + \underline{tt}\Big(|F_{r(v)} \setminus F_{l(v)}| + 1\Big)\}\\
& && \text{(since } p_{\Omega_l} + p_{L(r(v))} = p_{\Omega_l \cup L(r(v))}\text{)}\\
&= \max_{\Omega_l \subseteq L(l(v))} \{est_{\Omega_l \cup L(r(v))} + p_{\Omega_l \cup L(r(v))} + \underline{tt}(|F_{\Omega_l}|) + \underline{tt}\Big(|F_{r(v)} \setminus F_{l(v)}| + 1\Big)\}\\
& && \text{(since } est_{\Omega_l} = est_{\Omega_l \cup L(r(v))}\text{)}\\
&\leq \max_{\Omega_l \subseteq L(l(v))} \{est_{\Omega_l \cup L(r(v))} + p_{\Omega_l \cup L(r(v))} + \underline{tt}\Big(|F_{\Omega_l \cup L(r(v))}|\Big)\} && \text{(by Lemma 2)}\\
&\leq ect^{LB3}_{L(v)} && \text{(by definition)}
\end{aligned}$$

□

COMPLEXITY We use bit sets to represent the set of families in each node. The space complexity of the Θ-tree is therefore $\mathcal{O}(n \cdot |\mathcal{F}|)$. The set operations we use are *union*, *intersection*, *difference* and *cardinality*. Using bit sets, the 3 former ones are $\mathcal{O}(1)$ and the latter one is $\mathcal{O}(\log(|\mathcal{F}|))$ with a *binary population count* [War13]. The time complexity of insertion and deletion of an activity in the Θ-tree is therefore $\mathcal{O}(\log(n) \cdot \log(|\mathcal{F}|))$.

Example 3. *Let us consider the activities presented in Figure 4.4 (left). The family transition matrix $tt^{\mathcal{F}}$ is given in Figure 4.4 (center). The pre-computed values of $\underline{tt}(k)$ are reported in Figure 4.4 (right). Figure 4.5 illustrates the extended Θ-tree when all activities are inserted. Note that the value at the root of the tree is indeed a lower bound since we have $ect^*_\Theta = 75 \leq ect^{LB3}_\Theta = 80 \leq ect_\Theta = 85$.*

	1	3	2	4
est	0	15	25	30
p	10	10	20	25
F	F_1	F_2	F_3	F_3

$$tt^{\mathcal{F}} = \begin{pmatrix} 0 & 10 & 15 \\ 5 & 0 & 10 \\ 5 & 15 & 0 \end{pmatrix}$$

$\underline{tt}(k)$	k
0	0
1	0
2	5
3	15

Figure 4.4: Four activities and their families (left), transition times for the families (center), and pre-computed lower bounds for the transition times (right).

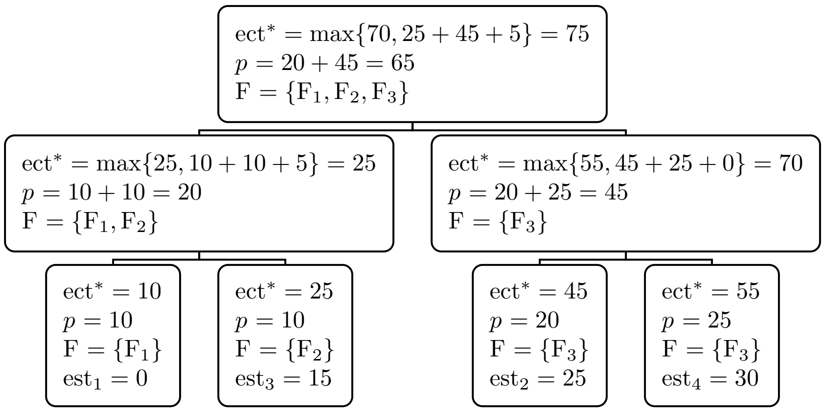

$\mathrm{ect}^* = \max\{70, 25+45+5\} = 75$
$p = 20 + 45 = 65$
$\mathrm{F} = \{\mathrm{F}_1, \mathrm{F}_2, \mathrm{F}_3\}$
$\mathrm{ect}^* = \max\{25, 10+10+5\} = 25$
$p = 10 + 10 = 20$
$\mathrm{F} = \{\mathrm{F}_1, \mathrm{F}_2\}$
$\mathrm{ect}^* = \max\{55, 45+25+0\} = 70$
$p = 20 + 25 = 45$
$\mathrm{F} = \{\mathrm{F}_3\}$
$\mathrm{ect}^* = 10$
$p = 10$
$\mathrm{F} = \{\mathrm{F}_1\}$
$\mathrm{est}_1 = 0$
$\mathrm{ect}^* = 25$
$p = 10$
$\mathrm{F} = \{\mathrm{F}_2\}$
$\mathrm{est}_3 = 15$
$\mathrm{ect}^* = 45$
$p = 20$
$\mathrm{F} = \{\mathrm{F}_3\}$
$\mathrm{est}_2 = 25$
$\mathrm{ect}^* = 55$
$p = 25$
$\mathrm{F} = \{\mathrm{F}_3\}$
$\mathrm{est}_4 = 30$

Figure 4.5: A Θ-tree when all activities of Figure 4.4 are inserted.

4.3.2 *Extended Θ-Λ-tree*

Algorithms 4.2.1 and 4.2.4 require an extension of the original Θ-tree, called Θ-Λ-tree [Vil07]. In this extension, leaves are marked as either *white* or *gray*. White leaves represent activities in the set Θ and gray leaves represent activities that are in a second set, Λ, with $\Lambda \cap \Theta = \emptyset$. In addition to ect_v^*, a lower bound to the *ect* of Θ, a Θ-Λ-tree also aims at computing $\overline{ect}_v^*$, which is a lower bound to $\overline{ect}_{(\Theta,\Lambda)}$, the largest *ect* obtained by including *one* activity from Λ into Θ:

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

In addition to p_v, ect_v^*, Vilím's original Θ-Λ-tree also maintains $\overline{p}_v$ and $\overline{ect}_v^*$, respectively corresponding to p_v and ect_v^*, *if* a single gray activity $i \in \Lambda$ in the sub-tree rooted at v maximizing $ect_{Leaves(v) \cup \{i\}}$ *was* included.

Our extension to the Θ-Λ-tree is similar to the one outlined in Section 4.3.1 for the Θ-tree: in addition to the previous values, each internal node also stores F_v and $\overline{F}_v$ in order to compute the lower bounds ect_v^* and $\overline{ect}_v^*$. Adapting the rules for the Θ-Λ-tree requires caution when families are involved. In [Vil07] and [DVCS15], the rules only use implicitly the information about *which* gray activity is considered in the update. In our case, the rules must consider explicitly where the used gray activity is located: either in the left subtree, denoted (L), or in the right subtree, denoted (R). The rules are then defined as:

$$\begin{aligned}
\overline{ect}_v^* &= \max \begin{cases} \overline{ect}_{l(v)}^* + p_{r(v)} + \underline{tt}\Big(|F_{r(v)} \setminus \overline{F}_{l(v)}| + 1\Big) & \text{(L)} \\ ect_{l(v)}^* + \overline{p}_{r(v)} + \underline{tt}\Big(|\overline{F}_{r(v)} \setminus F_{l(v)}| + 1\Big) & \text{(R)} \\ \overline{ect}_{r(v)}^* & \text{(R)} \end{cases} \\
\overline{F}_v &= \begin{cases} \overline{F}_{l(v)} \cup F_{r(v)} & \text{(L)} \\ F_{l(v)} \cup \overline{F}_{r(v)} & \text{(R)} \end{cases} \\
\overline{p}_v &= \begin{cases} \overline{p}_{l(v)} + p_{r(v)} & \text{(L)} \\ p_{l(v)} + \overline{p}_{r(v)} & \text{(R)} \end{cases}
\end{aligned}$$

In the rules above, the choice of which formula to use for $\overline{F}_v$ and $\overline{p}_v$ depends on the letter, either (L) or (R), associated with the term maximizing $\overline{ect}_v^*$, hence this value must be computed first. If a leaf v represents an activity i, then we simply have $\overline{ect}_v^* = ect_i$, $\overline{p}_v = p_i$, and $\overline{F}_v = \{F_i\}$. The rules for p_v, ect_v, and F_v are as presented in Section 4.3.1, but one must also define, for a gray leaf v, $ect_v^* = -\infty$, $p_v = 0$, and $F_v = \emptyset$.

As for the extended Θ-tree introduced in Section 4.3.1, the time complexity for the insertion and the deletion of an activity is $\mathcal{O}(\log(n) \cdot \log(|\mathcal{F}|))$.

4.3.3 *Strengthening ect_Θ^* and $\overline{ect}_{(\Theta,\Lambda)}^*$*

ect_Θ^* is a lower bound for ect_Θ^{LB3}. One can actually strengthen the value computed with the Θ-tree to get a value closer to ect_Θ^{LB3}. An idea from [BT96, VB12] that is also used in [AF08] is to pre-compute the exact minimum total transition time for every subset of families[3].

3 The approach can also be used for sets of activities. The description focuses here on families since it was initially used in the context of family-based transition times.

For a subset of families $\mathcal{F}' \subseteq \mathcal{F}$, let $tt\left(\mathcal{F}'\right)$ denote the minimum total transition time used for any activity set Θ such that $F_\Theta = \mathcal{F}'$. Assuming $tt\left(F_\Theta\right)$ is accessible in $\mathcal{O}(1)$, each time we access to the value ect^*_Θ in the algorithms of Section 4.2.3, we can also compute

$$ect^{tsp}_\Theta = est_\Theta + p_\Theta + tt\left(F_\Theta\right)$$

without changing the complexity of the algorithms. $tt\left(F_\Theta\right)$ must be precomputed for all subsets of families, so this is tractable only if there are few families as it requires solving many Traveling Salesman Problems of increasing sizes. est_Θ can be easily maintained in the Θ-tree, and the values p_Θ and F_Θ can be obtained in $\mathcal{O}(1)$ in the root node of the Θ-tree.

ect^{tsp}_Θ can be larger than ect^*_Θ because it uses $tt\left(F_\Theta\right)$ instead of $\underline{tt}(|F_\Theta|)$. This typically occurs when $ect^{tsp}_\Theta = ect^{LB3}_\Theta$. On the contrary, because ect^{tsp}_Θ considers all activities in Θ and never a subset, it might be smaller than ect^*_Θ. We therefore use the maximum of those two values in the different propagators.

One can also consider the family of the updated activity: similarly to $tt\left(\mathcal{F}'\right)$, let us write $tt\left(F_i \rightarrow \mathcal{F}'\right)$ the minimum total transition time when the processing starts with some activity of the family $F_i \in \mathcal{F}'$, and $tt\left(\mathcal{F}' \rightarrow F_i\right)$ when it completes with an activity of the family $F_i \in \mathcal{F}'$. We can pre-compute these values for every set of families $\mathcal{F}' \subseteq \mathcal{F}$ and every family $F_i \in \mathcal{F}'$ with a dynamic program running in $\Theta(|\mathcal{F}|^2 \cdot 2^{|\mathcal{F}|})$ and requiring $\Theta(|\mathcal{F}| \cdot 2^{|\mathcal{F}|})$ of memory. For instance, for $tt\left(F_i \rightarrow \mathcal{F}'\right)$, one defines:

$$\begin{cases} tt\left(F_i \rightarrow \{F_i\}\right) = 0 & \forall F_i \in \mathcal{F} \\ tt\left(F_i \rightarrow \{\mathcal{F}' \cup F_i\}\right) = \min\limits_{F_j \in \mathcal{F}'} \{tt^{\mathcal{F}}_{F_i,F_j} + tt\left(F_j \rightarrow \mathcal{F}'\right)\} & \forall \mathcal{F}' \subseteq \mathcal{F}, \forall F_i \in \mathcal{F} \setminus \mathcal{F}' \end{cases}$$

For instance, for the Detectable Precedences algorithm, the update rule becomes:

$$est_i \leftarrow \max\left\{est_i, ect^*_\Theta + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}, est_\Theta + p_\Theta + tt\left(F_\Theta \rightarrow F_i\right)\right\}$$

The same idea can be used to strengthen $\overline{ect}^*_{(\Theta,\Lambda)}$:

$$\overline{ect}^{tsp}_{(\Theta,\Lambda)} = \min\{est_\Theta, est_r\} + p_{\Theta\cup\{r\}} + tt\left(F_{\Theta\cup\{r\}}\right)$$

where r is the gray responsible activity (see line 11 in Algorithm 4.2.4). A subtle point is that the responsible activity r is not accessible from the Θ-Λ-tree as for $\overline{ect}^*_{(\Theta,\Lambda)}$, so we should iterate over all $r' \in \Lambda$ to maximize $\overline{ect}^{tsp}_{(\Theta,\Lambda)}$. We therefore use the responsible activity of $\overline{ect}^*_{(\Theta,\Lambda)}$ to compute $\overline{ect}^{tsp}_{(\Theta,\Lambda)}$.

4.4 LOWER BOUNDS ON THE MINIMUM TOTAL TRANSITION OF A SET OF ACTIVITIES

In this section, we describe different lower bounds [DVCS15] for Equation 4.4, recalled hereafter:

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

For each k, one has to find the shortest node-distinct $(k-1)$-edge path between any two nodes of the (family) transition graph (see Section 4.1), which is NP-hard as it can be cast into a resource-constrained shortest path problem. Even though $\underline{tt}(k)$ is to be precomputed, it is desirable to have polynomial precomputation, which justifies the use of the lower bounds explained in this section. A more detailed description can be found in [Dej16], we summarize them here so that the chapter is self-contained. Notice that none of the lower bounds subsume each other, so the final lower bound for a given cardinality k will be the maximum between the different lower bounds for this cardinality.

MINIMUM WEIGHT FOREST This lower bound consists of finding the set of $k-1$ edges with a minimum cost. Basically, we use Kruskal's algorithm [Kru56] to prevent cycles in our selection. As soon as $k-1$ edges have been selected, the algorithm is stopped. The result being a minimum weight forest in the general case, it is a lower bound of our original problem since it does not ensure to obtain a simple path in the graph.

SHORTEST WALK A dynamic program can be used to compute a lower bound on the minimum transition in a set of cardinality k. The idea is to compute a shortest walk with $k-1$ edges in the transition graph. Formally, we define $SW(e,n)$ as the shortest walk with e edges from any node to node n. To compute this value for all number of edges e and all nodes n, we rely on the following $\mathcal{O}(k \cdot T^2)$ dynamic program:

$$SW(0,x) = 0, \forall x \in [1,T]$$
$$SW(e+1,n) = \min_x SW(e,x) + w_{x,n}$$

where $e < k$ and $w_{x,n}$ is the weight on the edge from the node x to the node n. The lower bound for a given cardinality k is finally:

$$\min_x SW(k,x)$$

Notice this lower bound ensures the solution to be a walk in the graph but it does not prevent cycles. However, as suggested in [CMT81], one can strengthen the bound by avoiding 1-cycles, i.e., cycles of the form $x \to y \to x$.

MINIMUM ASSIGNMENT A lower bound based on a Minimum Assignment problem was proposed by Brucker and Thiele [BT96]: two sets containing all the nodes of the transition graph are constructed and a minimum assignment of k edges is searched for, that is, the edges always link an activity of one set with an activity of the other set. One can model this problem as a Minimum-Cost Maximum-Flow problem in a manner similar to the reduction of a minimum weight bipartite matching.

LAGRANGIAN RELAXATION To find the shortest simple path with k edges in the transition graph, one can add a source (node 0) and a sink node (node n + 1) to the transition graph so that the edges from the source node to all nodes (but the sink one) and the edges from the nodes (but the source one) to the sink node have a transition of zero. Then, one can solve the problem by searching for the shortest path from the source to the sink with $k+2$ edges. This can be solved with the following integer linear program:

$$\begin{aligned}
\textbf{minimize} \quad & \sum_i \sum_j tt_{i,j} \cdot x_{i,j} \\
\textbf{such that} \quad & \sum_j x_{0,j} - \sum_j x_{j,0} = 1 \\
& \sum_j x_{n+1,j} - \sum_j x_{j,n+1} = -1 \\
& \sum_j x_{i,j} - \sum_j x_{j,i} = 0 \\
& \sum_i \sum_j x_{i,j} = k \qquad \text{(CARD)} \\
& x_{i,j} \in \{0,1\}
\end{aligned}$$

This problem is NP-hard, so we propose to solve a lagrangian relaxation: we remove the edge cardinality constraint (i.e., Equation CARD) and penalize its violation in the objective function. Without the cardinality constraint, the shortest path can be computed with the Bellman-Ford algorithm [Bel56, Moo59] that is also able to detect a negative cycle. If this occurs, we use a classic linear relaxation instead of using the Bellman-Ford algorithm.

EXACT SHORTEST PATH FOR EVERY SUBSET Using the definitions given in Section 4.3.3, one can compute the best possible lower bound based on the cardinality of a set of activities/families. We compute the value of the shortest path for every subset, and for each cardinality k, we take the smallest shortest path of all subsets of cardinality k:

$$\underline{tt}(k) = \min_{|\mathcal{F}'|=k} tt\left(\mathcal{F}'\right)$$

The other lower bounds described before are upper bounded by this approach. However, it is not polynomial, so it can only be used for problems with a few activities/families.

4.5 EXPERIMENTATIONS

We split our evaluation in two parts: first, we consider the case where there are no optional activities, which was more studied in the literature. The experiments were conducted on Job-Shop Problem with Sequence Dependent Transition

Times (JSPSDTT) instances. In a second time, we consider the same problem with alternative machines, that is modeled using optional activities from the resource point of view.

SETTING We used AMD Opteron processors (2.7 GHz), the Java Runtime Environment 8 and the constraint solver *OscaR* [Osc12]. The memory consumption was limited to 4 GB.

4.5.1 *Experimentations without Optional Activities*

Problem instances

We have used two sets of instances. First, we used the standard **t2ps** instances from Brucker and Thiele [BT96]. However, there are only 15 of them, and we wanted to evaluate instances with more families, jobs, and machines in order to challenge the scalability of the different approaches. We therefore generated a new set of 315 instances, here referred to as **uttf**, with up to 50 jobs, 15 machines and 30 families.[4]

State-of-the-art filtering with Families

Based on the definition of $tt\,(F_i \to \{\mathcal{F}'\})$, two propagators are introduced in [AF08]:

- A DP-like propagator called UPDATEEARLIESTSTART running in $\mathcal{O}(n^2 \cdot \log(n))$.
- An EF-like propagator called PRIMALEDGEFINDING running in $\mathcal{O}(|\mathcal{F}| \cdot n^2)$.

Although the filtering obtained with these propagators can be stronger than their counterpart from [Vil07] and our extensions, the time complexity of the propagators is quite high as compared to $\mathcal{O}(n \cdot \log(n) \cdot \log(|\mathcal{F}|))$. In addition, they do not make use of a Not-First/Not-Last rule and the pre-computation of the minimum exact transition times for every subset of family is only tractable for small (typically less than 10) values of $|\mathcal{F}|$.

Compared Propagators

We compare models with the following propagators for Equation (URTTO):

- *decomp*: binary decomposition of Equation (URTTO) only.
- *urtt*: propagators for URTT from [DVCS15].
- art_{ex}: propagators of [AF08] using exact values for $tt\,(\mathcal{F})$, $tt\,(F \to \mathcal{F})$ and $tt\,(\mathcal{F} \to F)$.

4 The instances are available at http://becool.info.ucl.ac.be/resources/uttf-instances.

- art_{lb}: propagators of [AF08] adapted to make use of cardinality-based lower bounds from Section 4.4 for $tt(\mathcal{F})$, $tt(F \to \mathcal{F})$ and $tt(\mathcal{F} \to F)$.
- $urttf_{ex}$: propagators introduced in this chapter making use of the exact values for $\underline{tt}(|\mathcal{F}|)$ computed with $\min_{\mathcal{F}':|\mathcal{F}'|=|\mathcal{F}|} tt(\mathcal{F}')$.
- $urttf_{lb}$: propagators introduced in this chapter making use of lower bounds of Section 4.4 for $\underline{tt}(|\mathcal{F}|)$.

Replay Evaluation

We used the Replay Evaluation described in Chapter 3. To generate the CBTs, the Conflict Ordering Search [Gay+15] was used, as it was shown to be a good search strategy for scheduling problems. The generation lasted for 300 seconds, and we enforced a timeout of 1,800 seconds for the replay. The running times reported here do not take into account the pre-computation step since they are negligible (generally less than 2 sec. and max 10 sec.). Our performance profiles have here a logarithmic scale and we use the initial definition from [DM02]: the set of baseline approaches $\mathcal{B}$ (see Definition 2.1 in Chapter 2) contains all the different models.

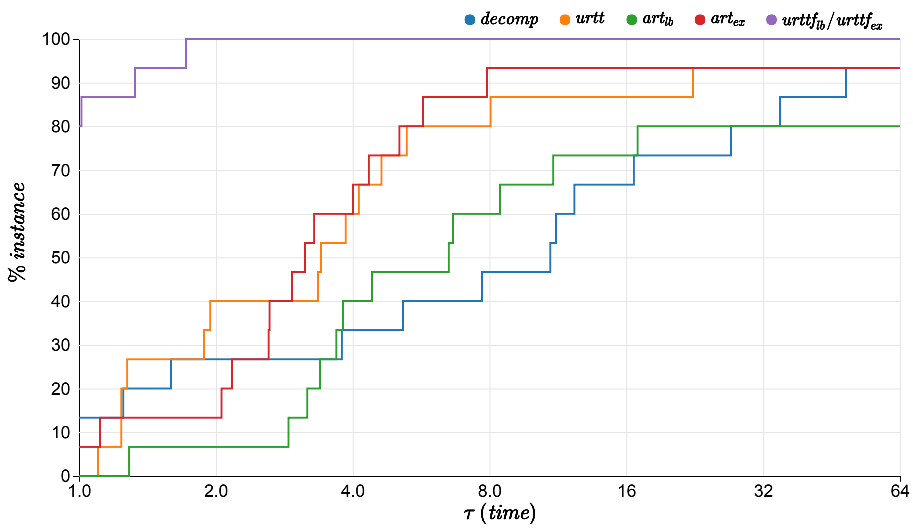


Figure 4.6: Performance profiles on **t2ps** instances for the time metric.

Results on the ***t2ps*** *Instances*

Figures 4.6 and 4.7 provide the performance profiles for the time and number of backtracks, respectively. Figure 4.7 shows that, interestingly, $urttf_{lb}$ prunes exactly as much as $urttf_{ex}$. This is due to the fact that our lower bounds are here able to compute the same values than $\min_{\mathcal{F}':|\mathcal{F}'|=|\mathcal{F}|} tt(\mathcal{F}')$. This suggests that we often do not have to compute the exact values for $tt(\mathcal{F})$ with the resource-consuming dynamic program, which is interesting since it is not tractable when there are many

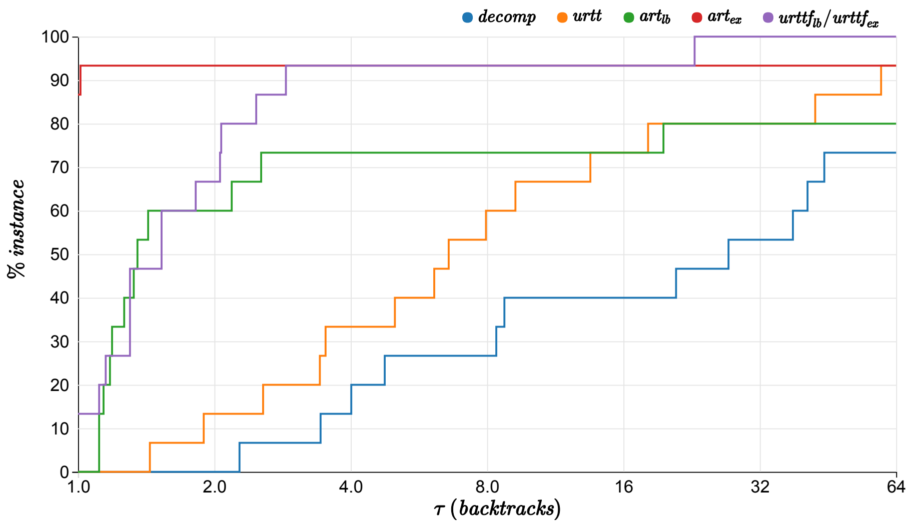


Figure 4.7: Performance profiles on **t2ps** instances for the number of backtracks metric.

families. We can see that from a time perspective (Figure 4.6), our approach is the fastest for $\sim 80\%$ of the instances ($urttf_{ex}$ being here equivalent to $urttf_{lb}$, see the function in $\tau = 1$ in Figure 4.6). But our approach is also robust, as the other instances (i.e., the remaining 20%) are solved within a factor $\tau < 2$ compared to the best model for those remaining instances. Considering the number of backtracks, our approach generally achieves less pruning than art_{ex} (not more than three times), but substantially more than $urtt$. This lack of pruning as compared to art_{ex} is compensated in practice by the low time complexity. Although not reported, we tried to combine $urttf_{ex}$ and art_{ex} and the performances were close to the ones of art_{ex} alone, thus only inducing a small overhead when $urttf_{ex}$ does not provide additional pruning.

*Results on the **uttf** Instances*

First of all, we consider the approaches art_{ex} and $urttf_{ex}$ unable to solve (i.e., times out by default) the 120 instances (out of 315) with 20 families or more, since the pre-computation becomes too expensive in terms of CPU and memory usage according to our 4 GB limitation.

Figures 4.8 and 4.9 provide the time performance profiles for the instances with strictly less than and with more than 20 families, respectively. Figure 4.8 shows that our approach still outperforms the other ones, even if it is the fastest on a smaller percentage of instances than for the **t2ps** instances. The instances being less structured, the gain in pruning is weaker as compared to the decomposition. However, our method catches up very quickly; for example, it is at most ~ 1.3 and 2 times slower than the best approach for almost 60% and 80% of the instances, respectively. Another interesting point is that $urttf_{ex}$ and $urttf_{lb}$ have very similar time performances, while the values for $\underline{tt}(k)$ were here generally different (not

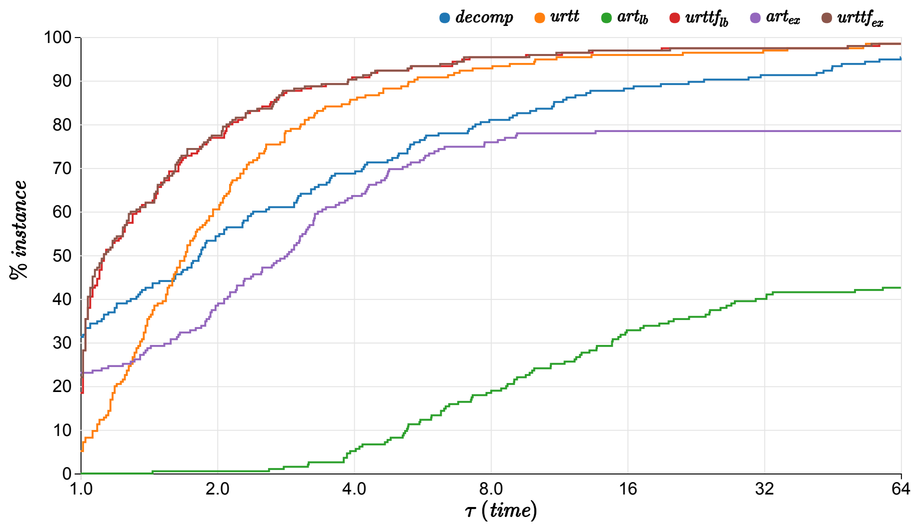


Figure 4.8: Performance profiles on **uttf** instances with strictly less than 20 families for the time metric.

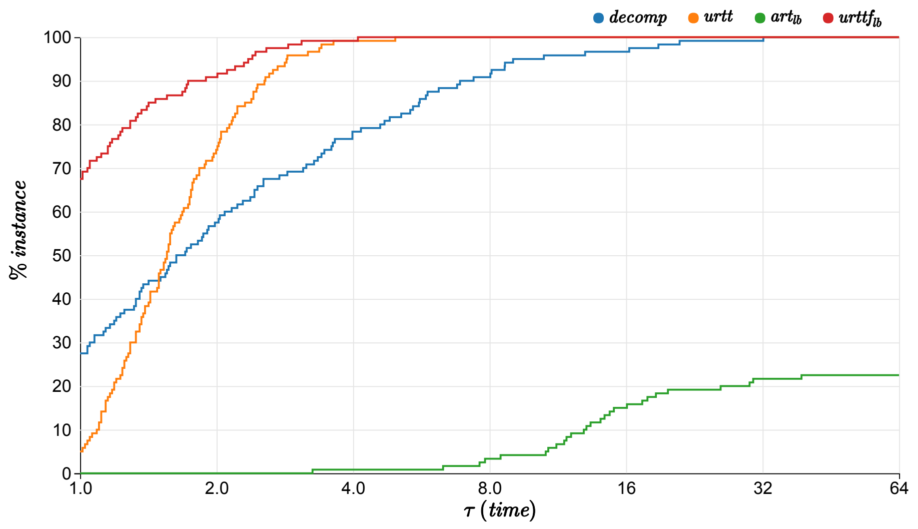


Figure 4.9: Performance profiles on **uttf** instances with more than 20 families for the time metric.

reported here). This means that computing the exact values for $tt(\mathcal{F})$ is not mandatory[5] when used with our propagators, which is profitable since we also target scalability in terms of the number of families.

Regarding the instances with more than 20 families (Figure 4.9), our approach is significantly better than the other ones, as we are the fastest on almost 70% of the

5 Still, if it is available at a low cost, it can be beneficial to use it.

instances and it is at most 4 times slower than the best approach on the remaining instances. This teaches us that when more families are involved, our approach is both efficient and robust.

4.5.2 *Experimentations with Optional Activities*

Optional activities are typically used when modeling problems where activities can be processed on a set of a alternative resources. Hence, in order to experiment with our approach when optional activities are involved, we experimented on JSPSDTT with alternative resources. In particular, we used an approach that consists in duplicating a times the activities and the resources of an original Job Shop problem [FLN00]. For each of the original activities, exactly one of its duplicates must then be executed on its corresponding duplicated machine. This amounts to solving the same problem as the original one, but with the additional liberty of choosing on which one of the a alternative machines an activity will be executed.

Formally, for a given activity i and a duplications, we write i_k the k^{th} duplicate of activity i. To ensure that one and only one of the alternative machines is used by the activity i, we force one and only one of the a duplicates i_k to be used by its corresponding duplicated machine:

$$\exists ! \, k \in [1, a] : v_{i_k}$$

Moreover, the job precedences between activities must be respected by all duplicates, i.e., if there is a precedence between two activities i and j in the original problem, then we must have:

$$\forall k \in [1, a] : c_{i_k} \leq s_{j_k}$$

SEARCH HEURISTIC To our knowledge, few search heuristics are actually devoted to the presence of optional activities. For our evaluation, we used a strategy from Barták that avoids taking decisions about optional activities that will actually not be executed in the final schedule [Bar08]. This is important, as it prevents the search to explore several times the exact same schedule.

The heuristic has two levels: on the first level, it decides whether an activity i is valid or not, i.e., it branches on v_i. On the left branch, it imposes $v_i = \mathit{true}$, and will then branch using the second level, as explained hereafter. On the right branch, $v_i = \mathit{false}$ is posted and the activity i will not be considered deeper in the tree. An other activity $j \neq i$ will then be considered to be branched on using the first level. In the second level, precedences between i and all activities $j : \neg(v_j = \mathbf{false})$ (i.e., still possibly running on the same resource) will be imposed, until no more precedences involving i can be decided. The first level of branching is then used with a different activity j.

To decide which activity should be branched on first, the activity with the smallest est is chosen (ties are broken by smallest duration and ect). Finally, once all decisions have been made, one can assign all activities to their est since the objective is here to minimize the makespan.

SETTINGS We generated 100 instances similar to the five small **t2ps** instances, i.e., with 10 jobs, 5 machines and 5 families. The instances are kept small because duplicating the alternatives already increase substantially the search space. The models we compared are the same ones as before, but the approach from Artigues et al., as they do not deal with optional activities. Our approach uses lower bounds for $\underline{tt}(|\mathcal{F}|)$. We also consider an additional model, called urV, that uses the filtering from Vilím.

We used the Replay evaluation (see chapter 3), where the baseline is the binary decomposition and the generation lasted at most 300 seconds. Finally, we filtered out instances that were solved within less than a second.

RESULTS First, we consider the problem with two alternative resources. The results are given in Figure 4.10. A first observation if that $urttf_{lb}$ is almost always the fastest and it solves all instances in $\tau < 2$, which makes our approach appealing. Interestingly, one can also see that the profiles of the other approaches are in this case quite similar. Finally, for $\sim 10\%$ of the instances, $urttf_{lb}$ provides a speed-up of ~ 32 as compared to the other approaches (see the profiles in $\tau = 32$ in Figure 4.10).

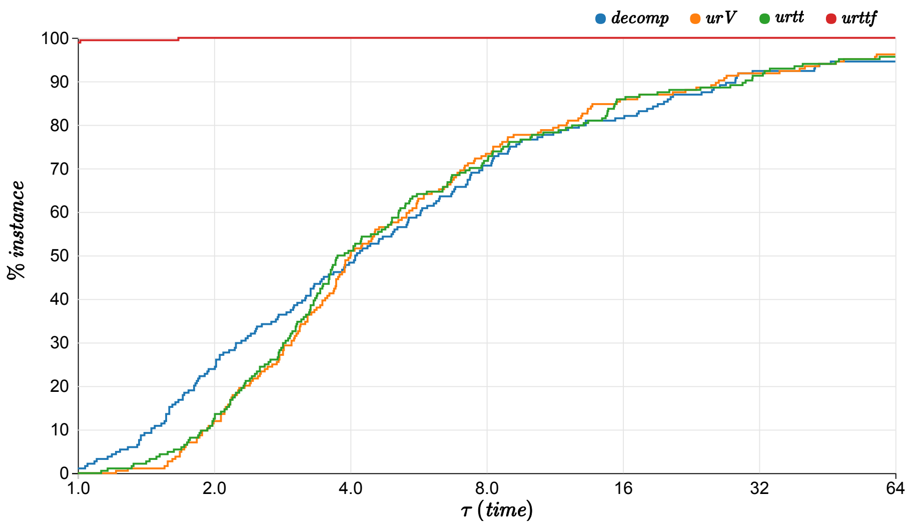


Figure 4.10: Performance profiles on generated instances of the Job Shop problem with two alternative resources.

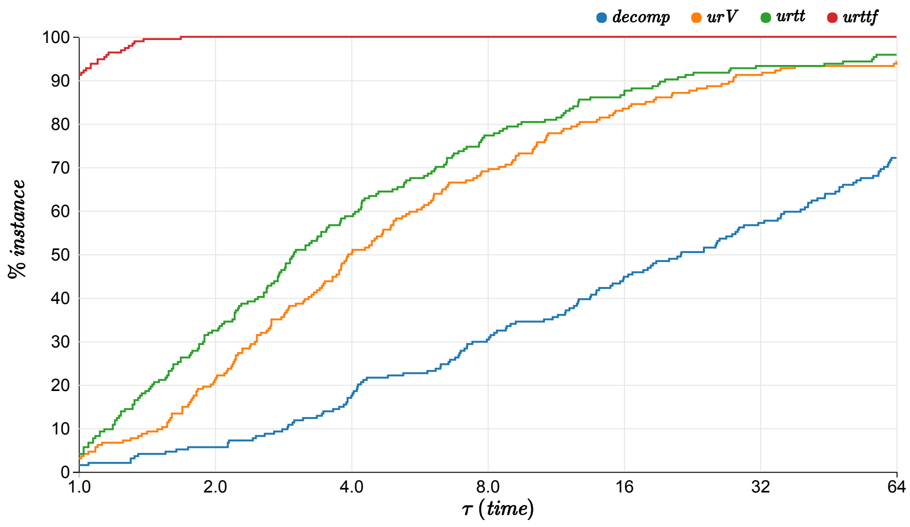


Figure 4.11: Performance profiles on generated instances of the Job Shop problem with three alternative resources.

Let us now consider the results (given in Figure 4.11) when we have three alternative resources. While our approach is still clearly the best one for similar reasons, one can now better separate *decomp*, *urV*, and *urtt*: *urV* is better than *decomp* and *urtt* is better than *urV*. Still, *urtt* and *urV* are close to each other, and tends to converge. This shows again the benefits of reasoning with families of activities.

5

RESOURCE-COST ALLDIFFERENT CONSTRAINT

In this chapter, we consider a family of optimization problems where a set of items, each requiring a possibly different amount of resource, must be assigned to different slots for which the price of the resource can vary. In particular, we first consider the domain of production scheduling in the presence of fluctuating renewable energy costs. Indeed, some countries tend to decrease the use of fossil and nuclear energies and to replace them with renewable energies [WB06]. For instance, in Europe, Germany has started a nuclear phase-out that should be completed by 2022. A consequence of the adoption of renewable energy is the larger fluctuation of energy prices. At the same time, the Electricity Price Forecast (EPF) at the daily basis becomes more and more accurate [Wer14]. This high volatility of the prices combined with EPF tools opens new perspectives and challenges for production scheduling optimization. A producer can get a competitive advantage if he is able to schedule his most energy consuming processes when the prices are low. The production planning should therefore leverage as much as possible the flexibility in the production process to translate it into energy cost savings.

In the setting we consider, the energy cost is assumed to be known at each future time slot (provided by an EPF module). The goal is then to schedule each item over the time slots such that the overall energy bill is minimized. For each item, its contribution to the cost is the energy required to produce it multiplied by the energy cost forecast at the time slot it is produced.

Example 1. *Let us illustrate this situation with the example given in Figure 5.1. 3 items a, b and c have to be produced and their consumption are $C(a) = 2$, $C(b) = 4$ and $C(c) = 3$ (see left plot). There are nine time slots where these items can be produced, each one being associated with an energy price $P(s)$. For instance, the slot 3 has a price of $P(3) = 15$. In the given schedule, a, b and c are* *produced at slot 1, 3 and 5, respectively. Hence, the total cost is* $T = 2 \cdot 20 + 4 \cdot 15 + 3 \cdot 15 = 145$*.*

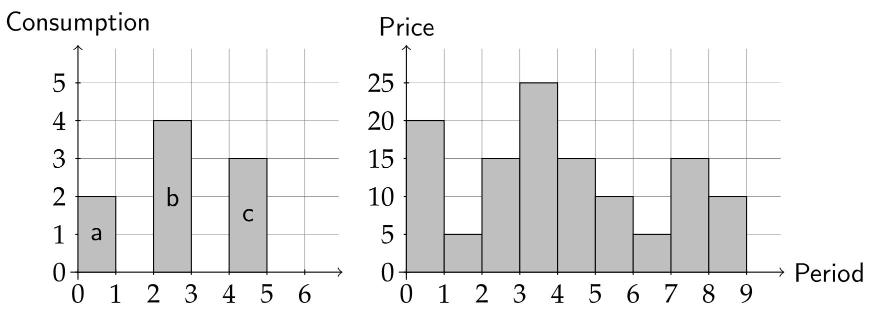


Figure 5.1: Energy Consumption of items (left) and energy prices (right).

For a given set of items $\mathcal{I}$, we can formally define the total cost as:

$$T = \sum_{i=1}^{|\mathcal{I}|} C(\mathcal{I}_i) \cdot P(s(\mathcal{I}_i)) \tag{5.1}$$

where $s(\mathcal{I}_i)$ is the time slot assigned to the item $\mathcal{I}_i$.

In CP, one can model this cost in two ways:

1. with a sum of ELEMENT [VHC88] constraints.
2. with a MINIMUMASSIGNMENT constraint that has a filtering based on reduced costs (referred to ILCALLDIFFCOST in [FLM99] and MINWEIGHTALLDIFF in [Sel02]). In this case, the cost of an edge linking a variable (i.e., an item) to a value (i.e., a time slot) is the product of the item consumption with the energy price of the time slot.

Unfortunately, both approaches have limitations. Modeling with a sum of ELEMENT constraints does not take into account the fact that the items must all be assigned to different time slots. For instance, in Figure 5.1, if all items could be assigned to any time slot, this approach would consider that all items can be assigned to the slots 2 or 7. The computed minimum cost would therefore be $5 \cdot (2+4+3) = 45$, which is clearly impossible since only 2 items can be produced at an energy price of 5.

The MINIMUMASSIGNMENT incorporates the all-different constraint, but the algorithm has quite a high time complexity of $\mathcal{O}(n^3)$, where n is the number of items[1].

CONTRIBUTION This chapter proposes a third approach by introducing the RESOURCECOSTALLDIFFERENT constraint and an associated scalable filtering algorithm. In the particular case where the total cost is computed with Equation 5.1,

1 Although in practice the computation in a search tree is incremental.

the constraint fills a gap between using a sum of individual ELEMENT constraints and solving the general matching problem with cost computed by MINIMUMASSIGNMENT. Its goal is to compute the total cost in a scalable and incremental manner, while considering the fact that all assignments must be different. Furthermore, efficient domain filtering can be performed in $\mathcal{O}(n \cdot m)$, where n is the number of unbound variables and m is the maximum domain size of unbound variables. While this is a better time complexity than $\mathcal{O}(n^3)$ required by MINIMUMASSIGNMENT, it comes at the price of a weaker inference. However, this trade-off pays off in practice for the two problems we considered, namely the *Continuous Casting Steel Production with Electricity Bill Minimization* and the *Product Matrix Travelling Salesman Problem*. The latter problem illustrates that the RESOURCECOSTALLDIFFERENT constraint can be used in other domains than production scheduling.

The results on the first problem demonstrate that MINIMUMASSIGNMENT is always slower than our approach. In addition, RESOURCECOSTALLDIFFERENT is often faster than the decomposition, sometimes with an important speed-up. This is especially true for large instances, for which an order of magnitude is gained in $\sim$ 20% of the cases, while $\sim$ 75% of them are solved faster. Moreover, our algorithm is robust, in the sense that when it is slower, it is by a small factor (3.2 at the very most). Results also illustrate that for a non-negligible number of the large instances, we can get an important gain in terms of number of backtracks as compared with a decomposition: a reduction of at least one order of magnitude is obtained for 30% of the instances.

For the second problem, the results show that our approach is the fastest in CP. Moreover, it outperforms the Concorde solver[App+06, App+11], a custom state-of-the-art Branch-and-Cut Mixed Integer Programming solver, for 90% of the instances, sometimes by an important factor (some instances could not be solved in a factor 32 as compared with our constraint).

RELATED WORK The need to compute the total cost related to some assignments in CP is not rare. It is for instance used to solve the Travelling Salesman Problem and the Travelling Salesman Problem with Time Windows. Focacci et al. proposed a global optimization constraint [FLM99, Foc+99] (they call it ILCALLDIFFCOST) to compute a lower bound on the total cost when all assignments must be different and to filter the domains based on reduced costs. However, they use a linear formulation to compute the reduced costs, and therefore do not obtain arc-consistency. In [Sel02], Sellmann calls this constraint MINWEIGHTALLDIFF and proposes an arc-consistent filtering algorithm in which exact reduced costs are used. More recently [DCP16], Ducomman et al. came up with a different computation of the exact reduced costs, by making use of an All-pairs Shortest path algorithm. In this chapter, we denote by MINIMUMASSIGNMENT these equivalent constraints and use the algorithm from [Foc+99] in particular for our experiments.[2] Finally, Régin introduced the more general GCCCOST global constraint [Rég02]

2 We also experimented with the version from [DCP16] to get exact reduced costs, but it appeared to be very slow for the instances we considered, so we do not report any results regarding that implementation.

for which several variables can be assigned to the same value as long as cardinality constraints are respected.

For all those constraints, there are no assumptions on the costs associated with the different variable-assignment pairs. The ResourceCostAllDifferent constraint we introduce is actually a particular case of a MinimumAssignment constraint for which the cost of assigning a variable X_i to a value s amounts to the product of a given *consumption* of X_i with the fixed *price* of assigning the value s to a variable. This particularity allows getting a more efficient filtering, as we shall see in the results.

For Scheduling problems, global constraints that consider (electricity) costs have been defined recently [SH10, SH11]. The particular case of the disjunctivecost constraint was then further studied when activities have variable durations [WB16]. Finally, a recent application of CP was successfully used to optimize a tissue manufacturing planning problem from an energy viewpoint [Dej+16].

CHAPTER OUTLINE We first formally define the constraint. We then describe an algorithm that checks feasibility, based on the computation of a lower bound for the total cost. Next, we present our filtering algorithm to prune unfeasible values of all variables. Finally, we evaluate our work on two problems: the *Continuous Casting Steel Production with Electricity Bill Minimization* that is an industrial use case, and the *Product Matrix Travelling Salesman Problem* that is more academic.

5.1 CONSTRAINT DEFINITION

Definition 12. *Let*

- *X be a sequence of integer variables that are the production time slots for each item to produce,*
- *C be a sequence of integer constants of length $|X|$ that are the amount of resource required to produce each item,*
- *H be an integer constant, that is the number of time slots (horizon),*
- *P be a sequence of H integer constants giving the price of the resource at each time slot, and*
- *T be an integer variable that is the total resource cost of the schedule,*

the constraint ResourceCostAllDifferent*(X, C, P, T) ensures that*

$$\wedge \begin{cases} \text{allDifferent}(X) \\ T = \sum_{i=1}^{|X|} C(X_i) \cdot P(X_i) \end{cases} \tag{5.2}$$

with $D(X_i) \subseteq [0..H[, \forall X_i \in X$.

Intuitively, all variables X_i must be assigned to a different value. Moreover, they all have a fixed consumption given by $C(X_i)$. Any variable X_i assigned to a value

s then implies a resource price of $P(s)$ multiplied by the consumption $C(X_i)$. The total cost T amounts to the sum, over all the variables, of the product of their consumption with the price of their assignment.

5.1.1 *Lower Bound Computation and Feasibility Check*

The constraint can be separated in two parts: all assignments must be different, and the assignments must respect the total cost constraint. For the first part, we rely on well-known propagators (see [Hoe01, Rég94]) devised for the allDifferent constraint. For the second part, we propose to compute a lower bound for the minimum total cost T, written T_{lb}. In the following, for a given sequence of variables X and a sequence of assignments A, we denote the total production cost of mapping the i^{th} element of X to the i^{th} element of A, by $Prod_{cost}(X, A) = \sum_{i=1}^{\min(|X|,|A|)} C(X_i) \cdot P(A_i)$.

LOWER-BOUND FOR THE COST There is an inherent matching problem involved in the filtering of allDifferent, as the domains of the variables can be different. To compute our lower bound efficiently, we relax the domain of the variables by assuming that all variables can be assigned to any value of any current domain, that is, any $s \in \bigcup_{X_i \in X} D(X_i)$. But we do not relax the all-different constraint. The matching problem can then be solved greedily on the relaxed problem.

Example 2. *Let us reconsider the example of Figure 5.1, and let us assume there are 6 more items d to i to be scheduled for production, with a consumption of $C(d) = 2$, $C(e) = 3$, $C(f) = 4$, $C(g) = 3$, $C(h) = 5$ and $C(i) = 6$. Moreover, $D(d) = D(f) = D(g) = \{2, 4, 6, 7, 8, 9\}$, $D(e) = \{2, 4, 8\}$ and $D(h) = D(i) = \{2, 7\}$. To compute our lower bound, we first compute the exact cost $C_{assigned}$ that has to be paid due to already assigned items. In our example a, b and c are assigned and $C_{assigned} = 145$. Secondly, one must compute the cost due to the set UV of unbound variables, $C_{unbound}$. We ease the matching of those items by assuming that they can be assigned to any slot $s \in \bigcup_{X_i \in UV} D(X_i) = \{2, 4, 6, 7, 8, 9\}$. To compute $C_{unbound}$, we map the item with the largest consumption with the lowest resource price slot, so i is matched with slot 2. We then proceed similarly for all remaining unbound items: h, f, e, g and d are respectively mapped with slot 7, 6, 9, 8 and 4. This computation can be done by sorting unbound variables and time slots by non-increasing order of consumption and non-decreasing order of price, respectively. In the end, $C_{unbound} = 6 \cdot 5 + 5 \cdot 5 + 4 \cdot 10 + 3 \cdot 10 + 3 \cdot 15 + 2 \cdot 25 = 220$. This is a lower bound since item e is matched with slot 9 so as to minimize the total cost, but this value is actually not in its domain. A lower bound for the total cost is then $T_{lb} = 145 + 220 = 365$.*

In the general case, T_{lb} can be computed as follows:

1. Compute the exact cost $C_{assigned}$ due to the set of assigned variables $X_{assigned}$.
2. Sort the unbound variables $UV = X \backslash X_{assigned}$ by non-increasing order of C, UV^{sorted}.

3. Compute the set of unmapped values that are still part of the domains of the unbound variables, i.e., $UA = \bigcup_{X_i \in UV} D(X_i)$.

4. Sort UA by non-decreasing order of P, UA^{sorted}.

5. Compute the minimal cost mapping of variables of UV^{sorted} to the values of UA^{sorted}, i.e., $C_{unbound} = Prod_{cost}(UV^{sorted}, UA^{sorted})$. Notice that $C_{unbound}$ is a lower bound for the cost due to the unbound variables, as a variable might be mapped to a value that is not in its domain.

6. The lower bound is then $T_{lb} = C_{assigned} + C_{unbound}$.

As described in Section 5.2, one can achieve this computation efficiently. First, the sorts of variable and value sequences can be done once and for all, as they remain correct for the whole search process. Incremental and reversible data structures allow keeping the use of those sorts at any time. Moreover, the cost $C_{assigned}$ due to bound variables and the set $\bigcup_{X_i \in UV} D(X_i)$ can be computed incrementally. Once we have computed T_{lb}, a first constraint inference is the feasibility check:

$$T_{lb} > \overline{T} \implies \textbf{Fail} \tag{FC}$$

5.1.2 *Domain Filtering*

In order to filter a value s of the domain of a variable X_i, one needs to compute the reduced cost (value of T_{lb} *if* $X_i = s$), written $T_{lb}^{X_i=s}$. One can then use the inference rule:

$$\forall X_i \in X \, \forall s \in D(X_i) : T_{lb}^{X_i=s} > \overline{T} \implies X_i \neq s \tag{DF}$$

To compute $T_{lb}^{X_i=s}$, one has to compute the value $C_{unbound}$ under the constraint $X_i = s$, i.e.:

$$C_{unbound}^{X_i=s} = C(X_i) \cdot P(s) + Prod_{cost}(UV^{sorted} \backslash \{X_i\}, UA^{sorted} \backslash \{s\}) \tag{5.3}$$

$T_{lb}^{X_i=s}$ is then defined as $C_{assigned} + C_{unbound}^{X_i=s}$.

We are interested in computing Equation 5.3 for all variables $X_i \in UV$ and all values $s \in D(X_i)$. An inefficient way would be to recompute the second term for each pair (X_i, s). However, one can notice that when a variable is assigned to a given value of its domain, to compute the optimal mapping for the other variables by means of UV^{sorted} and UA^{sorted}, some variables are mapped to the same assignment as in the original optimal assignment (see Figure 5.2). Moreover, the variables that must be mapped to other assignments must all be mapped to the predecessor/successor of the value they were mapped with in the original matching (see Figure 5.2).

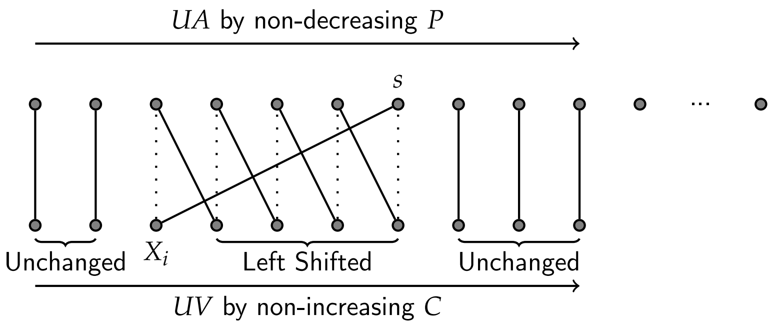


Figure 5.2: Optimal mapping of variables if the variable $X_i = s$.

This observation allows us to compute the second term of Equation 5.3 in $\mathcal{O}(1)$, provided we have precomputed the 3 arrays CS, CS^{left} and CS^{right}, for which the i^{th} element is defined as:

$$\begin{aligned} CS_i &= Prod_{cost}(UV^{sorted}_{1..i}, UA^{sorted}) \\ CS^{left}_i &= Prod_{cost}(UV^{sorted}_{2..i}, UA^{sorted}) \\ CS^{right}_i &= Prod_{cost}(UV^{sorted}_{1..i}, UA^{sorted}_{2..|UA|}) \end{aligned}$$

Let us call X_i^{pos} and s^{pos} the positions of X_i in UV^{sorted} and s in UA^{sorted}, respectively. Let us assume $s^{pos} > X_i^{pos}$ as in Figure 5.2 (there is symmetric reasoning for the case $s^{pos} < X_i^{pos}$, and the case $s^{pos} = X_i^{pos}$ is already managed by the feasibility checker). Then, computing the optimal mapping can simply be done with:

$$\underbrace{CS_{X_i^{pos}-1}}_{Unchanged} + \overbrace{C(X_i) \cdot P(s)}^{Assignment} + \underbrace{CS^{left}_{s^{pos}-1} - CS^{left}_{X_i^{pos}-1}}_{LeftShifted} + \overbrace{CS_{|UV|} - CS_{s^{pos}}}^{Unchanged}$$

Example 3. *Let us reconsider values from Example 2 and let us assume* $\overline{T} = 380$. *Prior to filtering, we can precompute* $CS = (30, 55, 95, 125, 170, 220)$, $CS^{left} = (25, 45, 75, 105, 135)$ *and* $CS^{right} = (30, 80, 120, 165, 240)$. *We now wish to compute* $C^{f=4}_{unbound}$. *Since* $f^{pos} = 3$ *and* $4^{pos} = 6$, *we have, in* $\mathcal{O}(1)$, $C^{f=4}_{unbound} = CS_2 + C(f) \cdot P(4) + CS^{left}_5 - CS^{left}_2 + CS_6 - CS_6 = 55 + 100 + 90 + 0 = 245$. *If we add* $C_{assigned}$, *we finally have a cost of* $390 > \overline{T}$ *and therefore* $4 \notin D(f)$.

TIME COMPLEXITY Computing CS, CS^{left} and CS^{right} is $\mathcal{O}(n)$ in time, where $n = |UV|$. Then one must compute $C^{X_i=s}_{unbound}$ for each pair $(X_i, s) : X_i \in UV, s \in D(X_i)$ and perform a feasibility check in $\mathcal{O}(1)$. The time complexity for checking all pairs variable-value is therefore $\mathcal{O}(n \cdot m)$, where $m = \max_{X_i \in UV} |D(X_i)|$.

5.2 ALGORITHMS

The implementation relies on several incremental and reversible data structures. First, we use reversible doubly-linked lists to maintain the sequences UV and UA, ordered by non-increasing C and non-decreasing P, respectively. They are maintained during search and their state is retrieved upon backtracking thanks to trailing (see [War83, AK99]). Second, we use an array of reversible sparse sets [SM+13], $X_{withValue}$, keeping track of the variables whose domain contain a given value s. This is useful to maintain UA efficiently. We also make use of *delta* sets. For a variable X_i, the set Δ_{X_i} contains all the values that were removed from the domain of X_i since the last call to the algorithm. This is done in an efficient manner, described in [SM+13]. For instance, retrieving the set Δ_{X_i} or its size is $\mathcal{O}(1)$. Finally, we maintain the total cost due to assigned variables $C_{assigned}$ as a reversible integer.

FEASIBILITY CHECKER Algorithm 5.2.1 allows computing T_{lb} and to perform the feasibility check. Lines 1-11 update the state by means of the freshly bound variables (i.e., not bound in the previous call): the variable is removed from the sequence of unbound variables UV, its exact contribution is added to $C_{assigned}$, and its assignment is removed from the possible assignments UA. Moreover, for each value removed from its domain, the variable is removed from the set of variables that could be assigned to this value. If this set gets empty, no variable can be assigned to the value and this time slot is therefore removed from the sequence of possible assignments. A first check is done with $C_{assigned}$. Lines 15-22 update the set of remaining available assignments, i.e., they remove from UA the values that are no more contained in any domain. To do so, the sequence UV is traversed and for each variable, if its delta set is not empty, we update the corresponding sparse sets of values present in the delta set. Line 23 computes a lower bound for the cost due to unbound variables, by traversing UV and UA. The feasibility check is done in lines 24-26 and the total cost is lower bounded in line 27.

DOMAIN FILTERING Algorithm 5.2.2 achieves domain filtering. We assume CS, CS^{left} and CS^{right} are computed according to current state. It can be done by simply traversing the UV and UA sequences. Line 1 computes $P_{assignment}$ that maps unbound assignments to their position in UA. The loops in lines 2-18 apply the domain filtering rule for each pair $(X_i \in UV, s \in D(X_i))$. Depending on the relative positions of X_i and s, $C^{X_i=s}_{unbound}$ is computed. At line 10, a feasibility check for this particular assignment is performed. If it is unfeasible, we remove the time slot of the domain of the variable and update the corresponding sparse set (and possibly UA if it is emptied).

5.3 CONTINUOUS CASTING STEEL PRODUCTION WITH ELECTRICITY BILL MINIMIZATION

To evaluate the RESOURCECOSTALLDIFFERENT (RCAD) constraint in practice, we first considered a real-life industrial problem, the Continuous Casting Steel Pro-

Algorithm 5.2.1 : Incremental Computation of T_{lb}. The algorithm also serves as a feasibility checker with inference rule FC.

```
forall $X_i \in UV : |D(X_i)| = 1$ do
    $UV \leftarrow UV \setminus \{X_i\}$
    $C_{assigned} \leftarrow C_{assigned} + C(X_i) \cdot P(X_i)$
    $UA \leftarrow UA \setminus \{X_i.value()\}$
    forall $s \in \Delta_{X_i}$ do
        $X_{withValue}(s) \leftarrow X_{withValue}(s) \setminus \{X_i\}$
        if $X_{withValue}(s) = \emptyset$ then
            $UA \leftarrow UA \setminus \{s\}$
        end
    end
end
if $C_{assigned} > \overline{T}$ then
    return Fail
end
forall $X_i \in UV$ do
    forall $s \in \Delta_{X_i}$ do
        $X_{withValue}(s) \leftarrow X_{withValue}(s) \setminus \{X_i\}$
        if $X_{withValue}(s) = \emptyset$ then
            $UA \leftarrow UA \setminus \{s\}$
        end
    end
end
$C_{unbound} \leftarrow Prod_{cost}(UV, UA)$
if $C_{assigned} + C_{unbound} > \overline{T}$ then
    return Fail
end
post($T \geq C_{assigned} + C_{unbound}$)
```

duction with Electricity Bill Minimization (CCSPEBM). We compared the performances of our filtering algorithm with those of existing approaches. All experiments were performed with the *OscaR* solver [Osc12], AMD Opteron processors (2.7 GHz), the Java Runtime Environment 8, and a memory consumption limit of 4 GB.

We first experimented with small-scale instances: RCAD provides generally additional pruning as compared with a decomposition in a sum of ELEMENT constraints, sometimes by one order of magnitude. This is reflected from a time perspective: one order of magnitude can be gained and RCAD is at the very most 3 times slower. MINIMUMASSIGNMENT is always slower than RCAD and almost only brings overhead if they are used together. Moreover, since the exact reduced costs are not

Algorithm 5.2.2 : Filter the domains by means of the inference rule DF.

```
$P_{assignment} \leftarrow map_{a \rightarrow p}$ /* Map from assignments to position in $UA$ */
forall $X_i \in UV$ do
    forall $s \in D(X_i)$ do
        $s^{pos} \leftarrow P_{assignment}(s)$
        if $X_i^{pos} < s^{pos}$ then
            $C^{X_i=s}_{unbound} \leftarrow CS_{X_i^{pos}-1} + C(X_i) \cdot P(s) + CS^{left}_{s^{pos}-1} - CS^{left}_{X_i^{pos}-1} + CS_{|UV|} - CS_{s^{pos}}$
        else if $X_i^{pos} > s^{pos}$ then
            $C^{X_i=s}_{unbound} \leftarrow CS_{s^{pos}-1} + C(X_i) \cdot P(s) + CS^{right}_{X_i^{pos}-1} - CS^{right}_{s^{pos}-1} + CS_{|UV|} - CS_{X_i^{pos}}$
        end
        if $C^{X_i=s}_{unbound} + C_{assigned} > \overline{T}$ then
            $D(X_i) \leftarrow D(X_i) \backslash s$
            $X_{withValue}(s) \leftarrow X_{withValue}(s) \backslash \{X_i\}$
            if $X_{withValue}(s) = \emptyset$ then
                $UA \leftarrow UA \backslash \{s\}$
            end
        end
    end
end
```

computed in MinimumAssignment as it is too expensive[3], it appears RCAD prunes generally more than MinimumAssignment.

To challenge the scalability of our approach, we also considered larger instances. We discovered that RCAD is faster than the decomposition in a sum of element constraints for $\sim 75\%$ of the instances and that at least one order of magnitude is gained for $\sim 20\%$ of the instances. From a number of backtracks point of view, an order of magnitude is gained for $\sim 30\%$ of the instances. Finally, RCAD is slower by a maximum factor of 3.2, showing our approach is not only efficient but also robust.

Before providing the different results in detail, this section gives the definition of the problem, a CP model to solve it, and describes the comparison methodology we used.

5.3.1 *Problem Definition*

A *Continuous Casting Steel Production Problem* is a scheduling problem, for which a CP approach has been proposed in [GSDS14]. The main difference in our problem is that steel is produced by melting scrap in an Electric Arc Furnace (EAF) instead of a blast furnace. The total electricity cost minimization becomes the objective of the problem.

The Continuous Casting Steel Production with Electricity Bill Minimization (CCSPEBM) consists in scheduling a set of programs $\mathcal{P}$. A program p is made of a sequence $\mathcal{B}_p$ of batches. Each batch must be processed by a given sequence of machines: the EAF, the Ladle Furnace (LF), the Argon Oxygen Decarburization (AOD), and finally the Caster (CAS). For a given program, the AOD might actually not be used (this is known a priori). A machine can process at most one batch at any time. An illustration of a CCSPEBM schedule is given in Figure 5.3.

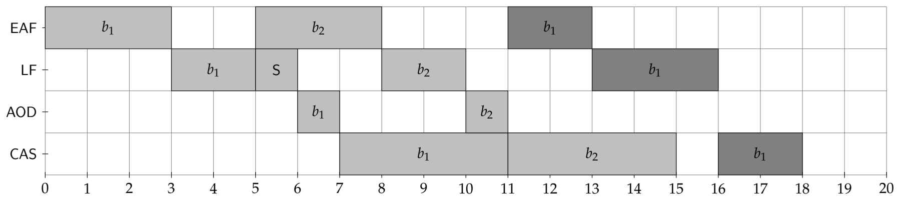


Figure 5.3: Example of a CCSPEBM schedule with two programs, respectively with two batches and one batch. Activities of the first/second program are in light/dark gray. Only the first program makes use of the AOD and its first batch is stocked by the LF during one time slot (represented by the S square).

In the process, a batch is first melt using the EAF. Because the steel must be kept at a high temperature, the subsequent machines must process the batch as soon as the previous machine has finished. In addition, a batch cannot stay too long in the whole process, that is, there is a maximum span between the EAF and the

3 According to preliminary experiments we conducted. We do not report any results when exact reduced costs are used.

CAS. The CAS must process all batches of a given program without interruption (see for instance the first program in Figure 5.3). The processing time of a batch by a machine is known a priori and cannot be modified. However, the LF/AOD might stock a batch during a small amount of time (see Figure 5.3).

The two electric machines requiring a significant amount of electrical energy are the EAF and the LF. A *consumption profile* is associated with the processing of a batch by a machine, that is, a function that provides the (non-null) consumption of each time period of the processing time. The profiles follow bounded ramp functions: the first slot requires a given amount and then stays constant with another higher consumption for the remaining duration. The profile can be different for each batch when processed by the EAF but it is the same for all batches processed by the LF. It is assumed that the LF does not consume energy while it stocks a batch since the consumption is negligible, i.e., the consumption required to stock a batch is 0. Finally, the electricity prices can vary at each time slot of the whole horizon.

5.3.2 *CP Model*

A standard CP scheduling approach is used for solving this problem. The processing of a batch by a machine is represented by an activity. Each activity is modeled with three integer variables used to represent its start, duration and end: s, d and e, such that $s + d = e$. For a machine m, s_m denotes the array of starting times of all activities executed on m, and s_m^b the starting time of the processing of the batch b by machine m (similar notations are used for duration and ending time). We allow d_{lf} and d_{aod} to vary since the LF/AOD may stock a batch for a moment (see the S square in Figure 5.3). For every batch b, the different activities must be scheduled continuously:

$$\forall p \in \mathcal{P}, \forall b \in \mathcal{B}_p : e_{eaf}^b = s_{lf}^b \wedge e_{lf}^b = s_{aod}^b \wedge e_{aod}^b = s_{cas}^b$$

All batches of a given program must be processed in sequence by the caster and without any interruption:

$$\forall p \in \mathcal{P}, \forall i \in 1..|\mathcal{B}_p| - 1 : e_{cas}^{b_i} = s_{cas}^{b_{i+1}}$$

A machine can only process one batch at a time. This is modeled with unary resource constraints [Vil04]:

$$\text{UNARY}(s_{eaf}, d_{eaf}, e_{eaf}) \wedge \text{UNARY}(s_{lf}, d_{lf}, e_{lf}) \wedge$$
$$\text{UNARY}(s_{aod}, d_{aod}, e_{aod}) \wedge \text{UNARY}(s_{cas}, d_{cas}, e_{cas})$$

Finally, a batch can not stay too long in the whole process:

$$\forall b \in \mathcal{B} : s_{cas}^b - e_{eaf}^b \leq max_{span}$$

where max_{span} is the maximum duration a batch can stay after the EAF and before the CAS.

ELECTRICITY BILL MINIMIZATION To model the objective, we split a given activity on the EAF/LF that must be processed during d time slots into d consecutive chunks with a duration of 1 time slot. Each of these chunks is then an item to be produced and has a consumption given by the batch consumption profile on the machine. The objective is modeled with Equation 5.1 and is to be minimized. We compare the following modelings of Equation 5.1:

- *SumElements*, using a sum of ELEMENT constraints. In this case, we do not split activities into chunks. We precompute the total cost of starting an activity at a given time and use this cost in the ELEMENT constraint.
- *SumElements* $\cup$ *MinAssignment*, using a MINIMUMASSIGNMENT constraint [Foc+99] additionally for each consuming machine. The cost of assigning a chunk at a time slot s is simply its consumption in the profile of the activity multiplied by the price at slot s. We do not use the exact reduced costs [DCP16] that appeared to require a prohibitive computation time for the size of the considered instances.
- *SumElements* $\cup$ *RCAD*, using our constraint additionally, since *RCAD* and *SumElements* actually do not subsume each other. *RCAD* alone can miss some filtering as compared to *SumElements* because of the union of domains relaxation in our procedure.
- *SumElements* $\cup$ *RCAD* $\cup$ *MinAssignment*, using all constraints altogether.

5.3.3 *Comparison Methodology*

REPLAY EVALUATION To compare the different models, we use the *Replay Evaluation* framework described in Chapter 3. The *baseline* model is *SumElements*. This model with then be extended with the different propagators considered in this chapter. To generate the CBTs, we use a simple first-fail search heuristic, the focus of this evaluation being on propagation: we first branch on the variable with the smallest domain. For the value heuristic, prior to search and for each consuming activity, we order the time slots by non-decreasing order of total processing cost of starting the activity at the time slot. Let s be the available slot minimizing cost of batch b on machine m, we then branch in a ternary fashion in this order: $s_m^b = s$, $s_m^b < s$ and $s_m^b > s$. For non-consuming activities, we assign the activity to its minimum starting time on the left branch, and remove this assignment on the right branch.

INSTANCE GENERATION This work being part of an industrial project, we discussed with consultants of the *N-side* company[4] in order to generate realistic instances, and we use real historical electricity prices on the EU market. Our instances have between 2 and 15 batches per program, and their duration is between 3 and 4 slots at the EAF, exactly 4 slots at the AOD/LF, and between 4 and 5

4 http://www.n-side.com/

slots at the CAS. 80% of the programs do use an AOD and possible consumptions vary between 1 and 100. We ensure that there are enough programs so that the caster is used between 60% and 80% of the horizon. Finally, max_{span} amounts to 150% of the sum of processing times of a batch by the LF and the AOD.

5.3.4 *Small Instances*

We first consider 158 small instances (96 time slots). Search trees were generated with a time limit of 30 s. and a backtrack limit of 500000. The results are given in Figures 5.4, 5.5, 5.6.

In Figure 5.4, one can see that $SumElements \cup RCAD$ is sometimes faster than $SumElements$, but not always. However, the factor is generally not large when it is slower (at most ~ 3). The number of backtracks is almost always reduced, sometimes significantly. For only a few instances there is no additional pruning, which explains the slowest execution times due to the overhead induced by our constraint. We perform an extended comparison of those approaches for larger instances in the next section.

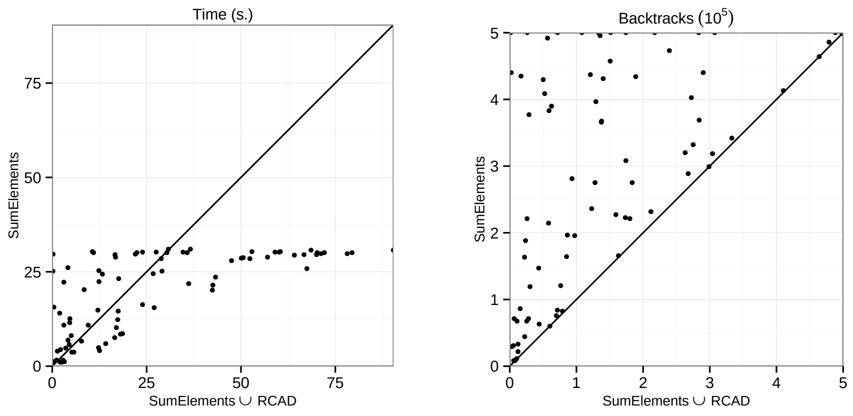


Figure 5.4: Comparison of $SumElements$ and $SumElements \cup RCAD$ on small instances.

Comparisons of $SumElements \cup MinAssignment$ and $SumElements \cup RCAD$ are reported in Figure 5.5.

An important observation is that $SumElements \cup MinAssignment$ is always slower than $SumElements \cup RCAD$. Interestingly, one can see that $SumElements \cup RCAD$ prunes generally more than $SumElements \cup MinAssignment$: the reason is that $RCAD$ is not subsumed by $MinAssignment$ since we use the version from Focacci et al. [FLM99] that uses the linear programming reduced costs and not the exact ones. It would be subsumed using the exact reduced costs as in [DCP16] but unfortunately computing them was too costly according to first preliminary experiments we conducted.

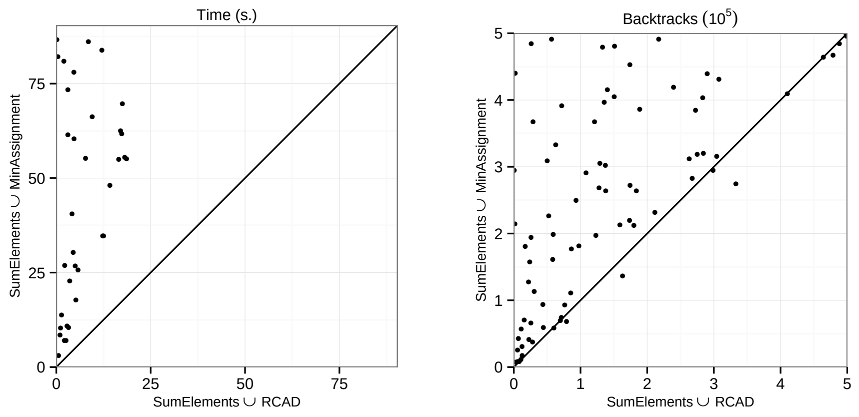


Figure 5.5: Comparison of *SumElements* ∪ *MinAssignment* and *SumElements* ∪ *RCAD* on small instances.

We therefore make a last comparison, between *SumElements* ∪ *MinAssignment* ∪*RCAD* and *SumElements* ∪ *RCAD* (see Figure 5.6). One can observe that MINIMUMASSIGNMENT often only brings overhead: the data points are generally close to the equality line for the number of backtracks, while the search times are slower. In conclusion, MINIMUMASSIGNMENT does not scale well and we will not use it for our comparison on larger instances.

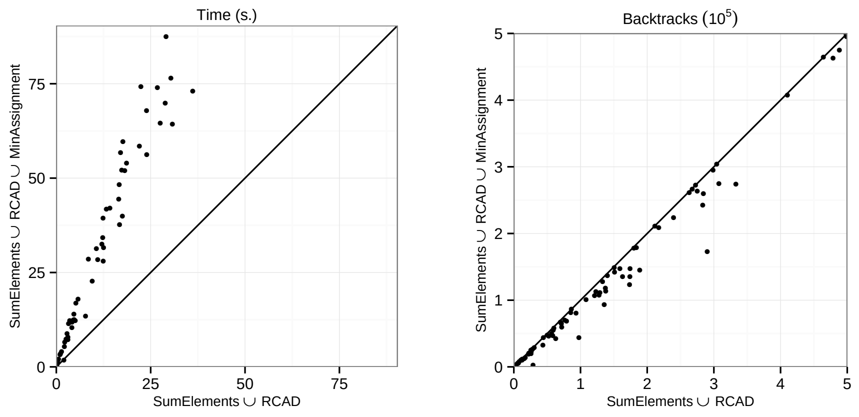


Figure 5.6: Comparison of *SumElements* ∪ *MinAssignment* ∪ *RCAD* and *SumElements* ∪ *RCAD* on small instances.

5.3.5 *Large Instances*

To challenge the scalability of our approach, we generated 227 larger instances with 480 time slots. We generated search trees using a time limit of 300 s., in order to grasp more information about the performances of both approaches during search. Results are given in Figure 5.7. As for small instances, one can see that there is often an important gain in terms of the number of backtracks. There is no additional pruning only for a few instances. From an execution time perspective, one can observe the replays for *SumElements* all takes around 300 s., which is expected since it is the model used to generate CBTs. Moreover, *SumElements* ∪ *RCAD* is faster for most of the instances.

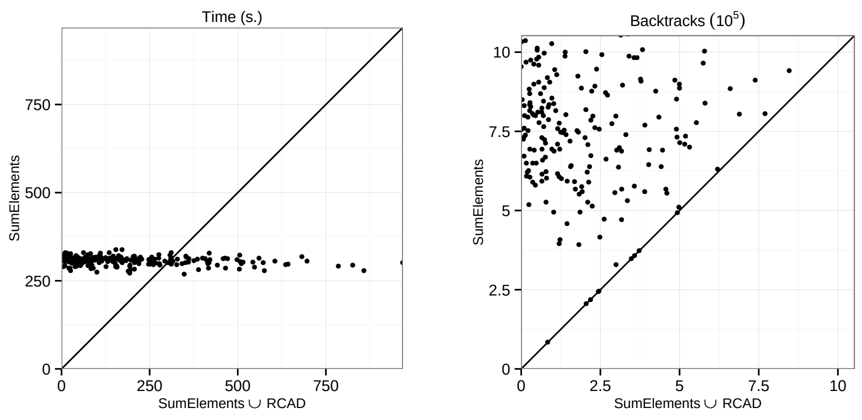


Figure 5.7: Comparison of *SumElements* and *SumElements* ∪ *RCAD* on large instances.

In order to quantify and better visualize the gain obtained by our approach, we constructed performance profiles following the definition given in Section 3.3.1 (i.e., only *SumElements* is part of the baseline set $\mathcal{B}$). Figures 5.8 and 5.9 respectively provide the profiles for the number of backtracks and the time metrics. Let us first read the profiles for the number backtracks: $F_{SumElements \cup RCAD}(1) \simeq 95\%$, so we achieve more pruning for $\sim 95\%$ of the instances. Furthermore, one can see that $F_{SumElements \cup RCAD}(0.1) \simeq 30\%$, which means that we gain at least one order of magnitude on the number of backtracks for $\sim 30\%$ of the instances.

From an execution time perspective, $F_{SumElements \cup RCAD}(1) \simeq 75\%$, so our approach is faster for $\sim 75\%$ of the instances. One can also observe that at least one order of magnitude is gained for $\sim 20\%$ of the instances. Finally, notice that $F_{SumElements \cup RCAD}$ reaches 100% at $\tau = 3.2$ This means that if *SumElements* ∪ *RCAD* is slower than *SumElements*, it is at most by a factor of 3.2. This illustrates that our approach is also robust as it does not deteriorate much the execution time in the case where it does not bring any additional filtering.

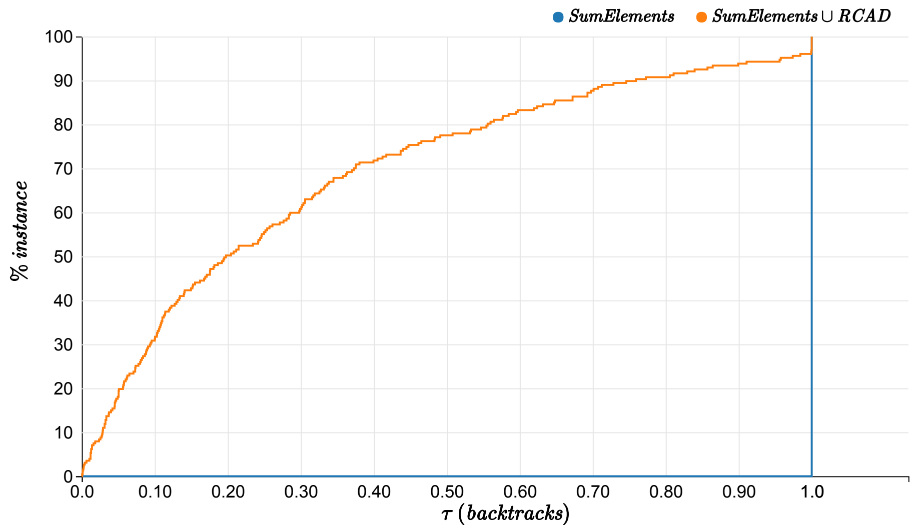


Figure 5.8: Backtracks performance profile on large instances of the CCSPEBM.

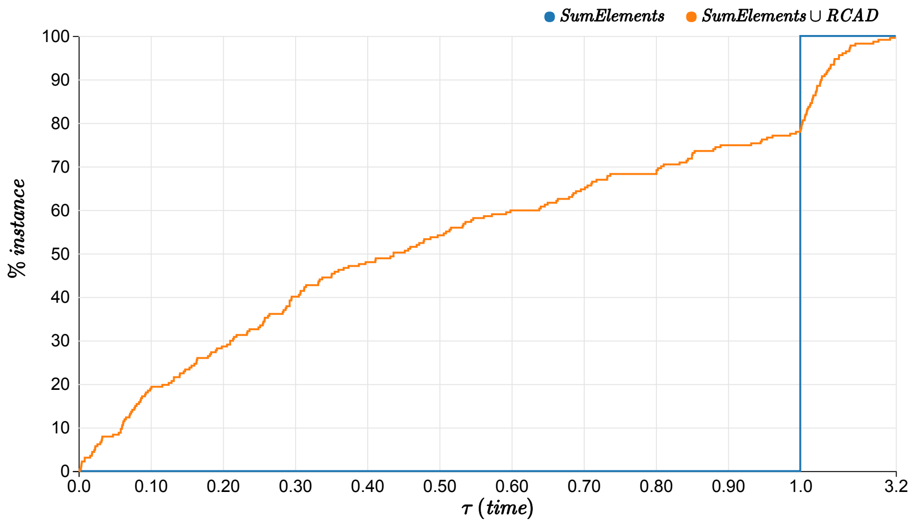


Figure 5.9: Time performance profile on large instances of the CCSPEBM.

5.4 THE PRODUCT MATRIX TRAVELLING SALESMAN PROBLEM

In order to evaluate our constraint on a second problem, let us consider a particular case of the well-known Asymmetric Travelling Salesman Problem (ATSP), the Product Matrix Travelling Salesman Problem (PMTSP), that was first formally described in [PLC87]. We recently added this problem to the CSPLib [CS]. We consider this problem because it is more academic than the previous one. The experiments show that our approach is the fastest for $\sim 90\%$ of the considered in-

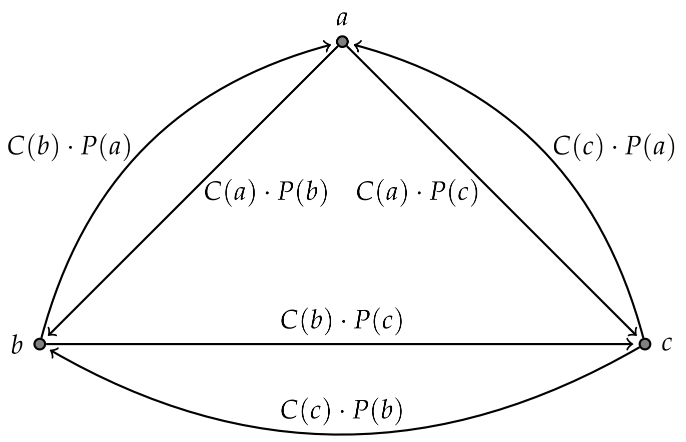


Figure 5.10: Instance of the Product Matrix Travelling Salesman Problem.

stances, even if we also experimented with the Concorde solver [App+06, App+11], a custom state-of-the-art Branch-and-Cut Mixed Integer Programming solver for the Travelling Salesman Problem (TSP).

DEFINITION Given two vectors of n elements C and P, one can construct a simple graph G with n vertices, such that the directed edge from the vertex i to the vertex $j \neq i$ has a cost equal to $C(i) \cdot P(j)$. The distance matrix is therefore the matrix product between the vectors C and P, hence the name of the problem. An instance of the PMTSP is illustrated in Figure 5.10. The problem consists in finding an Hamiltonian circuit of minimum total cost in the graph G. The problem was proven to be NP-hard in [Sar80, GLS85].

SOLVING In CP, the ATSP is usually solved with a successor model (see [Pes+98]): an array of n variables *succ* is used to represent the successors for each vertex. One then impose that the array is an Hamiltonian circuit (filtering from [Pes+98]) :

$$\textsc{circuit}(succ)$$

One can model the objective with either: 1) a sum of ELEMENT constraints, 2) a MINIMUMASSIGNMENT constraint, or 3) the RESOURCECOSTALLDIFFERENT constraint. The sum of ELEMENT constraints misses a lot of pruning in this case, so we will not consider it here. Let us respectively call *RCAD*, *MinAssignment* and *MinAssignment* ∪ *RCAD* the models with RESOURCECOSTALLDIFFERENT, MINIMUMASSIGNMENT, and both algorithms.

RESULTS We generated 50 instances with 200 vertices. We compared the performances of the different CP models and those of the Concorde solver[App+06, App+11], a custom state-of-the-art Branch-and-Cut Mixed Integer Programming solver for the TSP. Since Concorde does not solve ATSPs, we used the standard reduction from ATSPs to TSP given in [JV83] to solve our instances with this solver. We used the *Conflict Ordering Search* heuristic [Gay+15] to solve the problems in CP. For all approaches, the instances were solved to optimality. All experiments were performed on a machine with an Intel Core i7 (2,2 GHz) processor. The results are given in Figure 5.11, as a standard performance profile as defined in [DM02] (i.e., all approaches are used in the baseline set $\mathcal{B}$).

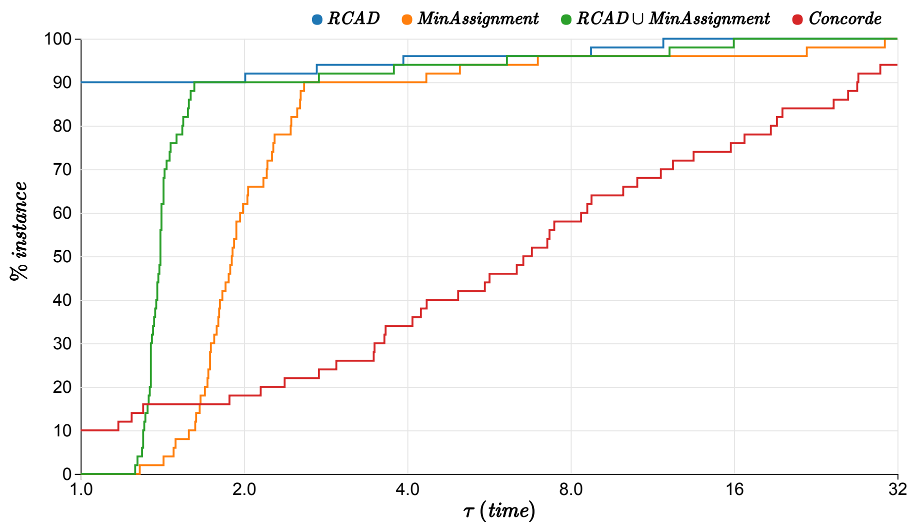


Figure 5.11: Performance profile for the different CP models and the Concorde solver on the 50 generated instances of the PMTSP.

First, one can observe that *MinAssignment* is the slowest of the CP approaches, which is a conclusion already done in the last section. Using our propagator *additionnaly* provides some speed-up since it can provide more pruning or detect a failure faster at a given node since it has a higher priority in the propagation queue. Yet, only using our algorithm is the best option, since it is always faster than the other CP models. The overhead of using MinimumAssignment to possibly get more pruning simply does not pay off in this case.

An interesting observation is that Concorde is only the fastest solver for $\sim 10\%$ of the instances, while *RCAD* is the fastest for the remaining $\sim 90\%$ instances. Moreover, the solving time ratio for Concorde, as compared with *RCAD*, is often large: 5% of the instances are even not solved in a factor 32 of the time required by *RCAD*. At the same time, when *RCAD* is not the fastest, it requires at the very most a factor of 12 as compared with Concorde. This indicates that even when *RCAD* is not the fastest, it is more robust. From this experiment, one can conclude that the CP technology should be the preferred one to solve this problem.

CONCLUSION AND PERSPECTIVES

CONCLUSION

Before we give a general conclusion, let us first provide a conclusion for each important contribution of this thesis.

GLOBAL CONSTRAINTS EVALUATION In this part, we introduced a tool that allows researchers of the OR community to easily build and export performance profiles from their experimental data. In addition, we proposed a framework to roughly estimate visually improvements brought to specific parts of a solving process. We believe this methodology can help to avoid falling into pitfalls, where improving the efficacy of a given algorithm used in a particular context is not worth experimentally. We showcased this approach for propagators in Constraint Programming and cuts in Mixed Integer Linear Programming.

Search heuristics can have a significant impact on the outcome of the evaluation of a global constraint (or more generally a filtering procedure). We therefore proposed a rigorous and yet simple framework that allows preventing any unfair advantages regarding the compared approaches, by *only measuring the effect of additional filtering*. Being able to measure exactly the time gain provided by a filtering algorithm permits to reduce the bias in empirical evaluations. We explained how to actually implement this framework.

Evaluating the potential advantages of reducing the cost of a given filtering procedure is of great importance to make our research efforts as fruitful as possible. As a first step in this direction, we proposed a systematic methodology to simulate the performance of *fictional implementations of a propagator having reduced activation cost*. This is done *before* starting time-consuming research activities to actually reduce the cost. A nice feature of deterministic replays is that measurements can be carried out in different replays, removing imprecisions due to measurement overhead.

We suggested in this work a broader usage of performance profiles in the CP community. We showcased that they allow deriving many informative conclusions. We evaluated several propagators for the following constraints on quite large benchmarks: ALLDIFFERENT, CUMULATIVE, BINPACKING and UNARY with transition times. In addition, a nice feature of our version of the performance profiles is that the whole community could continuously add new data and update them on a central repository, as a common effort to improve knowledge about the performance of propagators. This can be done as long as the baseline model remains the same.

As for the estimation of the impact of reducing the cost of a propagator, we illustrated the approach for Energetic Reasoning and Revisited Cardinality Reasoning for BinPacking over popular sets of instances. We found that reducing the propagator cost, *even to the point of making it negligible*, might actually be beneficial only on a small subset of a given instance set. Furthermore, this outcome can differ substantially depending on the considered benchmark and on the search strategy. We also briefly studied the shortfalls of not being able to achieve bound consistency for the CUMULATIVE constraint. Interestingly, from a pruning point a view, there is still a lot to gain.

GENERALIZED UNARY RESOURCE WITH TRANSITION TIMES This chapter has extended the algorithms and data structures for the unary resource, taking into account family-based transition times in order to perform additional propagation. The method also handles optional activities so that one can model more general problems (e.g., involving alternative resources). The original data structures and algorithms have been adapted accordingly. The approach is lightweight from both the time and space perspectives. Experiments conducted on the Job-Shop Problem with Sequence Dependent Transition Times have demonstrated that our work provides a substantial gain and is quite robust to changes in instance characteristics (e.g., number of activities and families).

RESOURCE-COST ALLDIFFERENT In this chapter, we considered problems where a set of items, each requiring a different amount of resource, must be assigned to different slots, and where the price for a unit of resource can vary at each slot. In this class of problems, the objective is to assign items such that the overall resource cost is minimized. We showed two ways of modeling such an objective in Constraint Programming (CP) and their limitations (limited inference or scalability). To cope with those limitations, we introduced a new filtering algorithm that we evaluate on a real and large-scale industrial problem, the Continuous Casting Steel Production with Electricity Bill Minimization. The results demonstrate that, especially for large instances, our approach is often faster, sometimes with an important speed-up: an order of magnitude is gained for $\sim$ 20% of the large instances and $\sim$ 75% of them are solved faster. Moreover, our algorithm is robust, in the sense that when it is slower, it is by a small factor (3.2 at the very most). Results also illustrate that for a non-negligible number of the large instances, we can get an important gain in terms of the number of backtracks as compared with a decomposition: a reduction of at least one order of magnitude is obtained for $\sim$ 30% of the instances. We also considered a more academic problem, the Product Matrix Travelling Salesman Problem. We compared our approach with existing ones in CP, as well as with Concorde, a custom TSP solver. The results show that our approach is the fastest in CP, and that it outperforms Concorde for $\sim$ 90% of the instances, sometimes by an important factor (some instances could not be solved in a factor 32 as compared with our approach).

MAIN CONCLUSION The aim of this thesis was to focus on the design of scalable propagators. Low time complexity of algorithms is, of course, always desirable, but in the case of propagators, it often happens that speed must be traded with inference strength, i.e., one must decide whether speed at each propagation call/search node must be preferred to the reduction of the total number of nodes. There is no final answer to this problem, and one must experiment with this trade-off when considering small instances. However, once we have to work with larger instances (and this will be the case), this trade-off does not matter anymore: we simply cannot use heavy propagators, even if they infer more. When designing new propagators, we must keep in mind the scalability constraint of our (new) algorithms. Heavy, more-powerful propagators should not be used in the general case in the future, as the instance sizes will keep growing. Some of our experiments made that conclusion for several constraints.

Yet, when designing a new filtering method, one does not necessarily have to directly come up with an efficient algorithm. One can always first derive a (maybe naive) inefficient implementation from the inference rules and then probe if reducing its time complexity will actually pay off.

PERSPECTIVES

Let us end this thesis by suggesting a few perspectives for the main contributions made in this thesis.

GLOBAL CONSTRAINTS EVALUATION We might consider generating more CBTs for a given instance, and gather the results. This would allow the evaluation approach to be even more robust. For instance, we could use several branching strategies or use Large Neighborhood Search in order to get more data for a given, large-scale, instance (and not only data from the beginning of the search tree). Another way to do so is to bound the search space to be replayed with a set of no-goods. Replaying a CBT with a model is not always possible, because the constraints used in the generator model must be subsumed by the one in the model, which is not always the case. For example, Time-Tabling and Edge-finding for the CUMULATIVE constraint do not subsume each other. Still we can generate a CBT into which all the replayed CBTs will be included in. To do so, when we generate the CBTs, we could use a model that prunes only when all the constraints used in replays are actually able to prune. A broader and long-term vision regarding propagator evaluation would be to extend this framework with statistical hypothesis testing. The aim would be to test if on (representatives) benchmarks, the use of a given propagator has a statistically significant positive impact. Finally, regarding the potential of necessary conditions, we could study the gain of activating a propagator only when it prunes several variables or values.

GENERALIZED UNARY RESOURCE WITH TRANSITION TIMES We would like to consider other types of problems (e.g., the Traveling Salesman Problem with Time Windows) and combine this work with the use of good lower bounds in a branch-and-bound setting. Moreover, when there are no families defined *a priori* in an instance, we want to study the benefit of first creating them by means of *clustering* algorithms and then using the filtering introduced in this chapter. This approach might prove to be helpful when the intra-cluster transition times are significantly smaller than the inter-cluster ones. Finally, we could generalize the definition of the families: all activities inside a given family could have the same positive constant transition time between each other (instead of always being 0). We would like to adapt our algorithms to take this into account.

RESOURCE-COST ALLDIFFERENT We would like to extend our algorithm to handle the production of several items at the same time slot. This would therefore be a particular case of GCCCOST for cardinalities larger than 1. Moreover, we would like to study the performances of our algorithm on other kinds of problems such as discrete lot sizing problems [Hou+14]. Finally, in the case of the CCSPEBM, an important future work is the evaluation of the robustness of the approach in function of errors in price prediction, as there is a certain level of uncertainty in price forecasts.

BIBLIOGRAPHY

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending CHIP in order to solve complex scheduling and placement problems." In: *Mathematical and Computer Modelling* 17.7 (1993), pp. 57–73.

[All+08] Ali Allahverdi, CT Ng, TC Edwin Cheng, and Mikhail Y Kovalyov. "A survey of scheduling problems with setup times or costs." In: *European Journal of Operational Research* 187.3 (2008), pp. 985–1032.

[AGS16] John Aoga, Tias Guns, and Pierre Schaus. "An efficient algorithm for mining frequent sequence with constraint programming." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer International Publishing. 2016, pp. 315–330.

[AGS17] John Aoga, Tias Guns, and Pierre Schaus. "Mining Time-constrained Sequential Patterns with Constraint Programming." In: *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR17)*. 2017.

[App+01] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. "TSP cuts which do not conform to the template paradigm." In: *Computational Combinatorial Optimization*. Springer, 2001, pp. 261–303.

[App+06] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *Concorde TSP solver*. 2006.

[App+11] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[ABF04] Christian Artigues, Sana Belmokhtar, and Dominique Feillet. “A new exact solution algorithm for the job shop problem with sequence-dependent setup times.” In: *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*. Springer, 2004, pp. 37–49.

[AF08] Christian Artigues and Dominique Feillet. “A branch and bound method for the job-shop problem with sequence-dependent setup times.” In: *Annals of Operations Research* 159.1 (2008), pp. 135–159.

[AK99] Hassan Aït-Kaci. *Warren’s Abstract Machine - A Tutorial Reconstruction*. 1999.

[BLP00] Philippe Baptiste and Claude Le Pape. “Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems.” In: *Constraints* 5.1-2 (2000), pp. 119–139.

[BLPN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Vol. 39. International Series in Operations Research & Management Science. Springer, 2001.

[Bar08] Roman Barták. “Search Strategies for Scheduling Problems with Optional Activities.” In: *Proceedings of CSCLP 2008 Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming, Rome*. 2008.

[BČ10] Roman Barták and Ondřej Čepek. “Incremental propagation rules for a precedence graph with optional activities and time windows.” In: *Transactions of the Institute of Measurement and Control* 32.1 (2010), pp. 73–96.

[BC94] Nicolas Beldiceanu and Evelyne Contejean. “Introducing global constraints in CHIP.” In: *Mathematical and computer Modelling* 20.12 (1994), pp. 97–123.

[Bel56] Richard Bellman. *On a Routing Problem*. Tech. rep. DTIC Document, 1956.

[Ben96] Frédéric Benhamou. “Heterogeneous constraint solving.” In: *International Conference on Algebraic and Logic Programming*. Springer. 1996, pp. 62–76.

[BCH14] David Bergman, André A. Ciré, and Willem Jan van Hoeve. “MDD Propagation for Sequence Constraints.” In: *J. Artif. Intell. Res. (JAIR)* 50 (2014), pp. 697–722.

[Ber+16] David Bergman, Andre A Cire, Willem-Jan van Hoeve, and JN Hooker. *Decision Diagrams for Optimization*. 2016.

[BHS11] Timo Berthold, Stefan Heinz, and Jens Schulz. "An Approximative Criterion for the Potential of Energetic Reasoning." In: *Theory and Practice of Algorithms in (Computer) Systems*. 2011, pp. 229–239.

[Bes06] Christian Bessiere. "Constraint Propagation." In: *Handbook of constraint programming* (2006), pp. 85–134.

[BD05] Christian Bessière and Romuald Debruyne. "Optimal and Suboptimal Singleton Arc Consistency Algorithms." In: *International Joint Conference on Artificial Intelligence*. 2005, pp. 54–59.

[BR97] Christian Bessiere and Jean-Charles Régin. "Arc consistency for general constraint networks: preliminary results." In: *International Joint Conference on Artificial Intelligence*. Citeseer, 1997, pp. 398–404.

[BVH03] Christian Bessiere and Pascal Van Hentenryck. "To be or not to be... a global constraint." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2003, pp. 789–794.

[Bra+07] Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter J. Stuckey, and Toby Walsh. "Encodings of the Sequence Constraint." In: *International Conference on Principles and Practice of Constraint Programming*. 2007, pp. 210–224.

[Bru99] Peter Brucker. "Complex scheduling problems." In: *Zeitschrift Oper. Res*. Citeseer. 1999.

[BT96] Peter Brucker and Olaf Thiele. "A branch & bound method for the general-shop problem with sequence dependent setup-times." In: *Operations-Research-Spektrum* 18.3 (1996), pp. 145–161.

[CS] Sascha Van Cauwelaert and Pierre Schaus. *CSPLib Problem 075: Product Matrix Travelling Salesman Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob075`.

[CY10] Kenil C. K. Cheng and Roland H. C. Yap. "An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints." In: *Constraints* 15.2 (2010), pp. 265–304.

[CMT81] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. "State-space relaxation procedures for the computation of bounds to routing problems." In: *Networks* 11.2 (1981), pp. 145–164.

[Dej16] Cyrille Dejemeppe. "Constraint programming algorithms and models for scheduling applications." PhD thesis. Université catholique de Louvain, Louvain-la-Neuve, 2016.

[DVCS15] Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. "The Unary Resource with Transition Times." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 89–104.

[Dej+16] Cyrille Dejemeppe, Olivier Devolder, Victor Lecomte, and Pierre Schaus. "Forward-Checking Filtering for Nested Cardinality Constraints : Application to an Energy Cost-Aware Production Planning Problem for Tissue Manufacturing." In: *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*. Springer. 2016, pp. 108–124.

[DHM00] Pierre Deransart, Manuel V Hermenegildo, and Jan Maluszynski. *Analysis and visualization tools for constraint programming: constraint debugging*. Vol. 1870. Lecture Notes in Computer Science. Springer, 2000.

[DP14] Alban Derrien and Thierry Petit. "A New Characterization of Relevant Intervals for Energetic Reasoning." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 289–297.

[DSVH90] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. "Solving large combinatorial problems in logic programming." In: *The Journal of Logic Programming* 8.1 (1990), pp. 75–93.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking optimization software with performance profiles." In: *Mathematical programming* 91.2 (2002), pp. 201–213.

[DB+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: *Workshop on Analytic Algorithmics and Combinatorics*. SIAM. 2013, pp. 80–90.

[DCP16] Sylvain Ducomman, Hadrien Cambazard, and Bernard Penz. "Alternative Filtering for the Weighted Circuit Constraint: Comparing Lower Bounds for the TSP and Solving TSPTW." In: *AAAI Conference on Artificial Intelligence*. 2016.

[EL90] Jacques Erschler and Pierre Lopez. "Energy-based approach for task scheduling under time and resources constraints." In: *International workshop on project management and scheduling*. 1990, pp. 115–121.

[FW86] Philip J Fleming and John J Wallace. "How not to lie with statistics: the correct way to summarize benchmark results." In: *Communications of the ACM* 29.3 (1986), pp. 218–221.

[FLN00] Filippo Focacci, Philippe Laborie, and Wim Nuijten. "Solving Scheduling Problems with Setup Times and Alternative Resources." In: *AIPS*. 2000, pp. 92–101.

[FLM99] Filippo Focacci, Andrea Lodi, and Michela Milano. "Integration of CP and OR methods for matching problems." In: *International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. 1999.

[Foc+99] Filippo Focacci, Andrea Lodi, Michela Milano, and Daniele Vigo. "Solving TSP through the integration of OR and CP techniques." In: *Electronic notes in discrete mathematics* 1 (1999), pp. 13–25.

[GHS15a] Steven Gay, Renaud Hartert, and Pierre Schaus. "Simple and scalable time-table filtering for the cumulative constraint." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 149–157.

[GHS15b] Steven Gay, Renaud Hartert, and Pierre Schaus. "Time-table disjunctive reasoning for the cumulative constraint." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Springer, 2015, pp. 157–172.

[GSDS14] Steven Gay, Pierre Schaus, and Vivian De Smedt. "Continuous Casting Scheduling with Constraint Programming." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 831–845.

[Gay+15] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. "Conflict ordering search for scheduling problems." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 140–148.

[GLS85] Paul C. Gilmore, Eugene L. Lawler, and David Shmoys. "Well-solved special cases of the traveling salesman problem." In: *The Traveling Salesman Problem*. Ed. by John Wiley & Sons. 1985.

[GH10] Diarmuid Grimes and Emmanuel Hebrard. "Job shop scheduling with setup times and maximal time-lags: A simple constraint programming approach." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2010, pp. 147–161.

[Har+15] Renaud Hartert, Pierre Schaus, Stefano Vissicchio, and Olivier Bonaventure. "Solving segment routing problems with hybrid constraint programming techniques." In: *International Conference on Principles and Practice of Constraint Programming*. Springer, Cham. 2015, pp. 592–608.

[HG95] William D Harvey and Matthew L Ginsberg. "Limited discrepancy search." In: *International Joint Conference on Artificial Intelligence*. 1995, pp. 607–615.

[HM09] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT press, 2009.

[Hoe01] Willem-Jan van Hoeve. "The alldifferent constraint: A survey." In: *arXiv preprint cs/0105015* (2001).

[Hoe+06] Willem Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. “Revisiting the Sequence Constraint.” In: *International Conference on Principles and Practice of Constraint Programming*. 2006, pp. 620–634.

[Hou+14] Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey, and Yves Deville. “The stockingcost constraint.” In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 382–397.

[Jef+10] Christopher Jefferson, Neil CA Moore, Peter Nightingale, and Karen E Petrie. “Implementing logical connectives in constraint programming.” In: *Artificial Intelligence* 174.16-17 (2010), pp. 1407–1429.

[JV83] Roy Jonker and Ton Volgenant. “Transforming asymmetric into symmetric traveling salesman problems.” In: *Operations Research Letters* 2.4 (1983), pp. 161–163.

[Knu74] Donald E. Knuth. “Structured Programming with Go to Statements.” In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301.

[KSS99] Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. “Benchmark instances for project scheduling problems.” In: *Project Scheduling*. Springer, 1999, pp. 197–212.

[Kru56] Joseph B Kruskal. “On the shortest spanning subtree of a graph and the traveling salesman problem.” In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[Lab03] Philippe Laborie. “Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results.” In: *Artificial Intelligence Journal* 143.2 (2003), pp. 151–188.

[LDD03] Ludovic Langevine, Pierre Deransart, and Mireille Ducassé. “A generic trace schema for the portability of CP(FD) debugging tools.” In: *International Workshop on Constraint Solving and Constraint Logic Programming*. Springer. 2003, pp. 171–195.

[LP+94] Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. “Time-versus-capacity compromises in project scheduling.” In: *Workshop of the UK Planning and Scheduling*. Citeseer, 1994.

[LBC12] Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. “A scalable sweep algorithm for the cumulative constraint.” In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 439–454.

[LM12] Michele Lombardi and Michela Milano. “Optimal methods for resource allocation and scheduling: a cross-disciplinary survey.” In: *Constraints* 17.1 (2012), pp. 51–85.

[LO+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: *International Joint Conference on Artificial Intelligence*. Vol. 3. 2003, pp. 245–250.

[MP15] Ciaran McCreesh and Patrick Prosser. "A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 295–312.

[MVH17] Laurent Michel and Pascal Van Hentenryck. "Constraint-Based Local Search." In: *Handbook of Heuristics*. Ed. by Rafael Martí, Pardalos Panos, and Mauricio G. C. Resende. Cham: Springer International Publishing, 2017, pp. 1–38.

[Moo59] Edward Moore. "The Shortest Path Through a Maze." In: *Proc. Internat. Sympos. Switching Theory*. 1959.

[MW95] Tobias Müller and Jörg Würtz. "Constructive disjunction in Oz." In: *Workshop Logische Programmierung*. Citeseer. 1995.

[Osc12] OscaR Team. *OscaR: Scala in OR*. Available from `https://bitbucket.org/oscarlib/oscar`. 2012.

[PSR13] François Pelsser, Pierre Schaus, and Jean-Charles Régin. "Revisiting the cardinality reasoning for BinPacking constraint." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, pp. 578–586.

[Pes+98] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. "An exact constraint logic programming algorithm for the traveling salesman problem with time windows." In: *Transportation Science* 32.1 (1998), pp. 12–29.

[PR10] David Pisinger and Stefan Ropke. "Large neighborhood search." In: *Handbook of metaheuristics*. Springer, 2010, pp. 399–419.

[PLC87] Robert D. Plante, Timothy J. Lowe, and R. Chandrasekaran. "The product matrix traveling salesman problem: an application and solution heuristic." In: *Operations Research* 35.5 (1987), pp. 772–783.

[Rég94] Jean-Charles Régin. "A filtering algorithm for constraints of difference in CSPs." In: *AAAI Conference on Artificial Intelligence*. Vol. 94. 1994, pp. 362–367.

[Rég02] Jean-Charles Régin. "Cost-based arc consistency for global cardinality constraints." In: *Constraints* 7.3-4 (2002), pp. 387–405.

[Rei91] Gerhard Reinelt. "TSPLIB—A traveling salesman problem library." In: *ORSA journal on computing* 3.4 (1991), pp. 376–384.

[SM+13] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. "Sparse-sets for domain implementation." In: *International workshop on Techniques foR Implementing Constraint programming Systems*. 2013, pp. 1–10.

[Sar80] V. I. Sarvanov. "On the complexity of minimizing a linear form on a set of cyclic permutations." In: *Dokl. Akad. Nauk SSSR*. Vol. 253. 3. 1980, pp. 533–535.

[Sch09] Pierre Schaus. "Solving balancing and bin-packing problems with constraint programming." PhD thesis. Université catholique de Louvain, Louvain-la-Neuve, 2009.

[Explorer97] Christian Schulte. "Oz Explorer: A Visual Constraint Programming Tool." In: *Proceedings of the Fourteenth International Conference on Logic Programming*. Ed. by Lee Naish. Leuven, Belgium: The MIT Press, July 1997, pp. 286–300.

[Sch99] Christian Schulte. "Comparing Trailing and Copying for Constraint Programming." In: *International Conference on Logic Programming*. Vol. 99. 1999, pp. 275–289.

[SS05] Christian Schulte and Peter J Stuckey. "When do bounds and domain propagation lead to the same search space?" In: *ACM Transactions on Programming Languages and Systems* 27.3 (2005), pp. 388–425.

[ST09] Christian Schulte and Guido Tack. "Weakly monotonic propagators." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2009, pp. 723–730.

[Sel02] Meinolf Sellmann. "An arc-consistency algorithm for the minimum weight all different constraint." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2002, pp. 744–749.

[Sha04] Paul Shaw. "A constraint for bin packing." In: *International Conference on Principles and Practice of Constraint Programming*. Springer, 2004, pp. 648–662.

[Shi+15] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. "Visual Search Tree Profiling." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015.

[Shi+16a] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. "Learning from Learning Solvers." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 455–472.

[Shi+16b] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. "Visual search tree profiling." In: *Constraints* 21.1 (2016), pp. 77–94.

[SH10] Helmut Simonis and Tarik Hadzic. "A family of resource constraints for energy cost aware scheduling." In: *Third International Workshop on Constraint Reasoning and Optimization for Computational Sustainability, St. Andrews, Scotland, UK*. 2010.

[SH11] Helmut Simonis and Tarik Hadzic. "A Resource Cost Aware Cumulative." In: *Recent Advances in Constraints: 14th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2009, Barcelona, Spain, June 15-17, 2009, Revised Selected Papers*. Springer, 2011, pp. 76–89.

[Sim+10] Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. "A generic visualization platform for CP." In: *International Conference on Principles and Practice of Constraint Programming*. Springer, 2010, pp. 460–474.

[Smi05] Barbara M Smith. "Modelling for constraint programming." In: *Lecture Notes for the First International Summer School on Constraint Programming* (2005).

[Tac09] Guido Tack. "Constraint Propagation - Models, Techniques, Implementation." PhD thesis. Saarland University, Germany, 2009.

[VB06] Peter Van Beek. "Backtracking search algorithms." In: *Handbook of constraint programming* (2006), pp. 85–134.

[VCDS18] Sascha Van Cauwelaert, Cyrille Dejemeppe, and Pierre Schaus. *The Unary Resource with Transition Times and Optional Activities*. Tech. rep. UCLouvain, 2018.

[VCLP14] Sascha Van Cauwelaert, Michele Lombardi, and Schaus Pierre. "Supervised Learning to Control Energetic Reasoning : Feasibility Study." In: *Proceedings of the Doctoral Program of CP2014*. 2014.

[VCLS15] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. "Understanding the potential of propagators." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Springer. 2015, pp. 427–436.

[VCLS16] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. *A Visual Web Tool to Perform What-If Analysis of Optimization Approaches*. Tech. rep. UCLouvain, 2016.

[VCLS18] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. "How efficient is a global constraint in practice ?" In: *Constraints* 23.1 (2018), pp. 87–122.

[VCS17a] Sascha Van Cauwelaert and Pierre Schaus. "Efficient Filtering for the Resource-Cost AllDifferent Constraint." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Springer. 2017.

[VCS17b] Sascha Van Cauwelaert and Pierre Schaus. "Efficient filtering for the Resource-Cost AllDifferent constraint." In: *Constraints* 22.4 (2017), pp. 493–511.

[VC+16] Sascha Van Cauwelaert, Cyrille Dejemeppe, Jean-Noël Monette, and Pierre Schaus. "Efficient Filtering for the Unary Resource with Family-Based Transition Times." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 520–535.

[VH89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. Vol. 5. MIT press Cambridge, 1989.

[VHC88] Pascal Van Hentenryck and Jean-Philippe Carillon. "Generality versus Specificity: An Experience with AI and OR Techniques." In: *AAAI Conference on Artificial Intelligence*. 1988, pp. 660–664.

[VHD87] Pascal Van Hentenryck and Mehmet Dincbas. "Forward Checking in Logic Programming." In: *International Conference on Logic Programming*. 1987, pp. 229–256.

[VRH04] Peter Van Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[Vil04] Petr Vilím. "$\mathcal{O}(n.log(n))$ Filtering Algorithms for Unary Resource Constraint." In: *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*. Ed. by Jean-Charles Régin and Michel Rueher. Springer, 2004, pp. 335–347.

[Vil07] Petr Vilım. "Global constraints in scheduling." PhD thesis. Charles University in Prague, Faculty of Mathematics, Physics, Department of Theoretical Computer Science, and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské námestı 2/25, 118 00 Praha 1, Czech Republic, 2007.

[Vil09a] Petr Vilım. "Edge finding filtering algorithm for discrete cumulative resources in O (kn log n)." In: *Principles and Practice of Constraint Programming-CP*. Vol. 5732. Springer. 2009, pp. 802–816.

[Vil09b] Petr Vilím. "Max energy filtering algorithm for discrete cumulative resources." In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer. 2009, pp. 294–308.

[VB12] Petr Vilım and Roman Barták. "Filtering algorithms for batch processing with sequence dependent setup times." In: *Proceedings of the 6th International Conference on AI Planning and Scheduling, AIPS*. 2012.

[WB16] Gilles Madi Wamba and Nicolas Beldiceanu. "The TaskIntersection Constraint." In: *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*. Springer. 2016, pp. 246–261.

[War83] David Warren. *An abstract Prolog instruction set*. Vol. 309. SRI International Menlo Park, California, 1983.

[War13] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

[Wer14] Rafał Weron. "Electricity price forecasting: A review of the state-of-the-art with a look into the future." In: *International Journal of Forecasting* 30.4 (2014), pp. 1030–1081.

[Wol09] Armin Wolf. "Constraint-Based Task Scheduling with Sequence Dependent Setup Times, Time Windows and Breaks." In: *GI Jahrestagung* 154 (2009), pp. 3205–3219.

[WM96] Jörg Würtz and Tobias Müller. "Constructive disjunction revisited." In: *Advances in Artificial Intelligence* (1996), pp. 377–386.

[WB06] Rolf Wüstenhagen and Michael Bilharz. "Green energy market development in Germany: effective public policy and emerging customer demand." In: *Energy policy* 34.13 (2006), pp. 1681–1696.

[Zam+13] Stéphane Zampelli, Yannis Vergados, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. "The berth allocation and quay crane assignment problem using a CP approach." In: *Principles and Practice of Constraint Programming*. Springer. 2013, pp. 880–896.