

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par : *l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)*

Présentée et soutenue le *13/05/2015* **par :**

MOHAMED SIALA

Search, propagation, and learning in sequencing and scheduling problems

JURY

CHRISTIAN ARTIGUES	Directeur de Recherche, LAAS-CNRS	Directeur de thèse
FAHIEM BACCHUS	Professeur, University of Toronto	Rapporteur
CHRISTIAN BESSIERE	Directeur de Recherche, LIRMM-CNRS	Rapporteur
HADRIEN CAMBAZARD	Maître de conférences, G-SCOP & Grenoble INP	Examinateur
EMMANUEL HEBRARD	Chargé de recherche, LAAS-CNRS	Directeur de thèse
GEORGE KATSIRELOS	Chargé de recherche, INRA Toulouse	Examinateur
CHRISTINE SOLNON	Professeur, INSA Lyon, LIRIS-CNRS	Examinateur

École doctorale et spécialité :
EDSYS : Informatique 4200018

Unité de Recherche :
Laboratoire d'analyse et d'architecture des systémes (LAAS-CNRS)

Directeur(s) de Thèse :
Christian Artigues et *Emmanuel Hebrard*

Rapporteurs :
Fahiem Bacchus et *Christian Bessiere*

Abstract

Sequencing and scheduling involve the organization in time of operations subject to capacity and resource constraints. We propose in this dissertation several improvements to the constraint satisfaction and combinatorial optimization methods for solving these problems. These contributions concern three different aspects: how to choose the next node to explore (search)? how much, and how efficiently, one can reduce the search space (propagation)? and what can be learnt from previous failures (learning)?

Our contributions start with an empirical study of search heuristics for the well known car-sequencing problem. This evaluation characterizes the key aspects of a good heuristic and shows that the search strategy is as important as the propagation aspect in this problem. Second, we carefully investigate the propagation aspect in a class of sequencing problems. In particular, we propose an algorithm for filtering a type of sequence constraints which worst case time complexity is lower than the best known upper bound, and indeed optimal. Third, we investigate the impact of clause learning for solving the car-sequencing problem. In particular, we propose reduced explanations for the new filtering. The experimental evaluation shows compelling evidence supporting the importance of clause learning for solving efficiently this problem. Next, we revisit the general approach of lazy generation for the Boolean variables encoding the domains. Our proposition avoids a classical redundancy issue without computational overhead. Finally, we investigate conflict analysis algorithms for solving disjunctive scheduling problems. In particular, we introduce a novel learning procedure tailored to this family of problems. The new conflict analysis differs from conventional methods by learning clauses whose size are not function of the scheduling horizon. Our comprehensive experimental study with traditional academic benchmarks demonstrates the impact of the novel learning scheme that we propose. In particular, we find new lower bounds for a well known scheduling benchmark.

Keywords: Artificial intelligence, constraint programming, Boolean satisfiability, combinatorial optimization, sequencing, scheduling.

Résumé

Les problèmes de séquencement et d'ordonnancement forment une famille de problèmes combinatoires qui implique la programmation dans le temps d'un ensemble d'opérations soumises à des contraintes de capacités et de ressources. Nous contribuons dans cette thèse à la résolution de ces problèmes dans un contexte de satisfaction de contraintes et d'optimisation combinatoire. Nos propositions concernent trois aspects différents : comment choisir le prochain nœud à explorer (recherche) ? comment réduire efficacement l'espace de recherche (propagation) ? et que peut-on apprendre des échecs rencontrés lors de la recherche (apprentissage) ?

Nos contributions commencent par une étude approfondie des heuristiques de branchement pour le problème de séquencement de chaîne d'assemblage de voitures. Cette évaluation montre d'abord les paramètres clés de ce qui constitue une bonne heuristique pour ce problème. De plus, elle montre que la stratégie de branchement est aussi importante que la méthode de propagation. Deuxièmement, nous étudions les mécanismes de propagation pour une classe de contraintes de séquencement à travers la conception de plusieurs algorithmes de filtrage. En particulier, nous proposons un algorithme de filtrage complet pour un type de contrainte de séquence avec une complexité temporelle optimale dans le pire cas. Troisièmement, nous investiguons l'impact de l'apprentissage de clauses pour résoudre le problème de séquencement de véhicules à travers une nouvelle stratégie d'explication réduite pour le nouveau filtrage. L'évaluation expérimentale montre l'importance de l'apprentissage de clauses pour ce problème. Ensuite, nous proposons une alternative pour la génération retardée de variables booléennes pour encoder les domaines. Finalement, nous revisitons les algorithmes d'analyse de conflits pour résoudre les problèmes d'ordonnancement disjonctifs. En particulier, nous introduisons une nouvelle procédure d'analyse de conflits dédiée pour cette famille de problèmes. La nouvelle méthode diffère des algorithmes traditionnels par l'apprentissage de clauses portant uniquement sur les variables booléennes de disjonctions. Enfin, nous présentons les résultats d'une large étude expérimentale qui démontre l'impact de ces nouveaux mécanismes d'apprentissage. En particulier, la nouvelle méthode d'analyse de conflits a permis de découvrir de nouvelle bornes inférieures pour des instances largement étudiées de la littérature.

Mot-clés : Intelligence artificielle, programmation par contraintes, satisfiabilité booléenne, optimisation combinatoire, séquencement, ordonnancement

Acknowledgments

I am tremendously grateful to my supervisor Emmanuel Hebrard, who guided this work. I consider myself very fortunate to work with him during all this period. I am indebted to Emmanuel for the freedom that he gave me, the countless discussions, his insistence of high research quality, and for including me in the reviewing process in top AI conferences and journals. I will remember his advices as well as his research methodologies.

I would like to thank my co-supervisor Christian Artigues. It was a great pleasure working with him. I am grateful for his continuous support and his constructive suggestions throughout the PhD.

I would like to thank my external reviewers Pr. Fahiem Bacchus and Dr. Christian Bessiere for their thorough examination. I'm truly honored that they reviewed my thesis. I would also like to thank the jury members Dr. Hadrien Cambazard, Dr. George Katsirelos, and Pr. Christine Solnon for their evaluation and their valuable feedback.

Many thanks to Marie-José Huguet, who contributed in many parts of this work. I appreciate all the time we spent discussing and working on sequencing problems.

I would like to thank my CP Doctoral Program mentors Ian Gent and Thierry Petit for their valuable advices.

I would like to thank Toby Walsh for offering me the opportunity to visit him in NICTA, Sydney. Although the work of the visit is not presented in this thesis, it did help me strengthen and consolidate my understanding of constraint programming. This visit yielded to a collaboration with Nina Narodytska and Thierry Petit to whom I express my deep gratitude. I want to thank Nina for the valuable discussions and precious advices she gave me during my visit to NICTA.

My thanks go also to Valentin Mayer Eichberger for the collaboration that we had. I enjoyed all the time we spent competing for the best model solving the car-sequencing problem.

My sincere thanks to the Cork Constraint Computation Centre (4C) for kindly granting access to its computing resources. I would also like to thank Barry O'Sullivan for the period I spend at Insight, Centre for Data Analytics in Cork prior to the PhD defense.

I would like to thank CNRS, Google and midi-Pyrénées region for the funding of the PhD. I thank also the LAAS administrative service and specially Christèle Mouclier.

I want to thank all the ROC team members at LAAS-CNRS who I shared this priceless experience with.

The list will never end if I am to mention all those who gave me valuable support in my PhD. I would only say thank you all.

Last but not least, I would like to thank my family and my friends for their continuous support and encouragement throughout my studies. Most importantly, I would like to thank my beloved mother, Sonia, who has always supported me and encouraged me to move forward.

Dedication

To the memory of my grandfather Abdelmajid,

To my grandmother Souad,

To my mother Sonia,

Contents

Contents **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
`Search` . . . 2
`Propagation` . . . 2
`Learning` . . . 3

2 Formal Background **9**
2.1 Constraint Programming . . . 9
2.1.1 Constraint Network . . . 9
2.1.1.1 Domains, Tuples, and Constraints . . . 9
2.1.1.2 Constraint Satisfaction Problem . . . 11
2.1.1.3 Backtracking Search . . . 12
2.1.2 Constraint Propagation . . . 13
2.1.2.1 Propagators . . . 13
2.1.2.2 Local Consistency . . . 15
2.1.3 Decomposition & Global Constraints . . . 17
2.1.3.1 Decomposition . . . 17
2.1.3.2 Global Constraints . . . 17
2.1.4 Search . . . 19
2.1.4.1 Boosting Search through Randomization and Restarts . 20
2.2 Boolean Satisfiability . . . 21
2.2.1 A Background on Propositional Logic . . . 21
2.2.2 Conflict Driven Clause Learning . . . 22
2.2.2.1 Conflict Analysis . . . 23
2.2.2.2 2-Watched Literals . . . 27
2.2.2.3 Activity-Based Branching . . . 27
2.2.2.4 Clause Database Reduction . . . 27
2.2.2.5 Restarts . . . 28
2.3 Clause Learning in CP . . . 28
2.3.1 A Baseline Hybrid Solver . . . 30
2.3.1.1 Domain Encoding . . . 30
The Direct Encoding . . . 30
The Order Encoding . . . 31

2.3.1.2 Solver Description . . . 31
2.3.2 Engineering a Hybrid Solver: Modern Techniques . . . 33
2.3.2.1 Backward Explanations . . . 33
2.3.2.2 Lazy (Atom) Generation . . . 34
2.3.2.3 Semantic Reduction . . . 35

3 An Empirical Heuristic Study for the Car-Sequencing Problem 37
3.1 The Car-Sequencing Problem . . . 38
3.1.1 Problem Description . . . 38
3.1.2 Modeling . . . 39
3.1.3 Related Work . . . 40
3.2 Heuristics Classification . . . 41
3.2.1 Classification Criteria . . . 41
3.2.1.1 Branching . . . 41
3.2.1.2 Exploration . . . 41
3.2.1.3 Selection . . . 42
3.2.1.4 Aggregation . . . 43
3.2.2 Heuristics Structure . . . 44
3.3 Evaluating the new Structure . . . 45
3.3.1 Impact of each Heuristic . . . 46
3.3.2 Criteria Analysis . . . 48
3.3.2.1 Branching Strategy . . . 48
3.3.2.2 Exploration . . . 49
3.3.2.3 Selection . . . 50
3.3.2.4 Aggregation . . . 51
3.3.3 A Summary Regarding the Criteria . . . 52
3.4 Search vs. Propagation . . . 53

4 Propagation in Sequencing Problems 55
4.1 Slack-Pruning . . . 56
4.1.1 Triggering Failure via Slack . . . 56
4.1.2 Filtering the Domains . . . 58
4.1.3 Time Complexity . . . 59
4.2 Sequence Constraints . . . 60
4.2.1 Decomposition via slide . . . 60
4.2.2 Chains of Among Constraints: . . . 61
4.2.2.1 Chain of AtMost Constraints . . . 61
4.2.2.2 Global Sequencing Constraint . . . 63
4.3 The AtMostSeqCard Constraint . . . 63
4.3.1 Finding a Support . . . 64
4.3.2 Filtering the Domains . . . 66
4.3.3 Algorithmic Complexity . . . 71
4.3.4 Achieving Arc-Consistency on AtMostSeqCard . . . 75
4.4 Extensions . . . 77
4.4.1 The AtMostSeqΔCard Constraint . . . 77
4.4.2 The MultiAtMostSeqCard Constraint . . . 79
4.5 Experimental Results . . . 80

4.5.1 Car-Sequencing . . . 81
4.5.2 Crew-Rostering . . . 83
Problem Description . . . 83
Models and Heuristics . . . 83
Benchmarks . . . 84

5 Learning **87**
5.1 Learning in Car-Sequencing . . . 88
5.1.1 Explaining AtMostSeqCard . . . 88
5.1.1.1 Explaining AtMostSeq & Cardinality . . . 89
Explaining AtMostSeq . . . 89
Explaining Cardinality . . . 90
5.1.1.2 Explaining the Extra-Filtering . . . 90
Explaining Failure . . . 91
Explaining Pruning . . . 94
5.1.2 Pseudo-Boolean & SAT Models for the Car-Sequencing Problem . 94
5.1.2.1 A Pseudo-Boolean Formulation . . . 94
5.1.2.2 From Pseudo-Boolean to SAT . . . 95
5.1.3 Experimental Results . . . 96
Hybrid CP/SAT . . . 97
SAT . . . 97
CP and Pseudo-Boolean Models . . . 97
5.2 Revisiting Lazy Generation . . . 99
5.2.1 The DomainFaithfulness Constraint . . . 99
5.2.1.1 Propagating DomainFaithfulness . . . 100
5.2.2 Incrementality . . . 101
Simulating UP: . . . 103
Channeling Between x and $b_1, \ldots, b_n$: . . . 105
5.2.3 Explaining DomainFaithfulness . . . 106
5.2.3.1 Explaining Failure . . . 107
5.2.3.2 Explaining Pruning . . . 107
5.3 Learning in Disjunctive Scheduling . . . 108
5.3.1 Modeling . . . 108
The Job Shop Problem . . . 109
The Open Shop Problem . . . 110
5.3.2 Search . . . 110
5.3.2.1 The Global Search Scheme . . . 110
5.3.2.2 Branching . . . 111
Variable Ordering . . . 111
Value Ordering . . . 112
5.3.3 Explaining Constraints . . . 112
5.3.3.1 Explaining Precedence(x, y, d) . . . 112
Explaining Failure . . . 112
Explaining Pruning . . . 112
5.3.3.2 Explaining Disjunctive(b, x, y, d_x, d_y) . . . 113
5.3.4 Disjunctive-Based Learning . . . 114
5.3.5 Experiments . . . 115

5.3.5.1 JSP Results . . . 117
Lawrence Instances . . . 117
Taillard Instances . . . 117
Taillard Statistics . . . 117
Lawrence Statistics . . . 123
Improving the Lower Bounds for Taillard Open Instances . 124
5.3.5.2 OSP Results . . . 126

6 Conclusion **133**

A Reproducing the Disjunctive Scheduling Experiments **151**

B Detailed Results for OSP Instances **153**

C Résumé étendu **159**
C.1 Introduction . . . 159
C.2 Contexte et définitions . . . 160
C.2.1 Programmation par contraintes . . . 160
C.2.2 Apprentissage de clauses dirigé par les conflits . . . 161
C.2.2.1 SAT . . . 161
C.2.2.2 Méthodes hybrides SAT/PPC . . . 162
C.3 Heuristiques de branchement pour le problème de car-sequencing . . . 163
C.4 Propagation dans une classe de problèmes de séquencement . . . 166
C.4.1 Propagation de la contrainte AtMostSeqCard . . . 166
C.4.2 Expérimentations avec le problème de car-sequencing . . . 167
C.4.3 Extensions . . . 168
C.5 Apprentissage de clauses . . . 168
C.5.1 Apprentissage de clauses pour le car-sequencing . . . 169
C.5.1.1 Explications pour AtMostSeqCard . . . 169
Explication pour l'échec . . . 169
Explications pour les règles de filtrage . . . 170
C.5.1.2 Résultats expérimentaux . . . 170
C.5.2 Le problème de redondance de clauses et l'apport de la contrainte DomainFaithfuln
C.5.3 Apprentissage dans les problèmes d'ordonnancement disjonctifs . 172
C.5.3.1 Modélisation . . . 173
Stratégie de recherche . . . 174
C.5.3.2 Apprentissage de clauses . . . 174
C.5.4 Résultats expérimentaux . . . 175
C.5.4.1 Minimisation du makespan . . . 175
C.6 Conclusion . . . 178

List of Figures

2.1 Example of implication graph 24
2.2 Cuts in the implication graph 24
2.3 Unique Implication Points in an implication graph 26
2.4 Example of an implication graph with a hybrid CP/SAT solver 29

4.1 Instantiation of an option with capacity 3/5. 58
4.2 Filtering when $d_j^{opt} \bmod p_j = 0$ 59
4.3 Filtering when $r = d_j^{opt} \bmod p_j \neq 0$ 59
4.4 Sequence of maximum cardinality obtained by `leftmost`. 65
4.5 Illustration of Lemma 4.19's proof. Horizontal arrows represent assignments. 69
4.6 Illustration of Lemma 4.20's proof. Horizontal arrows represent assignments. 70
4.7 AC on AtMostSeqCard($p = 4,\ q = 8,\ d = 12, [x_1, \ldots, x_n]$) 77

5.1 Assigning $b_1, \ldots, b_n$ 101

C.2 AtMostSeqCard$(4, 8, 12, [x_1, \ldots, x_{22}])$ 167

List of Tables

3.1 Values of the selection criteria for each option 43
3.2 Classes' scores using the parameter d^{opt} 44
3.3 Scores & Heuristic decisions 44
3.4 Comparison of heuristics averaged over propagation rules 47
3.5 Evaluation of the branching variants 49
3.6 Evaluation of the exploration variants 50
3.7 Evaluation of the selection variants 51
3.8 Evaluation of the aggregation variants 52
3.9 *Confidence* and *Significance* for each factor 52
3.10 Evaluation of the filtering variants (averaged over all heuristics) 53
3.11 Best results for filtering variants 53

4.1 Maximal cardinality instantiations. 79
4.2 Evaluation of the filtering methods (solved instances count) 81
4.3 Evaluation of the filtering methods (CPU time on solved instances) . . . 82
4.4 Evaluation of the filtering methods (search tree size on solved instances) 82
4.5 Evaluation of the filtering methods: static branching (highest success counts are in bold fonts
4.6 Evaluation of the filtering methods (dynamic branching) 85

5.1 Experimental comparison of CP, SAT, hybrid, and Pseudo-Boolean models for the car-seque
5.2 Job Shop: Lawrence (la-01-20) detailed results 118
5.3 Job Shop: Lawrence (la-21-40) detailed results 119
5.4 Job Shop: Taillard (tai01 – tai25) detailed results 120
5.5 Job Shop: Taillard (tai26 – tai50) detailed results 121
5.6 Job Shop: Taillard (tai51 – tai70) detailed results 122
5.7 Job Shop: Taillard statistics 124
5.8 Job Shop: Lawrence Statistics 124
5.9 Lower bound experiments for open Taillard instances 125
5.10 OSP results: Brucker et al. instances (j3-per0-1 – j5-per20-2) 127
5.11 OSP results: Brucker et al. instances (j6-per0-0 – j8-per20-2) 128
5.12 OSP: Brucker et al. reduced clause database results (j3-per0-1 – j5-per20-2) 129
5.13 OSP: Brucker et al. reduced clause database results (j6-per0-0 – j8-per20-2) 130
5.14 Open Shop Brucker et al. instances: Statistics 131

B.1 OSP: Gueret and Prins instances (GP03-01 – GP05-10) 154
B.2 OSP: Gueret and Prins instances (GP06-01 – GP08-10) 155
B.3 OSP: Gueret and Prins instances (GP09-01 – GP10-10) 156
B.4 OSP: Taillard instances (tai04_04_01 – tai07_7_10) 157

B.5 OSP: Taillard instances (tai10_10_01 – tai20_20_10) 158

C.1 *confiance* et *importance* pour chaque critère 165
C.2 Car-sequencing : évaluation des méthodes de filtrage 168
C.3 Apprentissage de clauses appliqué au Car-sequencing 171
C.4 Job Shop : Statistiques 176
C.5 Nouvelles bournes inférieures 178

Chapter 1

Introduction

Many real world problems involve sequencing a set of operations subject to resource constraints. Depending on the problem at hand, the objective might be optimizing an economic-related cost or simply finding satisfactory solutions. Sequencing and scheduling problems have direct applications in a variety of areas such as manufacturing, project management, and timetabling. The work presented in this thesis considers solving problems of this family in a combinatorial context. From a computational complexity theory perspective, many of these problems are NP-hard. Therefore, there is no known polynomial time algorithm for solving them.

There exist numerous techniques for solving combinatorial optimization problems ranging from heuristic to exact methods. Integer Linear Programming (ILP) is probably the best known and used approach. In this framework, the problem must be formulated as a system of linear equations. Typically, an ILP solver uses a branch-and-bound algorithm in which the lower bound is the optimal solution of the linear relaxation of the problem. Another restricted format is the one used by SAT solvers. The problem is stated using clauses, each of which being a disjunction of literals, where each literal is a propositional variable or its negation. Modern SAT solvers [95] are essentially based on the Davis-Putnam-Logemann-Loveland (DPLL) [41] algorithm augmented with resolution [112]. DPLL is a backtracking system using a simple form of inferences called Unit-propagation (UP). The integration of resolution within DPLL enables a strong inference through new clauses derived from conflicts during search. Constraint programming (CP) is another declarative paradigm for solving combinatorial problems based on a far richer language than ILP and SAT. In CP, a problem is defined with a set of relations, called constraints, operating on variables associated to sets of possible values called domains. CP solvers typically rely on propagating the constraints while exploring a search space. Constraint propagation is a fundamental concept in CP aiming at pruning the search space as much as possible. In fact, each constraint is associated to a propagator (or filtering algorithm) responsible for reducing the domains according to

some rules. In CP, we often distinguish **search** from **propagation**, and slightly more recently, from **learning**.

I report in this dissertation several contributions on each one of these aspects within constraint programming approaches to sequencing and scheduling problems. This case study strongly supports my thesis, that *modern constraint programming solvers may not underestimate any of these three aspects.*

`Search` Constraint programming solvers are typically implemented on top of backtracking systems. The search space is explored via a tree where every node corresponds to a decision restricting the search space to a smaller problem. The tree is often explored following a Depth-First Search (DFS) scheme. Whenever a failure is encountered, the solver backtracks to the last node, reverses the last decision, then resumes the exploration. The 'search' aspect in CP is related to the decisions made to explore the search tree.

A decision in CP is usually performed heuristically by shrinking a specific variable domain to a value. We often make the distinction between variable ordering and value ordering heuristics. Variable ordering heuristics are typically designed following the 'fail-first' principle [73, 129, 13]: «To succeed, try first where you are most likely to fail.». As such, one tries to prune inconsistent subtrees as soon as possible. Value ordering is usually less important and follows generally an opposite principle, called 'succeed-first' or 'promise' [61]. Indeed, the value with best chances to lead to a solution is preferred. These heuristics can be customized to the problem at hand or follow a standard scheme. Examples of standard variable ordering heuristics include: lexicographical order, minimum domain size, and maximum variable degree (i.e., how much a variable is constrained). General purpose value heuristics are less common, trivial ones (such as branching on the minimum or maximum value in the domain) are often used by default. When we have some information about the structure of the problem, however, dependent heuristics can be useful. We quote for instance [54, 130, 126, 51, 122].

Search strategies can have a dramatic effect on the overall efficiency as they guide the exploration of the search space [73, 28, 9, 63, 104]. In fact, a "bad" decision can cause the exploration to become trapped in an unsatisfiable sub-tree that can take an exponential time to explore.

`Propagation` Constraint propagation is a fundamental concept in CP aiming at reducing the search space by pruning dead-end branches. The level of pruning is usually characterized by a property called *local consistency*. The principle is that if an assignment is part of no solution of a relaxation of the problem, then it can not be part of a solution of the complete problem; it is inconsistent. Often, the problem is relaxed simply

by considering a subset of the constraints and/or variables. For instance, Arc Consistency[1](AC) [89, 90] considers constraints one at a time. Algorithms implementing AC were proposed in the 70s by Waltz [146] and Gaschnig [60]. Subsequently, 'higher' local consistencies were introduced, for instance by Montanari in [94] and Freuder in [56, 57]. The propagation methods based on local consistencies were originally "generic" in the sense that the constraint relation is part of the input. As a consequence, combining strong pruning and computational efficiency is difficult. The notion of 'global constraint' moves the relation from the input to the definition of the problem, making it far easier to reconcile these two objectives. The idea is to capture patterns occurring in many problems and to design dedicated algorithm to filter out inconsistent values for these particular cases.

There is a significant amount of work in the CP literature regarding the proposition, reformulation, and extension of global constraints [109, 110, 111, 21, 139, 96, 97, 121, 26]. The canonical example of global constraint is the AllDifferent constraint, ensuring that all variables are pair-wise distinct. Take for instance three variables x_1, x_2, x_3 subject to $x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3$. We can rewrite this as AllDifferent(x_1, x_2, x_3). Now assume that the domain for x_1 and x_2 is $\{1, 2\}$ and for x_3 is $\{1, 2, 3\}$. Enforcing AC on each constraint separately does not change the domains. However, the fact that all variables must have pairwise different values prevents the assignment of x_3 to 1 or 2 to be part of any solution. Making this inference via stronger local consistencies would take exponential time. However, it is possible to enforce AC on the AllDifferent constraint in polynomial time [109].

`Learning` When exploring a search tree, we repeat many times the same decisions. It is therefore natural to try to learn from a failure (a dead-end in the tree), in order to avoid doing the same mistake again. By definition, an exact set of decisions is never explored twice is a search tree. However, it may happen that only a part of the current branch, a 'nogood', entails a failure. When this is the case, it is possible to learn something useful in order to avoid failing more than once with the same reason.

The notion of nogood goes back originally to Stallman in the 70s [133]. The first formal adaptation to CP was proposed by Dechter in [43]. Other approaches to nogood recording were proposed later in [105, 113, 66]. In these approaches, a nogood is defined as a set of assignments that can not lead to any solution. This definition prevented learning from being more broadly used in constraint solvers. The success of nogood learning in the SAT community was, however, spectacular in the decade following Dechter's seminal work. This success is due to papers by Bayardo and Schrag [76], Marques-Silva and Sakallah [123, 124], Moskewicz et al. [95] and Zhang et al. [147]. Conflict Driven Clause

[1]The terms 'Domain Consistency' and 'Generalized Arc Consistency' are also used in the literature.

Learning (CDCL) [95] constitutes the backbone of modern SAT-solvers. In CDCL, nogoods are built by computing cuts in the graph drawn from the deductions made by Unit-propagation.

Nogood recording has gained considerable attention in the CP literature essentially during the past decade and a half [79, 78, 80, 77, 82, 35, 87, 34, 36, 37, 101, 106]. The notion of 'explanation' is the central component in these works. In order to compute a nogood, every propagation outcome should be explained in the form of a set of decisions and/or earlier propagations that logically imply it. Learning in CP has taken a new start in the past decade thanks to Katsirelos's Generalized nogoods [82, 81] and more recently to Lazy Clause Generation (LCG) [100, 101]. The latter mimics propagators in CDCL by considering them as generators of clauses. Propagators in LCG are allowed to use literals of the form $[\![x = v]\!]$, $[\![x \neq v]\!]$, $[\![x \leq v]\!]$, and $[\![x \geq v]\!]$ to express any domain change. All these types of literals can be used to explain any filtering outcome in a clausal form.

CP solvers can benefit from learning by 'discovering' new filtering rules, in the form of clauses, that propagators alone are not able to perform. Potentially, hybrid CP/SAT solvers have features coming from both approaches such as powerful propagation mechanisms, clause learning, adaptive branching, etc. However, this holds only when propagators, including those proposed for global constraints, are able to explain all their pruning.

Thesis Overview

This dissertation shows, by a thorough case-study of a class of sequencing and scheduling problems that all these aspects are important and must all be taken into account in order to design efficient solution methods.

We give a summary of the contributions presented in this thesis.

1. **An empirical heuristic study for the car-sequencing problem**

   Car-sequencing is a well known sequencing problem coming from the automotive industry. In 2005, a challenge has been organized by the French Operations Research and Decision Support Society (ROADEF[2]) for solving optimization versions of the problem provided by the RENAULT[3] automobile manufacturer [131]. In this problem, a set of cars has to be sequenced on an assembly line subject to capacity and demand constraints. Each car belongs to a class of vehicles that is defined with a set of options to install (like the sunroof and the air-conditioner).

[2] http://challenge.roadef.org/2005/en
[3] http://group.renault.com

We investigate the 'search' component for efficiently solving this problem. First, we propose a new heuristic classification for this problem. This classification is based on a set of four criteria: branching variables, exploration directions, selection of branching variables and aggregation functions for this selection. Thanks to this classification, we discovered new combinations of existing criteria leading to superior heuristics.

Based on large experimental tests, we indicate with a relatively high confidence which is the most robust strategy, or at least outline a small set of potentially best strategies. Specifically, we show that the way of selecting the most constrained option is critical, and the best choice is fairly reliably the "load" of an option, that is the ratio between its demand and the capacity of the corresponding machine. Similarly, branching on the class of vehicle is more efficient than branching on the use of an option. Finally, we show that the choice of the heuristic is often as important as the propagation method in this problem.

2. **Propagation in sequencing problems**

   Motivated by a simple observation in [111] about finding failures for the car-sequencing problem, we design a simple filtering rule called Slack-Pruning. This filtering relies on reasoning simultaneously about capacity and demand constraints. However, it is applicable with very limited branching scenarios. We propose therefore to generalize the Slack-Pruning in the form of a complete filtering for a new global constraint that we call AtMostSeqCard. This constraint can be used to model a number of sequencing problems including car-sequencing and crew-rostering.

   AtMostSeqCard can in fact be considered as a particular case of well known constraints. In [139], two algorithms designed for the AmongSeq constraint were adapted to this constraint with an $O(2^q n)$ and $O(n^3)$ worst case time complexity, respectively. In [91], another algorithm similarly adaptable to filter the AtMostSeqCard constraint was proposed with $O(n^2.log(n))$ time complexity down a branch of the search tree with an initial compilation of $O(q.n^2)$. We propose a complete filtering algorithm for this constraint with an $O(n)$ (hence optimal) worst case time complexity. Furthermore, we show that this algorithm can be adapted to achieve a complete filtering for some extensions of this constraint. In particular, the conjunction of a set of m AtMostSeqCard constraints sharing the same scope can be filtered in $O(nm)$.

   The experimental results on car-sequencing and crew-rostering benchmarks show how competitive and efficient our filtering is compared to state-of-the-art propagators.

3. **Learning in car-sequencing**

We investigate the learning aspect for solving car-sequencing instances using our filtering for AtMostSeqCard. In order to use AtMostSeqCard in a hybrid CP/SAT solver, one has to explain every single domain change made by the propagator. We therefore propose a procedure explaining AtMostSeqCard that runs in linear time complexity in the worst case. Any hybrid model using these explanations benefits from the complete filtering for this constraint along with clause learning and potentially many other CP/SAT features.

Our experiments include a variety of models with Pseudo-Boolean and SAT formulations. We show how clause learning improves the global performances in most cases. We witness a strong correlation between advanced propagation and finding solutions quickly for this problem. Moreover, for building proofs, clause learning is the most important ingredient and propagation is less useful.

4. **Revisiting lazy generation**

   We revisit in this part the lazy generation of Boolean variables for encoding the domains. The issue that we address is related to the redundancy of clauses used when lazily encoding a domain [53]. In fact, when a Boolean variable $[\![x \leq u]\!]$ has to be generated, the clauses $\neg[\![x \leq a]\!] \vee [\![x \leq u]\!]$; $\neg[\![x \leq u]\!] \vee [\![x \leq b]\!]$ are added where a and b are the nearest generated bounds to u. After adding these clauses, the clause $\neg[\![x \leq l]\!] \vee [\![x \leq u]\!]$ becomes redundant. The DomainFaithfulness constraint that we propose avoids such redundancy while ensuring the same level of consistency without any computational overhead. The novel lazy generation method is used in the next part with a large number of disjunctive scheduling instances.

5. **Learning in disjunctive scheduling**

   The last part of our contributions addresses the impact of clause learning for solving disjunctive scheduling problems. We propose a novel conflict analysis procedure tailored to this family of problems. In fact, we use a property of disjunctive scheduling allowing to learn clauses using a number of Boolean variables that is not function of the domain size. Our propositions give good experimental results and outperform the standard CP model in most cases. Furthermore, we observe a relationship between the instance size, the branching choice, and the conflict analysis scheme. Our method improved the best known lower bounds on several instances of a classic data set.

The work presented in this thesis is funded by CNRS[4] and 'midi-Pyrénées' region[5][6]. The CNRS grant was attributed to the ROC team[7] at LAAS-CNRS[8] jointly with a

[4] http://www.cnrs.fr
[5] http://www.midipyrenees.fr
[6] The region grant number is 11050449.
[7] https://www.laas.fr/public/en/roc
[8] https://www.laas.fr/public/en

Google research award on SAT-based scheduling[9]. Many parts of the dissertation has been published in the following international journals and conferences:

1. Two clause learning approaches for disjunctive scheduling. Mohamed Siala, Christian Artigues, and Emmanuel Hebrard. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31-September 4. Proceedings (to appear)*, 2015 [119]

2. A study of constraint programming heuristics for the car-sequencing problem. Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. *Engineering Applications of Artificial Intelligence*, 38(0):34 – 44, 2015 [122].

3. SAT and hybrid models of the car sequencing problem[10]. Christian Artigues, Emmanuel Hebrard, Valentin Mayer-Eichberger, Mohamed Siala, and Toby Walsh. In *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, pages 268–283, 2014 [5].

4. An optimal arc consistency algorithm for a particular case of sequence constraint. Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. *Constraints*, 19(1):30–56, 2014 [121].

5. An optimal arc consistency algorithm for a chain of atmost constraints with cardinality[11]. Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 55–69, 2012 [120].

The structure of the dissertation follows globally the contributions order. Chapter 2 introduces the formal background and all notations used throughout the thesis. We present in Chapter 3 our heuristic study for the car-sequencing problem. In Chapter 4, we investigate the propagation aspect in a class of sequencing problems. We present in Chapter 5 our threefold learning propositions: learning in car-sequencing; revisiting lazy generation; learning in disjunctive scheduling problems. Finally, we conclude the thesis in Chapter 6 and give potential future research directions.

[9] http://www2.cnrs.fr/presse/communique/2093.htm

[10] This part constitutes a joint work with Valentin Mayer-Eichberger and Toby Walsh. While the experimental observations were discussed together, the rest of the paper is organized in two clear different parts. The SAT part is solely proposed by Mayer-Eichberger and Walsh while the hybrid propositions constitutes our own contributions.

[11] The paper is awarded with an “Honorable mention”.

Chapter 2

Formal Background

Introduction

We present in this chapter the necessary background and notions used throughout the thesis. This chapter is divided in three sections: Constraint programming (Section 2.1), Boolean Satisfiability (Section 2.2), and learning in CP (Section 2.3).

2.1 Constraint Programming

Constraint programming is a framework for modeling and solving combinatorial problems. Unknowns are modeled with variables drawing their values from a discrete domain, and the possible relations between variables are represented as constraints. The Constraint Satisfaction Problem (CSP) consists in deciding whether there exists an assignment of the variables satisfying all the constraints. In this section, we formally define this formalism and introduce several notational conventions.

2.1.1 Constraint Network

2.1.1.1 Domains, Tuples, and Constraints

Let Δ be a set. We use the notation $\left|\Delta\right|$ to denote the cardinality of Δ. A `sequence` $\mathcal{S}$ defined in Δ is an ordered list of elements in Δ. We use the same notation $\mathcal{S}$ to denote a sequence $\mathcal{S}$ or the set of elements in $\mathcal{S}$.

Let $n \in \mathbb{N}^*$ and $\mathcal{X} = [x_1, \ldots, x_n]$ be a finite sequence of distinct variables A `domain` for $\mathcal{X}$, denoted by $\mathcal{D}$, is a mapping from variables to finite sets of values. For each variable x, we call $\mathcal{D}(x)$ the `domain of the variable` x. We suppose that $\mathcal{D}(x_i)$ is a finite subset

of $\mathbb{Z}$ for all $i \in [1, \ldots, n]$. We use $min(x_i)$ to denote the minimum value in $\mathcal{D}(x_i)$ and $max(x_i)$ to denote the maximum value in $\mathcal{D}(x_i)$. A domain $\mathcal{D}$ is `singleton` iff $\forall x \in \mathcal{X}$, $\left|\mathcal{D}(x)\right| = 1$. A `fail domain` is the special domain $\perp$ where all variables $x \in \mathcal{X}$ have a domain equal to $\varnothing$ (i.e., $\left|\mathcal{D}(x)\right| = 0$). The domain of a variable x is called `Boolean` iff $\mathcal{D}(x) = \{0, 1\}$. In a propositional context, we sometimes denote 0 by $false$ and 1 by $true$. When a domain $\mathcal{D}(x)$ is equal to a set of values of the form $\{l, l+1, l+2, .., u\}$ (where l and u are two integers s.t. $u \geq l$), we say that $\mathcal{D}(x)$ is a `range` domain and will be denoted by $[l, u]$. Finally, we say that v is `assigned` to the variable x iff $\mathcal{D}(x) = \{v\}$.

Given two domains $\mathcal{D}_1$ and $\mathcal{D}_2$ defined over the same sequence of variables $\mathcal{X} = [x_1, \ldots, x_n]$, we say that $\mathcal{D}_1$ is `stronger` (respectively `strictly stronger`) than $\mathcal{D}_2$ iff $\forall x, \mathcal{D}_1(x) \subseteq \mathcal{D}_2(x)$ (respectively $\forall x, \mathcal{D}_1(x) \subseteq \mathcal{D}_2(x)$ and $\exists x_i,\ \mathcal{D}_1(x_i) \subset \mathcal{D}_2(x_i)$). In this case, $\mathcal{D}_2$ is said to be `weaker` (respectively `strictly weaker`) than $\mathcal{D}_1$.

A n–`tuple` (or simply a tuple) $\tau = \langle v_1, \ldots, v_n \rangle$ is a sequence of n values. We use $\tau[i]$ to denote the value v_i. Given a tuple $\tau = \langle v_1, \ldots, v_n \rangle$ and a sub-sequence $S = [x_{s_1}, \ldots, x_{s_k}] \subseteq \mathcal{X}$, we denote by $\tau_{\pi S}$ the k–tuple $\tau' = \langle v_{s_1}, \ldots, v_{s_k} \rangle$ and is called the `projection` of τ on S.

Let $\mathcal{X}$ be a sequence of variables, $\mathcal{D}$ a domain for $\mathcal{X}$, and $\mathcal{S} = [x_1, \ldots, x_k]$ a sequence of variables in $\mathcal{X}$. A `constraint` C defined over S is a finite subset of $\mathbb{Z}^k$. S is called the `scope` of C (denoted by $\mathcal{X}(C)$) and $\left|S\right|$ is called the `arity` of C. We sometimes use the notation $C(\mathcal{S})$ to denote a constraint C having $\mathcal{S}$ as a scope. An `instantiation` of $\mathcal{S}$ is a k–tuple τ. τ is said to be:

- `consistent` for C (or `satisfying` C) if it belongs to C.
- `inconsistent` for C (or `violating` C) if it is not consistent for C.
- `valid` in $\mathcal{D}$ if $\tau[i] \in \mathcal{D}(x_i)$ for all $i \in [1, \ldots, n]$.

We distinguish two classes of constraints: firstly constraints given in extension (called also Table Constraints) where all the acceptable tuples are given explicitly in a list; secondly constraints expressed intentionally by a formula. Example 2.1 shows two possible representations for the same constraint.

Example 2.1. *A constraint defined intentionally and extensionally.*

Let x_1, x_2 and x_3 be three variables s.t. $\mathcal{D}(x_1) = \mathcal{D}(x_2) = \mathcal{D}(x_3) = \{1, 2, 3\}$. The AllDifferent$(x_1, x_2, x_3)$ *stating that the three variables should have pairwise different values can be defined intentionally by the formula: $x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3$ or extensionally using the following list of acceptable tuples $\langle 1, 2, 3 \rangle$, $\langle 1, 3, 2 \rangle$, $\langle 2, 1, 3 \rangle$, $\langle 2, 3, 1 \rangle$, $\langle 3, 1, 2 \rangle$, $\langle 3, 2, 1 \rangle$.*

All constraints used in this thesis are defined intentionally. A `constraint type` is a family of constraints sharing a general definition. The AllDifferent(x_1, x_2, x_3) constraint given in Example 2.1 is nothing but an instance of the constraint type AllDifferent where all variables in the scope should have pairwise different values. The AllDifferent constraint type is defined as follows:

Definition 2.1. AllDifferent$([x_1, \ldots, x_n])$: $x_i \neq x_j$ for all $i \neq j$.

Another typical example of constraint type is the Cardinality constraint given in Definition 2.2 where $[x_1, \ldots, x_n]$ is a sequence of Boolean variables.

Definition 2.2. Cardinality$([x_1, \ldots, x_n], d)$: $\sum_{i=1}^{n} x_i = d$

Cardinality is in fact a particular case of a more general constraint type called Pseudo-Boolean. Given a sequence of Boolean variables $[x_1, \ldots, x_n]$, a `Pseudo-Boolean` constraint[1] has the form of $\sum_{i=1}^{i=n} a_i \times x_i \blacktriangleleft k$ where $a_i, k \in \mathbb{Z}$ and $\blacktriangleleft$ is an operator in $\{\leq, \geq, =\}$.

We shall use the term constraint to denote either a constraint or a constraint type where no ambiguity is possible.

2.1.1.2 Constraint Satisfaction Problem

Definition 2.3. Constraint network

A `constraint network` (`CN`) is defined by a triplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where

- $\mathcal{X} = [x_1, \ldots, x_n]$ is a sequence of variables
- $\mathcal{D}$ is a domain for $\mathcal{X}$
- $\mathcal{C}$ is a set of constraints defined over subsets of $\mathcal{X}$.

A `solution` for a constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is an instantiation τ defined in $\mathcal{D}$ s.t. for all $C \in C$, $\tau_{\pi_{\mathcal{X}(C)}}$ is consistent for C. A constraint network is said to be `satisfiable` if it has a solution; `unsatisfiable` otherwise. We assume throughout the thesis for every variable $x \in \mathcal{X}$ that x is in the scope of at least one constraint and that x has a non-empty (initial) domain.

A `Constraint Satisfaction Problem` (`CSP`) consists of deciding whether a constraint network has a solution or not.

Since the SAT problem [39] can be considered as a particular case of CSP (the domain of each variable is $\{0, 1\}$ and each clause is considered as a constraint) then the constraint

[1] That is what Mixed Integer Programming people call a linear constraint on binary variables.

satisfaction problem is *NP-Hard* in general. Moreover, if all constraints are checkable in polynomial time, i.e., each constraint C has a function $\textsc{Check}_C : \mathbb{Z}^{|\mathcal{X}(C)|} \rightarrow \{false, true\}$ computable in polynomial time and answers «*true*» iff the tuple given in input is consistent for C, then the constraint satisfaction problem becomes *NP-Complete*.

We find mainly three approaches in the literature for solving constraint satisfaction problems: backtracking algorithms, local search and algebraic resolution. We consider in this thesis, only (complete) backtracking algorithms where the solver explores the search tree according to some strategies while performing propagation and possibly learns from conflict.

2.1.1.3 Backtracking Search

We give in Algorithm 1 a baseline backtracking Solver. One call of the recursive function TreeSearch() determines the satisfiability of the current constraint network. The final outcome will therefore indicates the satisfiability of the initial problem.

This algorithm uses a basic checking function (Algorithm 2) to find failures. The decisions are made based on Algorithm 3 «Decide()».It uses a simple form of decisions: it chooses an unassigned variable, and assigns it to a value in its domain. The decision is applied in Line 3. The choice of the next decision to make is typically performed following a variable/value heuristic.

Algorithm 1: TreeSearch()

```
1  if !Check() then
       return false
   else
       if D is singleton then
           return true
       else
           oldDomain ← D ;
2          (x, v) = Decide() ;
3          D(x) ← {v} ;
           if TreeSearch() then
               return true;
           else
               D ← oldDomain ;
               D(x) ← D(x) ∖ {v} ;
               return TreeSearch();
```

Backtracking algorithms can naturally be traced into trees. Vertices stand for calls to TreeSearch() and there is an edge between two calls if they are parent and child.

Algorithm 2: Check()

```
if ∃x s.t. D(x) is empty then
    return false ;
1 foreach C ∈ C do
    if The domain of X(C) is singleton then
        if C is not satisfied then
            return false ;
return true ;
```

Algorithm 3: Decide()

```
x ← Choose one unassigned variable ;
v ← Choose one value in D(x) ;
return (x, v) ;
```

The term 'search' is used throughout the thesis to describe any process related to the decisions made to explore the search tree.

With Backtracking Solvers, the domain will be subject to several changes. We will therefore suppose that $\mathcal{D}$ (respectively $\mathcal{D}(x)$) denotes the current domain of $\mathcal{X}$ (respectively the variable x), and $\mathcal{D}_{initial}$ (respectively $\mathcal{D}_{initial}(x)$) the initial domain (respectively of the variable x).

In constraint programming, backtracking solvers are augmented with reduction rules (known as propagators or filtering algorithms) that are usually characterized by some conditions they enforce (called local consistency). Reduction rules aim to reduce the search space using inferences based on the current state of the constraint network. When the constraints are given in intention, CP solvers typically use domain-based tightening. That is, operations on networks, keeping the same set of constraints and solutions, while returning stronger domains.

2.1.2 Constraint Propagation

2.1.2.1 Propagators

We use a similar formalism to [115, 114] for defining propagators.

Definition 2.4. Let $C([x_1, \ldots, x_k])$ be a constraint. A `propagator` f for C is a mapping from domains to domains respecting the following properties for any domain $\mathcal{D}$:

- $f(\mathcal{D})$ is stronger than $\mathcal{D}$ [Filtering property].

- Any tuple satisfying C that is valid in $\mathcal{D}$ is also valid in $f(\mathcal{D})$ [Correctness property].
- If $\mathcal{D}(x_i) = \{v_i\}\ \forall i \in [1..k]$, then $f(\mathcal{D}) = \mathcal{D}$ if $\langle v_1, \ldots, v_k \rangle$ satisfies C, and $f(\mathcal{D}) = \bot$ otherwise [Checking property] .

The scope of C is also called the scope of f and is denoted by $\mathcal{X}(f)$. We assume that f operates only on $\mathcal{X}(f)$. That is, if $f(\mathcal{D}) \neq \bot$, then $\forall x \notin \mathcal{X}(C)$, $f(\mathcal{D})(x) = \mathcal{D}(x)$.

Returning a fail domain $\bot$ is interpreted as finding a failure. That is, there is no possible way to satisfy the constraint under the domain $\mathcal{D}$. We suppose that all propagators return $\bot$ if there exists a variable whose domain is empty. By default we denote any propagator with the same name as the constraint.

Example 2.2. *Propagating* CARDINALITY$([x_1, \ldots, x_n], d)$

We show in Algorithm 4 a possible propagator for CARDINALITY$([x_1, \ldots, x_n], d)$. *This algorithm satisfies the filtering, correctness, and checking properties.*

Algorithm 4: CARDINALITY$([x_1, \ldots, x_n], d)$

```
if |{x_j | D(x_j) = {1}}| > d then
1   D ← ⊥ ;
if |{x_j | D(x_j) = {0}}| > n − d then
2   D ← ⊥ ;
if |{x_j | D(x_j) = {1}}| = d then
    foreach i ∈ {1..n} do
        if D(x_i) = {0,1} then
3           D(x_i) ← {0} ;
else
    if |{x_j | D(x_j) = {0}}| = n − d then
        foreach i ∈ {1..n} do
            if D(x_i) = {0,1} then
4               D(x_i) ← {1} ;
return D ;
```

Propagators are executed within backtracking search sequentially before taking any decision. We describe the basic *Generic Iteration Algorithm* used in [4, 114, 20] to iterate over a set of propagators. Algorithm 5 depicts a possible pseudo-code that returns a Boolean indicating if propagation finish without finding a failure.

In this algorithm, $\mathcal{F}$ is a set of propagators and *Open* is a list, initialized with $\mathcal{F}$, containing a subset of propagators to execute. Each iteration in the main loop chooses

Algorithm 5: Propagate()

$Open \leftarrow \mathcal{F}$;
while $Open \neq \varnothing$ **do**
  Choose $f \in Open$;
  $Open \leftarrow Open \setminus \{f\}$;
  $\mathcal{D} \leftarrow f(\mathcal{D})$;
  if $\mathcal{D} = \perp$ **then**
    return $false$;
  for $x \in \mathcal{X}(f)$ *s.t.* $\mathcal{D}(x)$ *changed* **do**
    $Open \leftarrow Open \cup \{g \mid g \in \mathcal{F} \wedge x \in \mathcal{X}(g)\}$;
return $true$;

a propagator f in $Open$; executes f; then updates the list $Open$. All propagators not in $Open$ and having at least one variable whose domain is changed by f will be added to $Open$. The filtering property that we used in the definition propagators makes Algorithm 5 terminates [4, 114].

The incorporation of propagators into a backtracking solver is simply done by replacing the checking function in the TreeSearch algorithm (Line 1 in Algorithm 1) with a call to Propagate(). Modern CP-Solvers deploy propagation based on Algorithm 5, however, with several improvements like the notion of idempotency and priority of propagators, among others. We shall not detail further this iterative process. We give, however, more attention on how to «measure» the filtering level between propagators.

Given two propagators f, g , we say that f is `stronger` than g iff $f(\mathcal{D})$ is stronger than $g(\mathcal{D})$ for all domain $\mathcal{D}$. In this case, we say also that f `subsumes` the filtering/pruning of g. The filtering of f and g is said to be `incomparable` iff none of them is stronger than the other. It is common in CP modeling to combine incomparable propagators together in order to prune further the search space. This was for instance the modeling choice in [11, 140, 139, 25]. There is of course a tradeoff between filtering strength and computational cost, and it is not always obvious to choose the most practical propagator. We shall draw a link to this modeling choice later when we introduce the notion of global constraint.

2.1.2.2 Local Consistency

Characterizing the level of filtering is usually associated with the notion of local consistency. A `local consistency` is a property that characterizes some necessary conditions on values (or instantiations) to belong to solutions [20]. The most known and widely used local consistency property is Arc Consistency.

Definition 2.5. Support

A `support` on a constraint C in a domain $\mathcal{D}$ is an instantiation of $\mathcal{X}(C)$ satisfying C and valid in $\mathcal{D}$.

We say that an `assignment` $x_i \leftarrow v$ has a support on a constraint $C([x_1, \ldots, x_k])$ in $\mathcal{D}$ iff there exists a support τ on C in $\mathcal{D}$ s.t. $\tau[i] = v$. Another way to look at the notion of support is that if a propagator for C prunes a value v from $\mathcal{D}(x_i)$, then necessarily $x_i \leftarrow v$ does not have a support in C (due to the correctness property).

Definition 2.6. `Arc Consistency`

A constraint $C([x_1, \ldots, x_k])$ is `Arc Consistent` (`AC`) on a domain $\mathcal{D}$ iff for all $i \in [1, k]$, any value $v \in \mathcal{D}(x_i)$ has a support on C in $\mathcal{D}$.

We shall use the term «complete filtering» to describe a propagator enforcing AC. Indeed, enforcing AC on a constraint C guarantees that every possible assignment can be part of a consistent instantiation for C.

Example 2.3. AC *on* CARDINALITY$([x_1, \ldots, x_n], d)$

The propagator depicted in Algorithm 4 enforces AC *on* CARDINALITY$([x_1, \ldots, x_n], d)$ *in* $O(n)$.

There is a close computational relationship between enforcing AC and solving. If deciding whether a given constraint C is satisfiable or not costs $O(\xi)$ time complexity, then enforcing AC on this constraint can run in $O(\xi \times \sum_{x \in \mathcal{X}(C)} \left|\mathcal{D}(x)\right|)$ by checking every possible assignment on $\mathcal{X}(C)$. The reverse sense works as follows: if AC runs in $O(\xi)$ time, then deciding the constraint runs in $O(\xi)$ and finding a solution costs $O(\left|\mathcal{X}(C)\right| \times \xi)$.

Arc Consistency is sometimes very costly to enforce. One may typically consider instead a weaker form of propagation called Bound Consistency.

Definition 2.7. Bound Support

A `bound support` on a constraint $C([x_1, \ldots, x_k])$ in a domain $\mathcal{D}$ is a k–tuple τ satisfying C s.t. $\forall i \in [1, k]$, $\tau[i] \in [min(x_i), max(x_i)]$.

Definition 2.8. `Bound consistency`

A constraint $C([x_1, \ldots, x_k])$ is `bound consistent` (`BC`) in a domain $\mathcal{D}$ iff for all $i \in [1, k]$, $min(x_i)$ and $max(x_i)$ have a bound support in $\mathcal{D}$.

Bound Consistency is obviously weaker than Arc Consistency. Note, however, that they are equivalent in some cases. Take for example the constraint $x_1 \leq x_2$. AC and BC are equivalent since for any bound support, we can easily build a support for this constraint.

We shall omit mentioning the domain $\mathcal{D}$ when describing supports, AC, and BC as it is supposed to be the current domain.

2.1.3 Decomposition & Global Constraints

2.1.3.1 Decomposition

We say that a constraint C can be `decomposed` into a finite set of constraints $\{c_1, \ldots, c_k\}$ iff for any solution τ for the constraint C^* defined by $c_1 \wedge \ldots \wedge c_k$, we have $\tau_{\pi_{\mathcal{X}(C)}}$ is a solution for C. Notice that there might exist some variables $\mathcal{X}^*_C$ in the scope of $c_1 \ldots c_k$ that do not belong to the scope of C. In this case we use the term `channeling` to denote the constraints having in their scope variables from both $\mathcal{X}(C)$ and $\mathcal{X}^*_C$.

It is known that decomposing constraints hinders propagation in general. Consider again the AllDifferent(x_1, x_2, x_3) constraint in Example 2.1 with $\mathcal{D}(x_1) = \mathcal{D}(x_2) = \{1, 2\}$ and $\mathcal{D}(x_3) = \{1, 2, 3\}$. Enforcing AC on each constraint of the decomposition would leave the domain as it is whereas there is no possible way to satisfy the original constraint when assigning 1 or 2 to x_3. In this example, the constraints $x_1 \neq x_2$, $x_2 \neq x_3$, and $x_1 \neq x_3$ are AC whereas AllDifferent(x_1, x_2, x_3) is not. Achieving AC on AllDifferent(x_1, x_2, x_3) in this case reduces the domain of x_3 to $\{3\}$.

There exists, however, a few particular cases where the decomposition maintains AC. We use in this thesis two known cases where AC on a constraint C is equivalent to enforcing AC on a decomposition. The first case, described below, is related to the notion of Berge acyclicity in the constraint graph, whereas the second case is related to the notion of monotonicity (a constraint of this type is studied in Section 4.2).

Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CN. The `constraint graph` of $\mathcal{P}$ is a hypergraph $\mathcal{H}_\mathcal{P}$ in which one associates each variable to a node and each constraint scope to an hyperedge. A `Berge cycle` [18] in $\mathcal{H}_\mathcal{P}$ is a sequence $[C_1, x_1, .., C_k, x_k, C_{k+1}]$ $(k > 1)$ where : $x_1 \ldots x_k$ are distinct variables; $C_1 \ldots C_k$ are distinct constraints; C_{k+1} is C_1; and x_i is in $\mathcal{X}(C_i)$ and $\mathcal{X}(C_{i+1})$. $\mathcal{H}_\mathcal{P}$ is said to be Berge cyclic if it contains a Berge cycle; and Berge acyclic otherwise. Notice than if two distinct variables x_1 and x_2 are in the scope of two constraints C_1 and C_2, then the constraint graph is necessarily Berge cyclic. The sequence $[C_1, x_1, C_2, x_2, C_1]$ is a Berge cycle in this case.

Let C be a constraint that can be decomposed into a finite set of constraints $\{c_1, .., c_k\}$. If the constraint graph of the CN formed by $c_1, \ldots, c_k$ is Berge acyclic, then C is AC iff c_i is AC for all $i \in [1, k]$ [14].

2.1.3.2 Global Constraints

The notion of global constraint [27, 137] is a fundamental concept in CP. We consider the definition of a `global constraint` as a constraint type defined over a non-fixed

number of variables. In practice, they represent sub-problems or patterns occurring in many problems.

The AllDifferent constraint given in Definition 2.1 is a typical example. There is a wide range of problems in which one can use AllDifferent. Sudoku for instance is a typical example where one can post an AllDifferent constraint for each row, column, and square. The same constraint can also be used in scheduling problems with unary resources. If all tasks of a machine $\mathcal{M}$ have a duration of one unit of time, then the resource constraint related to $\mathcal{M}$ is nothing but an AllDifferent constraint on the variables representing the start time of each task.

A global constraint is usually introduced in the CP literature together with a polynomial time filtering algorithm. The fact that they occur in several applications has attracted a lot of attention to develop special-purpose propagators making them practical tools for tackling hard combinatorial problems. The global constraint catalog [2] [16] contains descriptions (in terms of graph properties, automata, or first order logical formula) for more than 400 global constraints. Such a rich language may sometimes make it difficult to make the best the modeling choices.

We give in the following the definition of the `Global Cardinality Constraint` (Gcc) and the `Global Sequencing Constraint` (Gsc) that are used throughout this thesis.

Let $[x_1, \ldots, x_n]$ be a sequence of variables and $\Delta = \bigcup_{i=1}^{n} \mathcal{D}(x_i)$. Let low and upp be two mappings on integers such that $low(j) \leq upp(j)$ for all j. The Global Cardinality Constraint Gcc [110] is defined as follows:

Definition 2.9. $\textsc{Gcc}(low, upp, [x_1, \ldots, x_n]) : \bigwedge_{j \in \Delta} low(j) \leq \left|\{i \mid x_i = j\}\right| \leq upp(j)$

$\textsc{Gcc}(low, upp, [x_1, \ldots, x_n])$ limits the occurrences of any value $j \in \Delta$ in the sequence $[x_1, \ldots, x_n]$ to be in the interval $[low(j), upp(j)]$. It can be seen as a generalization of AllDifferent if we restrict the intervals $[low(j), upp(j)]$ to be $[0, 1]$. Arc Consistency on Gcc can be enforced in $O(\left|\Delta\right|.n^2)$ [110]. Quimper et al. showed a Bound Consistency algorithm for this constraint running in $O(t + n)$ where t is the time to sort the bounds of the domains of the variables [107].

The Global Sequencing Constraint Gsc is defined with a conjunction between a Gcc and a chain of Among constraints. An Among constraint (Definition 2.10) limits the occurrences of values of a set of integers ν to be bounded between two integer l and u $(l < u)$.

Definition 2.10. $\textsc{Among}(l, u, [x_1, \ldots, x_q], \nu) \Leftrightarrow l \leq \left|\{i \mid x_i \in \nu\}\right| \leq u$

The Gsc constraint is defined as follows:

[2] The latest version is available via `http://sofdem.github.io/gccat/`.

Definition 2.11. $\textsc{Gsc}(l, u, q, low, upp, [x_1, \ldots, x_n], \nu)$:

$$\bigwedge_{i=0}^{n-q} \textsc{Among}(l, u, [x_{i+1}, \ldots, x_{i+q}], \nu) \wedge \textsc{Gcc}(low, upp, [x_1, .., x_n])$$

We mention now an important complexity property related to AC for global constraints. For a more complete background on the subject, we refer the reader to [24].

Definition 2.12. AC-poly-time [24]

An AC-poly-time decomposition of a global constraint is a decomposition where AC can be enforced in polynomial time w.r.t. the size of the original constraint and domains.

Theorem 2.13. *[24]*

If enforcing AC *on a global constraint is NP-Hard, then there is no* AC*-poly-time decomposition of the original constraint that achieves* AC *on* C*.*

Theorem 2.13 gives a clear statement when to consider lower filtering compared to AC. Obviously, one does not use in practice AC algorithms when they are NP-Hard. Instead, lower filtering (usually BC) is typically used is this case since any decomposition would hinder propagation anyway. Arc Consistency on Gsc for instance is NP-Hard [22]. Régin and Puget proposed a reformulation of this constraint into a set of Gcc constraints. Their filtering is therefore hindering propagation.

The modeling choice between several global constraints should take into account the filtering level to enforce along with the complexity of such propagation. This tradeoff is often the motivation behind proposing new global constraints. The latter are usually either extensions or particular cases of other global constraints that might occur in a number of applications. It should be noted that the more general is a constraint, the higher the complexity of enforcing a given level of consistency on it. For instance, enforcing AC on Gcc can be done in $O(\left|\Delta\right|.n^2)$ time [110] while enforcing AC on AllDifferent takes $O(\left|\Delta\right|.n^{1.5})$ time [109]. Sometimes, generalizing constraints can make them intractable. For example, consider Gcc in which, instead of integer bounds of occurrences (i.e.,$low(j)$ and $upp(j)$ for all $j \in \Delta$), we have variables. That is, the occurrence of each value $j \in \Delta$ has to be equal to a variable δ_j. AC for this constraint is NP-Hard to enforce [108].

2.1.4 Search

The search aspect is related to the decisions made to explore the search tree. A decision in CP is often performed heuristically by reducing a specific variable domain to a value (in a similar way to Algorithm 3). Variable ordering heuristics are typically designed

following the 'fail-first' principle [73, 129, 13]: «To succeed, try first where you are most likely to fail.». As such, one tries to avoid inconsistent subtrees as soon as possible. Value ordering is usually less important and follows generally an opposite principle, called 'succeed-first' or 'promise' [61]. Indeed, the value with best chances to lead to a solution is preferred. These heuristics can be customized to the problem at hand or follow a standard scheme. Examples of problem dependent heuristics can be found in [54, 130, 126, 51, 122]. Examples of standard variable ordering heuristics include: lexicographical order, minimum domain size, and maximum variable degree (i.e., how much a variable is constrained). General purpose value heuristics are less common, trivial ones (such as branching on the minimum or maximum value in the domain) are often used by default.

Search strategies can have a dramatic effect on the overall efficiency as they guide the exploration of the search space [73, 28, 9, 63, 104]. In fact, a "bad" decision can cause the exploration to become trapped in an unsatisfiable sub-tree that can take an exponential time to explore.

2.1.4.1 Boosting Search through Randomization and Restarts

The authors of [67], have shown that the 'hardness' of finding solutions is not entirely related to the instance at hand, but rather to the combination 'instance $\oplus$ deterministic algorithm'. This observation is supported by the efficiency gain witnessed when adding randomization to a deterministic search algorithm. Randomization is typically performed when making decisions. For instance, one can use randomization when tie breaking choices that rank equally with respect to the heuristic at hand. Another example is to choose randomly across a number of best choices.

It was observed int [67] that at any time during the experiment there is a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before [67]. This phenomenon explains that runtime distributions on random instances, or on random runs for a given instance, are often `heavy tailed`.

`Restarts` has been proposed as a solution to avoid this phenomenon. The search is bounded by a given cutoff. Once the cutoff reached, the exploration is stopped, and restarted from the search root. One usually uses the number of failures as a restart cutoff. Using randomization when branching on nodes makes the explored trees differ from restart to restart.

We find in the literature two common restart policies. A `geometric` restart [144] uses a limit of $b \times f^{k-1}$ for the k^{th} restart where b is called a base and f is called a factor. A `Luby` policy [88], on the other hand, follows the sequence 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1,

2, 4, 8, ... multiplied by a base b. The i^{th} element of the luby sequence ψ_i is defined recursively by the formula:

$$\begin{cases} 2^{k-1} \ if \ \exists k \in \mathbb{N}, i = 2^k - 1 \\ \psi_{i-2^{k-1}+1} \ if \ \exists k \in \mathbb{N}, 2^{k-1} \leq i < 2^k - 1 \end{cases} \tag{2.1}$$

2.2 Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) is the question of deciding a Boolean expression defined in a Conjunctive Normal Form. That is, a conjunction of clauses, each of which is a disjunction of literals, and each literal represents a Boolean variable or its negation. As such, SAT can be considered as a particular case of CSP. This restriction has made SAT solvers benefit from several enhancements that are not available in pure constraint programming solvers.

We describe in this section the organization of modern SAT solvers by formally defining this formalism and introducing some related notions.

2.2.1 A Background on Propositional Logic

An `atom` a is a propositional (i.e., Boolean) variable. A `literal` p denotes either an atom a or its negation $\neg a$. The former is called positive literal whereas the latter is called negative literal. We use the notations a and $\neg a$ for each atom a to denote its positive and negative literals respectively. We extend the negation operator to literals following the rule $\neg\neg p = p$. A `clause` c is a disjunction of literals $p_1 \vee \ldots \vee p_k$. We suppose, without loss of generality, that all literals in a clause are pairwise distinct and there is no literals p, $\neg p$ in the same clause. We use the two notations: $p_i \in c$ for any literal appearing in the clause c; and $\left|c\right|$ as the size of the clause (i.e., the number of literals in the disjunction). Let c, c' be clauses and p be a literal. We denote by: $p \vee c$ the clause obtained by the disjunction of p with all literals in c; and $c \vee c'$ the clause defined by the disjunction of all literals in c and c'. Finally, a propositional formula Φ is given in a `Conjunctive Normal Form` (`CNF`) if it is defined by a conjunction of clauses $c_1 \wedge .. \wedge c_n$.

With that being defined, a CNF can be considered as a constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ s.t. $\mathcal{X}$ is the set of atoms, and $\mathcal{C}$ is the set of clauses. The Boolean Satisfiability Problem (SAT) is to decide the satisfiability of a CNF formula [39].

A literal p is said to be:

- *true* iff p is positive and its atom is assigned to the value 1 or p is negative and its atom is assigned to the value 0.
- $false$ iff $\neg p$ is *true*.

A literal p is said to satisfy a clause c iff $p \in c$ and p is *true*. Conversely, a literal p is said to strengthen a clause c iff $p \in c$ and p is $false$.

A clause c is satisfied iff there exists a literal satisfying c. Similarly, c is violated iff $\forall p \in c$, p falsifies c. A clause c is called unit when it contains exactly one unassigned literal and the rest strengthen c. Finally, an empty clause $\bot_{clause}$ is a clause with no literals.

2.2.2 Conflict Driven Clause Learning

`Conflict Driven Clause Learning` (`CDCL`) [123, 124, 95, 48] is a state-of-the-art complete algorithm underlying most modern SAT solvers. CDCL is essentially based on the `Davis-Putnam-Logemann-Loveland` (`DPLL`) [41] algorithm augmented with `resolution` [112]. DPLL is a backtracking system using one type of propagation called `Unit-propagation` (`UP`).

We associate two values to each assigned literal p: `level(`p`)` represents the number of decisions in the path between the root and the node in which p is assigned; and `rank(`p`)` represents the rank of p in the sequence of assignments of its level, in chronological order. We shall start counting *rank* from 0 at each level. Therefore, any decision has a rank equal to 0.

We introduce the notion of propagation rule as a mechanism to describe the outcome of some propagation.A `propagation rule` is a logical implication of the form $\Psi \Rightarrow p$ where Ψ is a conjunction of literals and p is either a literal or a failure $\bot$. Ψ is said to be the explanation for (propagating) p and will be denoted by `explain(`p`)`.

UP triggers propagation in two possible ways. First, whenever a clause c becomes unit, it enforces the only unassigned literal p in c to be *true* since it is the only possible way to satisfy c. The propagation rule describing this filtering is $\bigwedge_{q \neq p \in c} \neg q \Rightarrow p$. Second, when all literals in a clause c falsifies c, a failure $\bot$ is triggered (c is said to be the `conflict clause` in this case). We use $\bigwedge_{q \in c} \neg q \Rightarrow \bot$ to describe this propagation. If q is the last propagated literal in the conflict clause, then we call q and $\neg q$ `conflicting literals`.

Finally a `nogood` is a conjunction of literals sufficient to make the CN unsatisfiable if they are *true*. It follows from any propagation rule of the form $\Psi \Rightarrow \bot$ that Ψ is a nogood.

As previously said, modern SAT solvers implement Conflict Driven Clause Learning (CDCL) [123, 124, 95, 48], i.e., essentially DPLL in which new clauses are learnt from failures [123, 124]. However, CDCL solvers feature many enhancements, we describe the most important.

2.2.2.1 Conflict Analysis

Whenever a failure occurs during search, a new nogood is computed. The latter is translated into a clause that will be added to the base and used to perform non-chronological backtrack (known with the term `backjump`). The whole machinery is called `conflict analysis` and is based on the notion of cuts in the `Implication Graph`

Definition 2.14. Implication graph

The implication graph $G(N, E)$ is a directed acyclic graph built as follows:

- Each assigned literal is associated to a vertex in N.
- There exists a directed edge in E from p to q ($p \neq q$) if $p \in explain(q)$.
- When a failure is detected by a clause c, we first add a vertex q s.t. q is the conflicting literal in c. Then, any literal $p \neq q \in \neg c$ is associated to a directed edge going from p to q. Finally, there is a special vertex $\perp$ having edges coming from q and $\neg q$.

From Definition 2.14, one can observe that all decisions have no incident edge in $G(N, E)$.

We give an example of implication graph. Suppose that the set of clauses contains the following five clauses, among others: (1) $\neg a \vee \neg b$; (2) $b \vee h \vee c$; (3) $\neg g \vee \neg c \vee \neg d$; (4) $\neg c \vee d \vee \neg e$; and (5) $\neg c \vee e$.

We suppose that: g and a are true and correspond to decisions made at levels 4 and 9 respectively; $\neg h$ is propagated at level 6; the propagation after assigning a follows the following propagation order: clause 1 propagates $\neg b$, clause 2 propagates c, clause 3 propagates $\neg d$, clause 5 propagates e, and clause 4 triggers failure.

We show a part of the implication graph leading to failure. A node $p^{l \bullet r}$ in the implication graph stands for the assignment of p as the r-th consequence of the l-th decision (i.e., $l = level(p)$ and $r = rank(p)$). Note that decisions will always have the form of $p^{n \bullet 0}$ since their rank is always equal to 0. Grey vertices are decisions while white vertices are propagated literals. The conflicting literals in this example are e and $\neg e$.

The implication graph is built while searching by recording for each assigned literal p its reason, that is, $explain(p)$ if p is propagated and *null* otherwise (i.e., p is a decision).

FIGURE 2.1: Example of implication graph

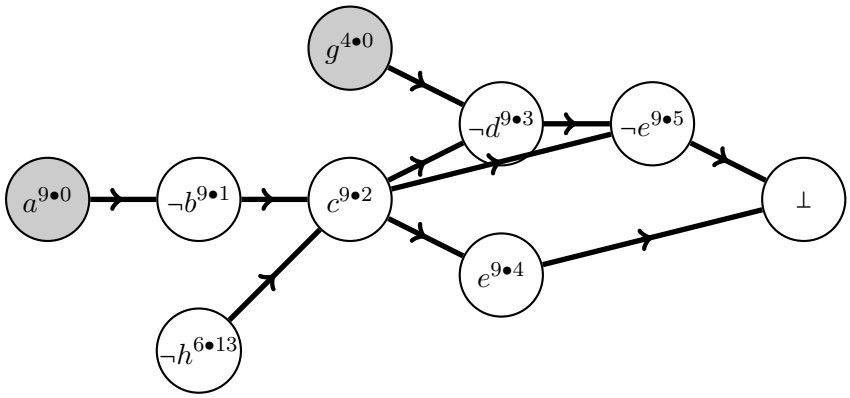
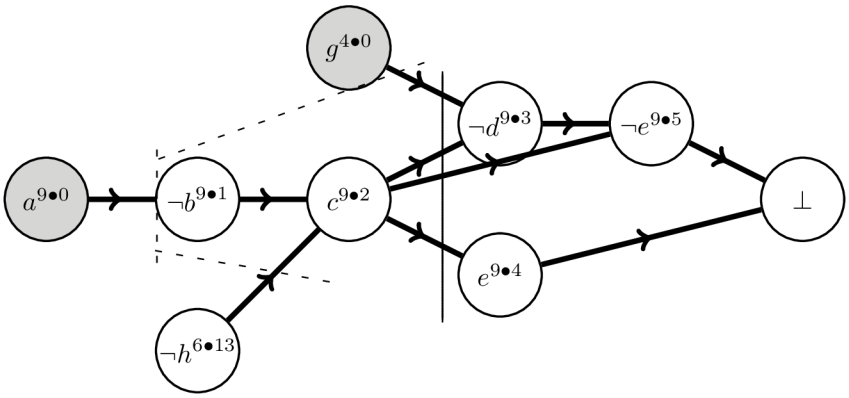


FIGURE 2.2: Cuts in the implication graph



During conflict analysis, new nogoods will be produced. They correspond to cuts in the implication graph. We define a `cut` as a bipartition of $G(N, E)$. We distinguish two disjoint sets: the conflict part and the reason part. The `conflict part` always contains the $\bot$ vertex whereas the `reason part` contains all decisions. The conjunction of literals in the reason side that have an edge going to at least one literal in the conflict side leads to a contradiction. It is therefore a nogood. The clause equal to the negation of the nogood is therefore logically implied by the CN. Different cuts will therefore produce different clauses. We show in Figure 2.2 two different cuts for the implication graph used in the previous example of Figure 2.1. The two cuts correspond to the nogoods $c \wedge g$ and $g \wedge a \wedge \neg h$. Hence we can learn the clauses $\neg c \vee \neg g$ and $\neg g \vee \neg a \vee h$.

When a nogood $\neg c$ is identified, c is firstly learnt and secondly used to perform non-chronological backtracking (called `backjumping`). The condition for backjumping is that

c contain only one literal p in the latest level. In this case, c can be seen as $\Psi \implies p$ where $\Psi = \neg p_1 \wedge .. \wedge \neg p_n$ s.t. $p_i \neq p \in c$ and p_i are assigned at previous levels. We first backtrack to the greatest level between $level(p_i)$, then c directly propagates p. For instance, with the clause $\neg g \vee \neg a \vee h$ in the previous example, we backtrack to level 6 and assign $\neg a$ to *true* immediately.

Learning schemes are essentially differentiated by their methods for building cuts. The first method proposed in the literature is the one used in the relsat system [76] where cuts are built s.t. the literal in the last level is always the latest decision. Modern SAT-solvers, however, use any Unique Implication Point (UIP), that is, a dominator of the conflicting literals in the last level.

Definition 2.15. `Domination` in the Implication Graph [147]

A vertex V dominates another vertex V' in the implication graph if any path from the decision vertex of the level of V to V' has to go through V.

Definition 2.16. Unique Implication point [147] A `Unique Implication point` (`UIP`) is a vertex in the current level that dominates both conflicting literals.

Choosing cuts based on UIPs was originally proposed in Grasp [123, 124]. As we can see in Figure 2.3, several UIPs can be found in a same implication graph. In this figure, every path from the latest decision (i.e., a) to the conflicting literals e and $\neg e$ has to pass through a, b and c. Three different UIP cuts are therefore possible in this example.

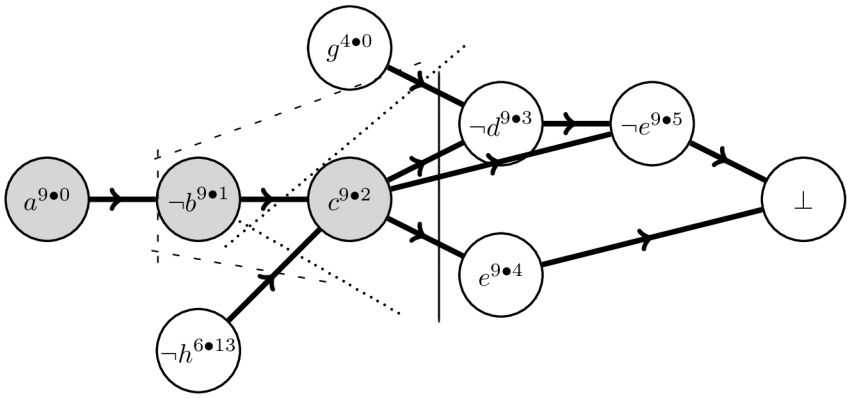
Among the several possibilities, there exists one UIP cut that is particularly interesting. By considering all UIPs by their reverse order of propagation, the first one (i.e., the nearest to the conflict), called the first Unique Implication point (`1-UIP`), guarantees the best backjump level (i.e. the nearest to the root). 1-UIP cuts have been shown to be extremely efficient in practice [147] and are widely used in modern SAT Solvers.

Algorithm 6: 1-UIP

$d \leftarrow current\ level$;
$\Psi \leftarrow explain(\bot)$;
while $\left|\{q \in \Psi \mid level(q) = d\}\right| > 1$ **do**
1 $\quad p \leftarrow \arg\max_q(\{rank(q) \mid level(q) = d \ \wedge \ q \in \Psi\})$;
2 $\quad \Psi \leftarrow \Psi \cup \{q \mid q \in explain(p) \wedge level(q) > 0\} \setminus \{p\}$;
return Ψ ;

Algorithm 6 shows a possible algorithm for computing the 1-UIP nogood. It returns a nogood Ψ having one literal assigned at the last decision level d. Ψ is initialized with the explanation of failure. Each iteration in the main loop substitutes a literal in Ψ with its explanation. The choice of the next literal to substitute is performed at Line 1 with the literal of Ψ assigned at the last decision level and of maximum rank.

FIGURE 2.3: Unique Implication Points in an implication graph



Algorithm 6 is usually implemented with a worst case time complexity of $O(\xi)$ where ξ is the number of propagated literals in the last level. Indeed this requires exploring the sequence of assigned literals in the latest level starting from the last propagated literal.

It should be pointed out that modern SAT solvers usually try to reduce the final nogood Ψ [132]. A common strategy of reduction is to eliminate literals having their explanation in Ψ.

In the example used in Figure 2.1, the 1-UIP clause is $\neg g \vee \neg c$, and the literal c is the first UIP. The solver then backtracks to the level of assigning g (i.e., 4 in this case), assigns c to $false$, then continues the exploration. We show a step-by-step execution of algorithm 6 for building the nogood in this example.

1. $\Psi \leftarrow c \wedge \neg d \wedge e$
2. $p \leftarrow e$
3. $\Psi \leftarrow c \wedge \neg d$ (i.e., $\Psi \leftarrow \Psi \cup \{c\} \setminus \{e\}$)
4. $p \leftarrow \neg d$
5. $\Psi \leftarrow c \wedge g$

We use the term `clause database` in the rest of this thesis to denote the set of learnt clauses.

2.2.2.2 2-Watched Literals

Unit-propagation is typically implemented with lazy data structures. The 2-watched literals [95, 62] is the most known lazy propagation scheme used with modern SAT solvers. Briefly, the idea is associate each clause c to two literals $p, q \in c$ (said to be watching c). No propagation check is needed for c as long as the two literals watching c are unassigned. Without loss of generality, if p becomes assigned, but strengthen the clause, Unit-propagation looks for a new unassigned literal to watch c. If no such literal exists, Unit-propagation assigns q to *true* if q is unassigned and triggers failure if q is assigned but falsifies c.

2.2.2.3 Activity-Based Branching

One of the most known and widely used variable ordering heuristic in SAT solvers is the so-called `Variable State Independent Decaying Sum` (`VSIDS`) [95]. This heuristic has been shown to be extremely efficient is practice. One can find a variety of implementations for VSIDS. The first description of a VSIDS ordering follows the following steps [95]:

- Each literal has an 'activity' value initialized to 0.
- Whenever a literal occurs in a learnt clause, its activity is incremented.
- The (unassigned) literal with the highest activity is chosen at each decision.
- All activity values are periodically divided by a constant so that literals in recent learnt clauses are preferred.

2.2.2.4 Clause Database Reduction

Learning clauses without controlling the clause database size can lead to a memory explosion with the increasing number of clauses. This explosion is likely to increase the amount of time required for enforcing UP. Several deletion strategies have been proposed in the literature [124, 95, 48, 7, 75]. One usually prefers the shortest clauses, or the most 'active' clauses. The latter are selected based on literal activities computed along with VSIDS. It is important to note that clauses responsible for propagating some literals in the current branch should not be deleted as they might be needed during conflict analysis.

2.2.2.5 Restarts

We have discussed in Section 2.1.4.1 the importance of restarts for combinatorial algorithms in general. CDCL can benefit further from restarts by using the learnt clauses and activity counting. The learnt clause prevents previous branches to be explored twice. Moreover, the activity of literals can be extremely useful to bring information from previous restarts to the search strategy.

2.3 Clause Learning in CP

When exploring a search tree, we repeat many times the same decisions. It is therefore natural to try to learn from failures, in order to avoid doing the same mistake again. By definition, an exact set of decisions is never explored twice is a search tree. However, it may happen that only a part of the current branch entails a failure. When this is the case, it is possible to learn something useful in order to avoid failing more than once with the same reason.

We have seen in the previous section how nogoods are derived from conflicts in SAT solvers. Nogood learning in CP, however, predates CDCL. Indeed, the notion of nogood goes back originally to the 70s in the seminal work of Stallman and Sussman [133]. And the first formal adaptation to CP was proposed by Dechter in [43]. A nogood (or conflict set in [43]) is defined as a set of assignments that can not lead to any solution. Other approaches to nogood recording were proposed later in [105, 113, 66].

Nogood learning in CP had not the same impact of CDCL in SAT solvers in the early days. It has gained, however, considerable attention progressively during the last decade and a half [79, 78, 80, 77, 82, 35, 34, 36, 37, 101, 106]. The notion of 'explanation' is the central component in these works. In order to compute a nogood, every propagation outcome should be explained in the form of a set of decisions and/or earlier propagations that logically imply it.

Learning in CP has taken a new start in the past decade thanks to Katsirelos's `generalized nogoods` [82, 81]. A generalized nogood extends the notion of nogood to contain both assignments and non-assignments (i.e., pruning). `Lazy clause generation`[3] (`LCG`) [100, 101] is a similar approach to Katsirelos'. However, propagators in LCG are allowed to use literals of the form $[\![x = v]\!]$, $[\![x \neq v]\!]$, $[\![x \leq v]\!]$, and $[\![x \geq v]\!]$ to express domain changes. All these types of literals can be used to explain domain reductions in a clausal form. The explanations are used essentially to mimic CDCL.

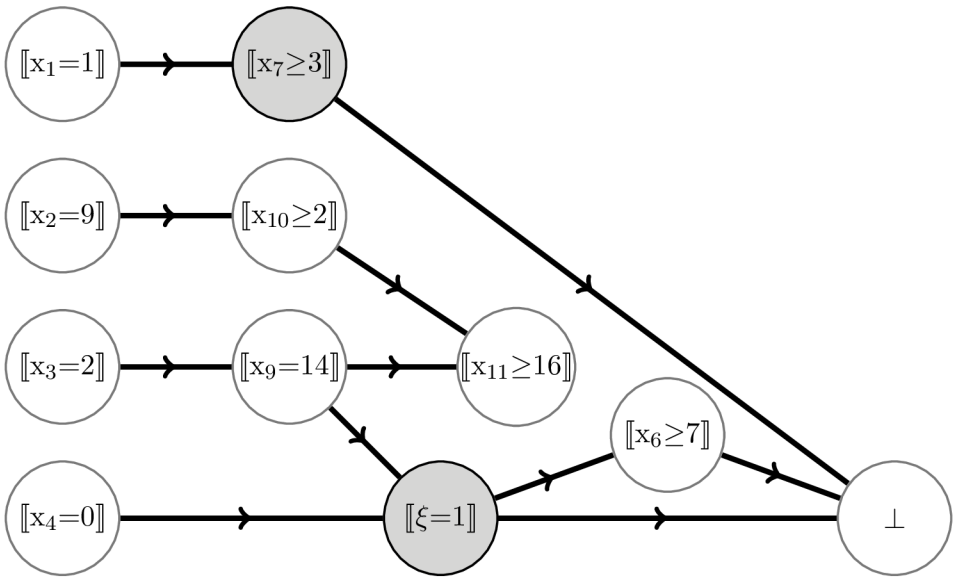
[3] Note that the term "lazy" might refer to completely different notions depending on the context (such as Integer Linear Programming). We therefore insist to mention that we use this term to respect the exact terminology used in [100, 101, 53, 52].

We give an illustrative example. Let ξ be a Boolean variable and $x_1 \dots x_{11}$ be variables with a domain defined by: $\mathcal{D}(x_1) = [1,30]$, $\mathcal{D}(x_2) = [9,30]$, $\mathcal{D}(x_3) = [0,3]$, $\mathcal{D}(x_4) = [0,30]$, $\mathcal{D}(x_5) = [24,50]$, $\mathcal{D}(x_6) = [5,10]$, $\mathcal{D}(x_7) = [2,10]$, $\mathcal{D}(x_8) = [9,30]$, $\mathcal{D}(x_9) = [13,16]$, $\mathcal{D}(x_{10}) = [0,3]$, and $\mathcal{D}(x_{11}) = [15,100]$. These variables are subject to the following constraints: (1) $x_1 + x_7 \geq 4$, (2) $x_2 + x_{10} \geq 11$, (3) $x_3 + x_9 = 16$, (4) $x_5 \geq x_8 + x_9$, (5) $\xi \leftrightarrow (x_9 - x_4 = 14)$, (6) $\xi \rightarrow (x_6 \geq 7)$, (7) $\xi \rightarrow (x_6 + x_7 \leq 9)$, and (8) $x_{11} \geq x_9 + x_{10}$. Observe that no pruning happens in the initial state of the problem. Now consider the following decisions in the chronological order:

1. Assign x_1 to 1: The only subsequent propagation is to make 3 the lower bound of x_7 by constraint (1), i.e., $[\![x_7 \geq 3]\!]$.
2. Assign x_2 to 9: Constraint (2) propagates $[\![x_{10} \geq 2]\!]$.
3. Assign x_3 to 2: In this case, constraint (3) enforces $[\![x_9 = 14]\!]$, then constraint (8) propagates $[\![x_{11} \geq 16]\!]$.
4. Assign x_4 to 0: Constraint (5) propagates ξ to 1. Constraint (6) enforces $[\![x_6 \geq 7]\!]$. And constraint (7) finds failure.

The implication graph corresponding to this example is shown in Figure 2.4. The solver learns the new clause $[\![x_7 \leq 2]\!] \vee [\![\xi = 0]\!]$ following the 1-UIP scheme, backtracks to the first level, assigns ξ to 0, and resumes the exploration of the search space.

FIGURE 2.4: Example of an implication graph with a hybrid CP/SAT solver



CP-solvers can benefit from clause leaning by 'discovering' new filtering rules, in the form of clauses, that propagators alone are not able to perform. In the previous example for

instance, when enforcing $[\![x_7 \geq 3]\!]$, no filtering suggest that ξ should be assigned to 0. It is only by means of the learnt clause $[\![x_7 \leq 2]\!] \vee [\![\xi = 0]\!]$ that the solver performs such filtering. Hybrid CP/SAT solvers may combine features coming from both approaches such as powerful propagation mechanisms, clause learning, and adaptive branching. However, this holds only when propagators, including those proposed for global constraints, are able to explain all their pruning.

In the rest of this section, we cover in more details the principles of Lazy Clause Generation [101, 100, 53] as it is the framework that we use to design the approach introduced in this dissertation. The latest architecture [53] is implemented on top of a CP-solver augmented with most SAT features (clause learning, non-chronological backtrack, adaptive-Branching, etc).

2.3.1 A Baseline Hybrid Solver

2.3.1.1 Domain Encoding

The atoms on which the learning is performed are related to some propositional facts about the variable domains. These atoms are channeled through a set of clauses to ensure a correct domain representation. The most known domain encodings in the literature are the `direct encoding` [42, 145] and the `order encoding` [40, 135] .

We assume without loss of generality that x is a variable with a domain $\mathcal{D}(x) = \{v_1, v_2, \ldots, v_k\}$ where $v_i < v_{i+1}$ for all $i \in [1, k-1]$.

The Direct Encoding The direct encoding uses k atoms denoted by $[\![x = v_j]\!]$ ($j \in [1, k]$) s.t. $[\![x = v_j]\!]$ is semantically equivalent to assigning x to v_j. Two types of clauses are used to represent the different relations between these atoms.

- at-least-one: a clause is used to express the fact that x has to be assigned to a value:

$$[\![x = v_1]\!] \vee [\![x = v_2]\!] \vee \ldots \vee [\![x = v_k]\!]$$

- at-most-one: $\frac{k^2-k}{2}$ clauses are used to express the fact that x has to be assigned to only one value.

$$\forall l < h \in [1, k],\ \neg[\![x = v_l]\!] \vee \neg[\![x = v_k]\!].$$

The Order Encoding Here also k atoms are used, however, each atom (denoted by $[\![x \leq v_j]\!]$, $j \in [1,k]$) is equivalent to have an upper bound for x less than v_j. As for the domain clauses, $k-1$ clauses are used as follows:

$$\forall j \in [1, k-1],\ \neg[\![x \leq v_{j+1}]\!] \vee [\![x \leq v_j]\!]$$

To make the notation lighter, we denote by $[\![x \neq v]\!]$ the literal $\neg[\![x = v]\!]$ and $[\![x \geq v]\!]$ the literal $\neg[\![x \leq v-1]\!]$.

Following lazy clause generation, we use these two types of atoms together. In this case, the domain related clauses have to ensure a complete domain representation between these atoms. For instance, if $[\![x \leq 3]\!]$ is true, then $[\![x = 4]\!]$ and $[\![x = 5]\!]$ should be set to false. A clausal representation of such relationships can be found in [100] under the term `Domain Faithfulness` (which is essentially a channeling between the direct and order encoding). Without loss of generality, for every variable x s.t. $D(x) = [l,u]$, we have the following clauses (referenced later by $\textsc{dom}(x)$):

1. $\neg[\![x \leq d]\!] \vee [\![x \leq d+1]\!],\ \forall d \in [l, u-1]$
2. $\neg[\![x = d]\!] \vee [\![x \leq d]\!],\ \forall d \in [l, u-1]$
3. $\neg[\![x = d]\!] \vee \neg[\![x \leq d-1]\!],\ \forall d \in [l+1, u]$
4. $[\![x = l]\!] \vee \neg[\![x \leq l]\!]$
5. $[\![x = d]\!] \vee \neg[\![x \leq d]\!] \vee [\![x \leq d-1]\!],\ \forall d \in [l+1, u]$
6. $[\![x = u]\!] \vee [\![x \leq u-1]\!]$

2.3.1.2 Solver Description

All domain related atoms and clauses described above are generated before search. The UP engine acts as a global constraint whose scope contains all these atoms, and whose semantics is given by the set of domain related clauses. During search, every propagator is expected to explain each domain change it performs. Since every domain change must be represented by a literal, propagators are limited to changes that can be expressed as conjunctions of literals of the following types:

- Assignment: an assignment operation assigns x to a value v in its domain, written $\mathcal{D}(x) \leftarrow \{v\}$.
- Pruning: conversely to assignments, a pruning operation removes a value v from a variable domain, written $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \setminus \{v\}$.

- Upper bound tightening: an upper bound tightening operation changes the upper bound of x to a value $u \in [min(x), max(x) - 1]$, written $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \cap]-\infty, u]$.
- Lower bound tightening: a lower bound tightening operation changes the lower bound of x to a value $l \in [min(x) + 1, max(x)]$, written $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \cap [l, +\infty]$.

The notion of 'explanation' and 'propagation rule' that we introduced in Section 2.2.2 for clauses are extended to propagators as follows. First, each domain operation is mapped to one of the literals $[\![x = v]\!]$, $[\![x \neq v]\!]$, $[\![x \leq v]\!]$, and $[\![x \geq v]\!]$ in the natural way. Second, once a domain operation is executed by a propagator f, the solver assigns the corresponding domain literal accordingly. For instance, if f enforces a new upper bound u for a variable x with $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \cap]-\infty, u]$, then the literal $[\![x \leq u]\!]$ is assigned to true. Any propagator executing a domain operation associated to a literal p is asked to `explain` p with a `propagation rule` of the form $\Psi \Rightarrow p$ where Ψ, called an `explanation` for p, is a conjunction of literals. The explanation Ψ should be of course valid in the sense where if the set of domain operations corresponding to literals in Ψ are called on the initial domain, then f executes (at least) the domain operation associated to p.

Example 2.4. *Propagation rule*

Let f be the propagator for $x \geq y + 10$ described in Algorithm 7. When tightening the lower bound of x to 13 because y is assigned to 3 (Line 2), f can generate the propagation rule $[\![y = 3]\!] \Rightarrow [\![x \geq 13]\!]$ which corresponds to the clause $[\![y \neq 3]\!] \vee [\![x \geq 13]\!]$.

Algorithm 7: $x \geq y + 10$

```
if min(y) + 10 > max(x) then
1   D ← ⊥ ;
else
    if (min(y) + 10) > min(x) then
2       D(x) ← D(x) ∩ [min(y) + 10, +∞[ ;
    if (max(x) − 10) < max(y) then
3       D(y) ← D(y) ∩ ]−∞, max(x) − 10] ;
return D ;
```

Similarly to CDCL, propagation rules are expanded to explain failures. That is, when a propagator f returns the fail domain $\bot$, a propagation rule associated to this failure is a logical implication $\Psi \Rightarrow \bot$ s.t. Ψ is a sufficient condition for f to detect a failure.

Example 2.5. *Explaining Failure*

Consider the same constraint $x \geq y + 10$ with $\mathcal{D}(x) = [3, 8]$ and $\mathcal{D}(y) = \{3\}$. In this case, the propagator f when triggering a failure (Line 1 in Algorithm 7)) can generate the explanation $[\![y = 3]\!] \wedge [\![x \leq 8]\!] \Rightarrow \bot$ which gives the conflict clause $[\![y \neq 3]\!] \vee [\![x \geq 9]\!]$.

Propagation rules are added to the UP-Engine as clauses already propagated. The same behavior applies when a conflict is raised by a propagator. The clause explaining the failure is added to the UP-Engine, however, as the conflict clause. The conflict analysis procedure is performed exactly the same way in CDCL.

It should be pointed out that any assignment by UP is reflected on the domain every time UP successfully terminates propagation. For instance, if UP propagates the literal $[\![x \leq 7]\!]$ to be *true* then the upper bound tightening $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \cap]-\infty, 7]$ is executed if $max(x) > 7$.

2.3.2 Engineering a Hybrid Solver: Modern Techniques

We describe here three modern techniques used in hybrid solvers: backward explanations, lazy generation, and semantic reduction.

2.3.2.1 Backward Explanations

The concept of `backward` (or lazy) explanations [59, 64, 98, 52] can simply be understood as generating explanations only when they are needed. The main motive behind using backward explanations is that generating a clause for each single propagation might make the clause database grow extremely large. Moreover these clauses do not make any difference to the propagation engine. They are only useful during conflict analysis, where only a fraction of them may be explored. Avoiding generating these clauses could therefore save time. We give a simple way for using backward explanations.

First, as usual, when a domain operation is being executed by a propagator f, the correspondent literal p should be assigned accordingly. However, instead of generating a propagation rule for l, the solver records f as the reason for assigning p. Any propagator using the backward mode is supposed to be able to generate a propagation rule for its actions during conflict analysis.

Algorithm 8 depicts a slightly modified version of the 1-UIP procedure in order to handle backward explanations. The difference between Algorithm 8 and Algorithm 6 is the use of a function called `reason(`p`)` to return the propagator f responsible for the domain operation represented by p. Moreover, the correspondent propagation rule is expected to be computed by the call to the function $explain(f, l)$. The same behavior applies when explaining a failure with $reason(\bot)$ and $explain(f, \bot)$.

Note that the way we presented Algorithm 8 allows any propagator to adapt any mode of generating explanations (i.e., eagerly at the moment of propagation, or during conflict analysis).

Algorithm 8: 1-UIP-backward

$d \leftarrow current\ level$;
$f \leftarrow reason(\bot)$;
1 $\Psi \leftarrow explain(f, \bot)$;
2 **while** $\left|\{q \in \Psi \mid level(q) = d\}\right| > 1$ **do**
3 $\quad p \leftarrow \arg\max_q(\{rank(q) \mid level(q) = d \ \wedge \ q \in \Psi\})$;
4 $\quad f \leftarrow reason(p)$;
$\quad \Psi \leftarrow \Psi \cup \{q \mid q \in explain(f, p) \wedge level(q) > 0\} \setminus \{p\}$;
5 **return** Ψ ;

2.3.2.2 Lazy (Atom) Generation

In order to have a reasonable number of atoms inside the UP engine, this technique is used to lazily generate atoms related to domain operations only when they are needed [53, 52]. Recall that for a variable domain of size k, the number of atoms is $2k$ and the number of clauses is about $4k$ (using the DOM(x) encoding). When the domain size is too large, hybrid models becomes hardly efficient because of the amount of time needed for propagating these clauses. The notion of 'lazy generation' appeared recently in the literature as a mechanism dealing with that issue.

We describe this mechanism following the latest propositions in [52] which are improvements of [53]. We use their term '`lazy generation`' to describe this technique.

The main transformation needed for using lazy generation is to reshape propagation rules to contain both literals and domain operations. The gain here is that one does not need the atoms generated from the beginning. Take for instance the propagation rule in Example 2.4 $[\![y = 3]\!] \Rightarrow [\![x \geq 13]\!]$. The propagator does not need to use the atom $[\![y = 3]\!]$ to explain $[\![x \geq 13]\!]$. Instead, it can inform the solver that the operation assigning y to 3 is responsible for the lower bound tightening of x to 13. We shall use the notations $[\![x = v]\!]$, $[\![x \neq v]\!]$, $[\![x \leq v]\!]$, and $[\![x \geq v]\!]$ for literals associated to generated atoms as well as the correspondent domain operations.

The skeleton of conflict analysis is the same as Algorithm 8. Few adaptations are, however, necessary. First, the nogood under construction Ψ can contain both literals and domain operations and p can be either a literal or a domain operation. Next, one should be able to recover the values of *level*, *rank* and *reason* for each domain constraint operation. Note that the rank is needed only in Line 3 to compute the last assigned literal in Ψ. Finally, before returning Ψ in Line 5, all domain operations in Ψ should either be replaced by their corresponding literals if they are already generated, or be associated to newly generated atoms.

Three scenarios are possible when lazily generating an atom $[\![x \leq u]\!]$.

1. If there is no value $a = max\{u' \mid [\![x \leq u']\!] \; is \; generated \; \wedge u' < u\}$, we add the clause $\neg[\![x \leq u]\!] \vee [\![x \leq b]\!]$ if there exists a value $b = min\{u' \mid [\![x \leq u']\!] \; is \; generated \; \wedge u' > u\}$.

2. If there is no value $b = min\{u' \mid [\![x \leq u']\!] \; is \; generated \; \wedge u' > u\}$, we add the clause $\neg[\![x \leq a]\!] \vee [\![x \leq u]\!]$ if there exists a value $a = max\{u' \mid [\![x \leq u']\!] \; is \; generated \; \wedge u' < u\}$

3. Otherwise, we add the clauses $\neg[\![x \leq a]\!] \vee [\![x \leq u]\!]$ and $\neg[\![x \leq u]\!] \vee [\![x \leq b]\!]$ where $a = max\{u' \mid [\![x \leq u']\!] \; is \; generated \; \wedge u' < u\}$ and $b = min\{u' \mid [\![x \leq u']\!] \; is \; generated \; \wedge u' > u\}$.

If an atom $[\![x = v]\!]$ has to be generated, one first generates $[\![x \leq v]\!]$ and $[\![x \leq v-1]\!]$ following the above way (if they are not already generated), then posts the clauses 2, 3, and 5 of $\text{DOM}(x)$.

The main problem with lazy generation is that there is a redundancy regarding the generation of bound literals. After adding the clauses $\neg[\![x \leq a]\!] \vee [\![x \leq u]\!]$ and $\neg[\![x \leq u]\!] \vee [\![x \leq b]\!]$ the clause $\neg[\![x \leq l]\!] \vee [\![x \leq u]\!]$ becomes redundant. There might be $n-2$ redundant clauses after generating n atoms for a given variable.

We shall propose in Section 5.2 a new way for using lazy generation in order to avoid this redundancy whilst being computationally equivalent to UP as if the atoms were generated from the beginning.

Recall that if the literals are eagerly generated then for any domain change, one assigns its corresponding literal to *true* which might trigger UP. Such a procedure is not necessary with lazy generation since not every domain operation is associated to a literal. Instead, the domain changes must be reflected on the generated literals. Feydy et al. [52] propose to associate a map for each variable x from values to domain operations[4]. Whenever $\mathcal{D}(x)$ changes, the map can be used to determine the newly executed domain operations already having an associated literal. These literals must then be assigned accordingly.

2.3.2.3 Semantic Reduction

In general, there is no complete qualitative evaluation for comparing different nogoods/explanations. Take for instance the nogoods $a \wedge \neg b \wedge c \Rightarrow \bot$, $e \wedge c \Rightarrow \bot$, and $a \wedge \neg b \Rightarrow \bot$. Unless we have additional information regarding a, b, and e, we cannot determine the best choice between $a \wedge \neg b \wedge c \Rightarrow \bot$ and $e \wedge c \Rightarrow \bot$ even though the latter is shorter. The strict inclusion, however, gives a simple and straightforward way for comparison. For instance $a \wedge \neg b \Rightarrow p$ is clearly preferable to $a \wedge \neg b \wedge c \Rightarrow p$.

[4]We had also a personal communication with Thibaut Feydy on the subject.

Reasoning about the semantic of each literal/domain operation enables a new way for reduction. Suppose that $\Psi : [\![x \leq 17]\!] \wedge [\![x \leq 10]\!] \wedge [\![y \geq 5]\!] \wedge [\![y \geq 9]\!] \wedge p \Rightarrow \bot$ is the final nogood found before converting domain operations into literals. Since $[\![x \leq 10]\!]$ can be considered as a plausible explanation for $[\![x \leq 17]\!]$, then we can safely remove it from Ψ. The same observation goes with $[\![y \geq 9]\!]$ as a reason for $[\![y \geq 5]\!]$. The final nogood in this case is $[\![x \leq 10]\!] \wedge [\![y \geq 9]\!] \wedge p \Rightarrow \bot$.

Semantic reduction revises the final nogood to contain for each variable the smallest possible upper bound and the largest possible lower bound literals. Not only has the final nogood a better quality, but also the number of lazily generated atoms is smaller.

Chapter 3

An Empirical Heuristic Study for the Car-Sequencing Problem

Introduction

Car-sequencing [102] is a well known sequencing problem coming from the automotive industry and has a long history in constraint programming [44, 17, 128, 111, 139]. In this problem, a set of cars has to be sequenced on an assembly line subject to capacity and demand constraints. Each car belongs to a class of vehicles that is defined with a set of options to install (such as sunroof and air-conditioner). In 2005, a challenge has been organized by the French Operations Research and Decision Support Society (ROADEF[1]) for solving optimization versions of the problem provided by the RENAULT[2] automobile manufacturer. We refer the reader to [131] for a survey regarding exact and heuristic methods used in this challenge.

In this chapter, we are interested in the search aspect for solving the car-sequencing problem. The latter is used as a test benchmark throughout this thesis. Through a comprehensive evaluation of search strategies for this problem. We show the interest of several new branching heuristics and we measure the overall impact of the choice of search strategy.

This empirical study is built on a new classification of heuristics for this problem. This classification is based on a set of four criteria: the type of branching decisions, the exploration directions, the selection of branching values ('options' in this model) and the aggregation function for this selection. In particular, we show that the way of selecting the most constrained option is critical, and the best choice is fairly reliably

[1] http://challenge.roadef.org/2005/en
[2] http://group.renault.com

the "load" of an option, that is the ratio between its demand and the capacity of the corresponding machine. Similarly, branching on the class of vehicle is more efficient than branching on the use of an option. Overall, even though results can vary greatly from instance to instance, we are able to indicate with a relatively high confidence which is the most robust strategy, or at least outline a small set of potentially best strategies.

The remaining of the chapter is organized as follows. In Section 3.1, we describe the car-sequencing problem and discuss the related constraint satisfaction models. In Section 3.2, we propose and classify a number of new and existing heuristics. And finally, we empirically evaluate and analyze the new classification in Section 3.3.

3.1 The Car-Sequencing Problem

3.1.1 Problem Description

In the car-sequencing problem, n vehicles have to be produced on an assembly line. There are k classes of vehicles and m types of options. Each class $c \in \{1, \ldots, k\}$ is associated with a demand d_c^{class}, that is, the number of occurrences of this class on the assembly line, and a set of options $\mathcal{O}_c \subseteq \{1, \ldots, m\}$. Each option is handled by a working station able to process only a fraction of the vehicles passing on the line. The capacity of an option j is defined by two integers p_j and q_j, such that no subsequence of size q_j may contain more than p_j vehicles requiring option j.

A solution of the problem is then a sequence of cars satisfying both demand and capacity constraints. This problem is NP-hard [84, 50].

For convenience, we shall also define, for each option j, the corresponding set of classes of vehicles requiring this option $\mathcal{C}_j = \{c \mid j \in \mathcal{O}_c\}$, and the option's demand $d_j^{opt} = \sum_{c \in \mathcal{C}_j} d_c^{class}$.

Example 3.1. *Consider a simple case of 5 slots (i.e., $n = 5$) with 3 classes $\{c_1, c_2, c_3\}$ and 4 options such that:*

- $\mathcal{O}_{c_1} = \{1, 2\}$, $\mathcal{O}_{c_2} = \{1, 3, 4\}$, $\mathcal{O}_{c_3} = \{2\}$.
- $d_{c_1}^{class} = 2$, $d_{c_2}^{class} = 2$, $d_{c_3}^{class} = 1$
- p_i/q_i *(lexicographically):* 3/4*;* 2/3*;* 1/3*;* 1/2.

From above, we obtain:

- $\mathcal{C}_1 = \{1,2\}$, $\mathcal{C}_2 = \{1,3\}$, $\mathcal{C}_3 = \{2\}$ *and* $\mathcal{C}_4 = \{2\}$
- $d_1^{opt} = 4$, $d_2^{opt} = 3$, $d_3^{opt} = 2$ *and* $d_4^{opt} = 2$

The sequence $[c_1, c_2, c_1, c_3, c_2]$ *is a possible solution for this instance.*

3.1.2 Modeling

We use a standard CP model[3] with two sets of variables. The first set corresponds to n integer variables $\{x_1, \ldots, x_n\}$ (called class variables) taking values in $\{1, \ldots, k\}$ and standing for the class of vehicles in each slot of the assembly line. The second set of variables corresponds to nm Boolean variables $\{y_1^1, \ldots, y_n^m\}$ (called option variables), where y_i^j stands for whether the vehicle in the i^{th} slot requires option j.

There are three sets of constraints.

1. *Demand constraints*: for each class $c \in \{1..k\}$, $\left|\{i \mid x_i = c\}\right| = d_c^{class}$. These constraints are usually enforced with a *Global Cardinality Constraint* (Section 2.1.3.2.
2. *Capacity constraints*: for each option $j \in \{1..m\}$, no subsequence of size q_j involves more than p_j cars requiring option j. That is, $\sum_{l=i}^{i+q_j-1} y_l^j \leq p_j$, $\forall i \in \{1, \ldots, n-q_j+1\}$. In order to factor out as much as possible the propagation aspect from the study, we use several models in order to diversify the data set. More precisely, we shall consider four models, differentiated by how capacity constraints are modeled and thus propagated. For each option j, these constraints can be expressed in one of the following alternatives:
   (a) A naive decomposition using sum constraints. This model is denoted DECOMPOSE.
   (b) Let *card* be a mapping on integers such that $card(c) = d_c^{class}, \forall c \in \{1, \ldots, k\}$. For each option j, we post the following *Global Sequencing Constraint* (Section 2.1.3.2):

$$\textsc{Gsc}(0, p_j, q_j, card, card, [x_1, \ldots, x_n], \mathcal{C}_j)$$

   This model is denoted GSC.
   (c) For each option j, we post the following AtMostSeqCard constraint (defined later in Section 4.3):

[3]This model can be found for instance in Ilog-Solver 6.7.

$$\textsc{AtMostSeqCard}(p_j, q_j, d_j^{opt}, [y_1^j, \ldots, y_n^j])$$

This model is denoted AMSC.

(d) We post both $\textsc{Gsc}(0, p_j, q_j, card, card, [x_1, \ldots, x_n], \mathcal{C}_j)$ and $\textsc{AtMostSeqCard}(p_j, q_j, d_j^{opt}, [y_1^j, \ldots, y_n^j])$ for each option.

This model is denoted GSC⊕AMSC.

3. *Channeling*: Option and class variables are channeled through simple constraints: $y_i^j = 1 \Leftrightarrow j \in \mathcal{C}_{x_i}, \forall j \in \{1, ..., m\}, \forall i \in \{1, ..., n\}$. Each constraint is implemented using a set of simple binary constraints $x_i = c \Rightarrow y_i^j = 1,\ \forall j \in \mathcal{O}_c$ and $x_i = c \Rightarrow y_i^j = 0,\ \forall j \in \{1, \ldots, m\} \setminus \mathcal{O}_c$.

3.1.3 Related Work

Regarding the search strategy, two main principles are known to be important for the car-sequencing problem. First, the sequence of variables to branch should follow the assembly line itself. Indeed, the structure in chain of capacity constraints makes it difficult to achieve any inference far away from a modified variable in the sequence [128]. Second, one should assign the most constrained class or option first. This has been perceived as a fail-first strategy, hence surprising since succeed-first strategies should be better for selecting the next branch to follow. However, as pointed out in [128], since the solutions to this problem are permutations of a multiset of values, choosing the most constrained one when it is still possible actually yields the least constrained sub-problem. Therefore, in this sense, it is indeed a succeed-first strategy.

In [128], a lexicographical exploration of the integer variables $x_1, \ldots, x_n$, standing for classes of vehicles, was advocated as an interesting search strategy. Three parameters were considered for choosing the most constrained class: the number of options per class (denoted as *max option*), the tightness of each option (i.e., the capacity constraint q/p) and the usage of each option (i.e., usage rate $\frac{d.q/p}{n}$).

In [111], the authors proposed to branch on option variables y_i^j, exploring the sequence consistently with their position on the assembly line, however starting from the middle towards the extremities. Indeed variables at both ends are subject to fewer capacity constraints than variables within the sequence. Moreover, they introduced for the first time the notion of slack for selecting the most constrained option.

In [68], several heuristics were compared for solving an optimization variant of this problem. These heuristics are based on the usage rate previously defined for selecting the next variables to assign in the sequence. They consider two ways for aggregating these values (using lexicographically the maximum value, or a simple sum) when branching on class variables. Two possibilities of using the usage rate were compared : static

and dynamic (i.e., updated at each node). Note that the static values of usage rate, load or slack are all equivalent. Their experiments showed essentially the interest of dynamic heuristics comparatively to static ones. The same observation is made in [29] where a dynamic load was used with class variable branching and a simple summation to aggregate the values.

3.2 Heuristics Classification

3.2.1 Classification Criteria

We propose to classify the heuristics related to this problem according to four criteria:

- The type of *branching* decisions: that is, whether we branch on classes or options.
- The order in which we *explore* the variables along the assembly line: one can start from the left of the sequence and progress to the right, or start from the middle of the assembly line widen to the sides.
- The measure used to *select* the most constrained options.
- The function used to *aggregate* the evaluation of the different options in order to choose the next class of vehicles to branch on.

Notice that among the many combinations of these four criteria, some correspond to existing heuristics, however some are novel. For each criterion, there are several alternatives, we present each of them below.

3.2.1.1 Branching

We can branch either by assigning a class to a slot, that is, branching on class variables x_i, or by assigning an option to a slot, that is, branching on option variables y_i^j. The former was used in [128], while the latter was proposed in [111]. Notice that when branching on option variables, we always set it to the value 1, which amounts at forcing the corresponding option to be represented in that slot. We therefore consider these two cases denoted respectively *class* and *opt*.

3.2.1.2 Exploration

Heuristics that do not follow the sequence of variables along the assembly line generally have poor performances [128]. We find in the literature two main exploration orders:

either following a lexicographical order on class variables or from the middle to the sides of the sequence. We therefore consider these two exploration cases denoted respectively *lex* and *mid*.

3.2.1.3 Selection

The best heuristics are those selecting first the most constrained option or class. Observe that since each class is defined by a set of options, then it all goes down to the hardness of the options. We therefore consider the following indicators proposed in the literature to select the most constrained option:

- The capacity q_j/p_j: The greater the ratio q_j/p_j, the more constrained is the option. In fact, a greater ratio q_j/p_j has more impact on neighboring slots as it is shown in Example 3.2.

  Example 3.2. *Let o_1 and o_2 be two options s.t. $p_1 = 1$, $p_2 = 2$, and $q_1 = q_2 = 3$. Consider now a sequence of 5 slots in which we have to choose between o_1 and o_2 in the third position. The two parts of the following figure show the impact of each option. In fact, by choosing o_1, all neighboring slots can no longer contain this option because of the at most 1/3 constraints.*

y_i^1					y_i^2				
0	***0***	*1*	***0***	***0***	.	.	*1*	.	.

- The residual demand d_j^{opt}: This value is equal to the total demand (of a given option) minus the number of cars containing this option already allocated ($d_j^{opt} = (d_j^{opt} - \sum_{i=1}^{n} min(y_i^j))$. Clearly, a greater demand makes it more difficult to fit the cars requiring this option on the assembly line.

- The load δ_j: This parameter combines the residual demand with the capacity ratio: $\delta_j = d_j^{opt} \times \frac{q_j}{p_j}$. In fact the ceiling of δ_j is always an upper bound for the number of slots required to mount d_j^{opt} times the option j. A greater value of the load is therefore more constrained.

- The slack σ_j: Let n_j be the number of slots available for option j. The slack of an option j is $\sigma_j = n_j - \delta_j$. Since we want higher values to indicate more constrained options, we use in fact $n - \sigma_j$.

- The usage rate ρ_j: This value is defined as the load divided by the number of remaining slots: $\rho_j = \delta_j/n_j$. It therefore represents how much of the remaining space will be occupied by vehicles requiring this option.

Based on these indicators, we consider four methods to evaluate the options. Each method returns an indicative value on how constrained is an option. In other words, the option maximizing the given parameter will be preferred in the next decision. In the following, we denote the above selection criteria by q/p, d^{opt}, δ, $n-\sigma$ and ρ, respectively.

3.2.1.4 Aggregation

In the case of *class* branching, since classes are defined as a set of options, the decision is most often made by summing up the "scores" of the options for each class. However, there are many ways to aggregate these values. We therefore propose to add the method used for the aggregation as a fourth criterion.
Let $f : \{1,\dots,m\} \mapsto \mathbb{R}$ be a scoring function. We denote $f(\mathcal{O}_c)$ the tuple formed by the sorted scores of class c's options, i.e., $f(\mathcal{O}_c) = \langle f(j_1),\dots,f(j_{|\mathcal{O}_c|})\rangle$ such that $\{j_1,\dots,j_{|\mathcal{O}_c|}\} = \mathcal{O}_c$ and $f(j_l) \geq f(j_{l+1})\ \forall l \in [1,\dots,|\mathcal{O}_c| - 1]$. We shall consider the following ordering relations between classes:

- Sum of the elements ($\leq_\Sigma$): $c_1 \leq_\Sigma c_2$ iff $\sum_{v\in f(\mathcal{O}_{c_1})} v \leq \sum_{v\in f(\mathcal{O}_{c_2})} v$.
- Euclidean norm ($\leq_{Euc}$): $c_1 \leq_{Euc} c_2$ iff $\sum_{v\in f(\mathcal{O}_{c_1})} v^2 \leq \sum_{v\in f(\mathcal{O}_{c_2})} v^2$.
- Lexicographical order ($\leq_{lex}$): $c_1 \leq_{lex} c_2$ iff $f(\mathcal{O}_{c_2})$ comes lexicographically after $f(\mathcal{O}_{c_1})$.

Example 3.3. *We give an illustrative example. We consider Example 3.1 and suppose that one branches on classes. In Table 3.1, we give the different values of each selection parameter for all options.*

TABLE 3.1: Values of the selection criteria for each option

Selection parameter \ Options	1	2	3	4
q/p	1,33	1.5	3	2
d^{opt}	4	3	2	2
δ	5,32	4.5	6	4
$n-\sigma$	5,32	4.5	6	4
ρ	1,064	0.9	1,2	0,8

In order to emphasize the impact of aggregation functions, we propose to study the different scores for each class using the d^{opt} parameter. Recall that each class is defined by a set of options, we obtain in Table 3.2 the corresponding values for each class.

In Table 3.3, we report the order of preferences given by the different aggregations. The class having the higher score will be selected first and so on.

TABLE 3.2: Classes' scores using the parameter d^{opt}

Options \ Classes	c_1	c_2	c_3
1	4	4	-
2	3	-	3
3	-	2	-
4	-	2	-

TABLE 3.3: Scores & Heuristic decisions

Agg.	Scores			Heuristic preferences
	c_1	c_2	c_3	
$\leq_{\sum}$	7	8	3	$[c_2, c_1, c_3]$
$\leq_{Euc}$	25	24	9	$[c_1, c_2, c_3]$
$\leq_{lex}$	[4, 3, -, -]	[4, 2,-,-]	[3,-,-,-]	$[c_1, c_2, c_3]$

Although we treat a simple case, one can observe that decisions can be influenced by aggregation functions. The behavior of $\leq_{\sum}$ *is different from* $\leq_{Euc}$ *and* $\leq_{lex}$*. It prefers* c_2 *whereas the others prefer* c_1*.*

3.2.2 Heuristics Structure

In the rest of the chapter, we denote the set of heuristics as follows: $\langle\{class, opt\}, \{lex, mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho, 1\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$. Note that we considered the constant function 1 as another possible selection criterion. This is proposed so that our classification also includes the *max option* heuristic [128] where each class is evaluated simply by its number of options.

Observe, however, that not all combinations make sense. For instance, the aggregation function does not matter when branching on options. Therefore, using the new classification, we obtain 42 possible heuristics:

- $\langle\{class\}, \{lex, mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$: The 30 heuristics that branches on *class* variables with the two exploration strategies $\{lex, mid\}$, the five selection parameters $\{q/p, d^{opt}, \delta, n-\sigma, \rho\}$ and the 3 aggregation techniques $\{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}$.
- $\langle\{opt\}, \{lex, mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \varnothing\rangle$: 10 heuristics branching on option variables with the two exploration possibilities $\{lex, mid\}$ and the five selection parameters $\{q/p, d^{opt}, \delta, n-\sigma, \rho\}$.

- $\langle\{class\}, \{lex, mid\}, \{1\}, \{\leq_\sum\}\rangle$: The two possible heuristics related to the particular case of *max option*.

Among the many combinations defined by this structure, there are several existing heuristics as well as new ones. In the literature, only few heuristics have been studied. First, the *max option* heuristic proposed in [128] branches on *class* variables lexicographically (lex) and the most constrained class is then selected using the sum ($\leq_\sum$) aggregation. It therefore corresponds to $\langle class, lex, 1, \leq_\sum\rangle$. Second, in [68], the authors proposed to use the usage rage with *class* branching, lexicographical exploration (lex) and $\leq_\sum$, $\leq_{lex}$ for aggregation. They correspond to $\langle class, lex, \delta, \{\leq_\sum, \leq_{lex}\}\rangle$. Similarly, the authors of [29] proposed a *class* branching using $\leq_\sum$ for aggregation in a lexicographical exploration (lex), however, using the load δ and the capacity q/p for selection (i.e., $\langle class, lex, \{\delta, q/p\}, \leq_\sum\rangle$). Finally, the heuristic proposed in [111] is based on *option* branching, exploring the sequence from the middle to the sides using the slack as a selection criteria. This heuristic corresponds to $\langle opt, mid, n-\sigma, \varnothing\rangle$.

To the best of our knowledge, all other heuristics are new and there is no comparative study for evaluating the impact of each classification criterion.

3.3 Evaluating the new Structure

In this section, we evaluate experimentally the impact of the proposed criteria classification for the heuristics. We slightly perform randomization as follows: with a low probability (2% for classes and 5% for options[4]), the second best choice (provided by the heuristic) is taken.

All the experiments were run on Intel Xeon CPUs 2.67GHz under Linux. The detailed results are available via `http://homepages.laas.fr/msiala/car-sequencing`. For each instance, we launched five randomized runs per heuristic with a 20 minutes time cut-off. All models are implemented using Ilog-Solver 6.7.

We use benchmarks available from the CSPLib [2] divided into three groups. The first group of the CSPLib contains 70 satisfiable instances having 200 cars, 5 options and from 18 to 30 classes, it is denoted by *set1*. The second group of the CSPLib corresponds to instances with 100 cars, 5 options and from 19 to 26 classes. In this group there are 4 satisfiable instances , denoted by *set2* and 5 unsatisfiable instances denoted by *set3*. The third group of the CSPLib contains 30 larger instances (ranging from 200 to 400 vehicles, 5 options and from 19 to 26 classes). The 7 instances from this group that are known to be satisfiable are grouped together in *set4*. At the top of each table, we mention,

[4]Those values were arbitrarily chosen. The impact of branching on an option variable being lower, a higher probability was necessary.

for each data set, the total number of instances with an indication on their feasibility (i.e., satisfiable: S and unsatisfiable U). The status of the 23 remaining instances *set5* is still unknown. They are often treated in an optimization context, hence they are not considered in these experimentations.

The set of heuristics $\langle\{class, opt\}, \{lex, mid\}, \{1, q/p, d^{opt}, \delta, n-\sigma, \rho\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$ combined with the four models DECOMPOSE, AMSC, GSC, and GSC⊕AMSC leads to 168 different configurations. The latter is applied to each set of instances (i.e., $70+4+5+7$ instances) with 5 randomized runs. The total CPU time for that was devoted to these experiments is around 244 days.

We say that a run (related to an instance and a given configuration) is `successful` if either a solution was found or unsatisfiability was proven. For each set of instances, we report the percentage of successful runs ($\%sol$) [5], the CPU time ($time$) in seconds both averaged over all successful runs and number of instances.

Experimental results are divided in thee parts. We first compare the many combinations of heuristic factors by giving the results for each one. Then, we study the proposed classification by evaluating each factor separately. Finally, we provide a comparison related to the efficiency and confidence of each factor

3.3.1 Impact of each Heuristic

In this paragraph, we report the results of each heuristic separately on each set of instances averaged over the four propagators.

The set of heuristics corresponds to all possible combinations of parameters given by: $\langle\{class, opt\}, \{lex, mid\}, \{1, q/p, d^{opt}, \delta, n-\sigma, \rho\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$ leading to the 42 heuristics presented in Section 3.2.

Table 3.4 shows the global results of our experiments. For each heuristic, we indicate in column ('Ref.') whether it is already known (with the corresponding reference) or not (with '-'). Recall that, in these experiments, we consider only dynamic evaluation with the four criteria : demand, load, usage rate and slack. For each set of instances, we report the percentage of successful runs ($\%sol$) and their average CPU time ($time$). The last two columns summarize the results over all set of instances. The column ($\%tot$) gives the total percentage of solved instances and the column ($\%dev$) gives the deviation in percent of a given heuristic to the heuristic solving the maximum number of instances. Bold values give the best heuristics w.r.t. $\%sol$.

[5] Since *set3* contains only unsatisfiable instances, then $\%sol$ corresponds to the percentage of instances for which the solver proves unsatisfiability.

TABLE 3.4: Comparison of heuristics averaged over propagation rules

Heuristics				Ref.	Instances								Total	
					set 1 $(70, S)$		set 2 $(4, S)$		set 3 $(5, U)$		set 4 $(7, S)$			
Sel.	Br.	Expl.	Aggr.		%sol	time	%sol	time	%sol	time	%sol	time	$\%tot$	$\%dev$
ρ	*class*	*lex*	$\leq_{lex}$	[68]	**100.00**	0.6	52.50	59.1	0.00	-	**25.71**	2.9	85.93	1.00
			$\leq_{\sum}$	[68]	**100.00**	0.6	48.75	0.2	0.00	-	10.71	84.4	84.53	2.61
			$\leq_{Euc}$	-	**100.00**	0.6	30.00	0.2	0.00	-	12.85	156.3	83.84	3.42
		mid	$\leq_{lex}$	-	99.92	0.5	53.75	163.5	0.00	-	16.42	50.0	85.17	1.88
			$\leq_{\sum}$	-	**100.00**	**0.5**	51.25	236.6	0.00	-	18.57	5.4	85.29	1.74
			$\leq_{Euc}$	-	**100.00**	**0.5**	51.25	249.3	0.00	-	17.14	30.2	85.17	1.88
	opt	*lex*	-	-	87.00	1.9	**75.00**	**33.3**	**25.00**	**211.3**	5.71	533.4	76.22	12.19
		mid	-	-	87.64	2.9	31.25	0.4	23.00	233.6	14.28	171.1	75.29	13.26
$n-\sigma$	*class*	*lex*	$\leq_{lex}$	-	**100.00**	0.6	52.50	59.2	0.00	-	**25.71**	**2.8**	85.93	1.00
			$\leq_{\sum}$	-	**100.00**	0.6	48.75	0.2	0.00	-	10.71	78.6	84.53	2.61
			$\leq_{Euc}$	-	**100.00**	0.6	48.75	0.1	0.00	-	10.71	79.4	84.53	2.61
		mid	$\leq_{lex}$	-	**100.00**	0.6	53.75	169.7	0.00	-	18.57	33.1	85.41	1.61
			$\leq_{\sum}$	-	**100.00**	**0.5**	51.25	236.9	0.00	-	22.14	29.0	85.58	1.41
			$\leq_{Euc}$	-	99.92	0.5	51.25	236.3	0.00	-	22.14	28.8	85.52	1.48
	opt	*lex*	-	-	32.71	21.7	43.75	236.8	13.00	190.7	0.00	-	29.42	66.11
		mid	-	[111]	38.14	13.0	26.25	33.7	18.00	260.8	0.00	-	33.31	61.62
δ	*class*	*lex*	$\leq_{lex}$	-	**100.00**	0.6	71.25	42.4	0.00	-	**25.71**	3.0	**86.80**	0.00
			$\leq_{\sum}$	[29]	**100.00**	0.6	48.75	0.3	0.00	-	10.71	100.2	84.53	2.61
			$\leq_{Euc}$	-	**100.00**	0.6	48.75	0.3	0.00	-	10.71	87.3	84.53	2.61
		mid	$\leq_{lex}$	-	**100.00**	**0.5**	37.50	38.2	0.00	-	15.00	51.5	84.36	2.81
			$\leq_{\sum}$	-	**100.00**	**0.5**	68.75	167.9	0.00	-	20.71	42.8	86.28	0.60
			$\leq_{Euc}$	-	**100.00**	**0.5**	68.75	166.5	0.00	-	20.00	16.2	86.22	0.67
	opt	*lex*	-	-	98.57	1.2	36.25	111.7	0.00	-	22.85	5.8	83.78	3.48
		mid	-	-	98.92	3.7	43.75	3.8	0.00	-	21.42	88.8	84.29	2.89
q/p	*class*	*lex*	$\leq_{lex}$	-	82.85	7.8	0.00	-	0.00	-	0.00	-	67.44	22.31
			$\leq_{\sum}$	[29]	83.35	10.1	18.75	0.1	0.00	-	0.00	-	68.72	20.84
			$\leq_{Euc}$	-	83.42	11.3	18.75	0.09	0.00	-	0.00	-	68.77	20.77
		mid	$\leq_{lex}$	-	84.71	7.9	18.75	95.7	0.00	-	0.00	-	69.82	19.56
			$\leq_{\sum}$	-	85.35	7.7	18.75	100.9	0.00	-	0.00	-	70.34	18.96
			$\leq_{Euc}$	-	84.64	7.5	18.75	96.0	0.00	-	0.00	-	69.77	19.63
	opt	*lex*	-	-	65.71	73.3	0.00	-	0.00	-	0.00	-	53.48	38.38
		mid	-	-	70.71	29.8	12.50	606.4	0.00	-	0.00	-	58.14	33.02
d^{opt}	*class*	*lex*	$\leq_{lex}$	-	90.92	1.2	37.50	47.4	0.00	-	**25.71**	55.3	77.84	10.32
			$\leq_{\sum}$	-	95.07	1.9	41.25	48.5	0.00	-	17.14	21.5	80.70	7.03
			$\leq_{Euc}$	-	94.50	0.7	43.75	106.5	0.00	-	23.57	40.2	80.87	6.83
		mid	$\leq_{lex}$	-	90.64	1.9	**75.00**	83.4	0.00	-	24.28	5.3	79.24	8.71
			$\leq_{\sum}$	-	94.71	0.6	67.50	68.9	0.00	-	13.57	53.9	81.33	6.30
			$\leq_{Euc}$	-	94.57	0.6	**75.00**	83.2	0.00	-	15.71	50.7	81.74	5.83
	opt	*lex*	-	-	73.78	2.9	56.25	79.5	0.00	-	0.71	282.0	62.73	27.73
		mid	-	-	77.28	13.7	43.75	5.2	0.00	-	7.85	16.5	65.58	24.45
1	*class*	*lex*	$\leq_{\sum}$	[128, 29]	86.92	13.2	18.75	0.1	0.00	-	0.00	-	71.62	17.49
		mid	$\leq_{\sum}$	-	89.92	8.3	63.75	20.3	0.00	-	0.00	-	76.16	12.26

For the easiest set (*set1*), 16 heuristics solve all instances in less than a second. Among them, 3 are known heuristics whereas 13 correspond to new combinations. It should be noted that all these configurations use a *class* branching and a load-based selection (i.e., $\rho, \delta, n-\sigma$). Interestingly, changing a single parameter of a heuristic can have a dramatic effect. For instance, the heuristic $\langle opt, lex, n-\sigma, \varnothing \rangle$ resolves only $32,71\%$ of this set whereas changing only the branching criterion to *class* (i.e., $\langle class, lex, n-\sigma, \{\leq_{lex}, \leq_{\sum}, \leq_{Euc}\}\rangle$) leads to a complete resolution (i.e., 100%).

For *set2* and *set3*, the heuristic $\langle opt, lex, \rho, \varnothing \rangle$ gives the best results with 75% in $33.3s$ for *set2* and 25% in $211.3s$ for *set3*. Also, the heuristics $\langle class, mid, d^{opt}, \{\leq_{lex}, \leq_{Euc}\}\rangle$ has

the same number of successful runs compared to $\langle opt, lex, \rho, \varnothing \rangle$ but with higher runtime. All of these heuristics correspond to new configurations.

Finally, for *set4*, the best heuristics resolve 25.71% in approximately $3s$ and correspond to the configurations $\langle class, lex, \{\delta, \rho, n-\sigma\}, \leq_{lex} \rangle$. Another heuristic $\langle class, lex, d^{opt}, \leq_{lex} \rangle$ obtains the same percentage but with higher runtime ($55.3s$).

Overall, the heuristic that has the best results across all data sets and therefore seems to be the more robust is $\langle class, lex, \delta, \leq_{lex} \rangle$ with 86.8% of solved instances (according to the column 'Total'). More generally, heuristics using load-based selection (i.e., δ, $n-\sigma$ and ρ) and class branching obtain better results than the other configurations.

3.3.2 Criteria Analysis

In this part, we aim to evaluate the relative impact of each classification criterion. For each criterion and each data set, we divide all the runs into as many sets as the number of possible values for this criterion. Then, we average the results within each set. For instance, *exploration* can be done either lexicographically (lex), or from the middle to the sides (mid). We will thus report two sets of statistics, one for lex and one for mid. Each average corresponds to one run per possible set of heuristics (21), filtering algorithms (4), randomized runs (5), and instances in the data set.

The following Tables (3.5, 3.6, 3.7 and 3.8) are split in two parts. We report in the upper part the results for each set and each possible criterion w.r.t. the criterion being used averaged over all other criteria. The lower part shows the best results obtained for any possible combination of the other criteria. In these tables, we report the percentage of successful runs ($\%sol$), the CPU time ($time$) in seconds both averaged over all successful runs, instances and heuristic criteria. Bold values indicate best results in terms of successful runs ($\%sol$). Moreover, in the upper tables, the last column ($\%tot$) gives the percentage of solved instances over all the sets.

3.3.2.1 Branching Strategy

Here we compare the two branching strategies: $class$ and opt. We tested all the possible combinations of heuristics for each strategy. However, as the constant selection parameter 1 is not defined for opt variables, we do not consider its heuristics in this part.

When branching on opt variables, we have defined 10 heuristics (since aggregation functions are omitted): $\langle opt, \{lex\ , mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \varnothing \rangle$, that is 200 tests for each

instance. To have consistent comparison with *class* branching, we separate its results by aggregation functions. That is $\langle class, \{lex\ , mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \leq_{lex}\rangle$, $\langle class, \{lex\ , mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \leq_{Euc}\rangle$ and $\langle class, \{lex\ , mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \leq_{\sum}\rangle$.

TABLE 3.5: Evaluation of the branching variants

Av. Bran.	*set1* $(70, S)$			*set2* $(4, S)$			*set3* $(5, U)$			*set4* $(7, S)$			Global
(×200)	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%tot
opt	73.0	102023.9	14.1	36.8	287139.5	82.0	**7.9**	53275.4	225.6	7.2	207502.8	107.9	62.2
$class, \leq_{lex}$	94.9	26120.0	2.0	45.2	481410.8	84.9	0.0	-	-	**17.7**	98707.8	22.5	80.7
$class, \leq_{\sum}$	**95.8**	27209.1	2.1	**46.3**	327601.5	95.7	0.0	-	-	12.4	156300.3	44.6	**81.1**
$class, \leq_{Euc}$	95.7	27563.3	2.1	45.5	463196.6	107.9	0.0	-	-	13.2	107599.7	52.9	81.0
Best Bran.													
opt	**100.0**	98577.4	10.3	75.0	7251.3	0.5	**40.0**	46211.8	261.8	25.7	629016.8	130.7	
$class, \leq_{lex}$	**100.0**	184.7	0.0	**100.0**	730687.4	89.5	0.0	-	-	**28.5**	29632.6	58.5	
$class, \leq_{\sum}$	**100.0**	184.2	0.0	95.0	904739.2	96.3	0.0	-	-	25.7	34705.3	54.8	
$class, \leq_{Euc}$	**100.0**	184.4	0.0	**100.0**	211830.5	128.8	0.0	-	-	**28.5**	47435.1	75.4	

The upper part of Table 3.5 shows that branching on classes is usually better than branching on options. However, the latter is more efficient on proving infeasibility (i.e., line *opt* on *set3*). The most efficient branching averaged over the other factors uses the $\leq_{\sum}$ aggregation, but the two other aggregation options ($\leq_{lex}$ or $\leq_{Euc}$) are close in performances. This result is confirmed by the lower part of the table.

3.3.2.2 Exploration

To evaluate the exploration parameters, we consider for each $\omega \in \{lex, mid\}$ the following heuristics:

- $\langle class, \omega, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$.
- $\langle opt, \omega, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \varnothing\rangle$.
- $\langle class, \omega, \{1\}, \{\leq_{\sum}\}\rangle$.

These three sets cover all possible combinations of heuristics leading to 420 tests for each parameter $\omega \in \{lex, mid\}$ and each instance. The results are shown in Table 3.6.

In the first part of Table 3.6, we can see that exploring the sequence from the middle then widening to the sides is in average slightly but consistently beneficial. Recall that the rationale for starting in the middle is that variables in the extremities are subject to fewer capacity constraints.

However, in the second part of Table 3.6, we can see that in terms of successful runs, exploring the sequence using the lexicographical order leads to better results for proving

TABLE 3.6: Evaluation of the exploration variants

Av. Expl. (×420)	*set1* $(70, S)$			*set2* $(4, S)$			*set3* $(5, U)$			*set4* $(7, S)$			Global
	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%tot
lex	89.2	50617.6	5.6	40.0	259229.0	46.6	1.8	52295.1	204.3	11.3	120652.6	54.2	75.5
mid	**90.3**	42167.0	4.1	**46.7**	479360.9	126.5	**1.9**	54184.0	245.5	**12.7**	139829.4	42.8	**76.8**
Best Expl.													
lex	**100.0**	184.8	0.0	**100.0**	730687.4	89.5	**40.0**	46211.8	261.9	**28.5**	29632.6	58.5	
mid	**100.0**	183.5	0.0	**100.0**	213028.8	129.1	36.0	63984.8	307.6	**28.5**	1357.4	9.2	

unsatisfiability. This could be explained by the fact that when starting in the middle of the sequence, we effectively split the problem into essentially disjoint subproblems (there is actually a weak link through demand constraints).

Overall, the exploration parameter does not seem to be as critical as the branching parameter.

3.3.2.3 Selection

Here, we evaluate the selection criterion for choosing the most-constrained option. In this case, there are two possible sets of heuristics for each parameter $\omega \in \{q/p, d^{opt}, \delta, n-\sigma, \rho\}$:

- $\langle class, \{lex, mid\}, \omega, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$
- $\langle opt, \{lex, mid\}, \omega, \varnothing\rangle$

That is 8 heuristics for each ω combined with the 4 propagators and the 5 runs. We therefore have 160 tests for each instance (reported in Table 3.7).

The special case of *max option* is presented separately at the end of Table 3.7 because the number of tested heuristics is different. In this case, there is only 2 heuristics $\langle class, 1, \{lex, mid\}, \{\leq_{\sum}\}\rangle$, that is 40 tests for each instance.

The upper part of Table 3.7 shows that using the *load* solves more instances in average over all the sets and for satisfiable sets (*set1*, *set2* and *set4*) only. Surprisingly, the *load* gives better results than *slack* and *usage rate*, despite the fact that both *slack* and *usage rate* are defined using the *load* and the number of available slots in the variable's sequence. However the *usage rate* criteria seems to work better both in average and for the best results for unsatisfiable instances. Moreover, in the second part of the table, one can note that the *demand* obtains good results.

This can be explained by the manner in which the benchmarks were generated. In fact, these instances, especially the hardest ones, are built in such way that they have a usage rate close to 1 [2]. Since the number of available slots is initially identical for all options,

TABLE 3.7: Evaluation of the selection variants

Av. Selec.	*set1* $(70, S)$			*set2* $(4, S)$			*set3* $(5, U)$			*set4* $(7, S)$			Global
(×160)	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%tot
ρ	96.8	1628.8	1.0	49.2	480035.3	99.9	**6.0**	49922.8	222.0	15.1	136850.7	81.7	82.6
$n-\sigma$	83.8	5773.4	2.3	47.0	699885.1	126.9	3.8	58466.5	231.4	13.7	103897.2	33.3	71.7
δ	**99.6**	3292.6	1.0	52.9	254264.1	74.8	0.0	-	-	**18.3**	98161.0	41.5	**85.1**
q/p	80.0	195896.5	17.7	13.2	135511.2	123.0	0.0	-	-	0.0	-	-	65.8
d^{opt}	88.9	25988.2	2.7	**55.0**	254347.0	68.8	0.0	-	-	16.0	185381.6	36.8	76.2
1 (×40)	88.4	130722.2	10.7	41.2	28165.2	15.8	0.0	-	-	0.0	-	-	73.8
Best Selec.													
ρ	**100.0**	184.8	0.0	75.0	7251.3	0.5	**40.0**	46211.8	261.9	25.7	4843.0	0.4	
$n-\sigma$	**100.0**	184.8	0.0	75.0	1009607.4	124.1	32.0	75445.9	351.0	25.7	4843.0	0.4	
δ	**100.0**	184.8	0.1	**100.0**	730687.4	89.5	0.0	-	-	25.7	4843.0	0.4	
q/p	98.8	7208.4	3.4	25.0	68.2	0.1	0.0	-	-	0.0	-	-	
d^{opt}	**100.0**	178.7	1.2	**100.0**	213028.8	129.1	0.0	-	-	**28.5**	29632.6	58.5	
1	99.7	58773.0	9.9	85.0	51740.9	36.9	0.0	-	-	0.0	-	-	

they also have the same (low) slack and the same (high) load. Therefore the heuristics based on these criteria (ie. *load*, *slack* and *usage rate*) cannot effectively discriminate values at the root of the search tree. However, recall that the load is defined as the product of the demand and the capacity. These two factors do not contribute equally, and therefore will favor different sets of options. In other words, one of them is bound to take a better decision, whilst the other is bound to take a worse one. We believe that this bias in the generation of the benchmarks explains the surprisingly good results of the demand (d^{opt}) as well as the bad results of the capacity q/p along with the *load*, the *slack* and the *usage rate*.

3.3.2.4 Aggregation

Aggregation functions are only used with *class* branching. For each parameter $\omega \in \{\leq_{lex}, \leq_{\sum}, \leq_{Euc}\}$, we have the 10 following heuristics combined with the propagators and the random runs (i.e., 200 tests for each ω and each instance):

- $\langle class, \{lex, mid\}, \{q/p, d^{opt}, \delta, n-\sigma, \rho\}, \omega\rangle$

The constant parameter for selection 1 is not considered in these experiments since it is only defined with the $\leq_{\sum}$ aggregation. The results are given in Table 3.8.

As we can see in the first part of this table, the three aggregation functions provide in average similar results except for the hardest instances (*set4*) where $\leq_{lex}$ solved more instances. Considering all instances, $\leq_{\sum}$ solves the largest number of problems. No solution was found for unsatisfiable instances as in our case, only *opt* branching can solve these instances (i.e., which by default does not use any aggregation function). However, regarding the best results in the second part of the table, when using $\leq_{lex}$ and $\leq_{Euc}$, one can obtain better performances in terms of resolved instances.

TABLE 3.8: Evaluation of the aggregation variants

Av. Agg. (×200)	*set1* $(70, S)$			*set2* $(4, S)$			*set3* $(5, U)$			*set4* $(7, S)$			Global
	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%tot
$\leq_{lex}$	94.9	26120.0	2.0	45.2	481410.8	84.9	0.0	-	-	**17.7**	98707.8	22.5	80.7
$\leq_{\Sigma}$	**95.8**	27209.1	2.1	**46.3**	327601.5	95.7	0.0	-	-	12.4	156300.3	44.6	**81.1**
$\leq_{Euc}$	95.7	27563.3	2.1	45.5	463196.6	107.9	0.0	-	-	13.2	107599.7	52.9	81.0
Best Agg.													
$\leq_{lex}$	**100.0**	184.7	0.0	**100.0**	730687.4	89.5	0.0	-	-	**28.5**	29632.6	58.5	
$\leq_{\Sigma}$	**100.0**	184.2	0.0	95.0	904739.2	96.3	0.0	-	-	25.7	34705.3	54.8	
$\leq_{Euc}$	**100.0**	184.4	0.0	**100.0**	211830.5	128.8	0.0	-	-	**28.5**	47435.1	75.4	

3.3.3 A Summary Regarding the Criteria

We have previously evaluated the average best choice of each criterion (in terms of solved instances). However, this choice is not the best on each set of instances. Instead, we can determine the best choice for each data set, called the "perfect" choice. The "`Confidence`" of the average best choice can then be defined by the ratio between the average best choice and the perfect choice. Similarly, we can consider the "worst" choice for each data set, and subsequently, define the "`Significance`" of a given factor using the ratio between the worst and the perfect choice as $1 - worst/perfect$.

TABLE 3.9: *Confidence* and *Significance* for each factor

	Confidence	*Significance*
Branching	0.989	0.247
Selection	0.995	0.231
Exploration	1.000	0.017
Aggregation	0.995	0.015

In Table 3.9, we give the values of *Confidence* and *Significance* for each factor (branching, selection, exploration, and aggregation).This table shows that there is high confidence for each selected average best choice (between 0.989 and 1.0): that is, exploration from middle to sides using a class branching, load selection, and a sum aggregation. When considering the *Significance* of each criterion, one can observe that only two of them (branching and selection) have a valuable impact. For the two other criteria (i.e., exploration and aggregation), there is little impact on the results when changing the parameters.

Therefore, the most robust heuristics will be those branching on classes variables and selecting options using the load criterion, that is $\langle class, \{lex, mid\}, \delta, \{\leq_{\Sigma}, \leq_{Euc}, \leq_{lex}\}\rangle$.

3.4 Search vs. Propagation

An empirical evaluation of our propositions regarding the propagation aspect is given in the next chapter. We consider here, however, how important is the search strategy compared to propagation. In addition to all the previous models, we consider a new one incorporating the Slack-Pruning (proposed in the next chapter, Section 4.1) within the decompose model. As we mention in Section 4.1, this rule can be applied only with *lex* branching. Therefore, we use the following set of heuristics $\langle\{class, opt\}, lex, \{1, q/p, d^{opt}, \delta, n-\sigma, \rho\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$. That is 21 different heuristics for each filtering algorithm. The experiments concern 9030 configurations per propagator.

Table 3.10: Evaluation of the filtering variants (averaged over all heuristics)

Filtering (×21)	*set1* (70 × 5)			*set2* (4 × 5)			*set3* (5 × 5)			*set4* (7 × 5)		
	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time
DECOMPOSE	75.8	190636.0	11.2	22.6	792179.8	44.4	0.0	-	-	7.7	194651.7	17.0
GSC	94.8	1639.4	4.2	44.0	38673.7	49.2	**2.8**	49417.9	260.8	12.1	35302.0	64.3
AMSC	91.2	36285.7	3.9	**49.2**	**411514.8**	**46.2**	1.5	68873.9	15.1	**13.1**	**239317.8**	**41.4**
GSC ⊕ AMSC	**95.1**	**1585.1**	**4.3**	44.0	35711.3	45.4	**2.8**	**46330.2**	**248.6**	12.5	32258.4	80.9
SLACK-PRUNING	90.5	55384.8	3.8	43.3	627443.4	43.9	1.7	82815.9	16.1	12.2	356073.4	34.8

Table 3.11: Best results for filtering variants

Filtering	*set1* (70 × 5)			*set2* (4 × 5)			*set3* (5 × 5)			*set4* (7 × 5)		
	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time	%sol	avg bts	time
DECOMPOSE	**100**	**184.8**	**0**	75	7251.3	0.5	0	-	-	25.7	4843	0.4
GSC	**100**	184.8	1.2	75	18073.7	58.2	**40**	46211.8	261.9	**28.5**	29632.6	58.5
AMSC	**100**	**184.8**	**0**	**100**	**730687.4**	**89.5**	20	60460.4	13.5	**28.5**	**31617.6**	**6**
GSC⊕AMSC	**100**	184.8	1.2	75	16923.7	55	**40**	**46196.7**	**259.7**	**28.5**	17252.6	40.8
SLACK-PRUNING	**100**	**184.3**	**0**	75	510189.0	35.1	20	70573.6	14	**28.5**	332430.9	34.3

Table 3.10 shows that the extra filtering of Slack-Pruning, AtMostSeqCard, or Gsc does help a lot. For instance, at least 90% of the instances of the first set are resolved irrespectively of the heuristic being used against 75,89% with the default decomposition (i.e., decompose). The difference is even greater for the other sets.

Consider now the propagation method as a fifth criterion (i.e., in addition to the heuristic factors). We calculate its *Confidence* and *Significance* according to the same formula given in Section 3.3.3. Their values are equal to 0.996 and 0.217, respectively. This is similar to the other criteria in terms of *Confidence* (i.e., close to 1.0), but slightly less than the *Significance* of branching and selection. This emphasizes the importance of these factors which are at least as important as the propagation level.

Overall, we observe that the choice of the search strategy has a very significant impact on the efficiency of the method. For instance, on the set of easiest instances, when averaging

across all heuristics, the "worst" filtering method (decomposition into sum constraints) is successful in about 20% less runs than the best (GSC+ ATMOSTSEQCARD). However, now averaging across all four models, the worst heuristic $\langle opt, lex, n-\sigma, - \rangle$, is successful 56% less runs than one of the many heuristics solving all easy instances (see Table 3.4). For harder instances (*set2*, *set3* and *set4*), these choices are even more important, with a 42% gap between the best and worst model, whilst the worst heuristics (in this case $\langle opt, lex, p/q, - \rangle$) do not solve any instances.

It is hardly a surprise to observe that the choice of search strategy is a critical one. However, whilst the aim of this study was to better understand what makes a good heuristic for the car-sequencing problem, it was relatively surprising to find out that minor variations around known heuristics would bring such a substantial gain.

Summary

We empirically studied in this chapter a large set of heuristics for the car-sequencing problem and proposed to classify these heuristics using 4 criteria: the type of branching decisions; the exploration order; the selection of the most constrained options; and the aggregation function for the options. Several new heuristics arise from this classification as untested combinations. Our experiments show that a single criterion can drastically impact the behavior of the heuristic. Moreover, it also gives a clear separation between the most important criteria (branching and selection) and the other factors (exploration and aggregation). Furthermore, this study shows that branching and variable ordering are as important as the propagation aspect in this problem.

Chapter 4

Propagation in Sequencing Problems

Introduction

Sequence constraints are useful in a number of applications. Constraints of this class enforce upper and/or lower bounds on all sub-sequences of variables of a given length within a main sequence. For instance, in crew-rostering, we may want to have an upper bound on the number of worked days in every sub-sequence to meet working regulations. Several constraints of this class have been studied in the CP literature such as Gen-Sequence and AmongSeq [111, 17, 32, 91, 139, 96]. An even more general constraint, Regular, can be used to enforce arbitrary patterns on all sub-sequences. However, as we explained in Section 2.1.3.2, the more general a constraint is, the higher is the complexity of reasoning about it. In this context, we focus on particular cases of sequence constraints where we have variables subject simultaneously to AtMost (i.e., of the form $\sum_{i=1}^{i=n} x_i \leq p$) and Cardinality (Section 2.3) constraints.

Our contributions start with a simple filtering rule that we call Slack-Pruning, dedicated to the car-sequencing problem. This rule reasons simultaneously about capacity and demand constraints. This simple filtering is generalized later as a new global constraint called AtMostSeqCard. The latter is useful in car-sequencing and crew-rostering problems. Following [139], ac on this constraint can be enforced with Gen-Sequence in $O(n^3)$ time or with cost-Regular in $O(2^q n)$ time where q is the size of the sliding window. Furthermore, the Gen-Sequence filtering of [91] is adaptable to AtMostSeqCard with $O(n^2.log(n))$ time complexity down a branch of the search tree with an initial compilation of $O(q.n^2)$. We propose a new algorithm achieving Arc Consistency on this constraint with an $O(n)$ (hence optimal) worst case time complexity. Next, we show that this algorithm can be easily modified to achieve

Arc Consistency on some extensions of this constraint. In particular, the conjunction of a set of m AtMostSeqCard constraints sharing the same scope can be filtered in $O(nm)$. The efficiency of our filtering is proven through a large experimental evaluation.

We start this chapter with the simple Slack-Pruning rule specially designed for solving the car-sequencing problem. Then, after giving a short background on sequence constraints in Section 4.2, we show how this reasoning can be generalized as a global constraint in Section 4.3. We show in Section 4.4 how to extend the new constraint without a computational overhead. The experimental results in Section 4.5 emphasize the efficiency of our filtering propositions.

4.1 Slack-Pruning

When analyzing the heuristics for the car-sequencing problem (Chapter 3), we have seen that selecting the options using load, slack, or usage rate is beneficial. In this section, we shall see that one can go one step further and use the same idea to prune the search tree at a very cheap computational cost. We suppose in this section that we are using the decompose model (Section 3.1.2) for the car-sequencing problem.

4.1.1 Triggering Failure via Slack

We first recall some of the notations that we used for car-sequencing in Section 3.1:

- n: the number of vehicles that have to be produced on the assembly line.
- k: the number of classes of vehicles.
- m: the number of types of options.
- d_c^{class}: the required demand for the class of vehicles c.
- d_j^{opt}: the required demand for the option j.
- $\mathcal{O}_c \subseteq \{1, \ldots, m\}$: the set of options defining the class of vehicles c.
- p_j and q_j: used to represent the capacity constraint related to an option j as follows: no subsequence of size q_j may contain more than p_j vehicles requiring option j.
- The load of an option j: $\delta_j = d_j^{opt} \times \frac{q_j}{p_j}$.
- The slack of an option j: $\sigma_j = n_j - \delta_j$ where n_j is the number of slots available for option j.

In [111], it is observed that if the slack (σ_j) of an option j is negative, then the problem is unsatisfiable. Indeed, the load (δ_j) tends to represent the number of required slots to mount all the occurrences of an option. Since the slack is the difference between the available number of slots and the load, a negative value suggests infeasibility since we need more slots than are available. However, one has to be careful about boundaries issues since the capacity constraints are truncated at the extremities of the assembly line. For instance, consider an option j with $p_j = 1$, $q_j = 3$ and $d_j^{opt} = 2$. The slack is negative as soon as there are less than six slots remaining ($n_j < 6$). However, a line with only four slots is sufficient if we put the two classes requiring this option on both ends of the line. In other words, the load is an accurate measure of how many slots are needed for a given option, however only for large values of demand and length of the assembly line.

We show in the following how to compute the the exact minimum number of slots to mount d_j^{opt} times an option j while respecting capacity constraints. We assume, however, that we explore the assembly line from left to right, and that the unassigned slots are contiguous in the assembly line.

Consider the following greedy rule (called `lex_assignment`):

1. Assign the first p_j variables to 1, and the $q_j - p_j$ next variables to 0.
2. Repeat step 1 ($\lceil d_j^{opt}/p_j \rceil - 1$) times.
3. Fill the remaining variables with the value 1.

Let δ'_j be the length of the sequence obtained by `lex_assignment`. The value of δ'_j is given by the formula:

$$\delta'_j = q_j(\lceil d_j^{opt}/p_j \rceil - 1) + \begin{cases} p_j & \text{if } d_j^{opt} \bmod p_j = 0 \\ d_j^{opt} \bmod p_j & \text{otherwise} \end{cases}$$

Proposition 4.1. *For each option j, δ'_j is the minimum number of contiguous slots to mount d_j^{opt} times the option j.*

Proof. The sequence returned by `lex_assignment` clearly satisfies all capacity constraints and has a cardinality equal to d_j^{opt}. Moreover, every subsequence of length q_j has exactly p_j times the value 1, therefore, it is not possible to obtain the same cardinality in a shorter sequence. Hence, δ'_j is the minimum length to mount d_j^{opt} times option j. □

In the following, the value of δ'_j is refereed as the 'real' load. Note that an equivalent formula can be found in [31].

4.1.2 Filtering the Domains

We suppose now that all variables up to a rank $i-1$ are assigned. To make the notation lighter we rename the sequence of unassigned variables $y_i, \ldots, y_n$ to: $y_0, \ldots, y_{n-i}$.

When the real load δ' is greater than the residual number of slots $n-i+1$, then we should fail since δ' is the minimum number of required slots. Moreover, we can prune inconsistent values in the domains of the option variables when the load is equal to the remaining number of slots. Khichane et al. [83] proposed to fix the first unassigned slot to contain the option at hand. We show that this filtering can be extended for many slots in the sequence. We illustrate this situation in Example 4.1.

Example 4.1. *Consider a sequence of unassigned variables* $y^j_0, \ldots, y^j_{16}$*, with capacity* 3/5 *and demand* 11*. Note that the load is* $\delta'_j = 5 \times (4-1) + 2 = 17$*, which is precisely equal to the number of unassigned slots. Consider the two slots indexed* 5 *and* 6*, corresponding to the variables* y^j_5 *and* y^j_6*. On the left, there are* 5 *slots, hence we can fit at most* 3 *vehicles with the option* j *since fitting* 4 *vehicles requires* $6 = 5(\lceil 4/3 \rceil - 1) + 4 \bmod 3$ *slots. Similarly, on the right, one cannot fit more than* 6 *vehicles with option* j *since fitting* 7 *vehicles would require* 11 *slots. Therefore, since the total demand is* 11*, we can conclude that* $11 - 6 - 3 = 2$ *vehicles with option* j *must fit in the slots* 5 *and* 6*. In other words, both* y^j_5 *and* y^j_6 *must be equal to* 1*. This example is depicted in Figure 4.1.*

FIGURE 4.1: Instantiation of an option with capacity 3/5.

y^j	0 **1**	1 **1**	2 1	3 0	4 0	5 **1**	6 **1**	7 1	8 0	9 0	10 **1**	11 **1**	12 1	13 0	14 0	15 **1**	16 **1**
	3					2		6									

Now we formally define the SLACK-PRUNING rule that can detect all such forced assignments (e.g., it detects all bold faced 1's in Figure 4.1).

Theorem 4.2. *The following filtering rule is correct:*

If $\delta'_j = n-i+1$*, then if* $d^{opt}_j \bmod p_j = 0$*, we impose* $y^j_i = 1$ *for all* i *such that* $i \bmod q_j < p_j$*. Otherwise (i.e.,* $d^{opt}_j \bmod p_j \neq 0$*), we impose* $y^j_i = 1$ *for all* i *such that* $i \bmod q_j < (d^{opt}_j \bmod p_j)$*.*

Proof. Suppose that $(d^{opt}_j \bmod p_j \neq 0)$. Then there exists two integers k and r such that $d^{opt}_j = k.p_j + r$. Notice that in this case, we have $\delta'_j = q_j.k + r$. Consider a subsequence $y^j_a, \ldots, y^j_b$ such that $a \bmod q_j = 0$ and $b = a + r - 1$, i.e., such that the rule above applies. Then there exist two integers α and β such that $a = \alpha \cdot q_j$ and $n - i - b = \beta \cdot q_j$ (since $n - i + 1 = \delta'_j = q_j.k + r$).

Now using $n-i-b = \beta \cdot q_j$, we show that $n-i+1 = \beta \cdot q_j + a + r$ then $n-i+1 = (\alpha+\beta) \cdot q_j + r$ and hence $k = \alpha + \beta$ (since $n - i + 1 = q_j.k + r$).

However, by definition of α and β, we may argue that the number of occurrences of the value 1 on $y_0^j, \ldots, y_{a-1}^j$ is at most $\alpha \cdot p_j$ and at most $\beta \cdot pj$ on $y_{b+1}^j, \ldots, y_{n-i}^j$.

Now since the demand $d_j^{opt} = (\alpha + \beta).p_j + r$ then all the p_j variables in the subsequence $y_a^j, \ldots, y_b^j$ must take the value 1.

We use a similar argument for the second case. Suppose that $d_j^{opt} \bmod p_j = 0$, consider a subsequence $y_a^j, \ldots, y_b^j$ such that $a \bmod q_j = 0$ and $b = a + p_j - 1$. Then there exist two integers α and β such that $a = \alpha \cdot q_j$ and $n - i - b = \beta \cdot q_j$. Therefore, the number of occurrences of the value 1 on $y_0^j, \ldots, y_{a-1}^j$ is at most $\alpha \cdot p_j$ and at most $\beta \cdot p_j$ on $y_{b+1}^j, \ldots, y_{n-i}^j$.

Now using the demand $d_j^{opt} = k \cdot p_j$, and $\delta' = q_j(\lceil d_j^{opt}/p_j \rceil - 1) + p_j$ we show that $n - i + 1 = q_j(k-1) + p$. However, since $b = a + p_j - 1$, $a = \alpha \cdot q_j$ and $n - i - b = \beta \cdot q_j$, then $k = \alpha + \beta + 1$ and all p_j variables the subsequence $y_a^j, \ldots, y_b^j$ must take the value 1. □

Figure 4.2 and 4.3 depict the proposed pruning. On the one hand, when $d_j^{opt} \bmod p_j = 0$, the only possible arrangement of vehicles that satisfy the capacity constraint is to start the sequence with p_j vehicles requiring the option, then $q_j - p_j$ vehicles not requiring the option and repeat (see Figure 4.2). Notice that because of the capacity constraint, all other variables must take the value 0. On the other hand, when $d_j^{opt} \bmod p_j \neq 0$, one must start the sequence with $d_j^{opt} \bmod p_j$ vehicles requiring the option, then the following $q_j - (d_j^{opt} \bmod p_j)$ slots can be filled arbitrarily as long as exactly p_j vehicles requiring this options are fitted in the q_j first slots. Here again, the initial sequence must be repeated throughout (see Figure 4.3).

FIGURE 4.2: Filtering when $d_j^{opt} \bmod p_j = 0$

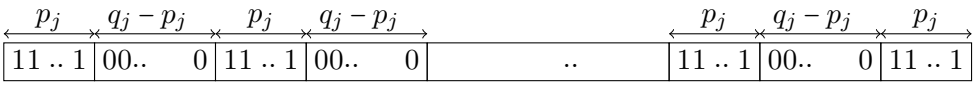


FIGURE 4.3: Filtering when $r = d_j^{opt} \bmod p_j \neq 0$



4.1.3 Time Complexity

This rule is extremely cheap to enforce. Once one has computed the real load, the domain filtering can be achieved in $O(k)$ where k is the number of option variables forced to take the value 1. Indeed, when $d_j^{opt} \bmod p_j \neq 0$ we can jump over the variables which are not forced to take the value 1, since their position is given by a simple recursion. In the worst case (i.e., when $d_j^{opt} \bmod p_j = 0$), k is equal to the number of unassigned variables and therefore the time complexity can reach $O(n)$.

In the next sections, we generalize the SLACK-PRUNING rule in the form of an Arc Consistency algorithm for a new global constraint that we call ATMOSTSEQCARD. The latter can be used for solving a large family of sequencing problems. This constraint will be introduced after a short background regarding sequence constraints.

4.2 SEQUENCE Constraints

There are several variants of the SEQUENCE constraints. We first review them and then motivate the need for the variant proposed in this chapter: the ATMOSTSEQCARD constraint which extends the SLACK-PRUNING.

4.2.1 Decomposition via SLIDE

We start with an important decomposition property related to sequence constraints introduced in [23]. For any constraint C, we denote by $\mathcal{D}_C^{\cup}$ the set of values $\bigcup_{x \in \mathcal{X}(C)} \mathcal{D}(x)$.

Definition 4.3. Monotonicity

A constraint C is said to be monotone iff there exists a total order $\prec$ on $\mathcal{D}_C^{\cup}$ s.t. for any two values $\alpha \prec \beta$, α can replace β in any support on C.

Example 4.2. *A monotone constraint*

Let $\sum_{i=1}^{i=n} x_i \leq p$ be the constraint ensuring that the sum of the Boolean variables $x_1 \dots x_n$ is at most p. We show that this constraint is monotone. The total order $\prec$ chosen here is the 'less than' (i.e. $<$) operator on integers. It is clear that the value 0 can replace the value 1 in any support for this constraint. □

Definition 4.4. The SLIDE meta-constraint

Let C be a constraint of arity k. The SLIDE$(C, [x_1, \dots, x_n])$ constraint is defined by the conjunction of all $C([x_i, \dots, x_{i+k-1}])$ where $i \in [1, n-k+1]$.

The SLIDE (meta-)constraint can be used to model many sequencing problems. The idea is to slide the same 'type' of constraints over a sequence of variables.

Theorem 4.5. *Arc Consistency on* SLIDE *[23]*

If C is monotone then AC *on* SLIDE$(C, [x_1, \dots, x_n])$ *is equivalent to* AC *on each constraint C.*

Theorem 4.5 gives an easy sufficient condition for making the decomposition of SLIDE not hindering propagation. This property is used in Section 4.2.2.1 to decompose a chain of ATMOST constraints.

4.2.2 Chains of AMONG Constraints:

In the following definitions, ν is a set of integers and l, u, q are integers. Sequence constraints are conjunctions of AMONG constraints, constraining the number of occurrences of a set of values in a set of variables.

Definition 4.6. $\text{AMONG}(l, u, [x_1, \ldots, x_q], \nu) \Leftrightarrow l \leq \left|\{i \mid x_i \in \nu\}\right| \leq u$

The AMONGSEQ constraint, first introduced in [17], is a chain of AMONG constraints of width q slid along a vector of n variables.

Definition 4.7. $\text{AMONGSEQ}(l, u, q, [x_1, \ldots, x_n], \nu) \Leftrightarrow \bigwedge_{i=0}^{n-q} \text{AMONG}(l, u, [x_{i+1}, \ldots, x_{i+q}], \nu)$

Note first that AMONG is not monotone in general. Therefore Theorem 4.5 does not apply and AC on each AMONG will not necessarily establish AC on AMONGSEQ. We use the same example given in [139] to show how decomposition hinders propagation. In $\text{AMONGSEQ}(2, 3, 5, [x_1, \ldots, x_7], \{1\})$ where $\mathcal{D}(x_1) = \mathcal{D}(x_2) = 1$, $\mathcal{D}(x_3) = \mathcal{D}(x_4) = \mathcal{D}(x_5) = \mathcal{D}(x_7) = \{0, 1\}$, and $\mathcal{D}(x_6) = 0$, each AMONG constraint is AC while the assignment $x_7 \leftarrow 0$ does not have a support on AMONGSEQ.

The first (incomplete) algorithm for filtering this constraint was proposed in 2001 [15]. Then, in [139, 138], two complete algorithms for filtering the AMONGSEQ constraint were introduced: firstly, a reformulation using the REGULAR constraint using 2^{q-1} states achieving AC in $O(2^q n)$ time; secondly, an algorithm achieving AC with a worst case time complexity of $O(n^3)$. Moreover, this last algorithm is able to handle arbitrary sets of AMONG constraints on consecutive variables (denoted GEN-SEQUENCE), however in $O(n^4)$. Last, two flow-based algorithms were introduced in [91]. The first achieves AC on AMONGSEQ in $O(n^{3/2} \log n \log p)$, while the second achieves AC on GEN-SEQUENCE in $O(n^3)$ in the worst case. These two algorithms have an amortized complexity down a branch of the search tree of $O(n^2)$ and $O(n^3)$, respectively.

4.2.2.1 Chain of ATMOST Constraints

Although useful in both applications, the AMONGSEQ constraint does not model exactly the type of sequences useful in car-sequencing and crew-rostering applications. First, there is often no lower bound for the cardinality of the subsequences, i.e., $l = 0$. Therefore AMONGSEQ is unnecessarily general in that respect. Moreover, the capacity constraint on subsequences is often paired with a cardinality requirement.

For instance, in car-sequencing, classes of car requiring a given option cannot be clustered together, because a working station can only handle a fraction of the cars passing on the line (*at most* p times in any sequence of length q). The total number of occurrences of

these classes is a requirement, and translates as an overall cardinality constraint rather than lower bounds on each subsequence.

In crew-rostering, allowed shift patterns can be complex, hence the REGULAR constraint is often used to model them. However, working in at most p shifts out of q is a useful particular case. If days are divided into three 8h shifts, ATMOSTSEQ with $p = 1$ and $q = 3$ makes sure that no employee work more than one shift per day *and* that there must be a 24h break when changing shifts. Moreover, similarly to car-sequencing, the lower bound on the number of worked shifts is global (monthly, for instance). In other words, we often have a chain of ATMOST constraints.

Definition 4.8. $\text{ATMOST}(p, [x_1, \ldots, x_q], \nu) \Leftrightarrow \text{AMONG}(0, p, [x_1, \ldots, x_q], \nu)$

To simplify notation, when the variables are Boolean and $\nu = \{1\}$, we denote by $\text{ATMOST}([x_1, \ldots, x_q], p)$ the $\text{ATMOST}(p, [x_1, \ldots, x_q], \nu)$ constraint. Note that $\text{ATMOST}([x_1, \ldots, x_q], p)$ is in fact the monotone constraint $\sum_{i=1}^{i=n} x_i \leq p$ given in Example 4.2. We can easily show that the general $\text{ATMOST}(p, [x_1, \ldots, x_q], \nu)$ is similarly monotone.

A chain of ATMOST constraints can be defined as follows:

Definition 4.9. $\text{ATMOSTSEQ}(p, q, [x_1, \ldots, x_n], \nu) \Leftrightarrow \bigwedge_{i=0}^{n-q} \text{ATMOST}(p, [x_{i+1}, \ldots, x_{i+q}], \nu)$

Observe that AC on ATMOSTSEQ is maintained using the decomposition of definition 4.9. In fact since ATMOST is monotone, then Arc Consistency is established on ATMOSTSEQ iff each ATMOST is AC.

A good tradeoff between filtering power and complexity can be achieved by reasoning about the total number of occurrences of values from the set ν together with the chain of ATMOST constraints.[1] We therefore introduce the ATMOSTSEQCARD constraint, defined as the conjunction of an ATMOSTSEQ with a cardinality constraint on the total number of occurrences of values in ν:

Definition 4.10. $\text{ATMOSTSEQCARD}(p, q, d, [x_1, \ldots, x_n], \nu) \Leftrightarrow$

$$\text{ATMOSTSEQ}(p, q, [x_1, \ldots, x_n], \nu) \wedge \left|\{i \mid x_i \in \nu\}\right| = d$$

The two AC algorithms introduced in [138] were adapted in [139] to achieve AC on the ATMOSTSEQCARD constraint. First, in the same way that AMONGSEQ can be encoded with a REGULAR constraint, ATMOSTSEQCARD can be encoded with a COST-REGULAR constraint, where the cost stands for the overall demand, and it is increased on transitions labeled with the value 1. This procedure has the same worst case time complexity, i.e., $O(2^q n)$ [139]. Second, the more general version of the polynomial algorithm

[1] This modeling choice is used in [139] on car-sequencing.

(Gen-Sequence) is used, to filter the following decomposition of the AtMostSeqCard constraint into a conjunction of Among:

$$\text{AtMostSeqCard}(p,q,d,[x_1,\ldots,x_n],\nu) \quad \Leftrightarrow \quad \bigwedge_{i=0}^{n-q} \text{Among}(0,p,[x_{i+1},\ldots,x_{i+q}],\nu) \quad \wedge \quad \text{Among}(d,d,[x_1,\ldots,x_n],\nu)$$

The algorithm of van Hoeve et al. [139] runs in $O(n^3)$ time complexity on this decomposition. Similarly, the algorithm of Maher et al. [91] runs in $O(n^2.log(n))$ down a branch of the search tree with an $O(q.n^2)$ initial compilation. The algorithm we propose in this chapter (first published as [120]) runs in linear time and is therefore optimal. Finally, another linear time algorithm based on the graph representation of [91] was subsequently proposed by Narodytska and Walsh in [136].

4.2.2.2 Global Sequencing Constraint

The Global Sequencing Constraint that we introduced in Definition 2.11 is in fact nothing but a conjunction between an AmongSeq and a Gcc. That is:

Definition 4.11. $\text{Gsc}(l,u,q,low,upp,[x_1,\ldots,x_n],\nu) \Leftrightarrow$

$$\text{AmongSeq}(l,u,q,[x_1,..,x_n],\nu) \wedge \text{Gcc}(low,upp,[x_1,..,x_n])$$

4.3 The AtMostSeqCard Constraint

In this section, we introduce a linear filtering algorithm for the AtMostSeqCard constraint. We first give a simple greedy algorithm for finding a support with an $O(nq)$ time complexity. Then, we show that one can use two calls to this procedure to enforce ac. Last, we show that its worst case time complexity can be reduced to $O(n)$.

It was observed in [139] and [91] that we can consider Boolean variables and $\nu = \{1\}$, since the following decomposition of Among (or AtMost) does not hinder propagation as it is Berge acyclic:

$$\text{Among}(l,u,[x_1,\ldots,x_q],\nu) \Leftrightarrow \bigwedge_{i=1}^{q}(x_i \in \nu \leftrightarrow x'_i = 1) \quad \wedge \quad l \leq \sum_{i=1}^{q} x'_i \leq u$$

Therefore, throughout the chapter, we consider $[x_1,\ldots,x_n]$ as a sequence of Boolean variables, and use the following restriction of the AtMostSeqCard constraint with $\nu = \{1\}$:

Definition 4.12.

$$\textsc{AtMostSeqCard}(p,q,d,[x_1,\ldots,x_n]) \Leftrightarrow \bigwedge_{i=0}^{n-q}(\sum_{l=1}^{q} x_{i+l} \leq p) \wedge (\sum_{i=1}^{n} x_i = d)$$

4.3.1 Finding a Support

Let w be an n-tuple in $\{0,1\}^n$, $\left|w\right| = \sum_{i=1}^{n} w[i]$ its cardinality, and $w[i:j]$ the projection of w on the subsequence $[x_i,\ldots,x_j]$.

We first show that one can find a support by greedily assigning variables in a lexicographical order to the value 1 whenever possible, that is, while taking care of not violating the AtMostSeq constraint. More precisely, doing so leads to an instantiation of maximal cardinality, which may easily be transformed into an instantiation of cardinality d.

The greedy procedure `leftmost` (Algorithm 9) computes an instantiation w that maximizes the cardinality of the sequence $(x_1,\ldots,x_n)$ subject to an AtMostSeq constraint (with parameters p and q),

Algorithm 9: `leftmost`

```
1  foreach i ∈ [1,...,n] do w[i] ← min(x_i);
   foreach i ∈ [1,...,q] do w[n+i] ← 0;
   c(1) ← w[1];
   foreach j ∈ [2,...,q] do c(j) ← c(j−1) + w[j];
   foreach i ∈ [1,...,n] do
2      if |D(x_i)| > 1 & max_{j∈[1,q]}(c(j)) < p then
           w[i] ← 1;
3          foreach j ∈ [1,...,q] do c(j) ← c(j) + 1;
4      foreach j ∈ [2,...,q] do c(j−1) ← c(j);
5      c(q) ← c(q−1) + w[i+q] − w[i];
   return w;
```

Algorithm `leftmost` works as follows. First, the tuple w is initialized to the minimum value in the domain of each variable in Line 1. Then, at each step $i \in [1,\ldots,n]$ of the main loop, the cardinality of the j^{th} subsequence involving the variable x_i with respect to the current value of w is stored in $c(j)$. In other words, at step i, we have $c(j) = \sum_{l=max(1,i-q+j)}^{min(n,i+j-1)} w[l]$.

When exploring variable x_i, such that $\mathcal{D}(x_i) = \{0,1\}$ we set $w[i]$ to 1 iff this would not violate the capacity constraints, that is, if $c(j) < p$ for all $j \in [1,\ldots,q]$ (Condition Line 2). In that case, the cardinality of every subsequence involving x_i is incremented (Line 3). Finally, when moving to the next variable, the values of $c(j)$ are shifted (Line 4), and the value of $c(q)$ is obtained by adding the value of $w[i+q]$ and subtracting $w[i]$ to its

previous value (Line 5).

From now on, we shall use the following notations:

- $\overrightarrow{w}$ denotes the instantiation found by `leftmost` on the sequence $x_1, \ldots, x_n$.
- $\overleftarrow{w}$ denotes the instantiation found by the same algorithm, however on the sequence $x_n, \ldots, x_1$, that is, from right to left. Notice that, in order to simplify the notations, $\overleftarrow{w}[i]$ shall denote the value assigned by `leftmost` to the variable x_i, and not x_{n-i+1} as it would actually be if we gave the reversed sequence as input.

Example 4.3. *We illustrate the behavior of* `leftmost` *on a simple example (see Figure 4.4). Let* $[x_1, \ldots, x_{22}]$ *be a sequence of variables with a capacity constraint of* 2/4*, that is, constrained by:* AtMostSeq$(2, 4, [x_1, \ldots, x_{22}])$*. Dots in the first row stand for unassigned variables. The second row shows the computed instantiation* $\overrightarrow{w}$*, and the next rows show the state of the variables* $c(1), c(2), c(3)$ *and* $c(4)$ *at the start of each iteration of the main loop. The last row stands for the maximum value of* $c(j)$*. The bold values indicate that* `leftmost` *assigns the value* 1*.*

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	**1**	0	0	1	**1**	0	0	0	**1**	0	1	0	0	1	**1**	0	0	**1**	**1**	0	0	1
$c(1)$	**0**	1	1	2	**1**	2	2	1	**0**	0	2	2	1	2	**1**	2	2	**1**	**1**	2	2	2
$c(2)$	**0**	1	2	1	**1**	2	1	0	**0**	2	2	1	2	1	**1**	2	1	**0**	**1**	2	2	1
$c(3)$	**0**	2	1	1	**1**	1	0	0	**1**	2	1	2	1	1	**1**	1	0	**0**	**1**	2	1	1
$c(4)$	**1**	1	1	1	**0**	0	0	1	**1**	1	2	1	1	1	**0**	0	0	**1**	**1**	1	1	1
$max(c)$	**1**	2	2	2	**1**	2	2	1	**1**	2	2	2	2	2	**1**	2	2	**1**	**1**	2	2	2

FIGURE 4.4: Sequence of maximum cardinality obtained by `leftmost`.

Lemma 4.13. `leftmost` *maximizes* $\sum_{i=1}^{n} x_i$ *subject to* AtMostSeq$(p, q, [x_1, \ldots, x_n])$.

Proof. Let $\overrightarrow{w}$ be the instantiation found by `leftmost`, and suppose that there exists another instantiation w (consistent for AtMostSeq$(p, q, [x_1, \ldots, x_n])$) such that $\left|w\right| > \left|\overrightarrow{w}\right|$. Let i be the smallest index such that $\overrightarrow{w}[i] \neq w[i]$. By definition of $\overrightarrow{w}$, we know that $\overrightarrow{w}[i] = 1$ hence $w[i] = 0$. Now, let j be the smallest index such that $\overrightarrow{w}[j] < w[j]$ (it must exists since $\left|w\right| > \left|\overrightarrow{w}\right|$).

We first argue that the instantiation w' equal to w except that $w'[i] = 1$ and $w'[j] = 0$ (as in $\overrightarrow{w}$) is consistent for AtMostSeq. Clearly, its cardinality is not affected by this swap, hence $\left|w'\right| = \left|w\right|$. Now, we consider all the sum constraints whose scopes are changed by this swap, that is, the sums $\sum_{l=a}^{a+q-1} w'[l]$ on intervals $[a, a+q-1]$ such that $a \leq i < a+q$ or $a \leq j < a+q$. There are three cases:

1. Suppose first that $a \leq i < j < a + q$: in this case, the value of the sum is the same in w and w', therefore it is lower than or equal to p.

2. Suppose now that $i < a \leq j < a + q$: in this case, the value of the sum is decreased by 1 from w to w', therefore it is lower than or equal to p.

3. Last, suppose that $a \leq i < a + q \leq j$: in this case, for any $l \in [a, a + q - 1]$, we have $w'[l] \leq \overrightarrow{w}[l]$ since j is the smallest integer such that $\overrightarrow{w}[j] < w[j]$, hence the sum is lower than or equal to p.

Therefore, given a sequence w of maximum cardinality that differs from $\overrightarrow{w}$ at rank i, we can build a sequence of equal cardinality that does not differ from $\overrightarrow{w}$ until rank $i + 1$. By iteratively applying this argument, we can obtain a sequence identical to $\overrightarrow{w}$, albeit with cardinality $\left|w\right|$, therefore contradicting our hypothesis that $\left|w\right| > \left|\overrightarrow{w}\right|$. □

Corollary 4.14. *Let $\overrightarrow{w}$ be the instantiation returned by* `leftmost`*. There exists a solution of* AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n])$ *iff the three following propositions hold:*

(1) AtMostSeq$(p, q, [x_1, \ldots, x_n])$ is satisfiable *(2)* $\sum_{i=1}^{n} min(x_i) \leq d$

(3) $\left|\overrightarrow{w}\right| \geq d$.

Proof. It is easy to see that these conditions are all necessary: (1) and (2) come from the definition, and (3) is a direct application of Lemma 4.13. Now, we prove that they are sufficient by showing that if these properties hold, then a solution exists. Since AtMostSeq$(p, q, [x_1, \ldots, x_n])$ is satisfiable, $\overrightarrow{w}$ does not violate the chain of AtMost constraints as the value 1 is assigned to x_i only if all subsequences involving x_i have cardinality $p - 1$ or less. Moreover, since $\sum_{i=1}^{n} min(x_i) \leq d \leq \left|\overrightarrow{w}\right|$, then there are at least $\left|\overrightarrow{w}\right| - d$ variables such that $min(x_i) = 0$ and $\overrightarrow{w}[i] = 1$. Assigning them to 0 in $\overrightarrow{w}$ does not violate the AtMostSeq constraint. Hence we can build a support for AtMostSeqCard. □

Lemma 4.13 and Corollary 4.14 give us a polynomial support-seeking procedure for AtMostSeqCard. Indeed, the worst case time complexity of Algorithm 9 is in $O(nq)$. There are n steps and on each step, Lines 2, 3 and 4 involve $O(q)$ operations. Therefore, for each variable x_i, a support for $x_i = 0$ or $x_i = 1$ can be found in $O(nq)$. Consequently, we have a naive AC procedure running in $O(n^2q)$ time.

4.3.2 Filtering the Domains

In this section, we show that we can filter out all the values inconsistent with respect to the AtMostSeqCard constraint within the same time complexity as Algorithm 9.

First, we show that there can be inconsistent values only in the case where the cardinality $\left|\overrightarrow{w}\right|$ of the instantiation returned by `leftmost` is exactly d: in any other case, the constraint is either violated (when $\left|\overrightarrow{w}\right| < d$) or AC, (when $\left|\overrightarrow{w}\right| > d$). The following lemma formalizes this:

Lemma 4.15. *The constraint* AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n])$ *is* AC *if the three following propositions hold:*

1. AtMostSeq$(p, q, [x_1, \ldots, x_n])$ *is* AC
2. $\sum_{i=1}^{n} min(x_i) \leq d$
3. $\left|\overrightarrow{w}\right| > d$

Proof. By Corollary 4.14 we know that AtMostSeqCard$(p, q, d+1, [x_1, \ldots, x_n])$ is satisfiable. Let w be a satisfying instantiation, and consider without loss of generality a variable x_i such that $\left|\mathcal{D}(x_i)\right| > 1$. Assume first that $w[i] = 1$. The solution w' equal to w except that $w'[i] = 0$ satisfies AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n])$. Indeed, $\left|w'\right| = \left|w\right| - 1 = d$ and since AtMostSeq$(p, q, [x_1, \ldots, x_n])$ was satisfied by w it must be satisfied by w'. Hence, for every variable x_i such that $\left|\mathcal{D}(x_i)\right| > 1$, there exists a support for $x_i = 0$.

Suppose that $w[i] = 0$, and let a (respectively b) be the largest (respectively smallest) index such that $a < i$, $w[a] = 1$ and $\mathcal{D}(x_a) = \{0, 1\}$ (respectively $b > i$, $w[b] = 1$ and $\mathcal{D}(x_b) = \{0, 1\}$). Let w' be the instantiation such that $w'[i] = 1$, $w'[a] = 0$, $w'[b] = 0$, and $w = w'$ otherwise. We have $\left|w'\right| = d$, and we show that it satisfies AtMostSeq$(p, q, [x_1, \ldots, x_n])$. Consider a subsequence $x_j, \ldots, x_{j+q-1}$. If $j + q \leq i$ or $j > i$ then $\sum_{l=j}^{j+q-1} w'[l] \leq \sum_{l=j}^{j+q-1} w[l] \leq p$, so only indices j s.t. $j \leq i < j + q$ matter. There are two cases:

1. Either a or b or both are in the subsequence ($j \leq a < j + q$ or $j \leq b < j + q$). In that case $\sum_{l=j}^{j+q-1} w'[l] \leq \sum_{l=j}^{j+q-1} w[l]$.
2. Neither a nor b are in the subsequence ($a < j$ and $j + q \leq b$). In that case, since $\mathcal{D}(x_i) = \{0, 1\}$ and since AtMostSeq$(p, q, [x_1, \ldots, x_n])$ is AC, we know that $\sum_{l=j}^{j+q-1} min(x_l) < p$. Moreover, since $a < j$ and $j + q \leq b$, there is no variable x_l in that subsequence such that $w[l] = 1$ and $0 \in \mathcal{D}(x_l)$. Therefore, we have $\sum_{l=j}^{j+q-1} w[l] < p$, hence $\sum_{l=j}^{j+q-1} w'[l] \leq p$.

In both cases w' satisfies all capacity constraints. Hence it is support for the value 1. □

Remember that achieving AC on AtMostSeq is trivial since AtMost is monotone. Therefore we focus of the case where AtMostSeq is AC, and $\left|\overrightarrow{w}\right| = d$. In particular,

Lemmas 4.16, 4.17, 4.19 and 4.20 only apply in that case. The equality $\left|\overrightarrow{w}\right| = d$ is therefore implicitly assumed in all of them.

Lemma 4.16. *If* $\left|\overrightarrow{w}[1:i-1]\right| + \left|\overleftarrow{w}[i+1:n]\right| < d$ *then* $x_i = 0$ *is not* AC.

Proof. Suppose that we have $\left|\overrightarrow{w}[1:i-1]\right| + \left|\overleftarrow{w}[i+1:n]\right| < d$ and suppose that there exists a consistent instantiation w such that $w[i] = 0$ and $\left|w\right| = d$.

By Lemma 4.13 on the sequence $x_1, \ldots, x_{i-1}$ we know that $\sum_{l=1}^{i-1} w[l] \leq \left|\overrightarrow{w}[1:i-1]\right|$.

By Lemma 4.13 on the sequence $x_n, \ldots, x_{i+1}$ we know that $\sum_{l=i+1}^{n} w[l] \leq \left|\overleftarrow{w}[i+1:n]\right|$.

Therefore, since $w[i] = 0$, we have $\left|w\right| = \sum_{l=1}^{n} w[l] < d$, thus contradicting the hypothesis that $\left|w\right| = d$. Hence, there is no support for $x_i = 0$. □

Lemma 4.17. *If* $\left|\overrightarrow{w}[1:i]\right| + \left|\overleftarrow{w}[i:n]\right| \leq d$ *then* $x_i = 1$ *is not* AC.

Proof. Suppose that we have $\left|\overrightarrow{w}[1:i]\right| + \left|\overleftarrow{w}[i:n]\right| \leq d$ and suppose that there exists a consistent instantiation w' such that $w'[i] = 1$ and $\left|w'\right| = d$.

By Lemma 4.13 on the sequence $x_1, \ldots, x_i$ we know that $\sum_{l=1}^{i} w'[l] \leq \left|\overrightarrow{w}[1:i]\right|$.

By Lemma 4.13 on the sequence $x_n, \ldots, x_i$ we know that $\sum_{l=i}^{n} w'[l] \leq \left|\overleftarrow{w}[i:n]\right|$.

Therefore, since $w'[i] = 1$, we have $\left|w'\right| = \sum_{l=1}^{i} w'[l] + \sum_{l=i}^{n} w'[l] - 1 < d$, thus contradicting the hypothesis that $\left|w'\right| = d$. Hence there is no support for $x_i = 1$. □

Lemmas 4.16 and 4.17 entail a pruning rule. In a first pass, from left to right, one can use an algorithm similar to `leftmost` to compute and store the values of $\left|\overrightarrow{w}[1:i]\right|$ for all $i \in [1, \ldots, n]$. In a second pass, the values of $\left|\overleftarrow{w}[i:n]\right|$ for all $i \in [1, \ldots, n]$ are similarly computed by simply running the same procedure on the same sequence of variables, however *reversed*, i.e., from right to left. Using these values, one can then apply Lemma 4.16 and Lemma 4.17 to filter out the value 0 and 1, respectively. We detail this procedure in the next section.

We first show that these two rules are complete, that is, if AtMostSeq is AC, and the overall cardinality constraint is AC then an instantiation $x_i = 0$ (respectively $x_i = 1$) is inconsistent iff Lemma 4.16 (respectively Lemma 4.17) applies. The following Lemma shows that the greedy rule maximizes the density of 1s on any subsequence starting on x_1, and therefore minimizes it on any subsequence finishing on x_n. Let `leftmost`(k) denote the algorithm corresponding to applying `leftmost`, however stopping whenever the cardinality of the instantiation reaches a given value k.

Lemma 4.18. *Let* w *be a satisfying instantiation of* AtMostSeq$(p, q, [x_1, \ldots, x_n])$. *If* $k \leq \left|w\right|$ *then the instantiation* $\overrightarrow{w}_k$ *computed by* `leftmost`(k) *is such that, for any* $1 \leq i \leq n$: $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$.

Proof. Let m be the index at which $\texttt{leftmost}(k)$ stops. We distinguish two cases. If $i > m$, for any value l in $[m+1,\dots,n]$, $\overrightarrow{w}_k[l] \leq w[l]$ (since $\overrightarrow{w}_k[l] = min(x_l)$), hence $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. When $i \leq m$, clearly for $i = 1$, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$ since $\left|\overrightarrow{w}_k\right| \leq \left|w\right|$. Now consider the case of $i \neq 1$. Since $\left|\overrightarrow{w}_k\right| \leq \left|w\right|$, then $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \left|w\right| - \sum_{l=1}^{i-1} \overrightarrow{w}_k[l]$. Thus, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l] + (\sum_{l=1}^{i-1} w[l] - \sum_{l=1}^{i-1} \overrightarrow{w}_k[l])$. Moreover, by applying Lemma 4.13, we show that $\sum_{l=1}^{i-1} \overrightarrow{w}_k[l]$ is maximum, hence larger than or equal to $\sum_{l=1}^{i-1} w[l]$. Therefore, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. □

Lemma 4.19. *If* AtMostSeq$(p, q, [x_1, \dots, x_n])$ *is* AC*, and* $\left|\overrightarrow{w}[1:i-1]\right| + \left|\overleftarrow{w}[i+1:n]\right| \geq d$ *then* $x_i = 0$ *has a support.*

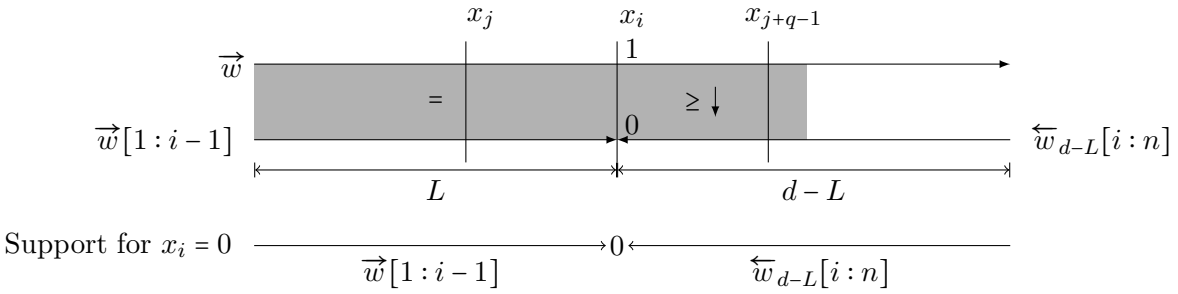


Figure 4.5: Illustration of Lemma 4.19's proof. Horizontal arrows represent assignments.

Proof. Let $\overrightarrow{w}$ be the instantiation found by $\texttt{leftmost}$. We consider, without loss of generality, a variable x_i such that $\mathcal{D}(x_i) = \{0,1\}$ and $\left|\overrightarrow{w}[1:i-1]\right| + \left|\overleftarrow{w}[i+1:n]\right| \geq d$, and show that one can build a support for $x_i = 0$. If $\overrightarrow{w}[i] = 0$ or $\overleftarrow{w}[i] = 0$ then there exists a support for $x_i = 0$, hence we only need to consider the case where $\overrightarrow{w}[i] = \overleftarrow{w}[i] = 1$.

Let $L = \left|\overrightarrow{w}[1:i-1]\right|$ and $\overleftarrow{w}_{d-L}$ be the result of $\texttt{leftmost}(d-L)$ on the subsequence $x_n, \dots, x_i$. We will show that w, defined as the concatenation of $\overrightarrow{w}[1:i-1]$ and $\overleftarrow{w}_{d-L}[i:n]$ is a support for $x_i = 0$.

First, we show that $w[i] = 0$, that is $\overleftarrow{w}_{d-L}[i] = 0$. By hypothesis, since $\left|\overrightarrow{w}[1:i-1]\right| + \left|\overleftarrow{w}[i+1:n]\right| \geq d$, we have $\overleftarrow{w}[i+1:n] \geq d-L$. Now, consider the sequence $x_i, \dots, x_n$, and let w' be the instantiation such that $w'[i] = 0$, and $w' = \overleftarrow{w}[i+1:n]$ otherwise. Since $w' = \overleftarrow{w}[i+1:n] \geq d-L$, by Lemma 4.18, we know that w' has a higher cardinality than $\overleftarrow{w}_{d-L}$ on any subsequence starting in i, hence $w[i] = \overleftarrow{w}_{d-L}[i] = w'[i] = 0$.

Now, we show that w does not violate the AtMostSeq constraint. Obviously, since it is the concatenation of two consistent instantiations, it can violate the constraint only on the junction, i.e., on a subsequence $x_j, \dots, x_{j+q-1}$ such that $j \leq i$ and $i < j+q$.

We show that the sum of any such subsequence is less than or equal to p by comparing with $\overrightarrow{w}$, as illustrated in Figure 4.5. We have $\sum_{l=j}^{j+q-1} \overrightarrow{w}[l] \leq p$, and $\sum_{l=j}^{i-1} \overrightarrow{w}[l] = \sum_{l=j}^{i-1} w[l]$.

Moreover, by Lemma 4.18, since $\left|\overrightarrow{w}[i:n]\right| = \left|\overleftarrow{w}_{d-L}\right| = d - L$ we have $\sum_{l=i}^{j+q-1} \overleftarrow{w}_{d-L}[l] \leq \sum_{l=i}^{j+q-1} \overrightarrow{w}[l]$ hence $\sum_{l=i}^{j+q-1} w[l] \leq \sum_{l=i}^{j+q-1} \overrightarrow{w}[l]$. Therefore, we can conclude that $\sum_{l=j}^{j+q-1} w[l] \leq p$. □

Lemma 4.20. *If* AtMostSeq$(p, q, [x_1, \ldots, x_n])$ *is* AC, *and* $\left|\overrightarrow{w}[1:i]\right| + \left|\overleftarrow{w}[i:n]\right| > d$ *then* $x_i = 1$ *has a support.*

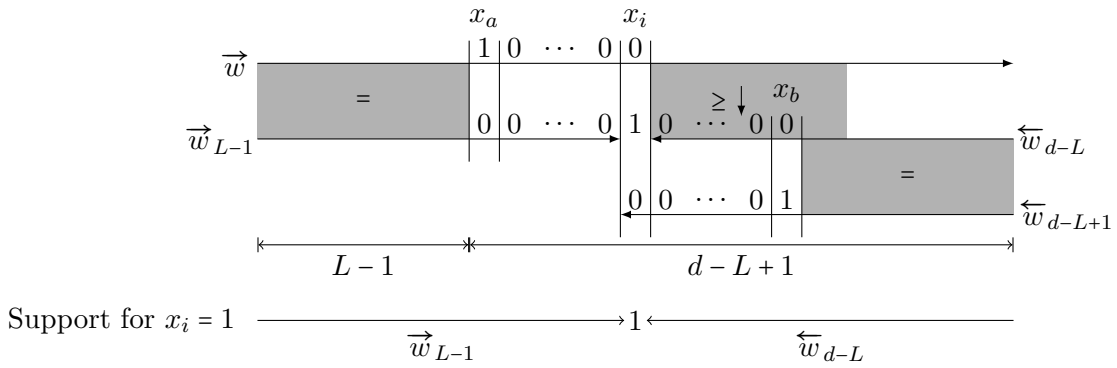


FIGURE 4.6: Illustration of Lemma 4.20's proof. Horizontal arrows represent assignments.

Proof. Let $\overrightarrow{w}$ and $\overleftarrow{w}$ be the instantiations found by `leftmost`, on respectively $x_1, \ldots, x_n$ and $x_n, \ldots, x_1$. We consider, without loss of generality, a variable x_i such that $\mathcal{D}(x_i) = \{0, 1\}$ and $\left|\overrightarrow{w}[1:i]\right| + \left|\overleftarrow{w}[i:n]\right| > d$, and show that one can build a support for $x_i = 1$. If $\overrightarrow{w}[i] = 1$ or $\overleftarrow{w}[i] = 1$ then there exists a support for $x_i = 1$, hence we only need to consider the case where $\overrightarrow{w}[i] = \overleftarrow{w}[i] = 0$.

Let $L = \left|\overrightarrow{w}[1:i]\right| = \left|\overrightarrow{w}[1:i-1]\right|$ (this equality holds since $\overrightarrow{w}[i] = 0$). Let $\overrightarrow{w}_{L-1}$ be the instantiation obtained by using `leftmost`$(L-1)$ on the subsequence $x_1, \ldots, x_{i-1}$, and let $\overleftarrow{w}_{d-L}$ be the instantiation returned by `leftmost`$(d-L)$ on the subsequence $x_n, \ldots, x_{i+1}$.

We show that w such that $w[i] = 1$, equal to $\overrightarrow{w}_{L-1}$ on $x_1, \ldots, x_{i-1}$ and to $\overleftarrow{w}_{d-L}$ on $x_{i+1}, \ldots, x_n$, is a support.

Clearly $\left|w\right| = d$, therefore we only have to make sure that all capacity constraints are satisfied. Moreover, since it is the concatenation of two consistent instantiations, it can violate the constraint only on the junction, i.e., on a subsequence $x_j, \ldots, x_{j+q-1}$ such that $j \leq i$ and $i < j + q$.

We show that the sum of any such subsequence is less than or equal to p by comparing with $\overrightarrow{w}$ and $\overleftarrow{w}_{d-L}$ (see Figure 4.6). First, note that on the subsequence $x_1, \ldots, x_{i-1}$, $\overrightarrow{w}_{L-1} = \overrightarrow{w}$, except for the largest index a such that $\overrightarrow{w}[a] = 1$ and $w[a] = 0$. Similarly on $x_n, \ldots, x_{i+1}$, we have $\overleftarrow{w}_{d-L} = \overleftarrow{w}_{d-L+1}$, except for the smallest b such that $\overleftarrow{w}_{d-L+1}[b] = 1$. There are two cases:

Suppose first that $j > a$. In that case, $\sum_{l=j}^{j+q-1} w[l] = \sum_{l=i}^{j+q-1} \overleftarrow{w}_{d-L+1}[l]$ if $j+q-1 \geq b$, and otherwise it is equal to 1. It is therefore always less than or equal to p since $i \geq j$ (and we assume $p \geq 1$).

Now suppose that $j \leq a$. In that case, consider first the subsequence $x_j, \ldots, x_i$. On this interval, the cardinality of w is the same as that of $\overrightarrow{w}$, i.e., $\sum_{l=j}^{i} w[l] = \sum_{l=j}^{i-1} \overrightarrow{w}_{L-1}[l] + 1 = \sum_{l=j}^{i} \overrightarrow{w}[l]$. On the subsequence $x_{i+1}, \ldots, x_{j+q-1}$, note that $\left|w[i+1:n]\right| = \left|\overrightarrow{w}[i+1:n]\right| = d-L$, hence by Lemma 4.18, we have $\sum_{l=i+1}^{j+q-1} w[l] = \sum_{l=i+1}^{j+q-1} \overleftarrow{w}_{d-L}[l] \leq \sum_{l=i+1}^{j+q-1} \overrightarrow{w}[l]$. Therefore $\sum_{l=j}^{j+q-1} w[l] \leq \sum_{l=j}^{j+q-1} \overrightarrow{w}[l] \leq p$. □

4.3.3 Algorithmic Complexity

Using Lemmas 4.16, 4.17, 4.19 and 4.20, one can design a filtering algorithm with the same worst case time complexity as `leftmost`. In this section, we introduce a linear time implementation of `leftmost`. We denote this algorithm `leftmost_count`, since we use it to compute an array "*count*" containing the values of $\left|\overrightarrow{w}[1:i]\right|$ for all values of i. We give the pseudo code for this procedure in Algorithm 10. The key idea that allows to reduce the complexity is that, at each step, a single new subsequence is to be considered. However, we also need to compute the new maximum across current subsequences, and increment all subsequences when assigning the value 1 to $w[i]$, both in constant time.

It is easy to see that `leftmost_count` has an $O(n)$ worst case time complexity. In order to prove its correctness, we will show that the instantiation computed by `leftmost_count` is the same as that computed by `leftmost`.

Lemma 4.21. *Algorithms 9 and 10 return the same instantiation w.*

Proof. We first prove the following three invariants, true at the beginning of each step of the main loop:

- The cardinality of the j^{th} subsequence is equal to $c[(i+j-2) \bmod q] + count[i-1]$.
- The number of subsequences of cardinality k is equal to $occ[n - count[i-1] + k]$.
- The cardinality maximum of any subsequence is equal to max_c.

Then, it is easy to check that `leftmost_count` computes the exact same instantiation as `leftmost`. Furthermore, at the end of the algorithm, we will have $count[i] = \left|\overrightarrow{w}[1:i]\right|$ for all $i \in [1, n]$.

Cardinality of the subsequences.

Let w_i denote the assignment w after $i-1$ steps of the loop. Notice that at the beginning and the end of the sequence of variables, subsequences are truncated. However, to

Algorithm 10: `leftmost_count`

Data: $p, q, [x_1, \ldots, x_n]$
Result: $count : [0, \ldots, n] \mapsto [0, \ldots, n]$
foreach $i \in [1, \ldots, n]$ **do**
  $w[i] \leftarrow min(x_i)$;
  $occ[i] = 0$;
foreach $i \in [0, \ldots, n]$ **do** $count[i] \leftarrow 0$;
$c[0] \leftarrow w[1]$;
foreach $i \in [1, \ldots, p]$ **do** $occ[n+i] = 0$;
foreach $i \in [1, \ldots, q]$ **do**
  $w[n+i] \leftarrow 0$;
  if $i < q$ **then** $c[i] \leftarrow c[i-1] + w[i+1]$;
  ;
  $occ[n + c[i-1]] \leftarrow occ[n + c[i-1]] + 1$;
$max_c \leftarrow max(\{c[i] \mid i \in [0, \ldots, q-1]\})$;
foreach $i \in [1, \ldots, n]$ **do**
1 **if** $max_c < p$ & $\left|\mathcal{D}(x_i)\right| > 1$ **then**
    $max_c \leftarrow max_c + 1$;
    $count[i] \leftarrow count[i-1] + 1$;
    $w[i] \leftarrow 1$;
  else $count[i] \leftarrow count[i-1]$;
2 $prev \leftarrow c[(i-1) \bmod q]$;
3 $next \leftarrow c[(i+q-2) \bmod q] + w[i+q] - w[i]$;
  $c[(i-1) \bmod q] \leftarrow next$;
  if $prev \neq next$ **then**
4 $occ[n + prev] \leftarrow occ[n + prev] - 1$;
5 $occ[n + next] \leftarrow occ[n + next] + 1$;
    if $next + count[i] > max_c$ **then** $max_c \leftarrow max_c + 1$;
    if $occ[n + prev] = 0$ & $prev + count[i] = max_c$ **then**
      $max_c \leftarrow max_c - 1$;
return $count$;

simplify the notations, we will consider that $w[-q], w[-q+1], \ldots, w[-1]$ exist and are equal to 0. Thus we can write that the cardinality of the j^{th} is equal to $\sum_{l=i-q+j}^{i+j-1} w_i[l]$.

We prove the first invariant by induction, i.e., let $P(i)$ denote the fact that the following equalities hold at the beginning of a step i:

$$(\sum_{l=i-q+j}^{i+j-1} w_i[l]) = (c[i+j-2 \bmod q] + count[i-1]) \quad \forall j \in [1, \ldots, q]$$

The base case $P(1)$ is easily checkable from the initialization of c.

Now suppose that $P(i)$ holds, and consider the state of c at the beginning of step $i+1$. First, note that at step i of the loop, only the value of $c[i-1 \bmod q]$ changes. Consider

$j \in [1, \ldots, q-1]$. In this case, $((i+1)+j-2 \bmod q) = (i+j-1 \bmod q) \neq (i-1 \bmod q)$. Therefore, $c[(i+1)+j-2 \bmod q]$ has not changed between step i and step $i+1$, and since $P(i)$ holds, we have:

$$(\sum_{l=i-q+(j+1)}^{i+(j+1)-1} w_i[l]) = (c[i+(j+1)-2 \bmod q] + count[i-1])$$

which can be rewritten as follows:

$$(\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_i[l]) = (c[(i+1)+j-2 \bmod q] + count[i-1])$$

Now there are two possibilities. Either *count* is incremented, i.e., $count[i] = count[i-1]+1$, and in that case $w_{i+1}[i] = w_i[i]+1$. Or *count* is not incremented, and in that case $w_{i+1}[i] = w_i[i]$.

In both cases we have:

$$\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_{i+1}[l] = \sum_{l=(i+1)-q+j;l\neq i}^{(i+1)+j-1} w_i[l] + w_{i+1}[i]$$

since $w_{i+1}[l] = w_i[l]$ for all $l \neq i$. Hence we obtain:

$$(\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_{i+1}[l]) = (c[i+(j+1)-2 \bmod q] + count[i-1]) - w_i[i] + w_{i+1}[i]$$

which can be rewritten as:

$$(\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_{i+1}[l]) = (c[(i+1)+j-2 \bmod q] + count[i])$$

Thus $P(i+1)$ holds.

Now we look at the last case: $j = q$. Here, at step i the value of $c[i-1 \bmod q]$ is set to $c[i+q-2 \bmod q] + w_{i+1}[i+q] - w_{i+1}[i]$. Since $P(i)$ holds, we can replace $c[i+q-2 \bmod q]$ by $\sum_{l=i}^{i+q-1} w_i[l]) - count[i-1]$, so at the beginning of step $i+1$ we have:

$$c[(i+1)+q-2 \bmod q] = (\sum_{l=i}^{i+q-1} w_i[l]) - count[i-1] + w_{i+1}[i+q] - w_{i+1}[i]$$

however, since $\sum_{l=i}^{i+q-1} w_i[l]) = w_i[i] + \sum_{l=i+1}^{i+q-1} w_{i+1}[l])$ we have:

$$c[(i+1)+q-2 \bmod q] = \sum_{l=i+1}^{i+q} w_{i+1}[l] - count[i-1] + w_i[i] - w_{i+1}[i]$$

Therefore, since $count[i] = count[i-1] + w_{i+1}[i] - w_i[i]$, the following holds:

$$c[(i+1)+q-2 \bmod q] = \sum_{l=i+1}^{i+q} w_{i+1}[l] - count[i]$$

We have shown that $P(i)$ implies $P(i+1)$, and we can therefore conclude that at the beginning of each step i of the loop $P(i)$ (that is, the first invariant) holds.

Occurrences of each cardinality.

We proceed as for the first invariant, and prove it by induction. The base case is easy to check since $count[0] = 0$, and since the array c is properly initialized.

Now we assume that there are exactly $occ[n - count[i-1] + k]$ subsequences involving x_i whose cardinality is equal to k in w_i, and we show that at the beginning of step $i+1$ there are $occ[n - count[i] + k]$ subsequences involving x_{i+1} of cardinality k in w_{i+1}.

There are two reasons for cardinalities to change.

First, when moving up to the next step in the loop, we move from subsequences involving x_i to subsequences involving x_{i+1}. There are $q-1$ subsequences involving both x_i and x_{i+1}. So we simply need to make sure that the occurrences are updated to reflect the fact that the subsequence $x_{i-q+1}, \ldots, x_i$ should not be counted anymore, whilst the subsequence $x_{i+1}, \ldots, x_{i+q}$ should now be. Let k_1 (respectively k_2) be the cardinality of the former (respectively latter) subsequence. As established by the first invariant, $k_1 = c[(i-1) \bmod q] + count[i-1]$, that is the value *prev* in Line 2 is set to $k_1 - count[i-1]$. Moreover, *next* is given the value $c[(i+q-2) \bmod q] + w[i+q] - w[i]$. However, from invariant 1, we have $c[(i+q-2) \bmod q] + count[i-1] = \sum_{l=i}^{i+q-1} w[l]$. It follows that

$$next = \sum_{l=i}^{i+q-1} w[l] + w[i+q] - w[i] - count[i-1] = \sum_{l=i+1}^{i+q} w[l] - count[i-1]$$

therefore $next = k_2 - count[i-1]$. To maintain invariant (2), we therefore need to increment the value of $occ[n - count[i-1] + k_2]$ and decrement the value of $occ[n - count[i-1] + k_1]$. This is precisely what is done in Lines 4 and 5.

Second, when the conditions in Line 1 are met, the value of $w[i]$ is set to 1. Since its value was previously 0, the cardinality of every subsequence involving $w[i]$ should be incremented before starting the next step $(i+1)$. This happens automatically because in this case the value of $count[i]$ will be set to $count[i-1] + 1$. Indeed, for any integer k, the number of occurrences of subsequences of cardinality $k-1$ at the beginning of step i is $occ[n - count[i-1] + k - 1]$. Therefore, since $count[i] = count[i-1] + 1$, at the beginning of step $i+1$, we have $occ[n - (count[i] - 1) + k - 1]$, that is, $occ[n - count[i] + k]$.

Cardinality maximum.

Here we show that the maximum value of the cardinalities of the current subsequences is properly maintained. When the number of occurrences of a cardinality k becomes non-null and if $k > max_c$, then max_c is set to k. Similarly, When the number of occurrences of a cardinality k becomes null and if $k = max_c$, then max_c is decreased. Last, when the cardinality of all subsequences is incremented, max_c is incremented too.

These operations are correct because from one step i to $i+1$, the value of max_c cannot change by more than 1. Indeed, only the first subsequence is removed, the other $q-1$ subsequences remain unchanged. Moreover, the first subsequence is replaced by the last subsequence to which a value $a \in [0,1]$ is added, and another value $b \in [0,1]$ is subtracted. Therefore its value cannot change by more than 1, hence max_c.

Now having these three invariants, one can check that at each step i the values of $w[i]$ will be the same as in Algorithm 9.

□

4.3.4 Achieving Arc-Consistency on AtMostSeqCard

Now, we can prove our main result, that AC on a constraint AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n])$ can be achieved in $O(n)$ time by Algorithm 11.

First, in Line 1, we achieve AC on AtMostSeq$(p, q, [x_1, \ldots, x_n])$, so that the first condition for Lemma 4.15 holds. Achieving AC on AtMostSeq can be done in linear time using a procedure essentially similar to `leftmost_count`. Indeed, since the constraint AtMost is monotone, we simply need to achieve AC on every AtMost. Moreover, a constraint AtMost$(p, [x_{i_1}, \ldots, x_{i_q}])$ may prune the domain of a variable only if p other variables in $[x_{i_1}, \ldots, x_{i_q}])$ are assigned to 1. To do that, we run a truncated version of `leftmost_count`: the values of $w[i]$ are never updated, i.e., they are set to the minimum value in the domain and we never enter the if-then-else block starting at condition 1 in Algorithm 10. Now, if at step i we have $max_c = p$, then there are p variables assigned to 1 in at least one subsequence involving x_i, hence it should be set to 0 if possible.

Second, in Line 2, we achieve AC on the cardinality constraint, in order to satisfy the second condition of Lemma 4.15.

Third, in Line 4 we compute the vector L that maps each index i to the value of $\left|\overrightarrow{w}[1:i]\right| - \sum_{j=1}^{j=i} min(x_j)$. This is given by the array *count* returned by `leftmost_count` on the sequence $[x_1, \ldots, x_i]$. Notice that, we work with the residual demand, computed in Line 3, rather than the original demand. At this point, the third condition of Lemma 4.15 can be checked, and we know whether the constraint is AC, inconsistent, or if some pruning may be possible.

In the latter case, we compute the vector R, that maps each index i to the value of $\left|\overleftarrow{w}[i:n]\right| - \sum_{j=i}^{j=n} min(x_j)$, in Line 5.

Finally, we can activate the pruning rules that are shown to be correct and sufficient by Lemmas 4.16 and 4.19 for Line 6, and Lemmas 4.17 and 4.20 for Line 7.

Algorithm 11: AC(AtMostSeqCard($p, q, d, [x_1, .., x_n]$))

```
1 if AC(AtMostSeq(p, q, [x_1, ..., x_n])) =⊥ then
    return ⊥ ;
2 if AC(∑_{i=1}^{n} x_i = d) =⊥ then
    return ⊥;
3 d_res ← d − ∑_{i=1}^{n} min(x_i);
4 L ← leftmost_count([x_1, ..., x_n], p, q);
  if L[n] = d_res then
5   R ← leftmost_count([x_n, ..., x_1], p, q);
    foreach i ∈ [1, ..., n] such that D(x_i) = {0, 1} do
6     if L[i] + R[n − i + 1] ≤ d_res then D(x_i) ← {0};
7     if L[i − 1] + R[n − i] < d_res then D(x_i) ← {1};
  else if L[n] < d_res then
    return ⊥ ;
  return D ;
```

Theorem 4.22. *Algorithm 11 achieves* AC *on* AtMostSeqCard *with an optimal worst case time complexity.*

Proof. The soundness of Algorithm 11 is a straight application of Lemmas 4.16 and 4.17. Its completeness is a consequence of Lemmas 4.15, 4.19 and 4.20.

Achieving AC on AtMostSeq (Line 1) can be done with one call to `leftmost_count`. Achieving AC on a simple cardinality constraint (Line 2) can be done trivially in $O(n)$ time. Finally, pruning the domains requires at most two calls to `leftmost_count`, plus going through the sequence of variable to actually change the domains, that is, $O(n)$ time.

The worst case time complexity of Algorithm 11 is then $O(n)$, hence optimal. □ □

Example 4.4. *We give an example of the execution of Algorithm 11 in Figure 4.7 for computing the* AC *of constraint* AtMostSeqCard *with* $p = 4$, $q = 8$ *and* $d = 12$.

The first line stands for current domains, dots are unassigned variables (hence $d_{res} = 10$*). The two next lines give the instantiations* $\overrightarrow{w}$ *and* $\overleftarrow{w}$ *obtained by running* `leftmost_count` *from left to right and from right to left, respectively. The third and fourth lines stand for the values of* $L[i] = \left|\overrightarrow{w}[1:i]\right| - \sum_{j=1}^{j=i} min(x_j)$ *and* $R[n-i+1] = \left|\overleftarrow{w}[i:n]\right| - \sum_{j=i}^{j=n} min(x_j)$*. The fifth and sixth lines correspond to the application of, respectively, Lemma 4.16*

$\mathcal{D}(x_i)$		.	0	.	.	.	.	.	.	0	1	0	.	.	.	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		**1**	0	**1**	**1**	**1**	0	0	0	0	1	0	**1**	**1**	**1**	0	0	0	**1**	0	**1**	**1**	1	
$\overleftarrow{w}[i]$		**1**	0	0	**1**	**1**	**1**	0	0	0	1	0	**1**	**1**	**1**	0	0	0	0	**1**	**1**	**1**	1	
$L[i]$	0	1	1	2	3	4	4	4	4	4	4	4	5	6	7	7	7	7	8	8	9	10	10	
$R[n-i+1]$		10	9	9	9	8	7	6	6	6	6	6	6	5	4	3	3	3	3	3	2	1	0	0
$L[i]+R[n-i+1]$		11	10	11	12	12	11	**10**	**10**	10	10	10	11	11	11	**10**	**10**	**10**	11	11	11	11	10	
$L[i-1]+R[n-i]$		**9**	10	10	10	10	10	10	10	10	10	10	**9**	**9**	**9**	10	10	10	10	10	**9**	**9**	10	
AC$(\mathcal{D}(x_i))$		**1**	0	.	.	.	.	**0**	**0**	0	1	0	**1**	**1**	**1**	**0**	**0**	**0**	.	.	**1**		1	

FIGURE 4.7: AC on ATMOSTSEQCARD($p=4,\ q=8,\ d=12, [x_1, \ldots, x_n]$)

and 4.17. Last, the seventh line gives the Arc Consistent domains. Bold values indicate pruning.

4.4 Extensions

In this section, we show that the filtering algorithm described in the previous section can be extended in a number of ways to enforce AC on more general constraints.

Some generalizations are straightforward. For instance, the parameter p does not need to be the same for all subsequences. Indeed neither Algorithm 9 nor Algorithm 10 relies on the fact that p is constant across all subsequences. We can easily give a list of upper bounds, one for each subsequence. Another relatively straightforward generalization is to have a variable, rather than a single value, for the demand d.

4.4.1 The ATMOSTSEQΔCARD Constraint

Let δ be a variable, we define the ATMOSTSEQΔCARD as follows:

Definition 4.23.

$$\text{ATMOSTSEQΔCARD}(p, q, \delta, [x_1, \ldots, x_n]) \Leftrightarrow \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq p) \wedge (\sum_{i=1}^{n} x_i = \delta)$$

We show how one can achieve AC on the above generalization. The changes to Algorithm 11 required to handle this generalization are minimal. Indeed, tight lower and upper bounds on δ are easy to compute.

They are given, respectively by $\sum_{i=1}^{n} min(x_i)$, and $\left|\overrightarrow{w}\right|$. Moreover, by Lemma 4.15, we know there can be inconsistent values for a variable x_i only if $\left|\overrightarrow{w}\right| \leq d$. It follows that we only need to care about the lower bound of δ. We show these changes in Algorithm 12. The domain of δ is updated in Line 2 for the lower bound, and Line 5 for the upper

bound. Also, the lower bound of δ ($min(\delta)$) is used to compute the residual demand to reach in Line 3 instead of d.

Algorithm 12: AC(AtMostSeqΔCard$(p, q, \delta, [x_1, .., x_n])$)

```
1 if AC(AtMostSeq(p, q, [x_1, ..., x_n])) =⊥ then
    return ⊥ ;
2 if AC(Σ_{i=1}^{n} x_i = δ) =⊥ then
    return ⊥ ;
3 d_res ← min(δ) − Σ_{i=1}^{n} min(x_i);
4 L ← leftmost_count([x_1, ..., x_n], p, q);
5 D(δ) ← D(δ) ∩ [0, L[n] + Σ_{i=1}^{n} min(x_i)];
6 if L[n] = d_res then
    R ← leftmost_count([x_n, ..., x_1], p, q);
    foreach i ∈ [1, ..., n] such that D(x_i) = {0,1} do
      if L[i] + R[n − i + 1] ≤ d_res then D(x_i) ← {0};
      if L[i − 1] + R[n − i] < d_res then D(x_i) ← {1};
7 else if L[n] < d_res then
    return ⊥ ;
return D ;
```

Theorem 4.24. *Algorithm 12 achieves* AC *on* AtMostSeqΔCard *with an optimal worst case time complexity.*

Proof. First, we need to filter inconsistent values from the domain of δ. By Lemma 4.13, the cardinality $\left|\overrightarrow{w}\right|$ of the instantiation returned by `leftmost` is a valid upper bound for δ. Moreover, because of the cardinality constraint, $\sum_{i=1}^{n} min(x_i)$ is a valid lower bound. It is easy to see that any value d within these bounds satisfies the conditions of Lemma 4.14. In other words, we can assign δ to any value in the interval $[\sum_{i=1}^{n} min(x_i), \left|\overrightarrow{w}\right|]$ and extend it to an AC support of AtMostSeqΔCard$(p, q, \delta, [x_1, \ldots, x_n])$. These bounds are therefore tight.

Second, we need to prune values in $\mathcal{D}(x_i)$ for all i in $1, \ldots, n$ that are not supported by any value in $\mathcal{D}(\delta)$. A naive algorithm for checking that would be to run `leftmost` for each value in $\mathcal{D}(\delta)$ and compute the union of possible values for the variables x_i. However, one can avoid this by distinguishing two cases after line 5. Suppose that $\left|\mathcal{D}(\delta)\right| > 1$, in this case, Line 1 and Line 2 and 5 imply that Lemma 4.15 holds for $d = min(\delta)$. Hence all values for the variables x_i are consistent and in this case we will never enter lines 6 and 7. Suppose now that $\left|\mathcal{D}(\delta)\right| = 1$, in this case, we can simply apply the same filtering (Line 6) that we proposed previously for a fixed cardinality.

The whole procedure requires at most two calls to `leftmost_count`, which takes $O(n)$ time. □

TABLE 4.1: Maximal cardinality instantiations.

x_i:	.	.	.	.	.	.	.	0	.	.	.	.	.	0	.	.	.	.	.	0	.	.	
$\overrightarrow{w}$ on 4.1:	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	$\lvert\overrightarrow{w}\rvert = 11$
$\overrightarrow{w}$ on 4.2:	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	$\lvert\overrightarrow{w}\rvert = 10$
$\overrightarrow{w}$ on 4.1 & 4.2:	1	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	$\lvert\overrightarrow{w}\rvert = 8$

4.4.2 The MULTIATMOSTSEQCARD Constraint

We show here that we can easily modify Algorithm 11 (or Algorithm 12) to achieve AC on the conjunction of several ATMOSTSEQCARD constraints.

For instance, in crew-rostering problems, the work pattern of an employee might require a conjunction of ATMOSTSEQCARD: one to limit the number of shifts per day, and another to limit the number of shifts per week. In the crew-rostering benchmarks that we consider in Section 4.5, we have a variable x_i for each working shift i. Moreover, we want each employee to work at most one shift per day, at most five shifts per week, and between 17 and 18 shifts on the whole period. We model this with two ATMOSTSEQΔCARD constraints: ATMOSTSEQΔCARD$(1, 3, \delta, [x_1, \ldots, x_n])$ and ATMOSTSEQΔCARD$(5, 21, \delta, [x_1, \ldots, x_n])$ s.t. $\mathcal{D}(\delta) = \{17, 18\}$. However, AC on these two constraints is not equivalent to AC on their conjunction. We illustrate this in Example 4.5 (using smaller instances of the constraints).

Example 4.5. *Consider the conjunction of the two following* ATMOSTSEQCARD *constraints:*

$$\text{ATMOSTSEQCARD}(1, 2, 9, [x_1, \ldots, x_{22}]) \quad \& \tag{4.1}$$

$$\text{ATMOSTSEQCARD}(2, 5, 9, [x_1, \ldots, x_{22}]) \tag{4.2}$$

Now, suppose that $\mathcal{D}(x_8) = \mathcal{D}(x_{14}) = \mathcal{D}(x_{20}) = \{0\}$*, whilst all other domains are equal to* $\{0, 1\}$*. The first line of Table 4.1 shows the domains of* $[x_1, \ldots, x_{22}]$*, with a dot (.) standing for a full domain (*$\{0, 1\}$*) and the value* 0 *standing for the domain reduced to the singleton* $\{0\}$*. The second and third lines show the results of* `leftmost` *on* $[x_1, \ldots, x_{22}]$ *for* $p/q = 1/2$ *and* $p/q = 2/5$*, respectively. Since the demand* d *is equal to* 9*, both constraints 4.1 and 4.2 are* AC*. Last, the third line shows an instantiation of maximum cardinality respecting simultaneously* ATMOSTSEQ$(1, 2, [x_1, \ldots, x_{22}])$ *and* ATMOSTSEQ$(2, 5, [x_1, \ldots, x_{22}])$*. It is obtained using the same principle as* `leftmost`*, however by checking two sets of subsequences, one for each* ATMOSTSEQCARD *constraint. It is easy to see that the arguments of Lemma 4.13 are still valid when considering any number of subsequences. Therefore, the total cardinality of* 8 *is a valid upper bound, and since* d *is equal to* 9*, the conjunction of the two constraints has no solution.*

We define the constraint MultiAtMostSeqCard, and show that the algorithm introduced in this chapter can be adapted to enforce ac on this constraint in $O(nm)$ time, where m is the number of chains of AtMost constraints.

Definition 4.25. $\text{MultiAtMostSeqCard}(p_1,..,p_m,q_1,..,q_m,d,[x_1,\ldots,x_n]) \Leftrightarrow$

$$\bigwedge_{k=1}^{m}\bigwedge_{i=0}^{n-q_k}(\sum_{l=1}^{q_k}x_{i+l}\leq p_k)\wedge(\sum_{i=1}^{n}x_i=d)$$

Theorem 4.26. *One can achieve* ac *on* MultiAtMostSeqCard *in* $O(nm)$ *time.*

Proof. The main argument to show that this theorem holds is that all previous proofs and algorithms can be easily adapted to this case. We therefore only sketch its proof.

First, note that one can modify the procedure `leftmost` (or `leftmost_count`) to handle a conjunction of AtMostSeq constraints instead of a single one. All we need to do is to duplicate m times the structures maintaining the cardinalities of the subsequences. We obtain a procedure that checks m chains in $O(nm)$ if we use Algorithm 10.

Second we show that Lemma 4.13 still holds with this new procedure, and with respect to several chains of AtMost constraints. In other words, greedily assigning the value "1" while respecting m chains of AtMost will produce a sequence of maximal cardinality. The argument used in the proof of Lemma 4.13 generalizes without modification to several chains. We show that if we make the hypothesis that an instantiation w of cardinality higher than of $\left|\overrightarrow{w}\right|$ found by the greedy procedure leads to a contradiction. For each value of q, the same three cases arise, and can be analyzed in exactly the same way. Hence we can show that w can be made equal to $\overrightarrow{w}$ without changing its cardinality, hence a contradiction.

In all subsequent proofs, we check subsequences of length q and show that they do not violate capacity constraints. Obviously, these proofs hold for any value of q (within $[1,n]$). In fact, the only difference is that when considering multiple chains, we might have to check subsequences of different lengths. □ □

4.5 Experimental Results

We tested our filtering algorithms on two benchmarks: car-sequencing and crew-rostering. Since Slack-Pruning is a particular case of AtMostSeqCard and in all cases cannot filter more than AtMostSeqCard then it will be omitted in these experiments. All models are implemented using Ilog-Solver 6.7. All experiments ran on Intel Xeon CPUs 2.67GHz under Linux. Since we compare propagators, we averaged the results across several branching heuristics to reduce the bias that these can have on the outcome.

Moreover, these heuristics were randomized and for each instance and each heuristic we launched 5 randomized runs with a 20 minutes time cutoff[2]. For each considered data set, we primarily compare the total number of successful runs, denoted "*#solved*". Then, we consider the CPU time in seconds and number of backtracks, denoted *#backtracks*, both restricted to successful runs. When appropriate, we emphasize the statistics of the best method using bold face fonts.

4.5.1 Car-Sequencing

We use the same configuration used in the previous chapter (Section 3.3). That is, 4 models (DECOMPOSE, GSC, AMSC, and GSC⊕AMSC) and 42 heuristics. For each model, we report the average number of solved instances in Table 4.2, the average CPU time on solved instances in Table 4.3 and the average number of backtracks in Table 4.4. In each table, we also report the minimum and maximum value (for any heuristic, though averaged over randomized runs) as well as the standard deviation over the different heuristics.

TABLE 4.2: Evaluation of the filtering methods (solved instances count)

propagation	#solved in *set1* (70×5)				#solved in *set2* (4×5)			
	avg	min	max	dev	avg	min	max	dev
DECOMPOSE	268.33	70.00	**350.00**	88.95	2.95	0.00	15.00	3.66
GSC	333.52	154.00	**350.00**	42.16	10.11	0.00	**20.00**	5.25
AMSC	321.35	80.00	**350.00**	64.05	**11.19**	0.00	**20.00**	5.22
GSC⊕AMSC	**334.11**	154.00	**350.00**	41.88	10.45	0.00	**20.00**	5.06
propagation	#solved in *set3* (5×5)				#solved in *set4* (7×5)			
	avg	min	max	dev	avg	min	max	dev
DECOMPOSE	0.00	0.00	0.00	0.00	2.35	0.00	9.00	2.65
GSC	0.73	0.00	**10.00**	2.35	4.64	0.00	**10.00**	3.69
AMSC	0.38	0.00	5.00	1.21	**5.09**	0.00	**10.00**	3.75
GSC⊕AMSC	**0.76**	0.00	**10.00**	2.41	4.80	0.00	**10.00**	3.65

Table 4.2 shows that in all cases, the best method is either GSC⊕AMSC or AMSC. In some cases a stronger filtering seems to be key and GSC⊕AMSC solves more instances than other methods: 95.46% of *set1* and 3.04% of *set3*. In other cases, exploration speed is more important and AMSC is better: 55.95% and 14.55% of solved instances for *set2* and *set4*, respectively. Overall, as witnessed by Table 4.4, GSC and GSC⊕AMSC usually require exploring a much smaller tree than AMSC. However, the propagator for GSC slows down the search by a substantial amount. Considering Table 4.3 as well as data from unsolved instances, we observed a factor 12.5 on the number of nodes explored per second between these two models. Moreover, the level of filtering obtained by these two

[2]The approximate total CPU time is one year.

TABLE 4.3: Evaluation of the filtering methods (CPU time on solved instances)

propagation	CPU time (in sec.) on *set1* (70×5)				CPU time (in sec.) on *set2* (4×5)			
	avg	min	max	dev	avg	min	max	dev
DECOMPOSE	10.49	0.02	1145.20	80.39	58.74	0.01	766.25	178.88
GSC	3.16	0.52	1100.54	33.17	109.45	0.11	1096.37	237.46
AMSC	3.79	0.03	1197.88	51.49	70.56	0.01	1014.57	186.87
GSC⊕AMSC	3.03	0.53	1017.74	33.60	99.71	0.11	1155.40	222.85
propagation	CPU time (in sec.) on *set3* (5×5)				CPU time (in sec.) on *set4* (7×5)			
	avg	min	max	dev	avg	min	max	dev
DECOMPOSE	-	-	-	-	30.85	0.03	985.75	136.43
GSC	276.06	29.22	988.79	308.64	53.61	1.63	975.03	147.35
AMSC	8.62	1.06	18.07	6.72	38.45	0.03	1171.78	124.29
GSC⊕AMSC	285.43	6.01	1131.19	337.24	61.61	1.62	1180.53	175.23

TABLE 4.4: Evaluation of the filtering methods (search tree size on solved instances)

propagation	#backtracks on *set1* (70×5)				#backtracks on *set2* (4×5)			
	avg	min	max	dev	avg	min	max	dev
DECOMPOSE	174017	148	25062202	1341281	1101723	78	15324348	3439897
GSC	1408	99	2320312	34519	131062	58	1595137	306448
AMSC	33600	92	13888040	468527	665205	61	10254401	1827516
GSC⊕AMSC	1007	92	1180605	23649	104823	56	1055307	244135
propagation	#backtracks on *set3* (5×5)				#backtracks on *set4* (7×5)			
	avg	min	max	dev	avg	min	max	dev
DECOMPOSE	-	-	-	-	378475	170	13767766	1754180
GSC	55365	5852	218590	63211	23897	151	467396	75097
AMSC	40326	5991	83454	29690	215349	146	5624744	653498
GSC⊕AMSC	57725	1120	244787	69705	22974	146	428523	71552

methods are incomparable. Therefore combining them is always better than using GSC alone.

In [139] the authors applied their method to *set1*, *set2* and *set3* only. For their experiments, they considered the best result provided by 2 heuristics. When using COST-REGULAR or GEN-SEQUENCE filtering alone, 50.7% of problems are solved and when combining either COST-REGULAR or GEN-SEQUENCE with GSC, 65.2% of problems are solved (with a time out of 1 hour). In our experiments, in average over the 42 heuristics and the 5 randomized runs, ATMOSTSEQCARD and GSC solve respectively 84.29% and 87.19% of instances and combining ATMOSTSEQCARD with GSC solves 87.42% instances in a time out of 20 minutes. Moreover, using the model GSC⊕AMSC, the best heuristic was able to solve 96.20% of these instances.

4.5.2 Crew-Rostering

Problem Description In this problem, working shifts have to be attributed to employees over a period, so that the required service is met at any time and working regulations are respected. The latter condition can entail a wide variety of constraints. Previous work [93, 103] used allowed (or forbidden) patterns to express successive shift constraints. For example, with 3 shifts of 8 hours per day: D (day), E (evening) and N (night), ND can be forbidden since employees need some rest after night shifts. We consider here a simple case involving 20 employees with 3 shifts of 8 hours per days where no employee can work more than one 8h shift per day, no more than 5 days per period of 7 days, and the break between two worked shifts must be at least 16h. The planning horizon is of 28 days, and each employee must work 17 shifts over the 4 weeks period (i.e., 34 hours per week in average).

Models and Heuristics We use a model with one Boolean variable e_{ij} for each of the m employees and each of the n shifts stating if employee i works on shift j. The demand d_j^s on each shift j is enforced through a sum constraint $\sum_{i=1}^{m} e_{ij} = d_j^s$. The other constraints are stated using two AtMostSeqCard constraints per employee, one with ratio $p/q = 1/3$, another with ratio $5/21$, and both with the same demand $d = 17$. We compare four models. In the first (decompose), we use a decomposition in a chain of AtMost constraints. In the second (amsc) we use two AtMostSeqCard constraints per employee j, of the form:

$$\text{AtMostSeqCard}(p, q, d, [e_{i1}, \ldots, e_{in}])$$

In the first constraint we have $p = 1, q = 3, d = 17$ and in the second constraint we have $p = 5, q = 21, d = 17$. Both are propagated using Algorithm 11. In the third model (gsc), we use the following Gsc constraint to encode the constraint $\text{AtMostSeqCard}(p, q, d, [e_{i1}, \ldots, e_{in}])$:

$$\text{Gsc}(0, p, q, \{0 : n - d, 1 : d\}, \{0 : n - d, 1 : d\}, [e_{i1}, \ldots, e_{in}], \{1\})$$

Note that in this case, since the domains are Boolean, the Gsc is in this case equivalent to AtMostSeqCard. Therefore, it cannot prune more since the latter enforces ac. However, it is stronger than the decomposition. Last, in the fourth model (mamsc) the conjunction of the two AtMostSeqCard constraints is propagated using Algorithm 12.

We used the following four variable ordering heuristics.

1. *Lexicographic*: Explores shifts chronologically and picks an employee at random;
2. *Middle*: Similar as above, however we start exploring shifts from the middle;

3. *Employee*: Picks an employee with min slack, then a possible shift of max demand;
4. *Shift*: Similar as above, however, the shift is selected before the employee.

In all cases, we branch by assigning the value 1 to the chosen pair (employee, shift).

Benchmarks We generated 341 instances, with worker availability ranging from 82% to 48% by increment of 0.1. This value denotes the probability that a given employee is willing to work during a given shift. It allows to vary the constrainedness of the problem. 228 of these instances were found feasible, 77 infeasible and 36 remain open. We report results for the satisfiable and unsatisfiable sets with 5 random runs per instance.

TABLE 4.5: Evaluation of the filtering methods: static branching (highest success counts are in bold fonts)

Lexicographic

Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
DECOMPOSE	0	-	-	-	-	170	0.05	0.02	86	452
GSC	25	308.93	344.29	74074	84301	175	2.56	9.71	262	1794
AMSC	125	164.36	239.56	1828347	2759080	213	1.76	21.95	22621	292152
MAMSC	**534**	87.29	188.81	685720	1491867	**271**	2.80	45.02	27150	444913

From the middle to the sides

Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
DECOMPOSE	1	166.76	0.00	5716015	0	160	0.04	0.00	0	0
GSC	7	253.20	301.63	53763	63110	165	1.07	0.08	0	0
AMSC	57	161.38	267.23	2207676	3621762	201	0.20	1.46	1622	15809
MAMSC	**336**	134.95	239.11	1410458	2525422	**265**	0.05	0.00	0	0

We report the results for the static heuristics in Table 4.5 and for the dynamic heuristics in Table 4.6. The first column indicates the total number of successful runs (#sol), then we report CPU time and number of backtracks, averaged over all instances and runs, as well as the standard deviation on this sample. Clearly, achieving AC on the (MULTI)ATMOSTSEQCARD constraint have a significant impact on the efficiency of the model. The decomposition into sum constraints cannot solve any satisfiable instance with lexicographic branching, and only one when starting from the middle of the sequence. The model using GSC offers a much more potent filtering, however, it is not as strong as AC on the ATMOSTSEQCARD constraint and moreover, it is much slower. On the other hand, the model using Algorithm 11 for the ATMOSTSEQCARD constraint

achieves AC whilst being as fast as the decomposed model in terms of exploration. Moreover, combining the two AtMostSeqCard constraints and using Algorithm 12 allows to solve about four times more satisfiable instances with *Lexicographic* branching and six times more with *Middle* branching.

The cost-Regular constraint could be used to enforce the same level of consistency as the combination of two AtMostSeqCard constraints. The possible patterns can be encoded through a finite automaton whilst the overall cardinality is encoded by the counter. Notice that using a Regular constraint (i.e., without cost) and modeling the overall work load with a cardinality constraint would not enforce a higher level of consistency than the decomposition into cardinality constraints (i.e., model DECOMPOSE) since AtMost constraints are monotone. A worst case analysis would indicate that the number of states in the automaton is too large.

Table 4.6: Evaluation of the filtering methods (dynamic branching)

Most constrained employee										
Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
DECOMPOSE	772	21.93	104.91	205087	1000794	165	0.06	0.00	0	2
GSC	746	65.75	180.29	14133	42235	175	0.98	0.09	0	3
AMSC	818	20.51	103.76	147479	761261	215	0.13	0.55	330	2582
MAMSC	**842**	20.78	111.00	125886	676061	**270**	0.05	0.01	0	2
Most constrained shift										
Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
DECOMPOSE	987	20.76	102.53	169964	853020	352	19.74	99.61	180161	967933
GSC	1006	33.30	107.08	8875	31586	335	15.97	95.36	5145	35824
AMSC	1061	10.07	65.02	90247	593928	362	12.19	77.37	108797	736775
MAMSC	**1074**	10.94	77.37	91222	667176	**377**	14.63	107.58	110244	834887

When using dynamic heuristics (see Table 4.6), the difference between the different models becomes much less spectacular. However, the trend is the same, with the model combining the pairs of AtMostSeqCard constraint dominating the other models.

Summary

We first proposed a simple filtering rule that reasons about capacity and demand constraints simultaneously for solving the car-sequencing problem. This pruning is then generalized to an optimal Arc Consistency algorithm for the AtMostSeqCard constraint.

Moreover, we showed how to adapt the filtering with more general constraints while keeping a reasonable worst case time complexity. Our computational results demonstrate the efficiency of our approach for solving car-sequencing and crew-rostering benchmarks.

Chapter 5

Learning

Introduction

In the past decade, hybrid CP/SAT solvers have been redesigned to benefit from CP and SAT features as much as possible. In this chapter, we show that enabling clause learning via hybrid models can greatly improve the performances of CP models in many sequencing and scheduling problems.

Lazy Clause Generation is a general framework for hybrid solvers in which propagators should be able to explain their pruning in a clausal form. A trend has subsequently emerged aiming at proposing effective and efficient explanations for (global) constraints (see for instance [47, 46, 116, 58, 55]). In this context, we investigate the learning aspect for solving car-sequencing benchmarks using our filtering for AtMostSeqCard in Section 5.1. We propose a procedure explaining AtMostSeqCard that runs in linear time complexity in the worst case. Any hybrid model using these explanations benefits from the complete filtering for this constraint along with clause learning and potentially many other CP/SAT features. We show experimentally how clause learning improves the global performances in most cases. We confirm a strong correlation between advanced propagation and finding solutions quickly for this problem. Moreover, for building proofs, clause learning appears in these experiments to be the most important ingredient while propagation is less useful.

The rest of the contributions presented in this chapter are related to the question of designing 'lazy' data structures in order to efficiently tackle large scaled instances. Backward explanations and lazy generation (see Section 2.3.2) are typically the type of 'lazy' data structures that we address. However, these techniques are relatively new in hybrid solvers and might be improved in a number of ways.

We revisit in Section 5.2 the lazy generation of Boolean variables for encoding the domains. The issue that we address is related to the redundancy of clauses used when lazily encoding a domain [53] (detailed in Section 2.3.2.2). The DomainFaithfulness constraint that we propose avoids such redundancy while ensuring the same level of consistency without any computational overhead.

Section 5.3 addresses the impact of clause learning for solving disjunctive scheduling problems. We consider a large number of disjunctive scheduling instances, on which we test the lazy generation method proposed in Section 5.2. Furthermore, we propose a novel conflict analysis scheme, called Disjunctive-based learning, tailored to this family of problems. Disjunctive-based learning uses a property of these problems allowing to learn clauses using a number of Boolean variables that is not function of the domain size. Our propositions give good experimental results and outperform the CP model in most cases. Furthermore, we confirm a correlation between the instance size, the branching choice, and the conflict analysis scheme. State-of-the-art lower bounds for a traditional benchmark are improved thanks to the new conflict analysis scheme.

5.1 Learning in Car-Sequencing

We investigate in this section the impact of clause learning for solving the car-sequencing problem. We first show how to explain our complete filtering for AtMostSeqCard. These explanations are later used in several hybrid models for solving the car-sequencing problem.

5.1.1 Explaining AtMostSeqCard

We first recall the definition of AtMostSeqCard. Given a sequence of Boolean variables $[x_1, \ldots, x_n]$ and three integers p, q, d, AtMostSeqCard is defined by a conjunction between a chain of AtMost constraints (called AtMostSeq) and Cardinality.

$$\text{AtMostSeqCard}(p, q, d, [x_1, \ldots, x_n]) \Leftrightarrow \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq p) \wedge (\sum_{i=1}^{n} x_i = d)$$

To explain AtMostSeqCard, we briefly recall the complete filtering that we proposed in Section 4.3. Let $[x_1, .., x_n]$ be a sequence of Boolean variables subject to $\text{AtMostSeqCard}(p, q, d, [x_1, .., x_n])$). The first step is to make sure that $\text{AtMostSeq}(p, q, [x_1, \ldots, x_n])$ and $\text{Cardinality}([x_1, \ldots, x_n], d)$ are ac. The remaining of the filtering is based on a greedy rule called `leftmost`. The outcome of `leftmost` is an instantiation w with a maximum cardinality on $[x_1, .., x_n]$ respecting all AtMost constraints. We use a linear time implementation of `leftmost` called `leftmost_count`

to complete the filtering. The procedure `leftmost_count` returns an array L where $L[i] = \sum_{j=1}^{j=i} w[j] - \sum_{j=1}^{j=i} min(x_j)$. The value of $L[i]$ represents the maximum possible cardinality that the sequence $[x_1, .., x_i]$ might additionally have while respecting all the AtMost constraints. We define the array R to be the result of `leftmost_count` on the reverse sequence $[x_n, .., x_1]$. Let $d_{res} = d - \sum_{i=1}^{n} min(x_i)$ be the remaining cardinality to satisfy. To complete the filtering, we use the following rules:

1. If $L[n] < d_{res}$, then a failure is raised.
2. If $L[n] = d_{res}$, then for all unassigned variable x_i:
   - If $L[i] + R[n-i+1] \leq d_{res}$, then x_i is assigned to 0.
   - If $L[i-1] + R[n-i] < d_{res}$, then x_i is assigned to 1.

Now in order to explain AtMostSeqCard, we make the distinction between the possible changes made by AtMostSeq or Cardinality on one hand, and the extra filtering that we obtain using `leftmost_count` on the other hand.

5.1.1.1 Explaining AtMostSeq & Cardinality

Explaining AtMostSeq We proceed here by propagating AtMostSeq$(p, q, [x_1, \ldots, x_n])$ with the decomposition into all possible AtMost constraints of size q. Recall that this decomposition does not hinder propagation (Section 4.2). Algorithm 13 shows an AC propagator for AtMost$([x_1, \ldots, x_q], p)$.

Algorithm 13: AtMost$([x_1, \ldots, x_q], p)$

```
if |{x_j | D(x_j) = {1}}| > p then
1   D ← ⊥ ;
else
2   if |{x_j | D(x_j) = {1}}| = p then
        foreach i ∈ {1..q} do
            if D(x_i) = {0,1} then
3               D(x_i) ← {0} ;
return D ;
```

On the one hand, when a failure is raised because of Line 1, the set of all variables assigned to 1 constitutes a possible reason triggering the failure. We therefore use the following propagation rule to explain a failure:

$$\bigwedge_{\mathcal{D}(x_i)=\{1\}} [\![x_i = 1]\!] \Rightarrow \bot$$

This explanation can be reduced as follows. Since $p+1$ assignments of the type $[\![x_i = 1]\!]$ are sufficient to have a failure on AtMost$([x_1, \ldots, x_q], p)$, then any conjunction defined on a subset of $\{[\![x_i = 1]\!] \mid \mathcal{D}(x_i) = \{1\}\}$ of size $p+1$ is a valid explanation of the failure.

On the other hand, any assignment made by this propagator (only of the type $\mathcal{D}(x_i) \leftarrow \{0\}$ in this case) in Line 3 is triggered because of the p assigned variables to 1 (i.e., the test in Line 2). We therefore return the set of assigned variables to 1 as an explanation for $[\![x_i = 0]\!]$.

$$\bigwedge_{\mathcal{D}(x_j)=\{1\}} [\![x_j = 1]\!] \Rightarrow [\![x_i = 0]\!]$$

Explaining Cardinality Notice first that filtering AtMost (Algorithm 13) is very close to filtering Cardinality as we proposed earlier in Algorithm 4. We use therefore similar reasoning to explain the following scenarios:

If a failure is raised in Line 1 (Algorithm 4):

$$\bigwedge_{\mathcal{D}(x_i)=\{1\}} [\![x_i = 1]\!] \Rightarrow \bot$$

Similarly to failures on AtMost, this explanation can be reduced by considering any subset of size $d+1$ from $\{[\![x_i = 1]\!] \mid \mathcal{D}(x_i) = \{1\}\}$.

If a failure is raised in Line 2 (Algorithm 4):

$$\bigwedge_{\mathcal{D}(x_i)=\{0\}} [\![x_i = 0]\!] \Rightarrow \bot$$

This explanation can also be reduced by considering any subset of size $n - d + 1$ from $\{[\![x_i = 0]\!] \mid \mathcal{D}(x_i) = \{0\}\}$.

To explain assignments, we return the set of assigned variables responsible for the domain change at hand:

$$\bigwedge_{\mathcal{D}(x_j)=\{1\}} [\![x_j = 1]\!] \Rightarrow [\![x_i = 0]\!] \quad \text{(propagated at Line 3, Algorithm 4)}$$

$$\bigwedge_{\mathcal{D}(x_j)=\{0\}} [\![x_j = 0]\!] \Rightarrow [\![x_i = 1]\!] \quad \text{(propagated at Line 4, Algorithm 4)}$$

5.1.1.2 Explaining the Extra-Filtering

We move now to explaining the extra-filtering of AtMostSeqCard. We start by giving a procedure explaining the failure triggered when $L[n] < d_{res}$. Next, we show how to use this procedure to explain domain reductions.

Explaining Failure The set of current assignments is a possible naive explanation for the failure. We propose in the following a procedure generating more compact explanations.

In example 5.1, the sequence $[x_1, .., x_6]$ is subject to AtMostSeqCard$(2, 5, 3, [x_1..x_6])$. The left part of the example shows the propagator triggering a failure on a domain $\mathcal{D}$ defined as follows: $\mathcal{D}(x_1) = \{1\}$, $\mathcal{D}(x_3) = \mathcal{D}(x_6) = \{0\}$, and all other variables are unassigned. The current sequence is unsatisfiable since $L[6] < d_{res}$. Consider now the same sequence, however, with a domain $\mathcal{D}'$ where all variables are unassigned except $\mathcal{D}'(x_6) = \{0\}$. This corresponds to the right part of the example. The results of `leftmost` on $\mathcal{D}$ and on $\mathcal{D}'$ are identical. Therefore the set of assignments in $\mathcal{D}$ and the set of assignments in $\mathcal{D}'$ are both valid explanations for this failure. They correspond respectively to the propagation rules $[\![x_1 = 1]\!] \wedge [\![x_3 = 0]\!] \wedge [\![x_6 = 0]\!] \Rightarrow \bot$ and $[\![x_6 = 0]\!] \Rightarrow \bot$. The second explanation is clearly preferable since it is strictly included in the first one.

Example 5.1. *Irrelevant assignments*

$\mathcal{D}$	1	.	0	.	.	0
w	1	1	0	0	0	0
L	0	1	1	1	1	1

$d_{res} = 2$
$L(6) = 1$
→ *Failure*

$\mathcal{D}'$	.	.	.	.	.	0
w	1	1	0	0	0	0
L	1	2	2	2	2	2

$d_{res} = 3$
$L(6) = 2$
→ *Failure*

The idea behind our algorithm for computing shorter explanations is to characterize some assignments with no impact on the behavior of the propagator, and thus can be removed from the naive explanation. The domain obtained by the assignments in the shorter explanation is clearly weaker than the domain from which the failure is triggered. We need to recall and define some notations related to `leftmost` in order to define this weaker domain and to prove our propositions.

Recall that `leftmost` computes an instantiation of maximum cardinality w that is consistent with all AtMost constraints. The instantiation w is initialized with $min(x_i)$ for all i. Afterwards, we greedily assign (from $i = 1$ to $i = n$) $w[i]$ to the value 1 if the following holds:

1. x_i is unassigned.
2. $max_{j \in [1,q]}(c(j)) < p$ where $c(j)$ is the cardinality in w of the jth subsequence including i.

We use in this paragraph slightly modified notations compared to Chapter 4. In fact, many notations are parametrized by the input domain $\mathcal{D}$ and even sometimes depend

on the ith iteration when computing `leftmost`. We therefore need to refer to $\mathcal{D}$ in d_{res}, w and L with $d_{res\mathcal{D}}$, $w_{\mathcal{D}}$ and $L_{\mathcal{D}}$ respectively. Furthermore, at the beginning of any iteration i, we denote by:

- $w^i_{\mathcal{D}}$ the current instantiation w.
- $max_{\mathcal{D}}(i)$ the value of $max_{j\in[1,q]}(c(j))$.
- $card_{\mathcal{D}}(I,i)$ the cardinality of a sub-sequence I.

Now we have all the notations needed to describe the shorter explanations and to prove our results.

Let $[x_1,\ldots,x_n]$ be a sequence of Boolean variables subject to AtMostSeqCard$(p,q,d,[x_1,\ldots,x_n])$. We associate any domain $\mathcal{D}$ for $x_1\ldots x_n$ to a weaker domain $\widehat{\mathcal{D}}$ defined as follows:

$$\begin{array}{ll}\widehat{\mathcal{D}}(x_i)=\{0,1\} & \text{if } \mathcal{D}(x_i)=\{0\}\wedge max_{\mathcal{D}}(i)=p\\ \widehat{\mathcal{D}}(x_i)=\{0,1\} & \text{if } \mathcal{D}(x_i)=\{1\}\wedge max_{\mathcal{D}}(i)\neq p\\ \widehat{\mathcal{D}}(x_i)=\mathcal{D}(x_i) & \text{otherwise}\end{array}$$

We prove in the following that the outcome of `leftmost` on $\mathcal{D}$ and $\widehat{\mathcal{D}}$ is the same. Hence the propagator behavior is the same on both domains.

Lemma 5.1. $w_{\widehat{\mathcal{D}}}=w_{\mathcal{D}}$.

Proof. Suppose that there exists an index $i\in[1..n]$ s.t. $w_{\widehat{\mathcal{D}}}[i]\neq w_{\mathcal{D}}[i]$ and let k be the smallest index verifying this property. Since $\widehat{\mathcal{D}}$ is weaker than $\mathcal{D}$ and `leftmost` is a greedy procedure assigning the value 1 whenever possible from left to right, it follows that $w_{\mathcal{D}}[k]=0$ and $w_{\widehat{\mathcal{D}}}[k]=1$. Hence $max_{\mathcal{D}}(k)=p$ and $max_{\widehat{\mathcal{D}}}(k)<p$. In other words, there exists a subsequence I containing x_k s.t. $card_{\mathcal{D}}(I,k)$ is equal to p, and $card_{\widehat{\mathcal{D}}}(I,k)$ is less than p. From this we deduce that there exists a variable $x_j\in I$ such that $w^k_{\mathcal{D}}[j]=1$ and $w^k_{\widehat{\mathcal{D}}}[j]=0$.

We show by contradiction that the latter statement cannot hold. Observe first that j must be greater than k because k is the smallest index where `leftmost` behaves differently. Next, from $w^k_{\mathcal{D}}[j]=1$ and $w^k_{\widehat{\mathcal{D}}}[j]=0$, only two cases are possible:

1. x_j is unassigned in $\mathcal{D}$ and $\widehat{\mathcal{D}}$: In this case, since $j>k$, then at iteration k both $w^k_{\mathcal{D}}(j)$ and $w^k_{\widehat{\mathcal{D}}}(j)$ are equal to 0 because `leftmost` changes the values of w greedily following the lexicographical order. Hence the first contradiction.

2. x_j is assigned in $\mathcal{D}$ but not in $\widehat{\mathcal{D}}$: It follows that $\mathcal{D}(x_j) = \{1\}$ since $w^k_{\mathcal{D}}[j] = 1$. Moreover, since $\widehat{\mathcal{D}}(x_j) = \{0,1\}$ then the definition of $\widehat{\mathcal{D}}$ implies that $max_{\mathcal{D}}(j) \neq p$. Recall now that $card_{\mathcal{D}}(I,k) = p$, therefore $max_{\mathcal{D}}(j) = p$ which is impossible.

□

Theorem 5.2. *If a failure is raised because* $L_{\mathcal{D}}[n] < d_{res\mathcal{D}}$ *, then*

$$\bigwedge_{\widehat{\mathcal{D}}(x_i)=\{1\}} [\![x_i = 1]\!] \wedge \bigwedge_{\widehat{\mathcal{D}}(x_i)=\{0\}} [\![x_i = 0]\!] \Rightarrow \bot$$

is a valid explanation.

Proof. We show that the set of assignments in $\widehat{\mathcal{D}}$ is sufficient to have a failure. In other words, we show that $L_{\widehat{\mathcal{D}}}[n] < d_{res\widehat{\mathcal{D}}}$. Let α be the number of variables having $\{1\}$ as a domain in D but unassigned in $\widehat{\mathcal{D}}$. It is clear that $d_{res\widehat{\mathcal{D}}} = d_{res\mathcal{D}} + \alpha$. By Lemma 5.1, we know that $w_{\mathcal{D}}$ and $w_{\widehat{\mathcal{D}}}$ are equal. It follows that $L_{\widehat{\mathcal{D}}}[n] = L_{\mathcal{D}}[n] + \alpha$. Therefore, since $L_{\mathcal{D}}[n] < d_{res}$ then $L_{\widehat{\mathcal{D}}}[n] < d_{res\widehat{\mathcal{D}}}$.

□

Theorem 5.2 gives us a linear time procedure to explain a failure. In fact, it is sufficient to compute the values $max_{\mathcal{D}}(i)$ in order to construct $\widehat{\mathcal{D}}$. All these values can be computed using one call to `leftmost_count` which is linear in time. Example 5.2 illustrates the explanation procedure.

Example 5.2. *Reducing the default explanation*

$\mathcal{D}$	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	.	.	.	1
$max_{\mathcal{D}}$	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	2	2	2
$w_{\mathcal{D}}$	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1
$L_{\mathcal{D}}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
	$d_{res\mathcal{D}} = 2$ *and* $L_{\mathcal{D}}[25] = 1 < d_{res\mathcal{D}} \implies$ *Failure*																								
$\widehat{\mathcal{D}}$	1	1	.	.	.	.	.	.	1	1	.	.	.	.	0	0	0	0	.	0	0	.	.	.	1
$w_{\widehat{\mathcal{D}}}$	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1
$L_{\widehat{\mathcal{D}}}$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	2	2	2	3	3	3	3
	$d_{res\widehat{\mathcal{D}}} = 4$ *and* $L_{\widehat{\mathcal{D}}}[25] = 3 < d_{res\widehat{\mathcal{D}}} \implies$ *Failure*																								

We illustrate here the explanation of a failure on AtMostSeqCard$(2,5,9,[x_1..x_{25}])$ *triggered using the extra-filtering rules. Observe first that* AtMostSeq *and* Cardinality *are both* AC*. Next, the propagator returns a failure since* $L_{\mathcal{D}}(25) = 1 < d_{res\mathcal{D}} = 2$*. The default explanation corresponds to the set of all the assignments in* $\mathcal{D}$*, whereas our procedure generates a more compact explanation by considering only the assignments in* $\widehat{\mathcal{D}}$*.*

Red values in the $max_{\mathcal{D}}$ line represent the indices corresponding to variables being unassigned in $\widehat{\mathcal{D}}$. As we can see, $w_{\widehat{\mathcal{D}}}$ is identical to $w_{\mathcal{D}}$. Therefore, the propagator behaves the same way on both domains. As a result, we reduce the size of the explanation from 22 *to* 11.

Note that this reduction is not optimal w.r.t. the explanation size. For instance, the first assignment $[\![x_1 = 1]\!]$ in Example 5.2 can be removed from the reduced explanation and the rest of the assignments still lead to a failure.

Explaining Pruning Suppose that an assignment $[\![x = v]\!]$ was triggered by the propagator for an input domain $\mathcal{D}$ at level l with a rank r. Consider the new domain $\mathcal{D}'$ identical to $\mathcal{D}$ at level l and rank $r-1$ except for x with $\mathcal{D}'(x) = \{1-v\}$ (i.e., the opposite of v). Since the pruning is correct, the constraint is unsatisfiable on $\mathcal{D}'$. Let $\Psi \Rightarrow \bot$ be the propagation rule explaining this failure using the previous mechanism. Observe that $[\![x = 1 - v]\!]$ has to be in Ψ, otherwise we have a failure without assigning x to $1 - v$ which contradicts our first hypothesis that $[\![x = v]\!]$ was triggered by the propagator on $\mathcal{D}$ at level l and rank r. The propagation rule can be reformulated as follows: $\Psi' \wedge [\![x = 1 - v]\!] \Rightarrow \bot$ (s.t. $\Psi' = \bigwedge_{q \neq [\![x=1-v]\!] \in \Psi}$) which is equivalent to $\Psi' \Rightarrow [\![x = v]\!]$. We can therefore use the same algorithm to explain failures and pruning.

5.1.2 Pseudo-Boolean & SAT Models for the Car-Sequencing Problem

We show first a Pseudo-Boolean model for the car-sequencing problem that serves as a starting point for the SAT formulations. The SAT models that we use are those proposed by Mayer-Eichberger and Walsh in [5, 92].

5.1.2.1 A Pseudo-Boolean Formulation

The DECOMPOSE model (Section 3.1) of this problem can be easily translated into a Pseudo-Boolean model since all constraints are in fact sum expressions. We use the same Boolean variables y_i^j standing for whether the vehicle in the i^{th} slot requires option j. Moreover, the integer domains of class variables $x_1, \ldots, x_n$ are expressed based on the direct encoding with $n \times k$ Boolean variables c_i^j standing for whether the ith vehicle is of class j. Since we use a Pseudo-Boolean model, we have the choice between using clauses to encode the different relationship between c_i^j or simply post one constraint per class variable using $\sum_j c_i^j = 1$ for all $i \in [1..n]$. The Pseudo-Boolean formulation of this problem that we adopt is the following.

1. *Demand constraints*: $\forall j \in [1..k]$, $\sum_i c_i^j = d_j^{class}$

2. *Capacity constraints*: $\sum_{l=i}^{i+q_j-1} y_l^j \leq p_j,\ \forall i \in \{1,\ldots,n-q_j+1\}$
3. *Channeling*:
   - $\forall i \in [1..n],\ \forall l \in [1..k]$, we have:
     - $\forall j \in \mathcal{O}_l,\ \overline{c_i^l} \vee y_i^j$
     - $\forall j \notin \mathcal{O}_l,\ \overline{c_i^l} \vee \overline{y_i^j}$
   - $\forall i \in [1..n],\ j \in [1..m], \overline{y_i^j} \vee \bigvee_{l \in \mathcal{C}_j} c_i^l$
4. *Class constraints*: $\forall i \in [1..n],\ \sum_j c_i^j = 1$

5.1.2.2 From Pseudo-Boolean to SAT

Notice that the above Pseudo-Boolean model contains only clauses, AtMost, and Cardinality constraints. A simple and straightforward way to formulate this problem into SAT is to encode each AtMost/Cardinality constraint into a CNF. The latter has been intensively studied in the last decade (see for instance [127, 49, 125, 6, 3]). We use, however, the three SAT encodings proposed for this problem by Mayer-Eichberger and Walsh in [5, 92]. They correspond in fact to three different ways of encoding AtMostSeqCard. All of them are based on the Sequential Counter [127]. We give a brief description for these models and refer the reader to [5] for more details.

The first step is to show the encoding used for Cardinality$([x_1,\ldots,x_n],d)$.

- Variables:
  - $s_{i,j}$: $\forall i \in [0..n],\ \forall j \in [0..d+1]$, $s_{i,j}$ is *true* iff $\sum_{k \in [1..i]} x_k \geq j$
- Clauses: $\forall i \in [1..n]$
  - $\forall j \in [0..d+1]$
    1. $\neg s_{i-1,j} \vee s_{i,j}$
    2. $x_i \vee \neg s_{i,j} \vee s_{i-1,j}$
  - $\forall j \in [1..d+1]$
    3. $\neg s_{i,j} \vee s_{i-1,j-1}$
    4. $\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}$
- Initial values:
    5. $s_{0,0}: true\ ;\ s_{0,1}: false\ ;\ s_{n,d}: true\ ;\ s_{n,d+1}: false\ ;$

In this encoding, $(n+1) \times (d+1)$ atoms $s_{i,j}$ are used in addition to the variables $[x_1, \ldots, x_n]$. An atom $s_{i,j}$ is semantically equivalent to have a lower bound at least equal to j in the sum $\sum_{k \in [1..i]} x_k$. The clauses 1 & 3 ensure the monotonicity of the sum, while clauses 2 & 4 perform a channeling between the variables x_i and $s_{i,j}$.

Adapting this encoding for an AtMost constraint is quite simple. In fact, it is sufficient to change the initial value of $s_{n,d}$ from *true* to unassigned. This way makes the constraint satisfied iff $\sum_{i \in [1..n]} x_i \leq d$.

Recall that AtMostSeqCard is defined by a conjunction of Cardinality and a set of AtMost constraints. We denote by: SAT_{Card} the (above) encoding for Cardinality; and SAT_{Atmost} the encoding applied to all AtMost constraints. Note that each AtMost constraint is encoded independently with new variables channeled only to option position variables.

Another possible way for encoding the chain of AtMost constraints can use similar encoding of the Gen-Sequence constraint [8, 32]. For each subsequence of size q whose latest index is i, we have the clause:

6. $\quad \neg s_{i,j} \vee s_{i-q,j-u}$

This encoding is denoted SAT_{Seq}.

Mayer-Eichberger and Walsh showed not only that the level of pruning of SAT_{Seq} is incomparable with SAT_{Atmost} but also combining SAT_{Card}, SAT_{Atmost}, and SAT_{Seq} maintains Arc Consistency on AtMostSeqCard [5]. Three SAT models for the car-sequencing problem are therefore proposed. They all encode the basic model using the following encodings of AtMostSeqCard:

1. CNF_A uses SAT_{Card} and SAT_{Atmost}.
2. CNF_S uses SAT_{Card} and SAT_{Seq}.
3. CNF_{A+S} combines SAT_{Card}, SAT_{Atmost} and SAT_{Seq}.

5.1.3 Experimental Results

We test the different approaches on the previous benchmarks of car-sequencing (used in Chapters 3 and 4). We reorganize the instances into three categories.

1. EasySat: It contains all instances from *set1* and *set2*. All these instances $(70 + 4)$ are satisfiable and easy for all the methods tested here.

2. HardSat: It contains the instances of *set4*. These instances (7) are known to be satisfiable but very hard to solve.

3. Unsat: It contains all unsatisfiable instances from *set3* in addition to the 23 unknown instances form *set5*.

We ran the following models:

Hybrid CP/SAT We use Mistral-2.0[1] as a hybrid CP/SAT solver with backward explanations. Our hybrid model is based on the Pseudo-Boolean formulation of the problem, however, by using AtMostSeqCard for capacity constraints. Note that the rest of the constraints are either Cardinality or AtMost constraints. We explain them in the same way we proposed previously in Section 5.1.1.1.

Using a hybrid solver has the advantage of using adaptive branching coming from the SAT component as well as problem-specific heuristics. We therefore propose to test the following configurations differentiated by the heuristic being used:

1. *Hybrid(VSIDS)*: using VSIDS.

2. *Hybrid(Slot)*: using the heuristic $\langle opt, mid, \delta, \varnothing \rangle$ (see Chapter 3).

3. *Hybrid(Slot/VSIDS)*: using first *Hybrid(Slot)*, then switching after 100 non-improving restarts to *Hybrid(VSIDS)*.

4. *Hybrid(VSIDS/Slot)*: the reverse of *Hybrid(Slot/VSIDS)*

SAT We use the three SAT models CNF_A, CNF_S, and CNF_{A+S} using Minisat[48] (version 2.2.0) with default parameter settings.

CP and Pseudo-Boolean Models We compare against the following «reference» approaches:

1. CP_{AMSC}: The pure CP model using AtMostSeqCard without clause learning with the same heuristic used in *Hybrid(Slot)* and the same solver Mistral-2.0.

2. *PBO-clauses*: A Pseudo-Boolean method relying on SAT encoding. We used MiniSat+ [49] on the Pseudo-Boolean encoding described in Section 5.1.2.1.

3. *PBO-cutting planes*: A Pseudo-Boolean method with dedicated propagation and learning based on cutting planes [45]. We used SAT4J [19] on the same model, with the «CuttingPlanes» algorithm.

[1] http://homepages.laas.fr/ehebrard/mistral.html

All experiments are realized on Intel Xeon CPUs 2.67GHz under Linux. For each instance, we ran 5 randomized runs with Luby restarts and a 20 minutes time cutoff. The summary of these results is given in Table 5.1. Recall that a run is said to be 'successful' iff a solution is found or the search space is completely explored without finding any solution. For each category of instances, we report: the total number of successful runs (*#suc*); the averaged number of failures (*avg fails*) and the averaged CPU time (*time*) in seconds. The statistics «*time*» and «*avg fails*» are computed only for the successful runs. We emphasize the statistics of the best method (w.r.t. *#suc*, ties broken by *time*) for each category using **bold** face fonts.

TABLE 5.1: Experimental comparison of CP, SAT, hybrid, and Pseudo-Boolean models for the car-sequencing problem

Method	EasySat (74×5)			HardSat (7×5)			Unsat (28×5)		
	#suc	*avg fails*	*time*	*#suc*	*avg fails*	*time*	*#suc*	*avg fails*	*time*
CNF_A	370	2073	1.71	28	337194	282.35	**85**	**249301**	**105.07**
CNF_S	370	1114	0.87	31	60956	56.49	65	220658	197.03
CNF_{A+S}	370	612	0.91	34	32711	36.52	77	190915	128.09
Hybrid(VSIDS)	370	903	0.23	16	207211	286.32	35	177806	224.78
Hybrid(VSIDS/Slot)	370	739	0.23	35	76256	64.52	37	204858	248.24
Hybrid(Slot/VSIDS)	370	132	0.04	34	4568	2.50	37	234800	287.61
Hybrid(Slot)	370	132	0.04	**35**	**6304**	**3.75**	23	174097	299.24
CP_{AMSC}	**370**	**43.06**	**0.03**	35	57966	16.25	0	-	-
PBO-clauses	277	538743	236.94	0	-	-	43	175990	106.92
PBO-cutting planes	272	2149	52.62	0	-	-	1	5031	53.38

From Table 5.1, we first note that CP and hybrid models outperform other approaches on satisfiable instances (i.e., EasySat and HardSat). The best method in average for both sets is the hybrid model using CP branching. By considering all the results on these instances, one can observe that models enforcing Arc Consistency on AtMostSeqCard are the best choices for finding solutions quickly. In fact, this claim is confirmed by the poor performances of Pseudo-Boolean models on satisfiable instances together with the distinguished results of CNF_{A+S} compared to other SAT models. Recall that CNF_{A+S} simulates ac on AtMostSeqCard. It is worth mentioning the importance of using the crafted heuristic compared to VSIDS, at least within hybrid models. For instance, on the dataset "HardSat", we move from solving 16 instances with *Hybrid(VSIDS)* to 35 instances with *Hybrid(Slot)*. In general, the results of satisfiable instances show that propagation is by far the most crucial factor for finding solutions. Moreover, the use of built-in heuristics is clearly beneficial compared to «blind» branching when using hybrid models.

Conversely to these observations, the results on the dataset "Unsat" instances clearly show that clause learning is the most important ingredient for proving unsatisfiability. There are a number evidences supporting this claim. First, while the CP model fails

to build proofs on any instance for this set, its equivalent hybrid model (*Hybrid(Slot)*) succeeds on 23 instances. We stress here the impact of VSIDS with hybrid models as we move from 23 to 37 instances with *Hybrid(Slot/VSIDS)* or *Hybrid(VSIDS/Slot)*. Next, the *PBO-clauses* model, which relies essentially on basic SAT encoding without any extra filtering, performs better that hybrid models on this set with 43 successful runs. Finally, the best results on this set come from the SAT models. Specifically, the «lightest» model CNF_A is, surprisingly, the best model for proving unsatisfiability with 85 instances.

To summarize the experimental findings, we first observed that clause learning improves the global performances generally. This is specially true when proving unsatisfiability. Second, we confirm a strong correlation between advanced propagation and finding solutions quickly for this problem. However, for building proofs, clause learning is the most crucial factor and propagation is less useful. Finally, regarding the choice of heuristic, adaptive-branching is very beneficial for building proofs while problem-specific heuristics are much helpful for finding solutions efficiently.

5.2 Revisiting Lazy Generation

We move now to the second part of our contributions regarding clause learning. We revisit the lazy generation of Boolean variables for encoding the domains. In particular, we show how to avoid the issue mentioned in Section 2.3.2.2. Recall that when lazily generating variables, clauses encoding the domains become redundant (see Section 2.3.2.2 for more details). The DomainFaithfulness constraint that we propose in this section avoids such redundancy while ensuring the same level of consistency without computational overhead.

This novel lazy generation is used in the next section with our hybrid models for solving scheduling problems. We consider only the lazy generation of atoms of the type $[\![x \leq u]\!]$ since all propagators in our models performs only bound tightening operations. Note that this type of domain reduction is the most used for scheduling problems in general. Nevertheless, the generalization of our propositions with atoms of the type $[\![x = v]\!]$ is quite simple and straightforward.

5.2.1 The DomainFaithfulness Constraint

We first recall the redundancy issue related to lazy generation. When an atom $[\![x \leq u]\!]$ has to be generated, we add the clauses $\neg[\![x \leq a]\!] \vee [\![x \leq u]\!]$; $\neg[\![x \leq u]\!] \vee [\![x \leq b]\!]$ where a and b are the nearest generated bounds to u with $a < u < b$. After adding these clauses, the clause $\neg[\![x \leq l]\!] \vee [\![x \leq u]\!]$ becomes redundant. We show how to avoid this redundancy.

Instead of generating clauses to encode the different relationships between the newly generated atoms, we propose to encode such relations through a new constraint called DomainFaithfulness. This constraint has a twofold role: firstly, it simulates UP as if the atoms were generated eagerly; secondly it performs a complete channeling between the range variable and all its domain atoms.

Let x be a Range variable (i.e., with a domain of the form $[l, u]$). Let $[v_1, \ldots, v_n]$ be a sequence of integer values, and $[b_1 \ldots b_n]$ be a sequence of lazily generated Boolean variables s.t. b_i is the atom $[\![x \leq v_i]\!]$. We assume that b_i is the ith generated Boolean variable for all i. We define the DomainFaithfulness constraint as follows.

Definition 5.3. DomainFaithfulness$(x, [b_1 \ldots b_n], [v_1, \ldots, v_n])$:

$$\forall i, b_i \leftrightarrow x \leq v_i$$

For each Range variable x, we use one DomainFaithfulness constraint (denoted by DomainFaithfulness(x)). Initially, the scope of DomainFaithfulness(x) contains only x. Afterwards, whenever an atom $b \Leftrightarrow [\![x \leq v]\!]$ has to be generated, we simply add b to the scope of DomainFaithfulness(x).

5.2.1.1 Propagating DomainFaithfulness

We present first a complete filtering procedure for DomainFaithfulness in Algorithm 14 running in $O(n)$ time complexity. Next, we show that one can enforce the same propagation level with a constant amortized time complexity down a branch of the search tree.

Algorithm 14: AC(DomainFaithfulness$(x, [b_1 \ldots b_n], [v_1, \ldots, v_n])$)

```
1  ub ← min(max(x), min(v_i | D(b_i) = {1}));
2  lb ← max(min(x), 1 + max(v_i | D(b_i) = {0}));
   if ub < lb then
3      return ⊥ ;
4  D(x) ← D(x) ∩ [lb, +∞[ ;
5  D(x) ← D(x)∩] − ∞, ub] ;
6  for i ∈ [1, n] do
       if v_i ≥ ub then
           D(b_i) ← {1};
       if v_i < lb then
           D(b_i) ← {0};
   return D ;
```

We assume that $n >= 1$, otherwise no propagation is needed since no atom is generated. The first step is to look for the tightest possible bounds for x. The new upper bound

ub is the minimum between the current upper bound of x and the minimum value v_i where b_i is assigned to 1. Similarly, the new lower bound lb is the maximum between the current lower bound and the maximum value $v_i + 1$ where b_i is assigned to 0. These new bounds are computed at the first two lines of Algorithm 14.

Regarding failure, there is only one way to make the constraint violated. This case corresponds to the situation when ub is less that lb (Line 3). The rest of the propagator is quite straightforward. First, we update the domain of x with the new bounds (Line 4 and Line 5). Then, we assign the atoms b_i in the natural way (Line 6). That is, any variable b_i is assigned to 1 if $v_i \geq ub$ and to 0 if $v_i < lb$. Figure 5.1 visualizes the effect of propagating DomainFaithfulness on $[b_1 \dots b_n]$.

FIGURE 5.1: Assigning $b_1, \dots, b_n$

$$\underbrace{0\ 0\ \dots\ 0}_{\{b_i \mid v_i < lb\}} \quad \underbrace{\{0,1\}\ \{0,1\}\ \dots\ \{0,1\}}_{\{b_i \mid lb \leq v_i < ub\}} \quad \underbrace{1\ 1\ \dots\ 1}_{\{b_i \mid v_i \geq ub\}}$$

Theorem 5.4. *Algorithm 14 enforces* AC *for* DomainFaithfulness *in* $O(n)$.

Proof. The time complexity for this algorithm is clearly $O(n)$. We show how to build supports for any possible assignment after propagating DomainFaithfulness. Assigning x to any value $v \in [lb, ub]$ has clearly a support by assigning any atom b_i to 1 if $v_i \geq v$ and to 0 otherwise. For the rest of assignments, it is also easy to find supports. We distinguish two cases for building supports for assignments of the type $[\![b_i = 1]\!]$.

- If $v_i \geq ub$ (i.e., b_i is already assigned to 1), we assign x to ub, and all unassigned b_j to 0.
- If $v_i < ub$ (i.e., b_i is unassigned), we assign x to lb, and all unassigned b_j to 1.

Similarly, we build supports for the assignments of the type $[\![b_i = 0]\!]$ as follows:

- If $v_i < lb$ (i.e., b_i is already assigned to 0), we assign x to lb, and all unassigned b_j to 1.
- If $v_i \geq lb$ (i.e., b_i is unassigned), we assign x to ub, and all unassigned b_j to 0.

□

5.2.2 Incrementality

We introduce here an incremental procedure to propagate DomainFaithfulness in a constant amortized time complexity down a branch of the search tree.

We use two arrays called s and g defined as follows: For each $i \in [1, n]$:

- If $\{v_k \mid v_k < v_i\} \neq \varnothing$, then $s[i] = \lambda$ where $v_\lambda = max\{v_k \mid v_k < v_i\}$, otherwise $s[i] = 0$. That is, the value of $s[i]$ represents the index j of the greatest value v_j that is smaller than v_i if such index exists, and 0 otherwise.
- If $\{v_k \mid v_k > v_i\} \neq \varnothing$, then $g[i] = \lambda$ where $v_\lambda = min\{v_k \mid v_k > v_i\}$, otherwise $g[i] = 0$. That is, $g[i]$ represents the index j of the smallest value v_j that is greater than v_i if such index exists, and 0 otherwise.

Consider now the example of $\mathcal{D}(x) = [17, 83]$ and an atom b_k corresponding to $[\![x \leq 64]\!]$ (i.e., $v_k = 64$). Suppose now that assigning b_k to 1 is the only new event before propagating DomainFaithfulness. It is easy to see that the only changes needed to maintain AC on this constraint are the tightening of the upper bound of x to 64 and the assignment of some atoms to 1. These atoms correspond to the set $\eta = \{b_{g[k]}, b_{g[g[k]]}, b_{g[g[g[k]]]} \dots b_{last_k}\}$ where b_{last_k} is unassigned and $b_{g[last_k]}$ is assigned to 1. The time complexity needed for this propagation is $O(|\eta|)$. Take now the same example, however, by having in addition to assigning b_k to 1, a new upper bound $ub^* = 48$ as an event. In this case, one can proceed exactly as in the previous example by assigning all atoms in $\eta = \{b_{g[k]}, b_{g[g[k]]}, b_{g[g[g[k]]]} \dots b_{last_k}\}$ to 1, then continue assigning other atoms to 1 to be consistent with the new upper bound. The new set of atoms is $\eta^* = \{b_{s[k]}, b_{s[s[k]]}, \dots, b_{ub}\}$ where $v_{ub} = min(v_k \mid v_k \geq ub^*)$. The time complexity in this case is $O(|\eta| + |\eta^*|)$.

Our incremental filtering is organized in two parts:

1. Simulating UP as if the atoms $b_1, \dots, b_n$ were eagerly generated with all domain clauses.
2. Performing the channeling between x and $b_1, \dots, b_n$.

Algorithm 15 depicts the main procedure for this incremental propagator. It uses algorithms 16, 17, and 18 as follows: Any event related to assigning an atom b_i to 1 is handled by Algorithm 16 ($UB(i, i_{ub})$); an event of assigning b_i to 0 is handled by Algorithm 17($LB(i, i_{lb})$); and the changes on $\mathcal{D}(x)$ are handled by Algorithm 18 ($Update_Range(i_{lb}, i_{ub}, lb, ub)$).

In Line 1 (respectively Line 2) of Algorithm 15, we setup i_{ub} (respectively i_{lb}) as the index of literal standing for the maximum (respectively minimum) value in $\{v_j \mid j \in [1, n]\}$. This initialization happens only in the first call. In subsequent calls, we use their updated values coming from the previous call. Moreover, these values are re-established when backtracking[2].

[2] i_{ub} and i_{lb} are implemented as a "reversible" integer.

Algorithm 15: Propagate(DomainFaithfulness($x, [b_1 \dots b_n], [v_1, \dots, v_n]$))

```
lb ← false;
ub ← false;
changed ← false;
//i_ub and i_lb can be modified later with UB(i, i_ub) and LB(i, i_lb) respectively.
1 i_ub ← arg max_j({v_j | j ∈ [1, n]}) ;
2 i_lb ← arg min_j({v_j | j ∈ [1, n]}) ;
//Θ is a list containing indices of newly assigned variables.
while Θ is not empty do
    i ← Θ.pop() ;
    if i >= 1 then
3       if D(b_i) = {1} then
            if UB(i, i_ub) then
                ub ← true ;
4       else
            if LB(i, i_lb) then
                lb ← true ;
        if D = ⊥ then
            return ⊥ ;
    else
        changed ← true ;
5 if changed then
    Update_Range(i_lb, i_ub, lb, ub) ;
return D ;
```

We use a list Θ containing indices of newly assigned variables in the scope of the constraint. We assume that Θ is globally modifiable by all algorithms and that the index of the variable x is 0 and b_i is i for all $i \in [1..n]$.

We show how the two parts of filtering are maintained by one call to Algorithm 15.

Simulating UP: Suppose that all atoms $b_1, \dots, b_n$ are eagerly generated with all domain clauses. The set of these clauses can be described with $\{\neg b_{s[i]} \vee b_i \mid i \in [1,n] \wedge s[i] \neq 0\}$ or $\{\neg b_i \vee b_{g[i]} \mid i \in [1,n] \wedge g[i] \neq 0\}$. There are two possible scenarios of propagation depending on the assignment of a variable b_i .

- b_i becomes assigned to 1: In this case, UP propagates the clause $\neg b_i \vee b_{g[i]}$ by assigning $b_{g[i]}$ to 1 or triggers failure if $\mathcal{D}(b_{g[i]}) = \{0\}$. If $b_{g[i]}$ becomes assigned, then UP should triggers propagation for clauses watched by $b_{g[i]}$. This scenario is triggered at Line 3 in Algorithm 15 and executed at Line 2, and Line 3 in Algorithm 16.

Algorithm 16: $UB(i, i_{ub}))$

```
//We first update the index i_ub
if v_{s[i]} < v_{i_ub} then
1   i_ub ← s[i] ;
//Here we simulate the propagation of the clause ¬b_i ∨ b_next
2 next ← g[i] ;
3 if next ≠ 0 then
      if max(b_next) = 0 then
4         D ←⊥;
          return false;
      else
5         D(b_next) ← {1} ;
          Θ.add(next) ;
//Now we perform the channeling between x and b_i
6 if min(x) > v_i then
7     D ←⊥;
      return false;
8 if v_i < max(x) then
      D(x) ← D(x)∩]−∞, v_i] ;
      return true ;
  else
      return false ;
```

Algorithm 17: $LB(i, i_{lb}))$

```
//We first update the value of i_lb
if v_{g[i]} > v_{i_lb} then
1   i_lb ← g[i] ;
//Here we simulate the propagation of the clause ¬b_next ∨ b_i
2 next ← s[i] ;
3 if next ≠ 0 then
      if min(b_next) = 1 then
4         D ←⊥;
          return false;
      else
5         D(b_next) ← {0} ;
          Θ.add(next) ;
//Now we perform the channeling between x and b_i
6 if max(x) < (v_i + 1) then
7     D ←⊥;
      return false;
8 if (v_i + 1) > min(x) then
      D(x) ← D(x) ∩ [v_i + 1, +∞[ ;
      return true ;
  else
      return false ;
```

Algorithm 18: $Update_Range(i_{lb}, i_{ub}, lb, ub)$)

```
next ← i_ub ;
bound ← max(x) ;
if not(ub) then
1   while next ≠ 0 do
        if v_next ≥ bound then
2           D(b_next) ← {1} ;
            next ← s[next] ;
        else
            next ← 0 ;
next ← i_lb;
bound ← min(x) ;
if not(lb) then
3   while next ≠ 0 do
        if v_next < bound then
4           D(b_next) ← {0} ;
            next ← g[next] ;
        else
            next ← 0 ;
```

- b_i becomes assigned to 0: Conversely to the previous case, UP propagates the clause $\neg b_{s[i]} \vee b_i$ by assigning $b_{s[i]}$ to 0 or triggers failure if $\mathcal{D}(b_{s[i]}) = \{1\}$. If $b_{s[i]}$ becomes assigned, then UP should trigger propagation for clauses watched by $b_{s[i]}$. This scenario is triggered at Line 4 in Algorithm 15 and executed at Line 2, and Line 3 in Algorithm 17.

Let η be the set of all atoms assigned by our algorithm. The worst case time complexity for simulating UP is clearly $O(|\eta|)$ which is the same as if UP propagates with the 2-watched literals. Therefore, the time complexity of this part is $O(n)$ down a branch of the search tree, and subsequently corresponds to a constant amortized complexity.

Channeling Between x and $b_1, \ldots, b_n$: There are two cases to distinguish when performing this channeling.

1. Changing $\mathcal{D}(x)$ based on newly assigned atoms: When an atoms $b_i \leftrightarrow [\![x \leq v_i]\!]$ becomes assigned to 1, one have to check:
   (a) If enforcing v_i as a new upper bound for x can make $\mathcal{D}(x)$ empty, and hence failure should be triggered. This test is performed at Line 6 of Algorithm 16.
   (b) If v_i can be the new upper bound of x. This is performed at Line 8 of Algorithm 16.

The case where b_i becomes assigned to 0 is similarly handled at Line 6 and Line 8 in Algorithm 17.

2. Assigning some atoms from $b_1, \ldots, b_n$ to be coherent with $\mathcal{D}(x)$. This propagation is handled by Algorithm 18 ($Update_Range(i_{lb}, i_{ub}, lb, ub)$). Clearly, when no change occurs on $\mathcal{D}(x)$ before calling Algorithm 15 «Propagate(DomainFaithfulness($x, [b_1 \ldots b_n], [v_1, \ldots, v_n]$))», then no propagation is needed. This is exactly what happens at Line 5 in Algorithm 15 using the Boolean *changed*. In the case where $\mathcal{D}(x)$ changed, we treat each type of domain change separately. We show the procedure used when the change concerns a new upper bound. The case of a new lower bound is similar. Let u be the new upper bound of x. We show that every atom b_i such that $v_i \geq u$ is assigned to 1 when the algorithm ends.

   (a) If there exists an atom b_j in the initial Θ list s.t. Algorithm 16 changes the upper bound of x to be v_j at Line 8, then no further propagation is needed.

   (b) Otherwise, every atom b_i with a value $v_i \geq max(x)$ should be assigned to 1. This is done by means of an index i_{ub} as follows: We first make sure that every atom with a value that is greater than $v_{i_{ub}}$ is already assigned to 1. Afterwards, we assign all atoms in the sequence $[b_{i_{ub}}, b_{s[i_{ub}]}, b_{s[s[i_{ub}]]} \ldots, b_{last_{ub}}]$ to 1 where $v_{last_{ub}} = min(v_k \mid v_k \geq max(x))$. This is exactly what happens in the loop of Line 1 in Algorithm 18. Now regarding the index i_{ub}, recall that it has to guarantee that all atoms with a value greater than $v_{i_{ub}}$ are already assigned to 1. Therefore, we initialize i_{ub} to the be the index of the greatest possible value v_i (Line 1 in Algorithm 15). Then, whenever we find an atom b_k newly assigned to 1 and associated to a value v_k that is smaller than the current $v_{i_{ub}}$, we update i_{ub} with the value $s[k]$. Recall that the part simulating UP guarantees that all atoms with a value $v \geq v_k$ are assigned to 1.

Regarding the complexity of this part, observe that considering i_{ub} and i_{lb} as reversible integers makes the running time of this part also $O(n)$ down a branch of the search tree and therefore corresponds to a constant amortized complexity.

5.2.3 Explaining DomainFaithfulness

Since DomainFaithfulness is used in a Hybrid CP/SAT Solver, we must explain all possible domain changes and failures triggered by this constraint.

5.2.3.1 Explaining Failure

There are several cases to find a failure by our algorithms. We give for each one a possible explanation using the current values of $next$, $min(x)$, $max(x)$, and i at the moment of propagation.

- Line 4 in Algorithm 16: $[\![b_{next} = 0]\!] \wedge [\![b_i = 1]\!] \Rightarrow \bot$
- Line 7 in Algorithm 16: $[\![x \geq min(x)]\!] \wedge [\![b_i = 1]\!] \Rightarrow \bot$
- Line 4 in Algorithm 17: $[\![b_{next} = 1]\!] \wedge [\![b_i = 0]\!] \Rightarrow \bot$
- Line 7 in Algorithm 17: $[\![x \leq max(x)]\!] \wedge [\![b_i = 0]\!] \Rightarrow \bot$

5.2.3.2 Explaining Pruning

Tightening the bounds of the range variable x is possible only when a Boolean variable with the same bound value was previously set to $true$ / $false$. We therefore use the following rules to explain $[\![x \geq l]\!]$ and $[\![x \leq u]\!]$ without saving any information (i.e., typically used with backward explanations):

- $[\![b_k = 0]\!] \Rightarrow [\![x \geq l]\!]$ s.t. $v_k = l - 1$
- $[\![b_k = 1]\!] \Rightarrow [\![x \leq u]\!]$ s.t. $v_k = u$

For the assignments of the type $[\![b = 1]\!]$ and $[\![b = 0]\!]$, we make a clear distinction whether they are assigned by Algorithms $UB(i, i_{ub})/LB(i, i_{lb})$ or by Algorithm $Update_Range(i_{lb}, i_{ub}, lb, ub)$.

- Line 5 in Algorithm 16: $[\![b_i = 1]\!] \Rightarrow [\![b_{next} = 1]\!]$
- Line 2 in Algorithm 18: $[\![x \leq max(x)]\!] \Rightarrow [\![b_{next} = 1]\!]$
- Line 5 in Algorithm 17: $[\![b_i = 0]\!] \Rightarrow [\![b_{next} = 0]\!]$
- Line 4 in Algorithm 18: $[\![x \geq min(x)]\!] \Rightarrow [\![b_{next} = 0]\!]$

All these explanations are computed eagerly and saved in an internal structure for later use during conflict analysis. The reason we compute them at the moment of propagation is to recover the exact literal responsible for assigning every b_{next}.

5.3 Learning in Disjunctive Scheduling

We investigate in this part the impact of clause learning for solving disjunctive scheduling problems. We introduce a novel learning mechanism tailored to this family of problems. Specifically, we use a property of these problems allowing to explain a conflict using a number of Boolean variables that is not function of the scheduling horizon. The novel conflict analysis procedure along with the alternative lazy generation mode that we proposed in Section 5.2 are experimentally tested on well known academic benchmarks. Both approaches give good experimental results and outperform the CP model in most cases. While the prior target of this study is the evaluation of the new learning mechanisms that we propose, numerous observations are made based on the experimental results. These observations include relations between the instance size, the heuristic choice, and the conflict analysis scheme. State-of-the-art lower bounds for a traditional benchmark are improved thanks to our approach.

Disjunctive scheduling refers to a large family of scheduling problems having in common the Unary Resource Constraint. That is, for each machine, no pair of tasks can overlap. For a long time, the focus in constraint programming was to design dedicated propagation algorithms for the Unary Resource Constraint. For instance, the Edge-Finding filtering [38, 99, 141] is inferring relationships of the form *« T must precede any task in Θ »* where: T is a task, and Θ is a set of tasks to be scheduled on the same machine of T. We refer the reader to [10] for a comprehensive introduction to filtering techniques used in scheduling in general.

We are interested in this section in the impact of clause learning rather than propagation. Our models use minimalist propagation mechanisms. Our approach is implemented on top of the so called `light model` used in [71, 69, 70]. We shall use the classical job shop problem (JSP) and open shop problem (OSP) as illustrations. The objective in both problems is to minimize the total scheduling duration (i.e., the makespan C_{max}). We start by describing the 'light' CP model for these problems.

5.3.1 Modeling

In the rest of this chapter, n and m denote two integers in $\mathbb{N}^*$. We consider the definition of a job as a set of tasks. Let $\mathcal{J} = \{J_i \mid 1 \leq i \leq n\}$ be the set of jobs, and $\mathcal{M} = \{M_k \mid 1 \leq k \leq m\}$ be the set of machines. Each job J_i is defined by m tasks $\{T_{ik} \mid 1 \leq k \leq m\}$ s.t. T_{ik} requires machine k. Conversely, each machine M_k is associated to n tasks $\{T_{ik} \mid 1 \leq i \leq n\}$. Each task T_{ik} is associated to a processing duration p_{ik} in which the machine M_k is allocated to job i. Let t_{ik} be the variable representing the starting time of task T_{ik}. For all $k \in [1, m]$, the `Unary Resource Constraint` for machine M_k can be

expressed as follows:

$$\forall i \in [1,n], \forall j \in [1,n] \text{ } s.t. \text{ } i < j \quad t_{ik} + p_{ik} \leq t_{jk} \vee t_{jk} + p_{jk} \leq t_{ik} \tag{5.1}$$

We use a simple decomposition into reified constraints with $O(n^2)$ Boolean variables δ_{kij} per machine M_k channeled to task variables as follows:

$\forall i \in [1,n], \forall j \in [1,n], i < j$

$$\delta_{kij} = \begin{cases} 0 & \Leftrightarrow \quad t_{ik} + p_{ik} \leq t_{jk} \\ 1 & \Leftrightarrow \quad t_{jk} + p_{jk} \leq t_{ik} \end{cases} \tag{5.2}$$

In the following, we shall refer to this channeling with the DISJUNCTIVE(b, x, y, d_x, d_y) constraint instantiated to (δ_{kij}, t_{ik}, t_{jk}, p_{ik}, p_{jk}).

The Job Shop Problem In addition to the DISJUNCTIVE constraints, this problem requires for each job a total order on its tasks. We therefore suppose that T_{iv_a} is the a^{th} task required by job J_i. Modeling the order of tasks for each Job is expressed by means of PRECEDENCE constraints. Let x, y be variables and d be an integer. The PRECEDENCE(x, y, d) constraint is defined as follows:

$$x + d \leq y \tag{5.3}$$

For each job i, we have the set of PRECEDENCE constraints:

$$\forall a \in [1, m-1] \quad \text{PRECEDENCE}(t_{iv_a}, t_{iv_{a+1}}, p_{iv_a}) \tag{5.4}$$

The JSP having the minimization of the makespan C_{max} as an objective can be defined as follows:

$$\text{minimize } C_{max} \text{ subject to}$$

$$\begin{array}{rl} \forall i \in [1,n] & t_{iv_m} + p_{iv_m} \leq C_{max} \\ \forall k \in [1,m], \forall i \in [1,n], \forall j \in [1,n], i < j & \text{DISJUNCTIVE}(\delta_{kij}, t_{ik}, t_{jk}, p_{ik}, p_{jk}) \\ \forall i \in [1,n], \forall a \in [1,m-1] & \text{PRECEDENCE}(t_{iv_a}, t_{iv_{a+1}}, p_{iv_a}) \end{array} \tag{5.5}$$

The Open Shop Problem The only difference compared to the JSP problem is that the order between tasks of the same job is part of the decision. In other words, two tasks of a same job cannot be executed at the same time but we are free to choose the processing order. A job can therefore be considered as a unary resource. Similarly to the disjunctions on machines, we introduce $O(m^2)$ Boolean variables ξ_{iab} for each job i and post the constraints $\text{DISJUNCTIVE}(\xi_{iab}, t_{ia}, t_{ib}, p_{ia}, p_{ib})$ for all $a < b \in [1, m]$. The OSP can therefore be defined as follows:

$$\text{minimize } C_{max} \text{ subject to}$$

$$\begin{aligned} &\forall i \in [1,n], \forall k \in [1,m] && t_{ik} + p_{ik} \leq C_{max} \\ &\forall k \in [1,m], \forall i \in [1,n], \forall j \in [1,n], i < j && \text{DISJUNCTIVE}(\delta_{kij}, t_{ik}, t_{jk}, p_{ik}, p_{jk}) \\ &\forall i \in [1,n], \forall a \in [1,m], \forall b \in [1,m], a < b && \text{DISJUNCTIVE}(\xi_{iab}, t_{ia}, t_{ib}, p_{ia}, p_{ib}) \end{aligned} \tag{5.6}$$

5.3.2 Search

Our search strategies are essentially based on those proposed in [71, 69].

5.3.2.1 The Global Search Scheme

Exploring the search space is performed in three steps. Firstly a greedy algorithm is used to compute an initial upper bound (u_{init}) for C_{max}. The initial lower bound (l_{init}) is initialized to be the largest sum of durations between all jobs/machines. Second, a dichotomic search is used to improve the initial upper/lower bounds for C_{max}. Each iteration is limited by a cutoff on CPU time and on the number of propagation calls. The initial dichotomy step starts with $[l_{init}, u_{init}]$ as a domain for C_{max}. In each dichotomy step i we try to solve the decision version of the problem (i.e., without an objective function) where the upper bound of C_{max} is equal to $(l_{i-1} + u_{i-1})/2$ s.t. the values l_{i-1} and u_{i-1} are the best bounds found after step $i - 1$. We update the bounds of C_{max} depending on the outcome of a dichotomic step i. If it is satisfiable then we store the value of C_{max} in the solution as u_i and change the upper bound of C_{max} accordingly. Otherwise, we set l_i to $(l_{i-1} + u_{i-1})/2$. However, observe that we change the lower bound of C_{max} only if the problem has been proven unsatisfiable at step i, but not if the limit has been reached. Finally, a branch and bound algorithm is launched with the best real lower/upper bound found (i.e., $[min(C_{max}), max(C_{max})]$).

5.3.2.2 Branching

It is very common in disjunctive scheduling to branch by fixing one of the possible precedences in the unary resource constraints. The authors of [71] proposed to branch on the Boolean variables in the DISJUNCTIVE constraints which simulates that behavior. Note that it is sufficient to have all these Boolean variables assigned to decide the problem. In fact, assigning all the tasks, along with the variable standing for makespan, to the minimum possible value in their domain returns a solution with the minimum possible value for C_{max} w.r.t. the assignment of the Boolean variables.

Variable Ordering The variable ordering heuristics are inspired from the conflict-driven domain/weighted-degree heuristic proposed in [30]. The idea is to assign first the variables involved in previous failures. The domain size $dom(t_{ik})$ of a task T_{ik} is equal to $max(t_{ik}) - min(t_{ik}) + 1$. The weight w_x of a variable x is equal to the number of times x is in the scope of the constraint triggering a failure. Every Boolean variable b in a DISJUNCTIVE(b, x, y, d_x, d_y) constraint can be evaluated using the following two heuristics:

1. $taskDom/bw$: $\frac{dom(x)+dom(y)}{w_b}$
2. $taskDom/tw$: $\frac{dom(x)+dom(y)}{w_x+w_y}$

In both heuristics, the final decision is randomly chosen between the two Boolean variables with minimum values.

We use slightly modified versions of the above heuristics in our hybrid models. First, following a remark in [70] stating that «the greater the minimum arity of constraints in a problem, the less discriminatory the weight-degree heuristic can be», we propose to update the variables weight in the conflicting clauses as follows. When a failure is triggered by a clause c, the weight of each variable in the clause is increased by $\frac{1}{|c|}$ instead of 1. Next, with $taskDom/tw$, instead on incrementing the weight of any Boolean variable b in c, we share this value between the two tasks in the DISJUNCTIVE constraint reified by b. This is proposed because the weight of the Boolean variables in these cases would not bring new information to $taskDom/tw$. Finally, if we use lazy generation, instead of updating the weight of the generated atoms $a \leftrightarrow [\![t_{ik} \leq v]\!]$, we consider increasing the weight of task T_{ik} (by $\frac{1}{|c|}$).

We shall also consider VSIDS as another variable ordering alternative in our hybrid models.

Value Ordering Similarly to the solution guided approach proposed in [12], we assign the chosen variable to the same value it has in the latest solution.

5.3.3 Explaining Constraints

Observe first that the constraints related to the makespan can be considered are Precedence constraints (i.e., of the form $x + d \leq y$). We therefore have two types of constraints to explain: Precedence, and Disjunctive. We give in the following how to generate explanations for these constraints. To make the notation lighter, we denote l_x (respectively u_x) the lower (respectively upper) bound in $\mathcal{D}(x)$.

5.3.3.1 Explaining Precedence(x, y, d)

To propagate Precedence, we need to update the upper bound of x and the lower bound of y. We give in Algorithm 19 a BC propagator for this constraint.

Algorithm 19: Precedence(x, y, d)

```
if min(x) + d > max(y) then
1   D ← ⊥ ;
else
    if max(x) > max(y) − d then
2       D(x) ← D(x) ∩ ]−∞, max(y) − d] ;
    if min(y) < min(x) + d then
3       D(y) ← D(y) ∩ [min(x) + d, +∞[ ;
return D ;
```

Explaining Failure The only way to have a failure in this constraint is when $max(x)$ is greater that $max(y) - d$ (Line 1 in Algorithm 19). The obvious explanation for this failure is:

$$[\![x \geq l_x]\!] \wedge [\![y \leq u_y]\!] \Rightarrow \bot$$

Explaining Pruning This propagator only tighten the upper bound of x and the lower bound of y. Let v be an integer. To explain the literal $[\![y \geq v]\!]$, it is clear that $[\![x \geq v - d]\!] \Rightarrow [\![y \geq v]\!]$ is a valid explanation. Similarly, if $[\![x \leq v]\!]$ is propagated by this constraint, then we use $[\![y \leq v + d]\!] \Rightarrow [\![x \leq v]\!]$ as an explanation for this propagation.

These explanations are computed in $O(1)$ regardless of the level/rank of the literals being explained. Furthermore, we do not need to keep track the exact bounds at the time it was changed through propagation. The backward explanation mode suits very well this constraint.

5.3.3.2 Explaining Disjunctive(b, x, y, d_x, d_y)

We start again by giving a full description of the filtering used for this constraint. We show a BC propagator in Algorithm 20.

Algorithm 20: Disjunctive(b, x, y, d_x, d_y)

```
if |D(b)| = 1 then
    if D(b) = {0} then
        return Precedence(x, y, d_x) ;
    else
        return Precedence(y, x, d_y) ;
else
    if l_x + d_x > u_y then
        D(b) ← {1} ;
        return Precedence(y, x, d_y) ;
    else
        if l_y + d_y > u_x then
            D(b) ← {0} ;
            return Precedence(x, y, d_x) ;
```

Algorithm 20 does not prune the domains of x nor y until $\mathcal{D}(b)$ becomes singleton. Furthermore, once $\mathcal{D}(b)$ is assigned, the constraint becomes a Precedence. Therefore, in order to explain Disjunctive, all the previous explanations are used along with the current state of b. That is, if we want to explain ω (ω is either a literal or a failure $\bot$) made by this propagator because of a call to Precedence, then it is sufficient to return $[\![b = v]\!] \wedge \Psi \Rightarrow \omega$ s.t. $\mathcal{D}(b) = \{v\}$ and Ψ is the explanation of ω based on the way we explain Precedence.

The only missing explanations to generate are the ones related to the assignments of b. We explain them using the following propagation rules:

$$[\![x \geq l_x]\!] \wedge [\![y \leq u_y]\!] \rightarrow [\![b = 1]\!]$$
$$[\![y \geq l_y]\!] \wedge [\![x \leq u_x]\!] \rightarrow [\![b = 0]\!]$$

The values l_x, u_x, l_y, and u_y must be those used at the time of propagation. We store these values once the propagator assigns b.

5.3.4 DISJUNCTIVE-Based Learning

We introduce a novel learning scheme as an alternative to the lazy generation mode. The main advantage offered by this novel learning mechanism is that the final nogoods do not contain any domain related atom.

Recall that our search strategies branch only on Boolean variables of the DISJUNCTIVE constraints. It follows that any bound literal (i.e., of the form $[\![x \leq v]\!]$ and $[\![x \geq v]\!]$) does not correspond to a decision. Therefore such literals are either propagated, hence have a non-null explanation, or have a level equal to the search root. Our new learning method exploits precisely this property. Instead of generating bound atoms before learning a new clause, we propose to start a second phase of conflict analysis.

The first step in the new DISJUNCTIVE-based learning is to perform conflict analysis as usual to compute the 1-UIP nogood Ψ. Next, we make sure that the latest literal in Ψ is not a bound literal. Otherwise, we keep explaining the latest literal in Ψ until having such UIP. We know that this procedure terminates because the worst case would reach the last decision which corresponds to a UIP that is not a bound literal. Let Ψ^* be the resulting nogood. Observe that the backjump level in Ψ^* might be different from the one given by the 1-UIP nogood.

Consider now $\mathfrak{I} = \{l_1, \ldots, l_n\}$ to be the set of bound literals in Ψ^* before generating atoms. Instead of performing lazy generation, we call the procedure «Substitute($\mathfrak{I}, \Psi^*$)» (algorithm 21) as a second phase of conflict analysis. This procedure keeps replacing any bound literal with its explanation until having a nogood composed by only literals related to some Boolean variables of the DISJUNCTIVE constraints. In Algorithm 21, we use:

- *visited*: to represent a set containing bound literals already explained
- ω: to represent the explanation of the current bound literal to resolve
- φ: to represent the set of bound literals in ω.

Starting from the first line in Algorithm 21, we split the nogood under construction in two parts: $\mathfrak{I}$ to contain bound literals; and Ψ for the rest of literals (i.e., literals associated to Boolean variables coming from the DISJUNCTIVE constraints). The idea of Algorithm 21 is to explain every bound literal in $\mathfrak{I}$ until no such literal exists. This is exactly what happens at each iteration of the main loop. $\mathfrak{I}$ is updated to contain new bound literals from φ at Line 2. The rest of literals in the current explanation ω goes in Ψ at Line 3 and Line 4.

The final nogood Ψ contains only some Boolean variables from the DISJUNCTIVE constraints without any bound literal. It should be noted that the backjump level remains

the same as in Ψ^* since resolving a literal l replaces it with a set of literals assigned at least at the same level of l.

Algorithm 21: Substitute($\mathfrak{I}, \Psi^*$)

```
1  Ψ ← Ψ* ∖ ℑ ;
   visited ← ∅ ;
   while |ℑ| > 0 do
       l ← choose l ∈ ℑ ;
       visited ← visited ∪ {l} ;
       f ← reason(l) ;
       ω ← explain(f, l) ;
       φ = {q | q ∈ ω ∧ q is a bound litteral} ;
2      ℑ ← ℑ ∪ {q | q ∈ φ ∧ level(q) > 0 ∧ q ∉ visited} ;
3      ω ← ω ∖ φ ;
4      Ψ ← Ψ ∪ {q | q ∈ ω ∧ level(q) > 0} ;
   return Ψ ;
```

The advantage of this approach is that the tasks' domains do not matter any more in size. The SAT engine focuses on learning clauses with only Boolean variables coming from the DISJUNCTIVE constraints. Note, however, that in this case conflict analysis is likely to take more time to finish compared to the lazy generation mode since there are more literals to explain.

5.3.5 Experiments

We implemented the learning mechanisms we propose within Mistral-2.0. This solver supports backward explanations and semantic reduction. The source code is available online via `https://github.com/siala/Hybrid-Mistral` and the tests can be reproduced following the guidance in Appendix A. All the experiments were performed on Intel i7-4770 processors running on Ubuntu 12.04. We compare the previous CP models against our new learning methods. The two heuristics *taskDom/bw* and *taskDom/tw* are tested in both CP and hybrid solvers. $VSIDS$ is also used as another hybrid model. We use a geometric restart with a base of 256 failures and a factor of 1.3. The total time limit is fixed to 3600s for all the experiments. Each dichotomy step is limited to a cutoff of 300s and $4 * 10^6$ propagation call. We ran 10 randomized runs with different seeds for each instance and configuration.

We use a clause reduction strategy based on the Size-Bounded Randomized (SBR) method [75]. Every f failures, we check whether the size of the clause database reached a given parameter ω. If so, a parametrized deletion procedure $reduceClauses(f, \omega, \alpha, k, \epsilon, \rho)$ is performed as follows. A clause c is considered 'locked' if there exists a literal p such that c is the reason for propagating p. All locked clauses

are not removed. The last α non-locked clauses are also kept. Afterwards, the clauses with size less than a parameter k are not deleted. The other clauses are deleted with a probability ρ. If the resulting number of clauses still greater than ω, we call again $reduceClauses$, however, after decreasing k by ϵ. We iterate this process until the clause database is of size smaller than w. The default values used for all the experiments for $< f, \omega, \alpha, k, \epsilon, \rho >$ are $< 5000, 75000, 50000, 12, 8, 90\% >$.

We shall evaluate experimentally the following models:

- Mistral(θ): The pure CP model using θ as a heuristic. The latter is denoted by
  - *bool* if we use $taskDom/bw$
  - $task$ if we use $taskDom/tw$
- Hybrid(θ, σ): The hybrid model where:
  - θ is the heuristic and is denoted by:
    - $vsids$ if we use VSIDS
    - *bool* if we use $taskDom/bw$
    - $task$ if we use $taskDom/tw$
  - σ indicates the learning method with '$disj$' in the case of using the DISJUNCTIVE-based learning and '$lazy$' with the lazy generation approach with DOMAINFAITHFULNESS.

We use a limit of $2.5 * 10^5$ generated atoms with the models Hybrid($\theta, lazy$). Once this limit is reached, we forget all clauses, delete the generated atoms, and restart.

We use the following format for all tables. Each instance results (i.e., using different seeds) is depicted in one line. Each model is associated to a column. We report for each model and instance: the average CPU time (T); the percentage of instances found optimal (%O); the minimum (min) and average (avg) upper bound (UB) across the different seeds. We shall denote in **bold** the minimum makespan found for each instance (can occur in different models). Furthermore, we add a line 'average' at the bottom of each table to show the average CPU time T and the average percentage of optimality %O for each model. The last line contains the average `PRD` (percentage relative deviation) of each model. The PRD of a model m for an instance C is computed with the formula: $100 * \frac{C_m - C_{best}}{C_{best}}$, where C_m is the minimum makespan found by model m for this instance (among the several randomized runs); and C_{best} is the minimum makespan found by all models for the instance C. The average PRD can be considered as an 'efficiency' measure for the models. The bigger this value, the less efficient a model is. The minimum possible value of a PRD is 0 and means that the model returns always the best makespan.

5.3.5.1 JSP Results

We use two well studied benchmarks for the job shop problem: Lawrence [86] and Taillard [134]. The former is much easier than the latter. We observed in these instances that $taskDom/tw$ performs slightly, but constantly, better than $taskDom/bw$. Therefore, the results that we report in this paragraph concern the models: Mistral($task$), Hybrid($vsids, disj$), Hybrid($vsids, lazy$), Hybrid($task, disj$), and Hybrid($task, disj$).

Lawrence Instances The detailed results of Lawrence instances are shown in Tables 5.2 and 5.3. The model Hybrid($vsids, disj$) has the best PRD with a value of 0.01 and the greatest percentage of optimal solutions (92%). The only case where the CP model returns the best makespan was with instance la27, however, without obtaining the best average. As a comparison between the different hybrid models, we observe that the DISJUNCTIVE-based learning outperforms the lazy approach regardless of the branching strategy. We are not able, however, to argue on a best heuristic here since VSIDS performs better with the DISJUNCTIVE-based learning whereas $taskDom/tw$ is the best choice of branching with lazy generation.

Taillard Instances These instances are much harder than Lawrence benchmark since a large number of them are still open in the literature and only 10 out of 70 instances are proved optimal in our experiments. We start by giving a global view analysis before empirically evaluating subsets of these instances.

The detailed results are given in Tables 5.4, 5.5, and 5.6. According to the global average PRD (shown at the end of Table 5.6), the best models for these instances are those using $taskDom/tw$. The CP model is completely outperformed by hybrid models with a PRD equal to 1.5474 compared to an average of 0.9487 with the models Hybrid($vsids, \theta$) and an average of 0.30185 with the models Hybrid($task, \theta$). Clearly, the branching choice is the most important criteria for hybrid models. The choice of the conflict analysis scheme does not seem to impact much the global behavior, although lazy generation performs slightly better.

These results do not confirm our earlier claim with Lawrence instances stating that Hybrid($vsids, disj$) is the best learning configuration. We therefore propose to classify the results according to the instance size.

Taillard Statistics In table 5.7, each line depicts several statistics for a given set of instances having the same number of disjunctions. We report for each model: the speed of exploration in terms of nodes explored by second (Nodes/s); the average size of learnt clauses (Size); and a performance metric M equal to the pair <%O,T> (%O is

TABLE 5.2: Job Shop: Lawrence (la-01-20) detailed results

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, disj$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
la01	0	100	**666**	666	0	100	**666**	666	0.01	100	**666**	666	0	100	**666**	666	0	100	**666**	666
la02	0.21	100	**655**	655	0.21	100	**655**	655	0.46	100	**655**	655	0.19	100	**655**	655	0.26	100	**655**	655
la03	0.06	100	**597**	597	0.07	100	**597**	597	0.15	100	**597**	597	0.08	100	**597**	597	0.11	100	**597**	597
la04	0.05	100	**590**	590	0.07	100	**590**	590	0.11	100	**590**	590	0.07	100	**590**	590	0.09	100	**590**	590
la05	0	100	**593**	593	0	100	**593**	593	0	100	**593**	593	0	100	**593**	593	0	100	**593**	593
la06	0.01	100	**926**	926	0.05	100	**926**	926	0.13	100	**926**	926	0.02	100	**926**	926	0.03	100	**926**	926
la07	3600	0	**890**	890	3600	0	**890**	890	3600	0	**890**	890	3600	0	**890**	890	3600	0	**890**	890
la08	0.03	100	**863**	863	0.07	100	**863**	863	0.09	100	**863**	863	0.03	100	**863**	863	0.04	100	**863**	863
la09	0	100	**951**	951	0.01	100	**951**	951	0.06	100	**951**	951	0.01	100	**951**	951	0.03	100	**951**	951
la10	0	100	**958**	958	0	100	**958**	958	0	100	**958**	958	0	100	**958**	958	0	100	**958**	958
la11	0.04	100	**1222**	1222	0.71	100	**1222**	1222	0.70	100	**1222**	1222	0.05	100	**1222**	1222	0.05	100	**1222**	1222
la12	0.11	100	**1039**	1039	0.22	100	**1039**	1039	0.36	100	**1039**	1039	0.07	100	**1039**	1039	0.09	100	**1039**	1039
la13	0.03	100	**1150**	1150	0.18	100	**1150**	1150	0.13	100	**1150**	1150	0.07	100	**1150**	1150	0.05	100	**1150**	1150
la14	0	100	**1292**	1292	0	100	**1292**	1292	0	100	**1292**	1292	0	100	**1292**	1292	0	100	**1292**	1292
la15	0.27	100	**1207**	1207	3.29	100	**1207**	1207	49.27	100	**1207**	1207	0.54	100	**1207**	1207	0.33	100	**1207**	1207
la16	0.31	100	**945**	945	0.27	100	**945**	945	0.53	100	**945**	945	0.26	100	**945**	945	0.48	100	**945**	945
la17	0.08	100	**784**	784	0.07	100	**784**	784	0.11	100	**784**	784	0.11	100	**784**	784	0.16	100	**784**	784
la18	0.03	100	**848**	848	0.04	100	**848**	848	0.06	100	**848**	848	0.05	100	**848**	848	0.06	100	**848**	848
la19	0.37	100	**842**	842	0.27	100	**842**	842	0.57	100	**842**	842	0.40	100	**842**	842	0.62	100	**842**	842
la20	0.11	100	**902**	902	0.06	100	**902**	902	0.12	100	**902**	902	0.10	100	**902**	902	0.15	100	**902**	902

TABLE 5.3: Job Shop: Lawrence (la-21-40) detailed results

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, disj$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
la21	3501.39	10	**1046**	1046.2	1319.54	100	**1046**	1046	3600	0	**1046**	1047	2948.05	40	**1046**	1046.6	3600	0	**1046**	1048.5
la22	78.96	100	**927**	927	78.92	100	**927**	927	335.46	100	**927**	927	64.92	100	**927**	927	119.64	100	**927**	927
la23	0.31	100	**1032**	1032	0.43	100	**1032**	1032	1.29	100	**1032**	1032	0.27	100	**1032**	1032	0.39	100	**1032**	1032
la24	190.55	100	**935**	935	96.02	100	**935**	935	607.17	100	**935**	935	146.52	100	**935**	935	458.56	100	**935**	935
la25	171.19	100	**977**	977	93.56	100	**977**	977	476.30	100	**977**	977	114.64	100	**977**	977	452.61	100	**977**	977
la26	1742.60	60	**1218**	1219.9	232.44	100	**1218**	1218	1130.81	100	**1218**	1218	62.97	100	**1218**	1218	992.87	90	**1218**	1218.1
la27	3600	0	**1241**	1259.9	3600	0	1246	1255.3	3600	0	1243	1270.6	3600	0	1243	1252.7	3600	0	1251	1262.2
la28	1487.91	80	**1216**	1216.7	898.47	90	**1216**	1216.5	2723.81	40	**1216**	1218.6	1427.41	80	**1216**	1216.9	1245.71	70	**1216**	1216.7
la29	3600	0	1183	1196	3600	0	**1168**	1178.6	3600	0	1189	1201.6	3600	0	1171	1185.8	3600	0	1176	1194.3
la30	2.01	100	**1355**	1355	4.95	100	**1355**	1355	9.73	100	**1355**	1355	1.78	100	**1355**	1355	2.14	100	**1355**	1355
la31	5.74	100	**1784**	1784	59.19	100	**1784**	1784	402.94	90	**1784**	1786.9	3	100	**1784**	1784	3.53	100	**1784**	1784
la32	2.54	100	**1850**	1850	4.31	100	**1850**	1850	5.09	100	**1850**	1850	2.49	100	**1850**	1850	2.65	100	**1850**	1850
la33	3.81	100	**1719**	1719	67.31	100	**1719**	1719	10.4	100	**1719**	1719	5.11	100	**1719**	1719	6.87	100	**1719**	1719
la34	72.37	100	**1721**	1721	1332.11	90	**1721**	1721.3	557.86	90	**1721**	1721.3	14.87	100	**1721**	1721	9.36	100	**1721**	1721
la35	4.17	100	**1888**	1888	17.91	100	**1888**	1888	44.78	100	**1888**	1888	3.84	100	**1888**	1888	3.61	100	**1888**	1888
la36	77.76	100	**1268**	1268	43.38	100	**1268**	1268	119.79	100	**1268**	1268	43.72	100	**1268**	1268	90.19	100	**1268**	1268
la37	246.81	100	**1397**	1397	264.10	100	**1397**	1397	634.73	100	**1397**	1397	310.01	100	**1397**	1397	528.60	100	**1397**	1397
la38	233.55	100	**1196**	1196	221.76	100	**1196**	1196	1246.13	100	**1196**	1196	204.52	100	**1196**	1196	730.26	100	**1196**	1196
la39	26.53	100	**1233**	1233	23.27	100	**1233**	1233	46.01	100	**1233**	1233	15.10	100	**1233**	1233	23.30	100	**1233**	1233
la40	229.06	100	**1222**	1222	284.70	100	**1222**	1222	1295.48	100	**1222**	1222	250.74	100	**1222**	1222	497.41	100	**1222**	1222
avg	471.97	88.75			396.20	92			602.51	88			410.55	90.50			489.25	89		
PRD				0.0321				0.0100				0.0489				0.0104				0.0372

TABLE 5.4: Job Shop: Taillard (tai01 – tai25) detailed results

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
tai01	18.57	100	**1231**	1231	9.55	100	**1231**	1231	34.64	100	**1231**	1231	8.31	100	**1231**	1231	14.96	100	**1231**	1231
tai02	136.43	100	**1244**	1244	121.97	100	**1244**	1244	533.13	100	**1244**	1244	116.84	100	**1244**	1244	288.61	100	**1244**	1244
tai03	116.05	100	**1218**	1218	59.30	100	**1218**	1218	304.75	100	**1218**	1218	115.11	100	**1218**	1218	412.97	100	**1218**	1218
tai04	62.39	100	**1175**	1175	35.23	100	**1175**	1175	179.47	100	**1175**	1175	34.89	100	**1175**	1175	84.88	100	**1175**	1175
tai05	1212.27	100	**1224**	1224	480.54	100	**1224**	1224	2573.16	70	**1224**	1224	1071.65	100	**1224**	1224	3169.86	50	**1224**	1225.70
tai06	3600	0	**1238**	1243.3	3600	0	**1238**	1240.2	3600	0	1239	1244.4	3600	0	**1238**	1242.1	3600	0	1239	1246.70
tai07	221.34	100	**1227**	1227	201.76	100	**1227**	1227	1112.44	100	**1227**	1227	226.89	100	**1227**	1227	1066.6	100	**1227**	1227
tai08	141.51	100	**1217**	1217	105.44	100	**1217**	1217	665.31	100	**1217**	1217	130.74	100	**1217**	1217	609.06	100	**1217**	1217
tai09	491.77	100	**1274**	1274	117.27	100	**1274**	1274	744.97	100	**1274**	1274	339.96	100	**1274**	1274	1450.87	100	**1274**	1274
tai10	161.88	100	**1241**	1241	46.87	100	**1241**	1241	243.83	100	**1241**	1241	104.35	100	**1241**	1241	457.06	100	**1241**	1241
tai11	3600	0	1396	1405.3	3600	0	**1374**	1385	3600	0	1375	1391.9	3600	0	1381	1397.8	3600	0	1386	1402.70
tai12	3600	0	1393	1403.8	3600	0	**1376**	1387.4	3600	0	1386	1400.7	3600	0	1388	1396.8	3600	0	1382	1393.30
tai13	3600	0	1350	1361.4	3600	0	**1342**	1353.2	3600	0	1358	1370.5	3600	0	1343	1353.5	3600	0	1354	1361.30
tai14	3600	0	**1345**	1351	3600	0	**1345**	1350.4	3600	0	1349	1355.9	3600	0	**1345**	1348.9	3600	0	**1345**	1350.10
tai15	3600	0	1375	1389.8	3600	0	1357	1371.1	3600	0	1363	1382.9	3600	0	**1354**	1374.4	3600	0	1360	1381.30
tai16	3600	0	1388	1404.9	3600	0	**1375**	1384.7	3600	0	1376	1398.6	3600	0	1379	1387.5	3600	0	1381	1391.50
tai17	3600	0	1476	1488.7	3600	0	1478	1485.7	3600	0	1477	1489.8	3600	0	1476	1488.9	3600	0	**1473**	1483.50
tai18	3600	0	1438	1455	3600	0	1426	1438.2	3600	0	**1425**	1455.1	3600	0	1427	1444.8	3600	0	1428	1447.60
tai19	3600	0	1367	1388.3	3600	0	1366	1377	3600	0	1371	1382.3	3600	0	1362	1378.1	3600	0	**1351**	1378.80
tai20	3600	0	1363	1380	3600	0	1361	1368.1	3600	0	1367	1376.6	3600	0	**1360**	1373.5	3600	0	1363	1375.90
tai21	3600	0	1662	1680.2	3600	0	1649	1662.9	3600	0	1650	1670.7	3600	0	1651	1670.2	3600	0	**1643**	1669.30
tai22	3600	0	1637	1652.1	3600	0	1624	1646	3600	0	1630	1651.2	3600	0	**1621**	1636.7	3600	0	1623	1645.30
tai23	3600	0	**1562**	1591.7	3600	0	1568	1578.4	3600	0	1576	1591.2	3600	0	1571	1585.7	3600	0	1567	1589.50
tai24	3600	0	**1645**	1655.1	3600	0	**1645**	1655.1	3600	0	1652	1670.8	3600	0	1653	1659	3600	0	1646	1659.10
tai25	3600	0	1627	1644.1	3600	0	**1601**	1615.7	3600	0	1618	1636.9	3600	0	**1601**	1628.3	3600	0	1615	1631.70

TABLE 5.5: Job Shop: Taillard (tai26 – tai50) detailed results

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
tai26	3600	0	1689	1696.1	3600	0	1679	1685.8	3600	0	1676	1693	3600	0	**1672**	1684.5	3600	0	1674	1689.20
tai27	3600	0	1701	1714.1	3600	0	1697	1704	3600	0	1701	1727	3600	0	**1693**	1704.8	3600	0	1694	1718.40
tai28	3600	0	1623	1633.9	3600	0	1616	1621.9	3600	0	**1603**	1622.9	3600	0	1617	1621.9	3600	0	**1603**	1621.80
tai29	3600	0	1642	1650.2	3600	0	1635	1639.2	3600	0	1635	1651.1	3600	0	**1630**	1640.3	3600	0	**1630**	1647.30
tai30	3600	0	1608	1633.8	3600	0	1608	1617.4	3600	0	1613	1625.7	3600	0	1608	1621.4	3600	0	**1607**	1627
tai31	3600	0	1853	1885.9	3600	0	1823	1853.9	3600	0	**1808**	1860.8	3600	0	1846	1863.6	3600	0	1825	1854.60
tai32	3600	0	1901	1931.3	3600	0	1876	1895.4	3600	0	1891	1906.3	3600	0	**1867**	1907.1	3600	0	1873	1900
tai33	3600	0	1914	1949	3600	0	1897	1921.1	3600	0	1876	1929.7	3600	0	**1869**	1916.8	3600	0	1897	1920.50
tai34	3600	0	1932	1965.1	3600	0	1927	1942.5	3600	0	**1914**	1947.9	3600	0	1916	1930.9	3600	0	1923	1937.30
tai35	3600	0	**2007**	2007.3	3600	0	**2007**	2016.3	3600	0	**2007**	2018.8	3600	0	**2007**	2007.8	3600	0	**2007**	2007
tai36	3600	0	1904	1939.9	3600	0	1886	1906	3600	0	**1878**	1904.5	3600	0	1897	1910.9	3600	0	1888	1909.30
tai37	3600	0	1861	1891.7	3600	0	1848	1870	3600	0	**1844**	1876.4	3600	0	1848	1871.8	3600	0	**1844**	1867.30
tai38	3600	0	1783	1803.7	3600	0	1764	1777.6	3600	0	**1752**	1779.4	3600	0	**1752**	1778	3600	0	1755	1780.70
tai39	3600	0	1854	1877.4	3600	0	1831	1853.3	3600	0	1832	1857.8	3600	0	1827	1848.4	3600	0	**1807**	1839.90
tai40	3600	0	1814	1840.6	3600	0	1780	1802.7	3600	0	**1763**	1808.4	3600	0	1789	1805.5	3600	0	1766	1804.70
tai41	3600	0	2151	2182.4	3600	0	2123	2145.1	3600	0	2114	2154.1	3600	0	**2110**	2134.4	3600	0	2115	2133.20
tai42	3600	0	2058	2082.7	3600	0	**2006**	2038.4	3600	0	2012	2045.2	3600	0	2010	2033.8	3600	0	2024	2039.60
tai43	3600	0	1996	2022.7	3600	0	1953	1975.8	3600	0	**1936**	1973.6	3600	0	1963	1982.7	3600	0	1961	1983.20
tai44	3600	0	2098	2140.4	3600	0	**2068**	2100.8	3600	0	2075	2115.9	3600	0	2085	2107	3600	0	2086	2107.80
tai45	3600	0	2089	2116.8	3600	0	**2058**	2081.6	3600	0	2075	2098.4	3600	0	**2058**	2088.9	3600	0	2063	2092.60
tai46	3600	0	2138	2157.9	3600	0	2108	2126.8	3600	0	**2095**	2127.3	3600	0	2103	2122.2	3600	0	2104	2123.20
tai47	3600	0	2037	2057.2	3600	0	1998	2017.6	3600	0	1991	2008.3	3600	0	**1988**	2017.4	3600	0	2015	2025.50
tai48	3600	0	2086	2109.2	3600	0	2055	2074	3600	0	**2028**	2067	3600	0	2048	2072.3	3600	0	2043	2059.30
tai49	3600	0	2099	2126.9	3600	0	**2061**	2082.3	3600	0	2073	2104.3	3600	0	2070	2096.4	3600	0	2063	2085.60
tai50	3600	0	2048	2082.4	3600	0	2018	2045.4	3600	0	2014	2047.1	3600	0	2025	2045.1	3600	0	**2001**	2049.50

TABLE 5.6: Job Shop: Taillard (tai51 – tai70) detailed results

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
tai51	3600	0	2834	2870	3600	0	2876	2889.5	3600	0	2868	2903.3	3600	0	2786	2815.1	3600	0	**2775**	2819.10
tai52	3600	0	2843	2869.4	3600	0	2845	2873.9	3600	0	2855	2903.3	3600	0	2785	2811.8	3600	0	**2769**	2798.70
tai53	3600	0	2777	2812.5	3600	0	2784	2801.6	3600	0	2768	2808.2	3600	0	**2729**	2756.1	3600	0	2740	2763.90
tai54	3600	0	2847	2865.5	3600	0	2841	2878.7	3600	0	2871	2887	3600	0	**2839**	2840.6	3600	0	**2839**	2842.60
tai55	3600	0	2802	2851.3	3600	0	2799	2845.2	3600	0	2813	2878.3	3600	0	**2734**	2788.2	3600	0	2763	2782
tai56	3600	0	2870	2892.9	3600	0	2867	2894.3	3600	0	2885	2939.6	3600	0	2839	2854.4	3600	0	**2829**	2844.80
tai57	3600	0	3002	3025.3	3600	0	3015	3047.5	3600	0	3044	3081.6	3318.84	10	**2943**	2973	3319.22	10	**2943**	2974.50
tai58	3600	0	2964	2996	3600	0	2961	2988.2	3600	0	2969	3040.7	3600	0	**2888**	2923	3600	0	2901	2922.30
tai59	3600	0	2788	2821.2	3600	0	2800	2837.3	3600	0	2835	2872.8	3600	0	2731	2758.1	3600	0	**2723**	2765.40
tai60	3600	0	2850	2867.5	3600	0	2846	2874.1	3600	0	2871	2903.6	3600	0	**2767**	2805.8	3600	0	2781	2807.80
tai61	3600	0	3076	3107.3	3600	0	3047	3081.9	3600	0	3044	3081.8	3600	0	**2985**	3020.3	3600	0	2988	3015.80
tai62	3600	0	3157	3185.9	3600	0	3126	3157.7	3600	0	3115	3183.6	3600	0	**3045**	3110.8	3600	0	3053	3110.80
tai63	3600	0	2896	2960.1	3600	0	2934	2954	3600	0	**2849**	2929.2	3600	0	2858	2891.6	3600	0	2861	2895.40
tai64	3600	0	2864	2895.5	3600	0	2885	2899.5	3600	0	2831	2899.6	3600	0	2811	2837.8	3600	0	**2794**	2842.40
tai65	3600	0	2948	2972.3	3600	0	2949	2961.9	3600	0	2878	2931.9	3600	0	2870	2895.7	3600	0	**2849**	2883.40
tai66	3600	0	3044	3073.5	3600	0	3045	3077	3600	0	3018	3077.3	3600	0	2956	2990.2	3600	0	**2915**	2983.60
tai67	3600	0	3032	3051.1	3600	0	3004	3019.5	3600	0	2950	2998.7	3600	0	2923	2968.1	3600	0	**2895**	2962.80
tai68	3600	0	2962	2988.1	3600	0	2935	2978.1	3600	0	2919	2977.6	3600	0	2881	2913.4	3600	0	**2869**	2906
tai69	3600	0	3178	3223.8	3600	0	3183	3222.8	3600	0	3176	3253.3	3600	0	3112	3154.3	3600	0	**3101**	3140.20
tai70	3600	0	3217	3254.4	3600	0	3203	3255.8	3600	0	3204	3251.9	3600	0	**3119**	3160.2	3600	0	3131	3171
avg	3173.74	12.85			3153.97	12.85			3228.45	12.42			3163.82	13			3241.06	12.28		
PRD				1.5474				0.9955				0.9019				0.3322				0.2715

the average optimality percentage and T is the average CPU time) for the set tai-01-10 and to the average PRD for the rest of sets. The choice of M is based on the fact that almost all instances have been proven optimal in the set tai-01-10 whereas the others are much harder and are not proved optimal (except one). We show the best values of M in **bold** values. We indicate also the number of disjunctions per set of instances in a separate column (Disj).

There are a number of clear observations from Table 5.7. First, as expected, the CP model is less efficient in general than any hybrid model for the instances tai11, ..., tai70. Second, the average size of the learnt clauses is always shorter with VSIDS than $taskDom/tw$. Take for example the set of instances tai11-20. The model Hybrid($vsids, disj$) learns clauses with size 31 (in average) whereas Hybrid($task, disj$) learns clauses with size 41. Third, according to the number of nodes explored by second, the CP model is faster than any hybrid model in general. As an illustration, with instances tai11-20, the speed of exploration of Mistral($task$) is 6509 Nodes/s while the fastest hybrid model Hybrid($vsids, disj$) explores 3970 Nodes/s. This behavior is expected because of the amount of time to propagate clauses and to learn from conflict.

Next, we observe that lazy generation slows down considerably the exploration speed compared to DISJUNCTIVE-based learning. For instance, with tai11-20, Hybrid($vsids, disj$) explores 3970 Nodes/s whereas Hybrid($vsids, lazy$) explores 520 Nodes/s. Furthermore the exploration speed seems to be constant on hard sets (tai-11 ... tai-70) irrespectively of the instance size. Indeed, it ranges from 413 to 698 Nodes/s. We believe that this behavior is due to the additional amount of time needed to propagate DOMAINFAITHFULNESS constraints compared to DISJUNCTIVE-based learning.

Finally, this table shows that $taskDom/tw$ is always slower than VSIDS with DISJUNCTIVE-based learning. Take again the set of instances tai-11-20, we move from 3970 Nodes/s with Hybrid($vsids, disj$) to 2715 Nodes/s with Hybrid($task, disj$).

Now regarding the overall efficiency, we can see that Hybrid($vsids, disj$) seems to be the best choice with small instances and Hybrid($task, lazy$) is by far the best choice with large instances. Moreover, $taskDom/tw$ is in general more efficient than VSIDS when the size of the instance grows. Finally, the DISJUNCTIVE-based learning performs much better than the $lazy$ mode with small/medium-sized instances and vice versa.

Lawrence Statistics In order to confirm our latest claims, we show the same statistics described above with Lawrence instances. We propose to give these statistics for the hardest instances in this set. An instance is considered «hard» if at least one model fails to prove its optimality at least once (i.e., using any seed). The hardest instances in this set are divided in two sets:

TABLE 5.7: Job Shop: Taillard statistics

Instances		Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, lazy$)			
	Disj	M		Nodes/S	Size	M		Nodes/S	Size	M		Nodes/S	Size	M		Nodes/S	Size	M		Nodes/S	Size
		%O	T			%O	T			%O	T			%O	T			%O	T		
tai 01-10	1575	90	616	8871	0	**90**	**477**	6814	18	87	999	1213	25	90	574	4869	21	85	1115	1261	34
		PRD				PRD				PRD				PRD				PRD			
tai 11-20	2850	3.2381		6509	0	3.0350		3970	31	1.8937		520	43	0.4808		2715	41	**0.1169**		539	66
tai 21-30	3800	0.7302		3935	0	0.2769		2424	33	0.4756		413	46	0.2485		1752	45	**0.1557**		437	73
tai 31-40	6525	1.7227		4503	0	0.7109		2598	51	**0.3043**		555	65	0.6016		1517	76	0.4103		566	111
tai 41-50	8700	2.2161		2570	0	0.4798		1530	70	**0.3036**		413	86	0.5420		994	97	0.6163		443	140
tai 51-60	18375	2.0798		1952	0	2.2847		2602	57	2.7990		562	44	**0.1621**		1131	91	0.2419		698	89
tai 61-70	24500	3.2381		1349	0	3.0350		2183	64	1.8937		522	50	0.4808		920	121	**0.1169**		584	123

1. *Open*: the set of instances for which all models fail to prove optimality. This set contains the instances la07, la27, and la29
2. *Opt*: the rest of hard instances. This set contains the instances la21, la26, la28, la31, and la34.

It should be noted that the number of disjunctions in these sets ranges from 525 to 4350. We can therefore consider then as small and medium-sized instances (w.r.t. Taillard instances). Table 5.8 gives the statistics for each set of instances in a separate line.

TABLE 5.8: Job Shop: Lawrence Statistics

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, lazy$)			
	%O	T	Nodes/S	Size	%O	T	Nodes/S	Size	%O	T	Nodes/S	Size	%O	T	Nodes/S	Size	%O	T	Nodes/S	Size
Opt	70	1362	11520	0	**96**	**768**	8507	26	64	1683	1746	31	84	891	6745	35	72	1170	3380	40
	PRD		Nodes/S	Size	PRD		Nodes/S	Size	PRD		Nodes/S	Size	PRD		Nodes/S	Size	PRD		Nodes/S	Size
Open	0.4280		18581	0	**0.1343**		10159	23	0.6530		1000	31	0.1393		6782	31	0.4969		1322	49

Table 5.8 shows clearly that Hybrid($vsids, disj$) outperforms the other models on these instances. This model proves 96% of the instances in *Opt* to optimality and has a PRD of 0.1343 on the set of instances *Open*. Overall, the statistics presented in this table supports our previous observations with Taillard instances such as the speed of exploration, the average size of learnt clauses, and more importantly the outstanding performances of DISJUNCTIVE-based learning compared to lazy generation with small/medium-sized instances.

Improving the Lower Bounds for Taillard Open Instances Many of the Taillard instances are still open in the literature. Our results do not improve any upper bound for these instances, but what about the lower bound? Recall that the way we perform dichotomy steps is focused only on improving the current upper bound. Indeed, if step i ends without finding a solution nor proving unsatisfiability, then we set l_i to $(l_{i-1} + u_{i-1})/2$. We propose to alter this particular behavior so that the purpose becomes

finding better lower bounds. This is simply done by starting the next iteration after setting u_i (instead of l_i) to $(l_{i-1} + u_{i-1})/2$.

We ran again the tests with the new dichotomy strategy for all open Taillard instances. We change the dichotomy breaking conditions to be only a 1400s time limit. All other settings are the same.

The new results are presented in Table 5.9. For each model and instance, we report the maximum (max) and average (avg) lower bound found for the 10 randomized runs. The best bound found by our models is shown in **bold** fonts for each instance. Moreover, the last column stands for the best known lower bound for each instance [1][3]

TABLE 5.9: Lower bound experiments for open Taillard instances

Instance	Mistral($task$) Lower bound		Hybrid($vsids, disj$) Lower bound		Hybrid($vsids, lazy$) Lower bound		Hybrid($task, disj$) Lower bound		Hybrid($task, lazy$) Lower bound		Best known
	max	avg	max	avg	max	avg	max	avg	max	avg	
tai11	1273	1266.90	**1294**	1287.70	1273	1266.90	1281	1271	1273	1269.70	1323
tai12	1297	1271.70	**1300**	1296.80	1275	1274	1298	1270.50	1276	1267.10	1351
tai13	1278	1268.50	**1305**	1296.40	1282	1268.50	1291	1284	1281	1268.70	1282
tai15	1283	1267	**1288**	1281	1270	1262.10	**1288**	1277.60	1284	1267.20	1304
tai16	1276	1267.40	**1293**	1288.40	1280	1275	1276	1273.20	1274	1258.30	1304
tai18	1303	1285	**1306**	1301.90	1281	1277.20	1300	1284.40	1300	1279.40	1369
tai19	**1202**	1202	**1202**	1202	**1202**	1202	**1202**	1202	**1202**	1202	1304
tai20	1306	1302.20	**1318**	1314.30	1306	1301.40	1313	1307.70	1307	1301.40	1318
tai21	1592	1586.60	**1613**	1607.40	1602	1598.70	1597	1591.90	1595	1587.30	1573
tai22	1522	1498.60	**1529**	1511.40	1520	1503.60	1524	1504	1524	1504.70	1542
tai23	1502	1495.60	**1514**	1502.50	1502	1497.80	1503	1499.40	1502	1497.80	1474
tai24	1571	1561.30	**1588**	1574.50	1573	1567.30	1573	1566.70	1572	1568.20	1606
tai25	1525	1519.20	**1543**	1535.80	1529	1522.10	1530	1523.60	1529	1523.40	1518
tai26	1557	1546.70	**1561**	1553.50	1552	1543.40	1559	1552	1555	1546.60	1558
tai27	1596	1590.70	**1607**	1600	1593	1588.80	1601	1597.80	1604	1598.30	1617
tai28	1568	1564.10	**1583**	1579.70	1579	1567.50	1568	1565.60	1578	1566.90	1591
tai29	1556	1542.90	**1573**	1562.30	1563	1555.90	1560	1554.40	1560	1547.30	1525
tai30	1499	1472.90	**1508**	1502.10	1504	1495.60	1500	1479.10	1474	1469.50	1485
tai32	**1774**	1774	**1774**	1774	**1774**	1774	**1774**	1774	**1774**	1774	1774
tai33	**1729**	1729	**1729**	1729	**1729**	1729	**1729**	1729	**1729**	1729	1778
tai34	**1828**	1828	**1828**	1828	**1828**	1828	**1828**	1828	**1828**	1828	1828
tai40	**1602**	1602	**1602**	1602	**1602**	1602	**1602**	1602	**1602**	1602	1631
tai41	**1830**	1830	**1830**	1830	**1830**	1830	**1830**	1830	**1830**	1830	1876
tai42	**1761**	1761	**1761**	1761	**1761**	1761	**1761**	1761	**1761**	1761	1867
tai43	**1694**	1694	**1694**	1694	**1694**	1694	**1694**	1694	**1694**	1694	1809
tai44	**1787**	1787	**1787**	1787	**1787**	1787	**1787**	1787	**1787**	1787	1927
tai45	**1731**	1731	**1731**	1731	**1731**	1731	**1731**	1731	**1731**	1731	1997
tai46	**1856**	1856	**1856**	1856	**1856**	1856	**1856**	1856	**1856**	1856	1940
tai47	**1690**	1690	**1690**	1690	**1690**	1690	**1690**	1690	**1690**	1690	1789
tai48	**1744**	1744	**1744**	1744	**1744**	1744	**1744**	1744	**1744**	1744	1912
tai49	**1758**	1758	**1758**	1758	**1758**	1758	**1758**	1758	**1758**	1758	1915
tai50	**1674**	1674	**1674**	1674	**1674**	1674	**1674**	1674	**1674**	1674	1807

Thanks to the model using VSIDS along with our new conflict analysis procedure (i.e., Hybrid($vsids, disj$)), we were able to find new lower bounds for 7 instances. These

[3]As by March 15th, 2015, we noticed an accepted paper to the CPAIOR'15 conference [142] in which the authors report several new bounds for these instances (and many other scheduling benchmarks). Their lower bounds are greater than or equal to the values found in our experiments. It should be noted, however, that they use a 30000s time cutoff, a parallelization phase with two threads, in addition to starting search by using the best known bounds as an additional information. Our approach is quite different since we start search from scratch without parallelization, and each instance is limited to 3200s time cutoff.

instances are tai13, tai21, tai23, tai25, tai26, tai29, and tai30. The old lower bounds are based on the work of [65] and are reported in [1]. The model Hybrid($vsids, disj$) solely find these new bounds and is by far the best choice for building proofs for all instances.

It should be noted that in general the difference between the average and the maximum bound per instance is not large. In fact, almost all averages for the instances with new lower bounds are better than the best known lower bound.

5.3.5.2 OSP Results

We use three benchmarks for this problem: Gueret and Prins [72]; Taillard [134]; and Brucker et al. [33]. Note that all these instances were previously closed thanks to [71]. Conversely to the previous problem, we observed that $taskDom/bw$ was slightly better than $taskDom/tw$ for this problem. We shall therefore report the results of: Mistral($bool$), Hybrid($vsids, disj$), Hybrid($vsids, lazy$), Hybrid($bool, disj$), and Hybrid($bool, disj$).

The first two benchmarks are extremely easy for all the models. Gueret and Prins instances are all solved to optimality within an average CPU time less than 0.02s for each instance with any model and any seed. Taillard instances are also solved to optimality, however, with slightly longer runtime. Their detailed results are shown in Appendix B. We shall give more attention to Brucker et al. instances in the rest of this evaluation. The number of disjunctions ranges from 18 to 448 in these instances. We can therefore consider them as (very) small instances. Tables 5.10 and 5.11 present the detailed results of these instances.

These tables show clearly that clause learning is particularly not helpful in these instances. First, the lazy generation mode decreases clearly the performances on these instances since only Mistral($bool$), Hybrid($vsids, disj$), and Hybrid($bool, disj$) succeed to prove optimality to all configurations. Moreover, the average running time per instance is equal to 31.21s with Mistral($bool$) and 119.71s with Hybrid($\theta, disj$).

To investigate further the of impact of clause learning in this set, we propose to decrease the clause database size from a limit of 75000 to 10000. The new parametrized reduction strategy is $< 5000, 10000, 500, 12, 8, 90\% >$ instead of $< 5000, 75000, 50000, 12, 8, 90\% >$.

The new results are shown in Tables 5.12 and 5.13. The performances of the models Hybrid($\theta, disj$) are greatly improved with an average runtime of 35.95 and 40.25 instead of 117.46s and 121.97s respectively. The CP model, however, has a slightly better runtime with 31.21s. It should be pointed out that the global performances of lazy generation are not improved with the new reduction strategy.

TABLE 5.10: OSP results: Brucker et al. instances (j3-per0-1 – j5-per20-2)

Instance	Mistral(*bool*)			Hybrid(*vsids*, *disj*)			Hybrid(*vsids*, *lazy*)			Hybrid(*bool*, *disj*)			Hybrid(*bool*, *lazy*)		
	T	%O	UB	T	%O	UB	T	%O	UB	T	%O	UB	T	%O	UB
	avg	avg	min avg	avg	avg	min avg	avg	avg	min avg	avg	avg	min avg	avg	avg	min avg
j3-per0-1	0	100	**1127** 1127	0	100	**1127** 1127	0	100	**1127** 1127	0	100	**1127** 1127	0	100	**1127** 1127
j3-per0-2	0	100	**1084** 1084	0	100	**1084** 1084	0	100	**1084** 1084	0	100	**1084** 1084	0	100	**1084** 1084
j3-per10-0	0	100	**1131** 1131	0	100	**1131** 1131	0	100	**1131** 1131	0	100	**1131** 1131	0	100	**1131** 1131
j3-per10-1	0	100	**1069** 1069	0	100	**1069** 1069	0	100	**1069** 1069	0	100	**1069** 1069	0	100	**1069** 1069
j3-per10-2	0	100	**1053** 1053	0	100	**1053** 1053	0	100	**1053** 1053	0	100	**1053** 1053	0	100	**1053** 1053
j3-per20-0	0	100	**1026** 1026	0	100	**1026** 1026	0	100	**1026** 1026	0	100	**1026** 1026	0	100	**1026** 1026
j3-per20-1	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000
j3-per20-2	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000
j4-per0-0	0	100	**1055** 1055	0	100	**1055** 1055	0	100	**1055** 1055	0	100	**1055** 1055	0	100	**1055** 1055
j4-per0-1	0	100	**1180** 1180	0	100	**1180** 1180	0	100	**1180** 1180	0	100	**1180** 1180	0	100	**1180** 1180
j4-per0-2	0	100	**1071** 1071	0	100	**1071** 1071	0	100	**1071** 1071	0	100	**1071** 1071	0	100	**1071** 1071
j4-per10-0	0	100	**1041** 1041	0	100	**1041** 1041	0	100	**1041** 1041	0	100	**1041** 1041	0	100	**1041** 1041
j4-per10-1	0	100	**1019** 1019	0	100	**1019** 1019	0	100	**1019** 1019	0	100	**1019** 1019	0	100	**1019** 1019
j4-per10-2	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000
j4-per20-0	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000
j4-per20-1	0	100	**1004** 1004	0	100	**1004** 1004	0	100	**1004** 1004	0	100	**1004** 1004	0	100	**1004** 1004
j4-per20-2	0	100	**1009** 1009	0	100	**1009** 1009	0	100	**1009** 1009	0	100	**1009** 1009	0	100	**1009** 1009
j5-per0-0	0.04	100	**1042** 1042	0.05	100	**1042** 1042	0.10	100	**1042** 1042	0.07	100	**1042** 1042	0.14	100	**1042** 1042
j5-per0-1	0	100	**1054** 1054	0	100	**1054** 1054	0	100	**1054** 1054	0	100	**1054** 1054	0	100	**1054** 1054
j5-per0-2	0.01	100	**1063** 1063	0.02	100	**1063** 1063	0.04	100	**1063** 1063	0.03	100	**1063** 1063	0.04	100	**1063** 1063
j5-per10-0	0	100	**1004** 1004	0	100	**1004** 1004	0	100	**1004** 1004	0	100	**1004** 1004	0	100	**1004** 1004
j5-per10-1	0	100	**1002** 1002	0	100	**1002** 1002	0	100	**1002** 1002	0	100	**1002** 1002	0	100	**1002** 1002
j5-per10-2	0	100	**1006** 1006	0	100	**1006** 1006	0.01	100	**1006** 1006	0	100	**1006** 1006	0.01	100	**1006** 1006
j5-per20-0	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000
j5-per20-1	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000	0	100	**1000** 1000
j5-per20-2	0	100	**1012** 1012	0	100	**1012** 1012	0.01	100	**1012** 1012	0	100	**1012** 1012	0.01	100	**1012** 1012

TABLE 5.11: OSP results: Brucker et al. instances (j6-per0-0 – j8-per20-2)

Instance	Mistral(*bool*)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($bool, disj$)				Hybrid($bool, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
j6-per0-0	6.31	100	**1056**	1056	13.57	100	**1056**	1056	83.34	100	**1056**	1056	14.97	100	**1056**	1056	76.84	100	**1056**	1056
j6-per0-1	0	100	**1045**	1045	0	100	**1045**	1045	0.01	100	**1045**	1045	0	100	**1045**	1045	0	100	**1045**	1045
j6-per0-2	0.05	100	**1063**	1063	0.04	100	**1063**	1063	0.09	100	**1063**	1063	0.06	100	**1063**	1063	0.13	100	**1063**	1063
j6-per10-0	0.04	100	**1005**	1005	0.04	100	**1005**	1005	0.09	100	**1005**	1005	0.05	100	**1005**	1005	0.10	100	**1005**	1005
j6-per10-1	0	100	**1021**	1021	0	100	**1021**	1021	0	100	**1021**	1021	0	100	**1021**	1021	0	100	**1021**	1021
j6-per10-2	0.02	100	**1012**	1012	0.02	100	**1012**	1012	0.04	100	**1012**	1012	0.03	100	**1012**	1012	0.04	100	**1012**	1012
j6-per20-0	0.02	100	**1000**	1000	0.04	100	**1000**	1000	0.07	100	**1000**	1000	0.03	100	**1000**	1000	0.06	100	**1000**	1000
j6-per20-1	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j6-per20-2	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0.01	100	**1000**	1000
j7-per0-0	547.13	100	**1048**	1048	2684.99	100	**1048**	1048	3600	0	**1048**	1049.40	2258.20	100	**1048**	1048	3600	0	**1048**	1048.80
j7-per0-1	2.04	100	**1055**	1055	1.84	100	**1055**	1055	6.25	100	**1055**	1055	2.38	100	**1055**	1055	6.30	100	**1055**	1055
j7-per0-2	1.01	100	**1056**	1056	0.98	100	**1056**	1056	2.96	100	**1056**	1056	1.11	100	**1056**	1056	3.15	100	**1056**	1056
j7-per10-0	1.78	100	**1013**	1013	1.97	100	**1013**	1013	7.63	100	**1013**	1013	2.78	100	**1013**	1013	8.27	100	**1013**	1013
j7-per10-1	0.06	100	**1000**	1000	0.10	100	**1000**	1000	0.13	100	**1000**	1000	0.08	100	**1000**	1000	0.16	100	**1000**	1000
j7-per10-2	70.52	100	**1011**	1011	148.98	100	**1011**	1011	1026.13	100	**1011**	1011	260.55	100	**1011**	1011	1106.26	100	**1011**	1011
j7-per20-0	0	100	**1000**	1000	0	100	**1000**	1000	0.01	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j7-per20-1	0.41	100	**1005**	1005	0.23	100	**1005**	1005	0.65	100	**1005**	1005	0.46	100	**1005**	1005	0.89	100	**1005**	1005
j7-per20-2	1.30	100	**1003**	1003	0.68	100	**1003**	1003	2.48	100	**1003**	1003	1.11	100	**1003**	1003	2.48	100	**1003**	1003
j8-per0-1	723.60	100	**1039**	1039	2399.62	100	**1039**	1039	3600	0	**1039**	1041.30	2775.71	100	**1039**	1039	3600	0	**1039**	1040.30
j8-per0-2	9.78	100	**1052**	1052	10.94	100	**1052**	1052	43.34	100	**1052**	1052	16.40	100	**1052**	1052	49.25	100	**1052**	1052
j8-per10-0	22.46	100	**1017**	1017	40.56	100	**1017**	1017	186.31	100	**1017**	1017	66.05	100	**1017**	1017	201.28	100	**1017**	1017
j8-per10-1	58.68	100	**1000**	1000	283.50	100	**1000**	1000	1442.98	90	**1000**	1000.10	302.86	100	**1000**	1000	907.38	90	**1000**	1000.10
j8-per10-2	177.67	100	**1002**	1002	519.64	100	**1002**	1002	2330.50	100	**1002**	1002	638.96	100	**1002**	1002	2827.89	100	**1002**	1002
j8-per20-0	0.12	100	**1000**	1000	0.12	100	**1000**	1000	0.29	100	**1000**	1000	0.32	100	**1000**	1000	0.19	100	**1000**	1000
j8-per20-1	0.01	100	**1000**	1000	0.02	100	**1000**	1000	0.02	100	**1000**	1000	0.02	100	**1000**	1000	0.02	100	**1000**	1000
j8-per20-2	0.04	100	**1000**	1000	0.06	100	**1000**	1000	0.10	100	**1000**	1000	0.11	100	**1000**	1000	0.07	100	**1000**	1000
average	31.21	100			117.46	100			237.18	95.96			121.97	100			238.29	95.96		

TABLE 5.12: OSP: Brucker et al. reduced clause database results (j3-per0-1 – j5-per20-2)

Instance	Mistral(*bool*)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid(*bool*, $disj$)				Hybrid(*bool*, $lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
j3-per0-1	0	100	**1127**	1127	0	100	**1127**	1127	0	100	**1127**	1127	0	100	**1127**	1127	0	100	**1127**	1127
j3-per0-2	0	100	**1084**	1084	0	100	**1084**	1084	0	100	**1084**	1084	0	100	**1084**	1084	0	100	**1084**	1084
j3-per10-0	0	100	**1131**	1131	0	100	**1131**	1131	0	100	**1131**	1131	0	100	**1131**	1131	0	100	**1131**	1131
j3-per10-1	0	100	**1069**	1069	0	100	**1069**	1069	0	100	**1069**	1069	0	100	**1069**	1069	0	100	**1069**	1069
j3-per10-2	0	100	**1053**	1053	0	100	**1053**	1053	0	100	**1053**	1053	0	100	**1053**	1053	0	100	**1053**	1053
j3-per20-0	0	100	**1026**	1026	0	100	**1026**	1026	0	100	**1026**	1026	0	100	**1026**	1026	0	100	**1026**	1026
j3-per20-1	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j3-per20-2	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j4-per0-0	0	100	**1055**	1055	0	100	**1055**	1055	0	100	**1055**	1055	0	100	**1055**	1055	0	100	**1055**	1055
j4-per0-1	0	100	**1180**	1180	0	100	**1180**	1180	0	100	**1180**	1180	0	100	**1180**	1180	0	100	**1180**	1180
j4-per0-2	0	100	**1071**	1071	0	100	**1071**	1071	0	100	**1071**	1071	0	100	**1071**	1071	0	100	**1071**	1071
j4-per10-0	0	100	**1041**	1041	0	100	**1041**	1041	0	100	**1041**	1041	0	100	**1041**	1041	0	100	**1041**	1041
j4-per10-1	0	100	**1019**	1019	0	100	**1019**	1019	0	100	**1019**	1019	0	100	**1019**	1019	0	100	**1019**	1019
j4-per10-2	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j4-per20-0	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j4-per20-1	0	100	**1004**	1004	0	100	**1004**	1004	0	100	**1004**	1004	0	100	**1004**	1004	0	100	**1004**	1004
j4-per20-2	0	100	**1009**	1009	0	100	**1009**	1009	0	100	**1009**	1009	0	100	**1009**	1009	0	100	**1009**	1009
j5-per0-0	0.04	100	**1042**	1042	0.05	100	**1042**	1042	0.11	100	**1042**	1042	0.07	100	**1042**	1042	0.14	100	**1042**	1042
j5-per0-1	0	100	**1054**	1054	0	100	**1054**	1054	0	100	**1054**	1054	0	100	**1054**	1054	0	100	**1054**	1054
j5-per0-2	0.01	100	**1063**	1063	0.02	100	**1063**	1063	0.04	100	**1063**	1063	0.03	100	**1063**	1063	0.04	100	**1063**	1063
j5-per10-0	0	100	**1004**	1004	0	100	**1004**	1004	0	100	**1004**	1004	0	100	**1004**	1004	0	100	**1004**	1004
j5-per10-1	0	100	**1002**	1002	0	100	**1002**	1002	0	100	**1002**	1002	0	100	**1002**	1002	0	100	**1002**	1002
j5-per10-2	0	100	**1006**	1006	0	100	**1006**	1006	0.01	100	**1006**	1006	0	100	**1006**	1006	0.01	100	**1006**	1006
j5-per20-0	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j5-per20-1	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j5-per20-2	0	100	**1012**	1012	0.01	100	**1012**	1012	0.01	100	**1012**	1012	0	100	**1012**	1012	0.01	100	**1012**	1012

TABLE 5.13: OSP: Brucker et al. reduced clause database results (j6-per0-0 – j8-per20-2)

Instance	Mistral(*bool*)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($bool, disj$)				Hybrid($bool, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
j6-per0-0	6.31	100	**1056**	1056	8.23	100	**1056**	1056	49.62	100	**1056**	1056	8.26	100	**1056**	1056	46.24	100	**1056**	1056
j6-per0-1	0	100	**1045**	1045	0.01	100	**1045**	1045	0.01	100	**1045**	1045	0	100	**1045**	1045	0	100	**1045**	1045
j6-per0-2	0.05	100	**1063**	1063	0.05	100	**1063**	1063	0.09	100	**1063**	1063	0.07	100	**1063**	1063	0.13	100	**1063**	1063
j6-per10-0	0.04	100	**1005**	1005	0.04	100	**1005**	1005	0.09	100	**1005**	1005	0.05	100	**1005**	1005	0.09	100	**1005**	1005
j6-per10-1	0	100	**1021**	1021	0	100	**1021**	1021	0.01	100	**1021**	1021	0	100	**1021**	1021	0	100	**1021**	1021
j6-per10-2	0.02	100	**1012**	1012	0.02	100	**1012**	1012	0.04	100	**1012**	1012	0.03	100	**1012**	1012	0.05	100	**1012**	1012
j6-per20-0	0.02	100	**1000**	1000	0.03	100	**1000**	1000	0.07	100	**1000**	1000	0.03	100	**1000**	1000	0.06	100	**1000**	1000
j6-per20-1	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j6-per20-2	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000	0.01	100	**1000**	1000
j7-per0-0	547.13	100	**1048**	1048	736.57	100	**1048**	1048	3600	0	**1048**	1048.60	735.44	100	**1048**	1048	3600	0	**1048**	1048.20
j7-per0-1	2.04	100	**1055**	1055	1.84	100	**1055**	1055	5.57	100	**1055**	1055	2.14	100	**1055**	1055	5.09	100	**1055**	1055
j7-per0-2	1.01	100	**1056**	1056	1.02	100	**1056**	1056	2.65	100	**1056**	1056	1.09	100	**1056**	1056	2.75	100	**1056**	1056
j7-per10-0	1.78	100	**1013**	1013	1.83	100	**1013**	1013	6.35	100	**1013**	1013	2.38	100	**1013**	1013	6.08	100	**1013**	1013
j7-per10-1	0.06	100	**1000**	1000	0.11	100	**1000**	1000	0.12	100	**1000**	1000	0.07	100	**1000**	1000	0.15	100	**1000**	1000
j7-per10-2	70.52	100	**1011**	1011	50.65	100	**1011**	1011	545.65	100	**1011**	1011	91.14	100	**1011**	1011	486.95	100	**1011**	1011
j7-per20-0	0	100	**1000**	1000	0.01	100	**1000**	1000	0.01	100	**1000**	1000	0	100	**1000**	1000	0	100	**1000**	1000
j7-per20-1	0.41	100	**1005**	1005	0.22	100	**1005**	1005	0.58	100	**1005**	1005	0.45	100	**1005**	1005	0.78	100	**1005**	1005
j7-per20-2	1.30	100	**1003**	1003	0.67	100	**1003**	1003	2.10	100	**1003**	1003	1.08	100	**1003**	1003	2.06	100	**1003**	1003
j8-per0-1	723.60	100	**1039**	1039	819.17	100	**1039**	1039	3600	0	**1039**	1039.80	916	100	**1039**	1039	3600	0	**1039**	1039
j8-per0-2	9.78	100	**1052**	1052	8.08	100	**1052**	1052	30.05	100	**1052**	1052	10.16	100	**1052**	1052	33.73	100	**1052**	1052
j8-per10-0	22.46	100	**1017**	1017	17.81	100	**1017**	1017	92.44	100	**1017**	1017	29.01	100	**1017**	1017	105.55	100	**1017**	1017
j8-per10-1	58.68	100	**1000**	1000	68.93	100	**1000**	1000	309.52	100	**1000**	1000	86.46	100	**1000**	1000	607.19	100	**1000**	1000
j8-per10-2	177.67	100	**1002**	1002	154.13	100	**1002**	1002	1352.30	100	**1002**	1002	208.54	100	**1002**	1002	1366.40	100	**1002**	1002
j8-per20-0	0.12	100	**1000**	1000	0.12	100	**1000**	1000	0.34	100	**1000**	1000	0.34	100	**1000**	1000	0.20	100	**1000**	1000
j8-per20-1	0.01	100	**1000**	1000	0.02	100	**1000**	1000	0.03	100	**1000**	1000	0.02	100	**1000**	1000	0.02	100	**1000**	1000
j8-per20-2	0.04	100	**1000**	1000	0.06	100	**1000**	1000	0.12	100	**1000**	1000	0.11	100	**1000**	1000	0.08	100	**1000**	1000
average	31.21	100			35.95	100			184.57	96.15			40.25	100			189.69	96.15		

In order to understand the behavior of the different models in this set, we propose to analyze the results of the two reduction strategies on the hardest instances in this set (j7-per0-0 and j8-per0-1). We give for each model: the average runtime T; the speed of exploration (Nodes/S); and the average learnt clauses size (Size). Table 5.14 presents these statistics.

Using lighter clause database improves the overall efficiency by essentially increasing the speed of exploration. For instance, with j7-per0-0, Hybrid($vsids, disj$) explores 6605 Nodes/s with the default strategy and 24498 Nodes/s with the reduced strategy. The latter is approximately the half of the speed in the CP model. The speed of explorations is clearly the most influential element in these instances. This explains the bad performances of lazy generation since it slows down considerably the speed of exploration. With j7-per0-0 for example, the factor of speed between Mistral(*bool*) and any model Hybrid($\theta, lazy$) is about 45 with the default strategy and $13,5$ with the reduced strategy. That is, the CP model explores about 45 (respectively $13,5$) faster the search space compared to any model using the lazy mode for the default (respectively reduced) strategy.

TABLE 5.14: Open Shop Brucker et al. instances: Statistics

	Default reduction strategy																
Instance	Disj	Mistral(*bool*)			Hybrid($vsids, disj$)			Hybrid($vsids, lazy$)			Hybrid($bool, disj$)			Hybrid($bool, lazy$)		
		T	Nodes/S	Size	T	Nodes/S	Size	T	Nodes/S	Size	T	Nodes/S	Size	T	Nodes/S	Size
j7-per0-0	294	**547**	47458	0	2684	6605	21	3600	1016	28	2258	6252	24	3600	1088	29
j8-per0-1	448	**723**	37864	0	2399	5730	24	3600	1028	30	2775	4905	26	3600	1072	30
	Reduced Clause Database																
Instance	Disj	Mistral(*bool*)			Hybrid($vsids, disj$)			Hybrid($vsids, lazy$)			Hybrid($bool, disj$)			Hybrid($bool, lazy$)		
		T	Nodes/S	Size	T	Nodes/S	Size	T	Nodes/S	Size	T	Nodes/S	Size	T	Nodes/S	Size
j7-per0-0	294	**547**	47458	0	736	24498	21	3600	3657	27	735	19839	24	3600	3262	29
j8-per0-1	448	**723**	37864	0	819	20236	23	3600	3361	28	916	16002	26	3600	2981	30

In conclusion, the «Light» CP models that we used from [71, 69, 70] are extremely efficient with small sized instances. These models benefit essentially from the fast exploration speed. The impact of clause learning is more and more glaring when the size of the instance grows. It is very interesting to see how the hybrid model using DISJUNCTIVE-based learning along with VSIDS outperforms the other models on medium sized instances. Conversely, the lazy generation mode with our variants of the weighted degree heuristic is by far the most efficient approach for large instances.

Summary

We showed in this chapter that the performances of CP models in many sequencing and scheduling problems can be greatly improved by means of clause learning.

First, we investigated the impact of clause learning for solving car-sequencing benchmarks via AtMostSeqCard. The explanation algorithm that we proposed for this constraint runs in $O(n)$ time complexity and generates compact explanations compared to a naive one. We compared the new hybrid model against CP, Pseudo-Boolean, and SAT encodings for this problem. The experimental results emphasize the importance of clause learning specially for building proofs. Furthermore, we observed a strong correlation between advanced propagation and finding solutions quickly for this problem. The experiments showed also that adaptive-branching is very beneficial for building proofs while problem-specific heuristics are much helpful for finding solutions efficiently.

Next, we revisited lazy generation by proposing a solution to the redundancy issue regarding the generation of domain clauses. The issue is avoided by means of a constraint called DomainFaithfulness simulating UP and performing a complete channeling between the domains and the lazily generated atoms. We showed also that one can enforce ac on this constraint in a constant time amortized complexity down a branch of the search tree.

Finally, we studied the impact of clause learning for solving disjunctive scheduling problems. We introduced a novel conflict analysis scheme, called Disjunctive-based learning, tailored to this family of problems. This method guarantees the learning of clauses without encoding the tasks domains. A large number of experiments were carried out on common job shop and open shop benchmarks using the new learning propositions. These experiments showed how CP models can greatly benefit from clause learning when the instance size grows. We observed that the new Disjunctive-based learning with VSIDS outperforms the other models on medium sized instances. Conversely, the lazy generation mode with our variants of the weighted degree heuristic is by far the most efficient approach for large instances. Finally, we were able to find new lower bounds for 7 open instances using VSIDS along with our Disjunctive-based learning.

Chapter 6

Conclusion

We brought contributions to each of the three aspects of constraint programming that are 'search', 'propagation' and 'learning' for efficiently solving sequencing and scheduling problems. This case study strongly supports my thesis, that *modern constraint programming solvers may not underestimate any of these three aspects.*

Case Study: Car-Sequencing We proposed a complete approach for tackling the car-sequencing problem, with contributions on search, propagation and learning. This approach represents the state of the art for complete methods for this problem.

We proposed a new classification of search heuristics for this problem. This classification is based on a set of four criteria: branching variables, exploration directions, selection of branching variables and aggregation functions for this selection. Thanks to this classification, many heuristics were used for the first time in the form of untested configurations. We were able to indicate with a relatively high confidence the most robust strategies. Furthermore, we showed that the choice of the heuristic is often as important as the propagation method.

We then introduced a family of filtering algorithms for a class of sequence constraints. We first designed a simple filtering rule called Slack-Pruning that can be used only when using a specific type of branching choices for the car-sequencing problem. This filtering relies on reasoning simultaneously about capacity and demand constraints. We proposed next a generalization of this pruning in the form of a complete filtering for a new global constraint that we called AtMostSeqCard. This constraint can be used to model a number of sequencing problems including car-sequencing and crew-rostering. The filtering that we proposed for AtMostSeqCard runs in $O(n)$ time in the worst case which is optimal, whereas the previously best known method required a cubic compilation phase and then ran in $O(n^2 log(n))$. Furthermore, we showed that this algorithm can be adapted to achieve a complete filtering for some extensions of this

constraint. In particular, the conjunction of a set of m AtMostSeqCard constraints sharing the same scope $[x_1, \ldots, x_n]$ can be filtered in $O(nm)$. The experimental results on car-sequencing and crew-rostering benchmarks showed how competitive and efficient our filtering is compared to state-of-the-art propagators.

Finally, we investigated clause learning by introducing this method in our constraint programming approach for the car-sequencing problem. In order to use AtMostSeqCard in a hybrid CP/SAT solver, one has to explain every single domain change made by the propagator. We therefore proposed a procedure explaining AtMostSeqCard that runs in linear time in the worst case. We used this procedure in the design of a hybrid model for the car-sequencing problem. The experiments in this part included a variety of models with Pseudo-Boolean and SAT formulations. We showed, in particular, how clause learning improves the global performances in most cases. We observed a strong correlation between advanced propagation and finding solutions quickly for this problem. Moreover, for building proofs, we observed that clause learning was the most important ingredient and propagation became less useful.

Clause Learning in CP Learning is relatively recent and not as established in CP as propagation and search. We introduced two new techniques useful for embedding clause learning techniques into a constraint programming approach.

The first contribution is a general purpose method for implementing the lazy generation of Boolean variables representing a domain. We addressed the issue related to the redundancy of clauses used when lazily encoding a domain [53]. The DomainFaithfulness constraint that we proposed avoids such redundancy while ensuring the same level of consistency as UP without any computational overhead. The novel lazy generation method was empirically evaluated on a large number of disjunctive scheduling instances.

The second contribution is a learning mechanism tailored for disjunctive scheduling problems. We used a property of disjunctive scheduling allowing to design a novel conflict analysis scheme that learns clauses using a number of Boolean variables that is not function of the domain size. Our approach outperforms the CP model introduced in [71, 69, 70] in most cases. Several best known lower bounds for a traditional benchmark have been improved thanks to our method.

Future Research

There are a number of potential future research directions in each of the questions tackled in this dissertation.

- **Car-Sequencing**

It would be interesting to adapt our threefold propositions for the car-sequencing problem to 'real' industrial situations such as those proposed in the ROADEF'05 challenge [131]. Furthermore, since we have isolated good branching criteria in particular, the combination of these heuristics with discrepancy search algorithms such as LDS [74], IDS [85], and DDS [143] seems promising.

- **Propagation via** AtMostSeqCard

  The first direct future work regarding AtMostSeqCard is to consider incrementality. Designing an AC algorithm for AtMostSeqCard with a constant time amortized complexity (i.e., $O(n)$ time down a branch) would be a great result.

  Next, as for any new global constraint, proposing new extensions/particular cases is always an important research avenue in CP. We have already proposed two useful extensions , namely AtMostSeqΔCard and MultiAtMostSeqCard. Other related constraints might also be useful in numerous applications. For instance, it is possible to have a circular form of AtMostSeqCard to model cyclic timetabling and crew-rostering problems. This can be modeled by having a few more AtMost constraints to transform the 'chain' into a 'cycle', however, achieving AC on this constraint in linear time is a tough challenge. Another extension consists in having Among constraints instead of AtMost. The global constraint built this way corresponds to a particular case of Gen-Sequence defined by a conjunction between Sequence and Cardinality.

- **Explaining** AtMostSeqCard

  First, we know that reducing the explanations is always interesting especially when learning with global constraints. Recall that our explanation algorithm for AtMostSeqCard does not guarantee minimality w.r.t. the size nor does it guarantee it in the inclusion sense. It would be interesting to investigate the question of how hard is generating minimal explanations for AtMostSeqCard? Next, the application of these explanations to other sequencing problems might be very helpful in practice. The crew-rostering problem that we introduced in Section 4.5.2 is a typical example.

- **Generalizing** DomainFaithfulness

  Recall that DomainFaithfulness was proposed to avoid the redundancy issue of lazy generation. The proposed version in this thesis supports only bound literals. Its generalization is quite simple to conceive following the semantics of the constraint. In fact, we only have to perform a complete channeling between the literals $[\![x = v]\!]$, $[\![x \neq v]\!]$, $[\![x \leq u]\!]$, and $[\![x \geq l]\!]$ with the corresponding domain. The resulting proposition can be used within any (standard) hybrid CP/SAT solver.

- **Learning in Scheduling Problems**

There are three possible research avenues:

- Application to other scheduling problems.

  It is quite straightforward to apply our learning models with other disjunctive scheduling problems. Examples of direct extensions include mainly variants of job shop scheduling with time lags, sequence dependent setup times, and earliness/tardiness costs. There are perhaps additional constraints to explain, however the underlying models/solver is essentially the same.

  Furthermore, it would be interesting to adapt our learning propositions for more (general) scheduling problems such as the Resource Constrained Project Scheduling Problem (RCPSP).

- Learning with global constraints.

  The learning mechanisms that we introduced for disjunctive scheduling are implemented on top of the 'Light' models proposed in [71, 69, 70]. These models use very limited propagation mechanisms. It would be interesting to add some extra filtering, such as edge-finding and detectable precedences [140], and evaluate the impact of clause learning with such models.

- Hand-crafted learning.

  The experimental results of the DISJUNCTIVE-based learning showed that classical conflict analysis is not necessarily the best choice in practice. An interesting future research avenue is to study possible hand-crafted learning schemes with other problems.

Bibliography

[1] A summary of the best known lower/upper bounds for Taillard Job Shop instances. `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/jobshop.dir/b` Accessed: March 01, 2015.

[2] CSPLib: A problem library for constraints. `http://www.csplib.org`, 1999.

[3] Ignasi Abìo, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodrìguez-Carbonell. A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, CP'13, Uppsala, Sweden*, pages 80–96, 2013.

[4] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.

[5] Christian Artigues, Emmanuel Hebrard, Valentin Mayer-Eichberger, Mohamed Siala, and Toby Walsh. SAT and hybrid models of the car sequencing problem. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming, CPAIOR'14, Cork, Ireland*, pages 268–283, 2014.

[6] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[7] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09, Pasadena, California, USA*, pages 399–404, 2009.

[8] Fahiem Bacchus. GAC via unit propagation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP'07, Providence, Rhodes Island, USA*, pages 133–147, 2007.

[9] Fahiem Bacchus and Paul van Run. Dynamic variable ordering in csps. In *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming, CP'95, Cassis, France*, pages 258–275, 1995.

[10] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International series in Operations Research and Management Science. Kluwer, 2001.

[11] Philippe Baptiste, Claude Le Pape, and Laurent Péridy. Global constraints for partial csps: A case-study of resource and due date constraints. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming, CP'98, Pisa, Italy*, pages 87–101, 1998.

[12] J. Christopher Beck. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research*, 29(1):49–77, 2007.

[13] J.Christopher Beck, Patrick Prosser, and RichardJ. Wallace. Trying again to fail-first. In BoiV. Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *Recent Advances in Constraints*, volume 3419 of *Lecture Notes in Computer Science*, pages 41–55. Springer Berlin Heidelberg, 2005.

[14] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[15] Nicolas Beldiceanu and Mats Carlsson. Revisiting the cardinality operator and introducing the cardinality-path constraint family. In *Proceedings of the 17th International Conference on Logic Programming, ICLP'01, Paphos, Cyprus*, pages 59–73, 2001.

[16] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

[17] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and computer Modelling*, 20(12):97–123, 1994.

[18] Claude Berge and Edward Minieka. *Graphs and hypergraphs*, volume 7. North-Holland publishing company Amsterdam, 1973.

[19] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[20] Christian Bessiere. Constraint propagation. In Peter van Beek Francesca Rossi and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29 – 83. Elsevier, 2006.

[21] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering Algorithms for the NValueConstraint. *Constraints*, 11(4):271–293, 2006.

[22] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The Slide Meta-Constraint. In *Proceedings of the 3dr International Workshop on Constraint Propagation and Implementation, Nantes, France*, 2006.

[23] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. SLIDE: A Useful Special Case of the CARDPATH Constraint. In *Proceedings of the 18th European Conference on Artificial Intelligence, ECAI'08, Patras, Greece*, pages 475–479, 2008.

[24] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The Complexity of Reasoning with Global Constraints. *Constraints*, 12(2):239–259, 2007.

[25] Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, Émilie Picard-Cantin, Claude-Guy Quimper, and Toby Walsh. The balance constraint family. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming, CP'14, Lyon, France*, pages 174–189, 2014.

[26] Christian Bessiere, Emmanuel Hebrard, Marc-André Ménard, Claude-Guy Quimper, and Toby Walsh. Buffered Resource Constraint: Algorithms and Complexity. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming, CPAIOR'14, Cork, Ireland*, pages 318–333, 2014.

[27] Christian Bessiere and Pascal Van Hentenryck. To Be or Not to Be ... a Global Constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP'03, Kinsale, Ireland*, pages 789–794, 2003.

[28] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[29] Simon Boivin, Marc Gravel, Michaël Krajecki, and Caroline Gagné. Résolution du problème de car-sequencing à l'aide d'une approche de type FC. In *Proceedings of Premières Journées Francophones de Programmation par Contraintes, JFPC'05, Lens, France*, pages 11–20, 2005.

[30] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'04, Valencia, Spain*, pages 146–150, 2004.

[31] Nils Boysen and Malte Fliedner. Comments on "solving real car sequencing problems with ant colony optimization". *European Journal of Operational Research*, 182(1):466–468, 2007.

[32] Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter J. Stuckey, and Toby Walsh. Encodings of the Sequence Constraint. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP'07, Providence, Rhodes Island, USA*, volume 4741, pages 210–224, 2007.

[33] Peter Brucker, Johann Hurink, Bernd Jurisch, and Birgit Wöstmann. A branch & bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76(1–3):43 – 59, 1997. Second International Colloquium on Graphs and Optimization.

[34] Hadrien Cambazard. *Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication*. PhD thesis, Ecole des mines de Nantes, 2006.

[35] Hadrien Cambazard and Narendra Jussien. Identifying and Exploiting Problem Structures Using Explanation-based Constraint Programming. *Constraints*, 11(4):295–313, 2006.

[36] Hadrien Cambazard and Narendra Jussien. Solving the Minimum Number of Open Stacks Problem with Explanation-based Techniques. In *Proceedings of the 2nd International Workshop on Explanation-Aware Computing, ExaCt'07, Vancouver, British Columbia, Canada*, pages 14–19, 2007.

[37] Hadrien Cambazard and Barry O'Sullivan. A reformulation-based approach to explanation in constraint satisfaction. In *Proceedings of the 7th International Symposium on Abstraction, Reformulation, and Approximation, SARA'07, Whistler, Canada*, pages 395–396, 2007.

[38] Jacques Carlier and Éric Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[39] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, STOC'71, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[40] James M. Crawford and Andrew B. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94, Seattle, Washington*, pages 1092–1097, 1994.

[41] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[42] Johan De Kleer. A Comparison of ATMS and CSP Techniques. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, IJCAI'89, Detroit, Michigan, USA*, pages 290–296, 1989.

[43] Rina Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[44] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *Proceedings of the 8th European Conference on Artificial Intelligence, ECAI'88, Munich, Germany*, pages 290–295, 1988.

[45] Heidi E. Dixon and Matthew L. Ginsberg. Inference Methods for a Pseudo-Boolean Satisfiability Solver. In *Proceedings of the 18th National Conference on Artificial Intelligence, AAAI'02, and the 14th Conference on Innovative Applications of Artificial Intelligence, IAAI'02, Edmonton, Alberta, Canada*, pages 635–640, 2002.

[46] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the 35th Australasian Computer Science Conference, ACSC'12, Melbourne, Australia*, pages 115–124, 2012.

[47] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining Flow-Based Propagation. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'12, Nantes, France*, pages 146–162, 2012.

[48] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, SAT'03, Santa Margherita Ligure, Italy*, pages 502–518, 2003.

[49] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[50] Bertrand Estellon and Frédéric Gardi. Car sequencing is NP-hard: a short proof. *Journal of the Operational Research Society*, 64(10):1503–1504, 2013.

[51] Jean-Guillaume Fages, Xavier Lorca, and Louis-Martin Rousseau. The salesman and the tree: the importance of search in CP. *Constraints*, pages 1–18, 2014.

[52] Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey. Semantic Learning for Lazy Clause Generation. In *Proceedings of TRICS Workshop: Techniques foR Implementing Constraint programming Systems, TRICS'13, Uppsala, Sweden*, 2013.

[53] Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP'09, Lisbon, Portugal*, pages 352–366, 2009.

[54] Mark S. Fox, Norman M. Sadeh, and Can A. Baykan. Constrained Heuristic Search. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, IJCAI'89, Detroit, Michigan, USA*, pages 309–315, 1989.

[55] Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints*, 19(1):1–29, 2014.

[56] Eugene C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[57] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, 1982.

[58] Graeme Gange, Peter J. Stuckey, and Pascal Van Hentenryck. Explaining Propagators for Edge-Valued Decision Diagrams. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, CP'13, Uppsala, Sweden*, pages 340–355, 2013.

[59] Graeme Gange, Peter J. Stuckey, and Vitaly Lagoon. Fast Set Bounds Propagation Using a BDD-SAT Hybrid. *Journal of Artificial Intelligence Research*, 38:307–338, 2010.

[60] John Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of the 12th Annual Allerton Conference on Circuit Systems Theory, Illinois USA*, pages 866–874, 1974.

[61] Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI'92, Vienna, Austria*, pages 31–35, 1992.

[62] Ian P. Gent. Optimal Implementation of Watched Literals and More General Techniques. *Journal of Artificial Intelligence Research*, 48:231–251, 2013.

[63] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming, CP'96, Cambridge, Massachusetts*, pages 179–193, 1996.

[64] Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy Explanations for Constraint Propagators. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages, PADL'10, Madrid, Spain*, pages 217–233, 2010.

[65] Anis Gharbi and Mohamed Labidi. Extending the single machine-based relaxation scheme for the job shop scheduling problem. *Electronic Notes in Discrete Mathematics*, 36(0):1057 – 1064, 2010.

[66] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[67] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence, AAAI'98, and the 10th Conference on Innovative Applications of Artificial Intelligence, IAAI'98, Madison, Wisconsin*, pages 431–437, 1998.

[68] Jens Gottlieb, Markus Puchta, and Christine Solnon. A study of greedy, local search, and ant colony optimization approaches for car sequencing problems. In *Proceedings of Applications of Evolutionary Computing, EvoWorkshop'03: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, and EvoSTIM, Essex, UK*, pages 246–257, 2003.

[69] Diarmuid Grimes and Emmanuel Hebrard. Job Shop Scheduling with Setup Times and Maximal Time-Lags: A Simple Constraint Programming Approach. In *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'10, Bologna, Italy*, pages 147–161, 2010.

[70] Diarmuid Grimes and Emmanuel Hebrard. Models and Strategies for Variants of the Job Shop Scheduling Problem. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming, CP'11, Perugia, Italy*, pages 356–372, 2011.

[71] Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the Open Shop: Contradicting Conventional Wisdom. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP'09, Lisbon, Portugal*, pages 400–408, 2009.

[72] Christelle Guéret and Christian Prins. A new lower bound for the open shop problem. *Annals of Operations Research*, 92(0):165–183, 1999.

[73] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263 – 313, 1980.

[74] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI'95, Montréal, Québec, Canada*, pages 607–615, 1995.

[75] Saïd Jabbour, Jerry Lonlac, Lakhdar Sais, and Yakoub Salhi. Revisiting the Learned Clauses Database Reduction Strategies. *CoRR*, abs/1402.1956, 2014.

[76] Roberto Bayardo Jr and Robert C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, and the 9th Conference on Innovative*

Applications of Artificial Intelligence, IAAI'97, Providence, Rhode Island, pages 203–208, 1997.

[77] Narendra Jussien. *The versatility of using explanations within constraint programming*. Habilitation à diriger des recherches, Université de Nantes, 2003.

[78] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. *Proceedings of TRICS Workshop: Techniques foR Implementing Constraint programming Systems, TRICS'00, Singapore*, pages 118–133, 2000.

[79] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining Arc-Consistency within Dynamic Backtracking. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP'00, Singapore*, pages 249–261, 2000.

[80] Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *Proceedings of the 11th Workshop on Logic Programming Environments, WLPE'01, Paphos, Cyprus*, 2001.

[81] George Katsirelos. *Nogood processing in CSPs*. PhD thesis, University of Toronto, 2008.

[82] George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In *Proceedings of the 20th National Conference on Artificial Intelligence, AAAI'05, and the 17th Conference on Innovative Applications of Artificial Intelligence, IAAI'05, Pittsburgh, Pennsylvania, USA*, pages 390–396, 2005.

[83] Madjid Khichane, Patrick Albert, and Christine Solnon. Integration of ACO in a constraint programming language. In *Proceedings of the 6th International Conference on Ant Colony Optimization and Swarm Intelligence, ANTS'08, Brussels, Belgium*, pages 84–95, 2008.

[84] Tamas Kis. On the complexity of the car sequencing problem. *Operations Research Letters*, 32(4):331–335, 2004.

[85] Richard E. Korf. Improved Limited Discrepancy Search. In *Proceedings of the 13th National Conference on Artificial Intelligence, AAAI'96, and the 8th Conference on Innovative Applications of Artificial Intelligence, IAAI'96, Portland, Oregon*, pages 286–291, 1996.

[86] Stephen R. Lawrence. Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1984.

[87] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):147–167, 2007.

[88] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173 – 180, 1993.

[89] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[90] Alan K. Mackworth. On Reading Sketch Maps. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence, IJCAI'77, Cambridge, Massachusetts, USA*, pages 598–606, 1977.

[91] Michael J. Maher, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Flow-based propagators for the SEQUENCE and related global constraints. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming, CP'08, Sydney, NSW, Australia*, pages 159–174, 2008.

[92] Valentin Mayer-Eichberger and Toby Walsh. SAT Encodings for the Car Sequencing Problem. In *Proceedings of the 4th Pragmatics of SAT Workshop, POS'13, Helsinki, Finland*, pages 15–27, 2013.

[93] Julien Menana and Sophie Demassey. Sequencing and Counting with the multicost-regular Constraint. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'09, Pittsburgh, PA, USA*, pages 178–192, 2009.

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

[95] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference, DAC'01, Las Vegas, Nevada, USA*, pages 530–535, 2001.

[96] Nina Narodytska. *Reformulation of Global Constraints*. PhD thesis, The University of New South Wales, Sydney, Australia, 2011.

[97] Nina Narodytska, Thierry Petit, Mohamed Siala, and Toby Walsh. Three Generalizations of the FOCUS Constraint. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI'13, Beijing, China*, pages 630–636, 2013.

[98] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[99] Wim Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach.* PhD thesis, Eindhoven University of Technology, 1994.

[100] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = Lazy Clause Generation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP'07, Providence, Rhodes Island, USA*, pages 544–558, 2007.

[101] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via Lazy Clause Generation. *Constraints*, 14(3):357–391, 2009.

[102] Bruce D. Parrello and Waldo C. Kabat. Job-Shop Scheduling Using Automated Reasoning: A Case Study of the Car-Sequencing Problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.

[103] Gilles Pesant. Constraint-Based Rostering. In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling, PATAT'98, Montréal, Canada*, 2008.

[104] Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini. Counting-Based Search: Branching Heuristics for Constraint Satisfaction Problems. *Journal of Artificial Intelligence Research*, 43:173–210, 2012.

[105] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[106] Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-based large neighborhood search. *Constraints*, 19(4):339–379, 2014.

[107] Claude-Guy Quimper, Alexander Golynski, Alejandro López-Ortiz, and Peter van Beek. An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint. *Constraints*, 10(2):115–135, 2005.

[108] Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. Improved Algorithms for the Global Cardinality Constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP'04, Toronto, Canada*, pages 542–556, 2004.

[109] Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94, Seattle, Washington*, pages 362–367, 1994.

[110] Jean-Charles Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence, AAAI'96, and the 8th Conference on Innovative Applications of Artificial Intelligence, IAAI'96, Portland, Oregon*, pages 209–215, 1996.

[111] Jean-Charles Régin and Jean-Francois Puget. A Filtering Algorithm for Global Sequencing Constraints. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming, CP'97, Linz, Austria*, pages 32–46, 1997.

[112] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[113] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3:48–55, 1993.

[114] Christian Schulte and Mats Carlsson. Chapter 14 - finite domain constraint programming systems. In Peter van Beek Francesca Rossi and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 495 – 526. Elsevier, 2006.

[115] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, 2008.

[116] Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Explaining Time-Table-Edge-Finding Propagation for the Cumulative Resource Constraint. In *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'13, Yorktown Heights, NY, USA*, pages 234–250, 2013.

[117] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark Wallace. Why cumulative decomposition is not as bad as it sounds. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP'09, Lisbon, Portugal*, pages 746–761, 2009.

[118] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and MarkG. Wallace. Solving rcpsp/max by lazy clause generation. *Journal of Scheduling*, 16(3):273–289, 2013.

[119] Mohamed Siala, Christian Artigues, and Emmanuel Hebrard. Two Clause Learning Approaches for Disjunctive Scheduling. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming, CP'15, Cork, Ireland*, 2015.

[120] Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. An Optimal Arc Consistency Algorithm for a Chain of Atmost Constraints with Cardinality. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP'12, Québec City, QC, Canada*, pages 55–69, 2012.

[121] Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. An optimal arc consistency algorithm for a particular case of sequence constraint. *Constraints*, 19(1):30–56, 2014.

[122] Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. A study of constraint programming heuristics for the car-sequencing problem. *Engineering Applications of Artificial Intelligence*, 38:34–44, 2015.

[123] João P. Marques Silva and Karem A. Sakallah. GRASP - a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design, ICCAD'96, San Jose, California, USA*, pages 220–227, 1996.

[124] João P. Marques Silva and Karem A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[125] João P. Marques Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP'07, Providence, Rhodes Island, USA*, pages 483–497, 2007.

[126] Helmut Simonis and Barry O'Sullivan. Search Strategies for Rectangle Packing. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming, CP'08, Sydney, NSW, Australia*, pages 52–66, 2008.

[127] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP'07, Providence, Rhodes Island, USA*, pages 827–831, 2005.

[128] Barbara M. Smith. Succeed-first or Fail-first: A Case Study in Variable and Value Ordering, 1996. Research Report 96.26 University of Leeds, School of Computer Studies.

[129] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *Proceedings of the 13th European Conference on Artificial Intelligence, ECAI'98, Brighton, UK*, pages 249–253, 1998.

[130] Stephen F. Smith and Cheng-Chung Cheng. Slack-based Heuristics for Constraint Satisfaction Scheduling. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI'93, Washington, DC*, pages 139–144, 1993.

[131] Christine Solnon, Van-Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the roadef'2005 challenge problem. *European Journal of Operational Research*, 191(3):912–927, 2008.

[132] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT'09, Swansea, UK*, pages 237–243, 2009.

[133] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135 – 196, 1977.

[134] Éric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278 – 285, 1993. Project Management anf Scheduling.

[135] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[136] The AtMostSeqCard Revisited. Nina narodytska and toby walsh. In *Proceedings of the 11th International Workshop on Constraint Modelling and Reformulation, ModRef'12, Quebec City, Canada*, 2012.

[137] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Peter van Beek Francesca Rossi and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169 – 208. Elsevier, 2006.

[138] Willem Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. Revisiting the Sequence Constraint. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming, CP'06, Nantes, France*, pages 620–634, 2006.

[139] Willem Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. New filtering algorithms for combinations of among constraints. *Constraints*, 14(2):273–292, 2009.

[140] Petr Vilím. *Global Constraints in Scheduling.* PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, 2007.

[141] Petr Vilím. Edge Finding Filtering Algorithm for Discrete Cumulative Resources in $\mathcal{O}(kn \log(n))$. In *Proceedings of the 15th International Conference on Principles*

and Practice of Constraint Programming, CP'09, Lisbon, Portugal, volume 5732, pages 802–816, 2009.

[142] Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming, CPAIOR'15, Barcelona, Spain*, pages 437–453, 2015.

[143] Toby Walsh. Depth-bounded Discrepancy Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI'97, Nagoya, Japan*, pages 1388–1395, 1997.

[144] Toby Walsh. Search in a Small World. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI'99, Stockholm, Sweden*, pages 1172–1177, 1999.

[145] Toby Walsh. SAT v CSP. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP'00, Singapore*, pages 441–456, 2000.

[146] David L. Waltz. Generating Semantic Descriptions From Drawings of Scenes With Shadows. Technical report, Cambridge, Massachusetts, USA, 1972.

[147] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design, ICCAD'01, San Jose, California*, pages 279–285, 2001.

Appendix A

Reproducing the Disjunctive Scheduling Experiments

We give in this appendix sufficient details to reproduce the scheduling experiments. The source code is available via github at `https://github.com/siala/Hybrid-Mistral`. After cloning the repository[1], the following command line is needed to use the exact version of the solver for all the tests:

```
$ git checkout 3d2fe4738746aade930ce6aa92aa9a33cae09d48
```

The command used to compile the source is the following:

```
$ make scheduler
```

The general command syntax for the tests is the following:

```
$ bin/scheduler BENCHNAME -type [jla|jsp|osp] [-options]
```

where BENCHNAME is the instance file location and '-type [jla | jsp | osp]' indicates its type. The option '-seed v' is needed to precise the value of the randomisation seed 'v'. The 10 seeds that we used in these tests range from 11041979 to 11041988.

The instances are available in:

- data/scheduling/jsp/taillard/ for JSP Taillard instances. The option -type should have the value 'jsp' (default value)
- data/scheduling/jla/Lawrence/ for JSP Lawrence instances. The option -type should have the value 'jla'

[1] https://help.github.com

- data/scheduling/osp/gueret-prins/ for OSP Gueret&Prins instances. The option -type should have the value 'osp'
- data/scheduling/osp/taillard/ for OSP Taillard instances. The option -type should have the value 'osp'
- data/scheduling/osp/hurley for OSP Brucker et al. instances. The option -type should have the value 'osp'

We show now the options used for each model:

- Mistral($bool$): no option needed
- Mistral($task$): -taskweight 2
- Hybrid($vsids, disj$): -fdlearning 2 -semantic 1 -keeplearning 1 -orderedexploration 1 -reduce 1 -fixedLearntSize 50000 -fixedlimitSize 75000 -vsids 1
- Hybrid($bool, disj$): -fdlearning 2 -semantic 1 -keeplearning 1 -orderedexploration 1 -reduce 1 -fixedLearntSize 50000 -fixedlimitSize 75000
- Hybrid($task, disj$): -fdlearning 2 -semantic 1 -keeplearning 1 -orderedexploration 1 -reduce 1 -fixedLearntSize 50000 -fixedlimitSize 75000 -taskweight 2
- Hybrid($vsids, lazy$): -fdlearning 2 -semantic 1 -keeplearning 1 -orderedexploration 1 -reduce 1 -fixedLearntSize 50000 -fixedlimitSize 75000 -lazygeneration 1 -vsids 1
- Hybrid($bool, lazy$): -fdlearning 2 -semantic 1 -keeplearning 1 -orderedexploration 1 -reduce 1 -fixedLearntSize 50000 -fixedlimitSize 75000 -lazygeneration 1
- Hybrid($task, lazy$): -fdlearning 2 -semantic 1 -keeplearning 1 -orderedexploration 1 -reduce 1 -fixedLearntSize 50000 -fixedlimitSize 75000 -lazygeneration 1 -taskweight 2

To use the reduced clause reduction strategy (mentioned at Section 5.3.5.2), the options '-fixedLearntSize' '-fixedlimitSize' should have the values 500 and 10000 respectively (instead of 50000 and 75000).

Finally, in order to use the configuration for improving the lower bounds of Taillard open instances, the option '-lbcutoff 1400' should be added to precise the new time limit for each dichotomy step.

Appendix B

Detailed Results for OSP Instances

We give the detailed results for Gueret and Prins instances in Tables B.1, B.2, and B.3; and for Taillard OSP instances in Tables B.4 and B.5. Note that we use the same presentation protocol used in Section 5.3.5.

TABLE B.1: OSP: Gueret and Prins instances (GP03-01 – GP05-10)

Instance	Mistral($bool$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($bool, disj$)				Hybrid($bool, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
GP03-01	0	100	**1168**	1168	0	100	**1168**	1168	0	100	**1168**	1168	0	100	**1168**	1168	0	100	**1168**	1168
GP03-02	0	100	**1170**	1170	0	100	**1170**	1170	0	100	**1170**	1170	0	100	**1170**	1170	0	100	**1170**	1170
GP03-03	0	100	**1168**	1168	0	100	**1168**	1168	0	100	**1168**	1168	0	100	**1168**	1168	0	100	**1168**	1168
GP03-04	0	100	**1166**	1166	0	100	**1166**	1166	0	100	**1166**	1166	0	100	**1166**	1166	0	100	**1166**	1166
GP03-05	0	100	**1170**	1170	0	100	**1170**	1170	0	100	**1170**	1170	0	100	**1170**	1170	0	100	**1170**	1170
GP03-06	0	100	**1169**	1169	0	100	**1169**	1169	0	100	**1169**	1169	0	100	**1169**	1169	0	100	**1169**	1169
GP03-07	0	100	**1165**	1165	0	100	**1165**	1165	0	100	**1165**	1165	0	100	**1165**	1165	0	100	**1165**	1165
GP03-08	0	100	**1167**	1167	0	100	**1167**	1167	0	100	**1167**	1167	0	100	**1167**	1167	0	100	**1167**	1167
GP03-09	0	100	**1162**	1162	0	100	**1162**	1162	0	100	**1162**	1162	0	100	**1162**	1162	0	100	**1162**	1162
GP03-10	0	100	**1165**	1165	0	100	**1165**	1165	0	100	**1165**	1165	0	100	**1165**	1165	0	100	**1165**	1165
GP04-01	0	100	**1281**	1281	0	100	**1281**	1281	0	100	**1281**	1281	0	100	**1281**	1281	0	100	**1281**	1281
GP04-02	0	100	**1270**	1270	0	100	**1270**	1270	0	100	**1270**	1270	0	100	**1270**	1270	0	100	**1270**	1270
GP04-03	0	100	**1288**	1288	0	100	**1288**	1288	0	100	**1288**	1288	0	100	**1288**	1288	0	100	**1288**	1288
GP04-04	0	100	**1261**	1261	0	100	**1261**	1261	0	100	**1261**	1261	0	100	**1261**	1261	0	100	**1261**	1261
GP04-05	0	100	**1289**	1289	0	100	**1289**	1289	0	100	**1289**	1289	0	100	**1289**	1289	0	100	**1289**	1289
GP04-06	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269
GP04-07	0	100	**1267**	1267	0	100	**1267**	1267	0	100	**1267**	1267	0	100	**1267**	1267	0	100	**1267**	1267
GP04-08	0	100	**1259**	1259	0	100	**1259**	1259	0	100	**1259**	1259	0	100	**1259**	1259	0	100	**1259**	1259
GP04-09	0	100	**1280**	1280	0	100	**1280**	1280	0	100	**1280**	1280	0	100	**1280**	1280	0	100	**1280**	1280
GP04-10	0	100	**1263**	1263	0	100	**1263**	1263	0	100	**1263**	1263	0	100	**1263**	1263	0	100	**1263**	1263
GP05-01	0	100	**1245**	1245	0	100	**1245**	1245	0	100	**1245**	1245	0	100	**1245**	1245	0	100	**1245**	1245
GP05-02	0	100	**1247**	1247	0	100	**1247**	1247	0	100	**1247**	1247	0	100	**1247**	1247	0	100	**1247**	1247
GP05-03	0	100	**1265**	1265	0	100	**1265**	1265	0	100	**1265**	1265	0	100	**1265**	1265	0	100	**1265**	1265
GP05-04	0	100	**1258**	1258	0	100	**1258**	1258	0	100	**1258**	1258	0	100	**1258**	1258	0	100	**1258**	1258
GP05-05	0	100	**1280**	1280	0	100	**1280**	1280	0	100	**1280**	1280	0	100	**1280**	1280	0	100	**1280**	1280
GP05-06	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269
GP05-07	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269	0	100	**1269**	1269
GP05-08	0	100	**1287**	1287	0	100	**1287**	1287	0	100	**1287**	1287	0	100	**1287**	1287	0	100	**1287**	1287
GP05-09	0	100	**1262**	1262	0	100	**1262**	1262	0	100	**1262**	1262	0	100	**1262**	1262	0	100	**1262**	1262
GP05-10	0	100	**1254**	1254	0	100	**1254**	1254	0	100	**1254**	1254	0	100	**1254**	1254	0	100	**1254**	1254

TABLE B.2: OSP: Gueret and Prins instances (GP06-01 – GP08-10)

Instance	Mistral(*bool*)				Hybrid(*vsids*, *disj*)				Hybrid(*vsids*, *lazy*)				Hybrid(*bool*, *disj*)				Hybrid(*bool*, *lazy*)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
GP06-01	0	100	**1264**	1264	0	100	**1264**	1264	0	100	**1264**	1264	0	100	**1264**	1264	0	100	**1264**	1264
GP06-02	0	100	**1285**	1285	0	100	**1285**	1285	0	100	**1285**	1285	0	100	**1285**	1285	0	100	**1285**	1285
GP06-03	0	100	**1255**	1255	0	100	**1255**	1255	0	100	**1255**	1255	0	100	**1255**	1255	0	100	**1255**	1255
GP06-04	0	100	**1275**	1275	0	100	**1275**	1275	0	100	**1275**	1275	0	100	**1275**	1275	0	100	**1275**	1275
GP06-05	0	100	**1299**	1299	0	100	**1299**	1299	0	100	**1299**	1299	0	100	**1299**	1299	0	100	**1299**	1299
GP06-06	0	100	**1284**	1284	0	100	**1284**	1284	0	100	**1284**	1284	0	100	**1284**	1284	0	100	**1284**	1284
GP06-07	0	100	**1290**	1290	0	100	**1290**	1290	0	100	**1290**	1290	0	100	**1290**	1290	0	100	**1290**	1290
GP06-08	0	100	**1265**	1265	0	100	**1265**	1265	0	100	**1265**	1265	0	100	**1265**	1265	0	100	**1265**	1265
GP06-09	0	100	**1243**	1243	0	100	**1243**	1243	0	100	**1243**	1243	0	100	**1243**	1243	0	100	**1243**	1243
GP06-10	0	100	**1254**	1254	0	100	**1254**	1254	0	100	**1254**	1254	0	100	**1254**	1254	0	100	**1254**	1254
GP07-01	0	100	**1159**	1159	0	100	**1159**	1159	0	100	**1159**	1159	0	100	**1159**	1159	0	100	**1159**	1159
GP07-02	0	100	**1185**	1185	0	100	**1185**	1185	0	100	**1185**	1185	0	100	**1185**	1185	0	100	**1185**	1185
GP07-03	0	100	**1237**	1237	0	100	**1237**	1237	0	100	**1237**	1237	0	100	**1237**	1237	0	100	**1237**	1237
GP07-04	0	100	**1167**	1167	0	100	**1167**	1167	0	100	**1167**	1167	0	100	**1167**	1167	0	100	**1167**	1167
GP07-05	0	100	**1157**	1157	0	100	**1157**	1157	0	100	**1157**	1157	0	100	**1157**	1157	0	100	**1157**	1157
GP07-06	0	100	**1193**	1193	0	100	**1193**	1193	0	100	**1193**	1193	0	100	**1193**	1193	0	100	**1193**	1193
GP07-07	0	100	**1185**	1185	0	100	**1185**	1185	0	100	**1185**	1185	0	100	**1185**	1185	0	100	**1185**	1185
GP07-08	0	100	**1180**	1180	0	100	**1180**	1180	0	100	**1180**	1180	0	100	**1180**	1180	0	100	**1180**	1180
GP07-09	0	100	**1220**	1220	0	100	**1220**	1220	0	100	**1220**	1220	0	100	**1220**	1220	0	100	**1220**	1220
GP07-10	0	100	**1270**	1270	0	100	**1270**	1270	0	100	**1270**	1270	0	100	**1270**	1270	0	100	**1270**	1270
GP08-01	0	100	**1130**	1130	0	100	**1130**	1130	0.01	100	**1130**	1130	0	100	**1130**	1130	0	100	**1130**	1130
GP08-02	0	100	**1135**	1135	0.01	100	**1135**	1135	0.01	100	**1135**	1135	0.01	100	**1135**	1135	0	100	**1135**	1135
GP08-03	0	100	**1110**	1110	0	100	**1110**	1110	0.01	100	**1110**	1110	0.01	100	**1110**	1110	0	100	**1110**	1110
GP08-04	0	100	**1153**	1153	0	100	**1153**	1153	0	100	**1153**	1153	0.01	100	**1153**	1153	0.01	100	**1153**	1153
GP08-05	0	100	**1218**	1218	0	100	**1218**	1218	0.01	100	**1218**	1218	0	100	**1218**	1218	0.01	100	**1218**	1218
GP08-06	0	100	**1115**	1115	0.01	100	**1115**	1115	0.01	100	**1115**	1115	0.01	100	**1115**	1115	0.01	100	**1115**	1115
GP08-07	0	100	**1126**	1126	0.01	100	**1126**	1126	0.01	100	**1126**	1126	0	100	**1126**	1126	0	100	**1126**	1126
GP08-08	0	100	**1148**	1148	0	100	**1148**	1148	0.01	100	**1148**	1148	0.01	100	**1148**	1148	0.01	100	**1148**	1148
GP08-09	0	100	**1114**	1114	0	100	**1114**	1114	0	100	**1114**	1114	0	100	**1114**	1114	0.01	100	**1114**	1114
GP08-10	0	100	**1161**	1161	0	100	**1161**	1161	0	100	**1161**	1161	0	100	**1161**	1161	0.01	100	**1161**	1161

TABLE B.3: OSP: Gueret and Prins instances (GP09-01 – GP10-10)

Instance	Mistral($bool$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($bool, disj$)				Hybrid($bool, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
GP09-01	0.01	100	**1129**	1129	0.01	100	**1129**	1129	0.01	100	**1129**	1129	0.01	100	**1129**	1129	0.01	100	**1129**	1129
GP09-02	0.05	100	**1110**	1110	0.02	100	**1110**	1110	0.02	100	**1110**	1110	0.02	100	**1110**	1110	0.03	100	**1110**	1110
GP09-03	0.01	100	**1115**	1115	0.01	100	**1115**	1115	0.01	100	**1115**	1115	0.01	100	**1115**	1115	0.01	100	**1115**	1115
GP09-04	0.01	100	**1130**	1130	0.01	100	**1130**	1130	0.01	100	**1130**	1130	0.01	100	**1130**	1130	0.01	100	**1130**	1130
GP09-05	0.01	100	**1180**	1180	0.01	100	**1180**	1180	0.01	100	**1180**	1180	0.01	100	**1180**	1180	0.01	100	**1180**	1180
GP09-06	0.04	100	**1093**	1093	0.02	100	**1093**	1093	0.02	100	**1093**	1093	0.02	100	**1093**	1093	0.03	100	**1093**	1093
GP09-07	0.02	100	**1090**	1090	0.01	100	**1090**	1090	0.01	100	**1090**	1090	0.02	100	**1090**	1090	0.03	100	**1090**	1090
GP09-08	0.01	100	**1105**	1105	0.02	100	**1105**	1105	0.02	100	**1105**	1105	0.01	100	**1105**	1105	0.02	100	**1105**	1105
GP09-09	0.01	100	**1123**	1123	0.01	100	**1123**	1123	0.01	100	**1123**	1123	0.01	100	**1123**	1123	0.02	100	**1123**	1123
GP09-10	0.03	100	**1110**	1110	0.02	100	**1110**	1110	0.03	100	**1110**	1110	0.02	100	**1110**	1110	0.03	100	**1110**	1110
GP10-01	0.08	100	**1093**	1093	0.08	100	**1093**	1093	0.11	100	**1093**	1093	0.10	100	**1093**	1093	0.12	100	**1093**	1093
GP10-02	0.02	100	**1097**	1097	0.02	100	**1097**	1097	0.03	100	**1097**	1097	0.03	100	**1097**	1097	0.03	100	**1097**	1097
GP10-03	0.31	100	**1081**	1081	0.17	100	**1081**	1081	0.27	100	**1081**	1081	0.35	100	**1081**	1081	0.47	100	**1081**	1081
GP10-04	0.04	100	**1077**	1077	0.05	100	**1077**	1077	0.06	100	**1077**	1077	0.04	100	**1077**	1077	0.06	100	**1077**	1077
GP10-05	0.51	100	**1071**	1071	0.25	100	**1071**	1071	0.42	100	**1071**	1071	0.45	100	**1071**	1071	0.74	100	**1071**	1071
GP10-06	0.05	100	**1071**	1071	0.04	100	**1071**	1071	0.05	100	**1071**	1071	0.05	100	**1071**	1071	0.07	100	**1071**	1071
GP10-07	0.10	100	**1079**	1079	0.04	100	**1079**	1079	0.06	100	**1079**	1079	0.09	100	**1079**	1079	0.11	100	**1079**	1079
GP10-08	0.04	100	**1093**	1093	0.05	100	**1093**	1093	0.07	100	**1093**	1093	0.05	100	**1093**	1093	0.06	100	**1093**	1093
GP10-09	0.03	100	**1112**	1112	0.04	100	**1112**	1112	0.05	100	**1112**	1112	0.04	100	**1112**	1112	0.04	100	**1112**	1112
GP10-10	0.05	100	**1092**	1092	0.03	100	**1092**	1092	0.05	100	**1092**	1092	0.05	100	**1092**	1092	0.05	100	**1092**	1092
average	0.02	100			0.01	100			0.01	100			0.02	100			0.02	100		

TABLE B.4: OSP: Taillard instances (tai04_04_01 – tai07_7_10)

Instance	Mistral($bool$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($bool, disj$)				Hybrid($bool, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
tai04_04_01	0	100	**193**	193	0	100	**193**	193	0	100	**193**	193	0	100	**193**	193	0	100	**193**	193
tai04_04_02	0	100	**236**	236	0	100	**236**	236	0	100	**236**	236	0	100	**236**	236	0	100	**236**	236
tai04_04_03	0	100	**271**	271	0	100	**271**	271	0	100	**271**	271	0	100	**271**	271	0	100	**271**	271
tai04_04_04	0	100	**250**	250	0	100	**250**	250	0	100	**250**	250	0	100	**250**	250	0	100	**250**	250
tai04_04_05	0	100	**295**	295	0	100	**295**	295	0	100	**295**	295	0	100	**295**	295	0	100	**295**	295
tai04_04_06	0	100	**189**	189	0	100	**189**	189	0	100	**189**	189	0	100	**189**	189	0	100	**189**	189
tai04_04_07	0	100	**201**	201	0	100	**201**	201	0	100	**201**	201	0	100	**201**	201	0	100	**201**	201
tai04_04_08	0	100	**217**	217	0	100	**217**	217	0	100	**217**	217	0	100	**217**	217	0	100	**217**	217
tai04_04_09	0	100	**261**	261	0	100	**261**	261	0	100	**261**	261	0	100	**261**	261	0	100	**261**	261
tai04_04_10	0	100	**217**	217	0	100	**217**	217	0	100	**217**	217	0	100	**217**	217	0	100	**217**	217
tai05_05_01	0.01	100	**300**	300	0.01	100	**300**	300	0.03	100	**300**	300	0.01	100	**300**	300	0.02	100	**300**	300
tai05_05_02	0	100	**262**	262	0.01	100	**262**	262	0.02	100	**262**	262	0	100	**262**	262	0.01	100	**262**	262
tai05_05_03	0.01	100	**323**	323	0.02	100	**323**	323	0.05	100	**323**	323	0.02	100	**323**	323	0.06	100	**323**	323
tai05_05_04	0.01	100	**310**	310	0.02	100	**310**	310	0.03	100	**310**	310	0.02	100	**310**	310	0.03	100	**310**	310
tai05_05_05	0.02	100	**326**	326	0.04	100	**326**	326	0.08	100	**326**	326	0.03	100	**326**	326	0.07	100	**326**	326
tai05_05_06	0	100	**312**	312	0.01	100	**312**	312	0.03	100	**312**	312	0.01	100	**312**	312	0.03	100	**312**	312
tai05_05_07	0.01	100	**303**	303	0.02	100	**303**	303	0.04	100	**303**	303	0.02	100	**303**	303	0.04	100	**303**	303
tai05_05_08	0.01	100	**300**	300	0.02	100	**300**	300	0.04	100	**300**	300	0.02	100	**300**	300	0.04	100	**300**	300
tai05_05_09	0.01	100	**353**	353	0.03	100	**353**	353	0.05	100	**353**	353	0.02	100	**353**	353	0.04	100	**353**	353
tai05_05_10	0.03	100	**326**	326	0.02	100	**326**	326	0.06	100	**326**	326	0.03	100	**326**	326	0.07	100	**326**	326
tai07_07_01	0.01	100	**435**	435	0.02	100	**435**	435	0.02	100	**435**	435	0.02	100	**435**	435	0.03	100	**435**	435
tai07_07_02	0.02	100	**443**	443	0.04	100	**443**	443	0.05	100	**443**	443	0.02	100	**443**	443	0.02	100	**443**	443
tai07_07_03	0.15	100	**468**	468	0.14	100	**468**	468	0.28	100	**468**	468	0.12	100	**468**	468	0.29	100	**468**	468
tai07_07_04	0.02	100	**463**	463	0.04	100	**463**	463	0.04	100	**463**	463	0.02	100	**463**	463	0.03	100	**463**	463
tai07_07_05	0.01	100	**416**	416	0.03	100	**416**	416	0.04	100	**416**	416	0.01	100	**416**	416	0.01	100	**416**	416
tai07_07_06	0.29	100	**451**	451	0.18	100	**451**	451	0.57	100	**451**	451	0.35	100	**451**	451	1.49	100	**451**	451
tai07_07_07	0.06	100	**422**	422	0.13	100	**422**	422	0.33	100	**422**	422	0.06	100	**422**	422	0.22	100	**422**	422
tai07_07_08	0	100	**424**	424	0.02	100	**424**	424	0.02	100	**424**	424	0.01	100	**424**	424	0	100	**424**	424
tai07_07_09	0	100	**458**	458	0.02	100	**458**	458	0.02	100	**458**	458	0	100	**458**	458	0.01	100	**458**	458
tai07_07_10	0.01	100	**398**	398	0.02	100	**398**	398	0.02	100	**398**	398	0.01	100	**398**	398	0.02	100	**398**	398

TABLE B.5: OSP: Taillard instances (tai10_10_01 – tai20_20_10)

Instance	Mistral(*bool*)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($bool, disj$)				Hybrid($bool, lazy$)			
	T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB		T	%O	UB	
	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg	avg	avg	min	avg
tai10_10_01	0.09	100	**637**	637	0.07	100	**637**	637	0.13	100	**637**	637	0.06	100	**637**	637	0.13	100	**637**	637
tai10_10_02	0.05	100	**588**	588	0.06	100	**588**	588	0.08	100	**588**	588	0.05	100	**588**	588	0.07	100	**588**	588
tai10_10_03	0.06	100	**598**	598	0.07	100	**598**	598	0.10	100	**598**	598	0.04	100	**598**	598	0.05	100	**598**	598
tai10_10_04	0.03	100	**577**	577	0.05	100	**577**	577	0.06	100	**577**	577	0.02	100	**577**	577	0.03	100	**577**	577
tai10_10_05	0.07	100	**640**	640	0.07	100	**640**	640	0.11	100	**640**	640	0.06	100	**640**	640	0.08	100	**640**	640
tai10_10_06	0.03	100	**538**	538	0.06	100	**538**	538	0.06	100	**538**	538	0.03	100	**538**	538	0.04	100	**538**	538
tai10_10_07	0.05	100	**616**	616	0.05	100	**616**	616	0.07	100	**616**	616	0.04	100	**616**	616	0.04	100	**616**	616
tai10_10_08	0.06	100	**595**	595	0.06	100	**595**	595	0.11	100	**595**	595	0.05	100	**595**	595	0.06	100	**595**	595
tai10_10_09	0.05	100	**595**	595	0.06	100	**595**	595	0.08	100	**595**	595	0.05	100	**595**	595	0.08	100	**595**	595
tai10_10_10	0.05	100	**596**	596	0.06	100	**596**	596	0.09	100	**596**	596	0.06	100	**596**	596	0.06	100	**596**	596
tai15_15_01	0.63	100	**937**	937	0.37	100	**937**	937	0.45	100	**937**	937	0.58	100	**937**	937	0.73	100	**937**	937
tai15_15_02	0.54	100	**918**	918	0.52	100	**918**	918	0.51	100	**918**	918	0.45	100	**918**	918	0.50	100	**918**	918
tai15_15_03	0.43	100	**871**	871	0.35	100	**871**	871	0.42	100	**871**	871	0.48	100	**871**	871	0.49	100	**871**	871
tai15_15_04	0.47	100	**934**	934	0.35	100	**934**	934	0.44	100	**934**	934	0.47	100	**934**	934	0.50	100	**934**	934
tai15_15_05	0.64	100	**946**	946	0.41	100	**946**	946	0.52	100	**946**	946	0.58	100	**946**	946	0.61	100	**946**	946
tai15_15_06	0.53	100	**933**	933	0.35	100	**933**	933	0.46	100	**933**	933	0.49	100	**933**	933	0.53	100	**933**	933
tai15_15_07	0.47	100	**891**	891	0.37	100	**891**	891	0.46	100	**891**	891	0.47	100	**891**	891	0.48	100	**891**	891
tai15_15_08	0.47	100	**893**	893	0.37	100	**893**	893	0.49	100	**893**	893	0.45	100	**893**	893	0.52	100	**893**	893
tai15_15_09	0.73	100	**899**	899	0.51	100	**899**	899	0.52	100	**899**	899	0.54	100	**899**	899	0.74	100	**899**	899
tai15_15_10	0.67	100	**902**	902	0.40	100	**902**	902	0.54	100	**902**	902	0.63	100	**902**	902	0.68	100	**902**	902
tai20_20_01	3.63	100	**1155**	1155	2.37	100	**1155**	1155	3.10	100	**1155**	1155	2.96	100	**1155**	1155	3.31	100	**1155**	1155
tai20_20_02	5.93	100	**1241**	1241	2.44	100	**1241**	1241	3.67	100	**1241**	1241	3.74	100	**1241**	1241	4.62	100	**1241**	1241
tai20_20_03	3.02	100	**1257**	1257	2.15	100	**1257**	1257	3.01	100	**1257**	1257	3.06	100	**1257**	1257	3.21	100	**1257**	1257
tai20_20_04	3.51	100	**1248**	1248	2.19	100	**1248**	1248	2.65	100	**1248**	1248	3.34	100	**1248**	1248	3.17	100	**1248**	1248
tai20_20_05	2.92	100	**1256**	1256	2.21	100	**1256**	1256	2.85	100	**1256**	1256	2.72	100	**1256**	1256	2.87	100	**1256**	1256
tai20_20_06	3.57	100	**1204**	1204	2.44	100	**1204**	1204	3.19	100	**1204**	1204	3.47	100	**1204**	1204	3.41	100	**1204**	1204
tai20_20_07	3.93	100	**1294**	1294	2.59	100	**1294**	1294	3.35	100	**1294**	1294	4	100	**1294**	1294	3.41	100	**1294**	1294
tai20_20_08	4.99	100	**1169**	1169	2.56	100	**1169**	1169	3.28	100	**1169**	1169	3.50	100	**1169**	1169	4.26	100	**1169**	1169
tai20_20_09	3.68	100	**1289**	1289	2.33	100	**1289**	1289	2.71	100	**1289**	1289	3.31	100	**1289**	1289	3.56	100	**1289**	1289
tai20_20_10	3.21	100	**1241**	1241	2.18	100	**1241**	1241	2.76	100	**1241**	1241	3.18	100	**1241**	1241	3.18	100	**1241**	1241
average	0.75	100			0.48	100			0.63	100			0.66	100			0.73	100		

Appendix C

Résumé étendu

Nous décrivons brièvement les contributions principales de la thèse dans ce résumé étendu.

C.1 Introduction

La Programmation Par Contraintes (PPC) est un paradigme riche et générique pour résoudre les problèmes combinatoires d'une manière efficace. Face à l'explosion combinatoire, la PPC s'appuie typiquement sur des mécanismes de filtrage (ou de propagation) afin de réduire l'espace de recherche. De plus, des approches hybrides récentes de type SAT/PPC ont montré des résultats remarquables pour résoudre des problèmes largement dominés par d'autres approches (voir par exemple [118]). Ces méthodes déploient essentiellement des procédures d'analyse de conflits permettant l'apprentissage de nouvelles contraintes sous forme de clauses ce qui permet un élagage avancé de l'espace de recherche. L'analyse de conflit se base essentiellement sur la notion d'explication de contrainte afin de simuler le comportement des solveurs SAT.

Cette thèse vise à contribuer à la résolution des problèmes de séquencement et d'ordonnancement dans un contexte de Programmation Par Contraintes avec apprentissage de clauses. Les contributions présentées dans cette thèse se résume en sept points.

1. Une étude approfondie des heuristiques de branchement pour le problème de car-sequencing
2. Une nouvelle règle de filtrage simple et efficace dédiée au problème de car-sequencing.
3. Un algorithme de filtrage complet et optimal pour une classe de problèmes de séquencement tel que le car-sequencing ou encore les problèmes de type «confection d'horaires d'équipages» (crew-rostering).

4. Deux généralisations de ce filtrage.
5. Une stratégie réduite d'explication de notre filtrage. Cette stratégie a été appliquée au problème de car-sequencing.
6. Une révision du mode d'apprentissage de clauses avec génération retardée d'atomes.
7. Une nouvelle procédure d'analyse de conflit pour les problèmes d'ordonnancement disjonctifs.

Nous optons pour une approche expérimentale afin d'étudier l'efficacité de nos propositions. Ce résumé décrit brièvement les contributions principales de la thèse.

C.2 Contexte et définitions

C.2.1 Programmation par contraintes

Un problème de Satisfaction de Contraintes (CSP) est défini par un triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ où $\mathcal{X}$ représente l'ensemble des variables, $\mathcal{D}$ l'ensemble des domaines finis pour chacune des variables et $\mathcal{C}$ l'ensemble des contraintes qui spécifient les combinaisons de valeurs autorisés sur des sous-ensembles de variables. On suppose que $\mathcal{D}(x) \subset \mathbb{Z}$ pour tout $x \in \mathcal{X}$ et on note $min(x)$ et $max(x)$ les valeurs minimale et maximale de $\mathcal{D}(x)$. Une instanciation d'un ensemble de variables $\mathcal{X}$ est un tuple w tel que $w[i]$ représente la valeur affectée à la variable x_i. Une contrainte $C \in \mathcal{C}$ portant sur un ensemble de variables $\mathcal{X}$ caractérise une relation, i.e., un sous-ensemble de tuples du produit cartésien du domaine des variables de $\mathcal{X}$. Une instanciation w est dite cohérente pour une contrainte C ssi elle appartient à la relation correspondante. Une contrainte C est dite *arc-consistante* (AC) ssi, pour toute valeur j de chaque variable x_i qu'elle met en jeu, il existe une instanciation cohérente w telle que $w[i] = j$.

Nous considérons la résolution des CSPs avec des méthodes de recherche complètes basées sur une exploration en profondeur avec des mécanismes de branchement et de propagation de contraintes. Chaque contrainte est associée à au moins un propagateur. Chaque propagateur (ou algorithme de filtrage) réduit l'espace de recherche en supprimant des valeurs qui n'appartiennent à aucune solution par rapport au domaine courant. Pour chaque variable x, on suppose que les propagateurs utilisent uniquement les contraintes unaires suivante : $[\![x \leq v]\!]$, $[\![x \geq v]\!]$, $[\![x = v]\!]$ et $[\![x \neq v]\!]$.

C.2.2 Apprentissage de clauses dirigé par les conflits

Nous allons donner une description générale du fonctionnement des solveurs SAT avant de passer aux méthodes hybrides.

C.2.2.1 SAT

Un problème SAT sous forme normale conjonctive (CNF) se définit par une conjonction de clauses. Une clause $l_1 \vee .. \vee l_n$ est une disjonction de littéraux. Chaque littéral l représente soit une variable booléenne soit sa négation. Par la suite, le terme «atome» désigne une variable booléenne.

Les meilleurs solveurs SAT de la littérature sont basés sur l'algorithme CDCL (Conflict Driven Clause Learning) [95]. Cet algorithme peut être considéré comme une recherche arborescente augmenté par des mécanismes d'apprentissage de clauses et de retours arrière non-chronologiques. On note qu'il y a un seul type de propagation pour les clauses appelé *propagation unitaire* (UP) : pour une clause $l_1 \vee .. \vee l_n$ telle que la valeur de vérité de $n-1$ littéraux est en contradiction avec la clause, alors le dernier littéral doit forcément être assigné de façon à la satisfaire. L'algorithme CDCL progresse en prenant des décisions, c'est à dire, en affectant des valeurs de vérité arbitraires à des littéraux et en appliquant la propagation unitaire jusqu'à atteindre un point fixe après chaque décision. Lorsque la propagation unitaire déclenche une contradiction, une routine d'analyse de conflit est appelée. Le but est d'analyser le conflit courant pour trouver une raison, sous la forme d'une nouvelle clause. Cette dernière sera apprise et utilisée pour un retour arrière non-chronologique.

Pour chaque littéral l, on utilise la notation $n(l)$ pour représenter le niveau de l'arbre de recherche au quel l est assigné, et $r(l)$ pour représenter le rang du littéral l dans la séquence des assignements effectués par l'algorithme CDCL, et $raison(l)$ pour représenter la raison d'affectation de l. Pour chaque littéral assigné l, $reason(l)$ est égale à

- null si l est une décision.
- $l_1 \wedge .. \wedge l_n$ avec $l_i \neq l \in c$ et c est la clause qui a propagé l.

D'une façon similaire, on définit la raison initiale du conflit $raison(\perp)$ comme $l_1 \wedge .. \wedge l_n$ avec $l_i \in c$ et c est la clause dont la propagation unitaire a provoqué l'échec.

L'algorithme 2 présente la procédure d'analyse de conflit la plus utilisée dans littérature.

Cet algorithme renvoie Ψ comme raison du conflit sous forme d'un ensemble d'affectations suffisantes pour impliquer un échec. La clause apprise après l'analyse de conflit est

Algorithm 2: Analyse de conflit

1 $\Psi \leftarrow reason(\bot)$;
while $\left|\{l \in \Psi \mid n(l) = d\}\right| > 1$ **do**
2 $\quad l \leftarrow \arg\max_{l \in \Psi}(r(l))$;
3 $\quad \Psi \leftarrow \Psi \cup \{q \mid q \in raison(l) \wedge n(q) > 0\} \setminus \{l\}$;
return Ψ ;

$\neg\Psi$. La première ligne de cet algorithme initialise Ψ avec $reason(\bot)$. Par la suite, on remplace tous les littéraux de Ψ du niveau courant d par leur raisons sauf le dernier. Le choix du prochain littéral à remplacer s'effectue à la ligne 2 en prenant le dernier littéral assigné.

C.2.2.2 Méthodes hybrides SAT/PPC

Il existe une multitude de systèmes hybrides SAT/PPC dans la littérature. Nous allons utiliser un formalisme très récent en se basant sur les travaux de la 'Génération de Clauses Retardée' désignée par (LCG) pour «Lazy Clause Generation» [101, 100, 53, 52]. L'idée principale est de garder l'utilisation des propagateurs et de les considérer comme générateurs de clauses afin de simuler CDCL.

Nous allons utiliser la notion de littéral l pour représenter une contrainte unaire de type $[\![x \leq v]\!]$, $[\![x \geq v]\!]$, $[\![x = v]\!]$, $[\![x \neq v]\!]$. Dans LCG chaque propagateur est sensé être capable d'«expliquer» toutes ses actions en termes de règles de propagation. Une règle de propagation est une implication logique sous la forme $\Psi \Rightarrow l$ telle que la contrainte unaire associée à l est appelée par le propagateur en question et Ψ est une conjonction de littéraux suffisante pour propager l. Pour chaque littéral l, nous allons noter par

- $propag(l)$: le propagateur responsable au changement de domaine représenté par l.
- $explication(f, l)$: la prémisse Γ de la règle de propagation $\Gamma \Rightarrow l$ retournée par le propagateur $reason(l)$.

Les notions de *propag* et *explication* sont étendues pour l'échec $\bot$.

D'une façon très similaire à l'analyse de conflit décrite précédemment pour CDCL, nous introduisons dans algorithme 2 un pseudo-code d'analyse de conflit utilisé dans les solveurs hybrides modernes (par exemple [52]).

Cet algorithme est présenté d'une manière assez flexible afin de supporter différents modes d'apprentissage. D'abord la génération des explications peut être effectuée d'une

Algorithm 3: Analyse de conflit hybride

$f \leftarrow propag(\bot)$;
1 $\Psi \leftarrow explication(f, \bot)$;
2 **while** $|\{l \in \Psi \ / \ level(l) = d\}| > 1$ **do**
3 $\quad l = \arg\max_{l \in \Psi}(r(l))$;
4 $\quad f \leftarrow propag(l)$;
$\quad \Psi \leftarrow \Psi \cup \{q \mid q \in explication(f, l) \wedge n(q) > 0\} \setminus \{l\}$;
5 **return** Ψ ;

façon retardée (sur demande) ou bien au moment de la propagation. Chaque propagateur a la liberté d'adopter un mode de génération d'explications approprié selon les spécificités de la contrainte en question. D'autre part, la génération des atomes peut également s'effectuer d'une façon proactive (avant de commencer la recherche) ou bien d'une façon retardée (juste avant d'apprendre les clauses). Dans tous les cas, tous les atomes générés doivent être associés à des clauses qui encodent les différentes relations entre ces atomes. Par exemple, si on génère dès le début tous les atomes possibles associés à une variables x telle que $D(x) = [l, u]$, il va falloir ajouter les clauses suivantes pour tout $d \in [l, u]$: $\neg[\![x \leq d]\!] \vee [\![x \leq d+1]\!]$; $\neg[\![x = d]\!] \vee [\![x \leq d]\!]$;$\neg[\![x = d]\!] \vee \neg[\![x \leq d-1]\!]$;$[\![x = d]\!] \vee \neg[\![x \leq d]\!] \vee [\![x \leq d-1]\!]$;$[\![x = l]\!] \vee \neg[\![x \leq l]\!]$;$[\![x = u]\!] \vee [\![x \leq u-1]\!]$.

C.3 Heuristiques de branchement pour le problème de car-sequencing

Les premières descriptions du problème de car sequencing remontent aux années 80 [102, 44]. Nous disposons d'un nombre de véhicules à fabriquer. Tous les véhicules sont issues d'un modèle de base auquel on ajoute différentes options (climatisation, toit ouvrant, etc). Ainsi, les voitures sont regroupées par classe (chacune requiert un ensemble d'options prédéfinies). Le nombre de voitures par classe est fixé. Des stations de travail dédiées à la réalisation des options sont placées sur la chaîne d'assemblage (une station de travail par option). Les équipes de travail sont placées sur chaque station et une limite de temps est imposée pour installer l'option spécifique. Toutefois, les voitures demandant une certaine option ne doivent pas être groupées ensemble, sinon la station ne sera pas capable d'installer toutes les options.

Modélisation

Nous nous sommes basés sur le modèle standard implémenté avec Ilog-Solver 6.7. Ce modèle comporte n variables entières représentant les classes de véhicules de chaque position dans la ligne d'assemblage et nm variables booléennes y_i^j représentant le fait que

le véhicule placé en i^{ieme} position nécessite l'option j. La demande de chaque classe est exprimée avec une contrainte de cardinalité globale (Gcc) et l'arc-consistance de cette contrainte est réalisée avec Ilog Solver [110]. Quatre modélisations pour les contraintes de capacité sont comparées (exprimant que pour chaque option j, chaque sous-séquence de taille q_j contient au plus p_j variables d'option fixées à 1) associées aux contraintes de demande de chaque option (déduites des contraintes de demande sur chaque classe). La première modélisation (*sum*) comprend une simple décomposition en une séquence de contraintes de somme pour les contraintes de capacité ainsi qu'une contrainte de somme supplémentaire exprimant la demande. La seconde modélisation, (*gsc*) utilise une contrainte de séquencement global Gsc [111] par option. La troisième, (*amsc*) est une application de la procédure d'ac introduite dans la prochaine section pour la contrainte AtMostSeqCard. Enfin, la quatrième modélisation, (*amsc+gsc*) combine les contraintes AtMostSeqCard et Gsc.

Nouvelle structuration des heuristiques

Nous proposons de classifier les heuristiques de branchement selon quatre critères :

- *Exploration* : La manière d'explorer la ligne d'assemblage soit de manière lexicographique (*lex*) soit depuis le milieu vers les bords (*mid*).
- *Branchement* : L'affectation d'une classe (*class*) ou d'une option (*opt*) à une position dans la ligne.
- *Sélection* : Le paramètre d'évaluation pour les options. Cinq choix sont possibles : la capacité p_j/q_j ; la demande résiduelle (d_j^{opt}) ; la charge $\delta_j = d_j^{opt}.\frac{q_j}{p_j}$; la marge $\sigma_j = n - (n_j - \delta_j)$; et le taux d'utilisation $\rho_j = \delta_j/n_j$ (où n_j représente le nombre de positions libre dans la séquence pour l'option j).
- *Agrégation* : Lorsqu'on utilise un branchement de type $class$, on a besoin d'agréger les scores des options pour chaque classe. Nous proposons d'utiliser les trois choix suivants : la somme($\leq_{\sum}$) ; la somme euclidienne ($\leq_{Euc}$) ; et l'ordre lexicographique ($\leq_{lex}$).

L'ensemble des heuristiques est noté par $\langle\{class, opt\}, \{lex, mid\}, \{1, q/p, d^{opt}, \delta, n-\sigma, \rho\}, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$.

Expérimentations

Nous considérons 3 groupes d'instances de la CSPLib [2] ont été considérés. Le premier groupe, appelé set1 par la suite, comporte 70 instances à 200 véhicules, toutes

satisfiables. Dans le second groupe, il y a 9 instances de 100 véhicules réparties en 4 instances satisfiables, notées set2 et 5 instances insatisfiables, noté set3. Le troisième groupe contient 30 instances de plus grande taille (jusqu'à 400 véhicules). Parmi elles, nous considérons les 7 instances connues pour être satisfiables, notées set4.

Nous avons testé toutes les combinaisons possibles d'heuristiques et de modèles avec 5 graines de randomisation pour chaque instance. Ces tests ont montré en moyenne que :

- Le meilleur choix de branchement est celui basé sur les variables de classes
- L'exploration *mid* est légèrement meilleure que l'exploration lexicographique
- Le meilleur paramètre de sélection est celui basé sur la charge δ
- La fonction d'agrégation $\leq_{\sum}$ est légèrement meilleure que $\leq_{lex}$ et $\leq_{Euc}$.

Notons que ces observations ne constituent pas nécessairement le meilleur choix pour chaque groupe d'instances. On peut déterminer plutôt le meilleur choix pour chaque groupe, appelé le choix *parfait*. Nous introduisons alors deux métriques d'évaluation de critère appelé *confiance* et *importance* définit comme suit. La *confiance* du meilleur choix en moyenne est définit par le quotient entre le meilleur choix en moyenne et le choix parfait. De même, on peut considérer le *pire* choix pour chaque groupe et par conséquence, introduite l'*importance* d'un critère comme le quotient entre le *pire* choix et le choix *parfait* choix avec la formule $1 - \frac{pire}{parfait}$.

La table C.1 présente les valeurs de *confiance* et d'*importance* pour chaque critère (y compris la propagation). Cette table montre qu'il y a une grande confiance pour chaque meilleur paramètre en moyenne. Toutefois, les critères de branchement et de sélection sont beaucoup plus importants que les autres. On peut donc considérer les heuristiques $\langle class, \{lex, mid\}, \delta, \{\leq_{\sum}, \leq_{Euc}, \leq_{lex}\}\rangle$ comme les plus robustes.

TABLE C.1: *confiance* et *importance* pour chaque critère

	confiance	*importance*
Branchement	0.989	0.247
Sélection	0.995	0.231
Exploration	1.000	0.017
Agrégation	0.995	0.015
Propagation	0.996	0.217

C.4 Propagation dans une classe de problèmes de séquencement

Dans cette section, nous introduisons un algorithme de complexité temporelle optimale pour effectuer la cohérence d'arc sur la contrainte AtMostSeqCard. Nous présentons par la suite des expérimentations pour évaluer l'efficacité de nos propositions sur le problème de car-sequencing. Enfin, nous décrivons deux adaptations de cet algorithme avec des contraintes plus générales.

C.4.1 Propagation de la contrainte AtMostSeqCard

Soit $p, q, d \in \mathbb{N}$ et $[x_1, \ldots, x_n]$ une séquence de variables booléennes. La contrainte AtMostSeqCard que nous proposons se définit par une conjonction d'une chaîne de contraintes AtMost (i.e. de type $\sum_{l=1}^{q} x_{i+l} \leq p$) avec une contrainte de cardinalité.

Definition C.1. AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n]) \Leftrightarrow$

$$\bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq p) \wedge (\sum_{i=1}^{n} x_i = d)$$

AtMostSeqCard peut être considérée comme un cas particulier d'autres contraintes de séquence. Les meilleurs complexités de la littérature pour un filtrage complet de cette contraintes sont de $O(n^3)$[139], $O(2^q n)$ [139] et $O(n^2.log(n))$ dans une branche de l'arbre de recherche avec une initialisation de $O(q.n^2)$ [91]. Nous proposons un algorithme de filtrage complet pour cette contrainte avec une complexité linéaire dans le pire cas, donc optimale.

Cet algorithme se base sur une règle gloutonne qu'on appelle `leftmost`. Le rôle de `leftmost` est de retourner une instanciation w qui maximise $\sum_{i=1}^{n} x_i$ tout en respectant les contraintes AtMost. Nous avons proposé une implémentation linéaire de `leftmost` appelée `leftmost_count` retournant un vecteur L tel que $L[i]$ représente le nombre de 1 ajoutés par `leftmost` de x_1 à x_i. De la même façon on définit le vecteur R comme le résultat de `leftmost_count` sur la séquence $[x_n, \ldots, x_1]$. L'Algorithme 4 détermine l'AC de la contrainte AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n])$.

On établit dans les deux premières lignes de cet algorithme l'AC de la chaîne de contraintes AtMost ainsi que la cardinalité $\sum_{i=1}^{n} x_i = d$. Ces deux procédures peuvent être implémentées en temps linéaire. La suite de l'algorithme complète le filtrage. La complexité au pire cas de l'Algorithme 4 est clairement $O(n)$. Un exemple d'exécution de l'algorithme 4 pour $p = 4,\ q = 8,\ d = 12$ est donné dans la Figure C.2. Dans cette figure, la première ligne représente les domaines courants, les points représentent des variables non instanciées.

Algorithm 4: AC(AtMostSeqCard)

Data: $[x_1, \ldots, x_n], p, q, d$
Result: AC on AtMostSeqCard$(p, q, d, [x_1, .., x_n])$
1 AC$(\bigwedge_{i=0}^{n-q}(\sum_{l=1}^{q} x_{i+l} \leq p))$;
2 AC$(\sum_{i=1}^{n} x_i = d)$;
$d_{res} \leftarrow d - \sum_{i=1}^{n} min(x_i)$;
$L \leftarrow$ `leftmost_count`$([x_1, \ldots, x_n], p, q, d)$;
if $L[n] = d_{res}$ **then**
  $R \leftarrow$ `leftmost_count`$([x_n, \ldots, x_1], p, q, d)$;
  foreach $i \in [1, \ldots, n]$ *such that* $\mathcal{D}(x_i) = \{0, 1\}$ **do**
3 **if** $L[i] + R[n - i + 1] \leq ub$ **then** $\mathcal{D}(x_i) \leftarrow \{0\}$;
4 **if** $L[i - 1] + R[n - i] < ub$ **then** $\mathcal{D}(x_i) \leftarrow \{1\}$;
else if $L[n] < d_{res}$ **then**
5 **return** $\perp$;

FIGURE C.2: AtMostSeqCard$(4, 8, 12, [x_1, \ldots, x_{22}])$

.

$\mathcal{D}(x_i)$		.	0	.	.	.	.	.	.	0	1	0	.	.	.	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		**1**	0	**1**	**1**	**1**	0	0	0	0	1	0	**1**	**1**	**1**	0	0	0	**1**	0	**1**	**1**	1	
$\overleftarrow{w}[i]$		**1**	0	0	**1**	**1**	**1**	0	0	0	1	0	**1**	**1**	**1**	0	0	0	0	**1**	**1**	**1**	1	
$L[i]$	0	1	1	2	3	4	4	4	4	4	4	4	5	6	7	7	7	7	8	8	9	10	10	
$R[n-i+1]$		10	9	9	9	8	7	6	6	6	6	6	6	5	4	3	3	3	3	3	2	1	0	0
$L[i]+R[n-i+1]$		11	10	11	12	12	11	**10**	**10**	10	10	10	11	11	11	**10**	**10**	**10**	11	11	11	11	10	
$L[i-1]+R[n-i]$		**9**	10	10	10	10	10	10	10	10	10	10	**9**	**9**	**9**	10	10	10	10	10	**9**	**9**	10	
AC$(\mathcal{D}(x_i))$		**1**	0	.	.	.	.	**0**	**0**	0	1	0	**1**	**1**	**1**	**0**	**0**	**0**	.	.	**1**		1	

Les deux lignes suivantes donnent les instanciations $\overrightarrow{w}$ et $\overleftarrow{w}$ obtenues par `leftmost` de gauche à droite et de droite à gauche. Les troisième et quatrième lignes donnent les valeurs de L et de R. Les cinquième et sixième ligne correspondent à l'application de nos des propositions de filtrage. La septième ligne donne l'arc-consistance.

C.4.2 Expérimentations avec le problème de car-sequencing

Nous présentons dans cette section les résultats expérimentaux de l'algorithme de propagation appliqué au problème de car-sequencing. Ces expériences ont été effectuées sur un processeur Intel Xeon à 2.67GHz sous Linux. Les développements ont été faits avec Ilog-Solver 6.7 avec 5 exécutions aléatoires de chaque instance de 20 minutes.

Pour comparer l'efficacité relative des propagateurs, les résultats sont donnés en moyenne sur plusieurs heuristiques pour réduire le biais que celles-ci pourraient introduire. Pour chaque ensemble de données, *#sol* représente le nombre d'instances résolues. Puis, on donne le temps CPU (*time*) en secondes et le nombre de retour-arrière (*avg bts*) (tous les deux calculés en moyenne sur le nombre de solutions obtenues, le nombre d'instances et

d'heuristiques) ainsi que le nombre maximum de retour-arrière (*max bts*). Les résultats (en termes de *#sol*) de la meilleure méthode sont notés en gras.

Nous considérons la même configuration qu'en Section C.3. En particulier, nous utilisions les mêmes heuristiques et les 4 modèles de propagation *sum*, *gsc*, *amsc* et *amsc+gsc*. Les résultats sont présentés dans la Table C.2. Dans tous les cas, la meilleure valeur en terme de nombre de problèmes résolus est obtenue soit avec *amsc+gsc* (pour les instances de petite taille ou insatisfiables) ou avec *amsc* seul (pour les instances de plus grande taille set2 et set4). Globalement, on note que Gsc permet d'élaguer beaucoup plus de valeurs incohérentes que AtMostSeqCard. Toutefois, la propagation de Gsc ralentit très significativement la recherche (on a observé un facteur 12.5 sur le nombre de nœuds explorés par seconde). Par ailleurs, les niveaux de filtrage obtenus par ces deux méthodes sont incomparables. En conséquent combiner ces deux contraintes est toujours bénéfique plutôt que d'utiliser Gsc seule.

Table C.2: Car-sequencing : évaluation des méthodes de filtrage

Méthodes	set1 ($70 \times 34 \times 5$)				set2 ($4 \times 34 \times 5$)				set3 ($5 \times 34 \times 5$)				set4 ($7 \times 34 \times 5$)			
	#sol	avg bts	max bts	time	#sol	avg bts	max bts	time	#sol	avg bts	max bts	time	#sol	avg bts	max bts	time
sum	8480	231.2K	25.0M	13.93	95	1.4M	15.3M	76.60	0	-	-	> 1200	64	543.3K	13.7M	43.81
gsc	11218	1.7K	2.3M	3.60	325	131.7K	1.5M	110.99	31	55.3K	218.5K	276.06	140	25.2K	321.9K	56.61
amsc	10702	39.1K	13.8M	4.43	**360**	**690.8K**	**10.2M**	**72.00**	16	40.3K	83.4K	8.62	**153**	**201.4K**	**3.2M**	**33.56**
amsc+gsc	**11243**	**1.2K**	**1.1M**	**3.43**	339	118.4K	1.0M	106.53	**32**	**57.7K**	**244.7K**	**285.43**	147	23.8K	371.0K	66.45

C.4.3 Extensions

Nous avons adapté le filtrage de AtMostSeqCard pour deux contraintes plus générales. La première contrainte AtMostSeqΔCard remplace simplement la demande d par une variable entière δ, alors que la deuxième contrainte MultiAtMostSeqCard est une conjonction de m AtMostSeqCard portant sur la même séquence avec la même demande. Nous avons proposé un filtrage complet de MultiAtMostSeqCard en $O(m.n)$ et de AtMostSeqΔCard en $O(n)$. Nos expérimentations sur un cas particulier de problèmes de confection d'horaires pour des équipes (crew-rostering) ont montré l'efficacité du filtrage complet pour MultiAtMostSeqCard.

C.5 Apprentissage de clauses

Nous présentons dans cette section nos contributions liées à l'apprentissage de clauses.

C.5.1 Apprentissage de clauses pour le car-sequencing

Nous nous intéressons dans cette partie à l'aspect apprentissage de clauses pour résoudre le problème de car-sequencing en utilisant l'algorithme de filtrage de AtMostSeqCard que nous avons proposé.

C.5.1.1 Explications pour AtMostSeqCard

Afin d'utiliser AtMostSeqCard dans un solveur hybride, il est nécessaire d'expliquer la contrainte sous forme de règles de propagations pour tous les changements possibles provoqués par algorithme 4. Pour cela, nous classifions ces changements en deux types : d'une part on a les changements dus au filtrage des contraintes AtMost ou de cardinalité (lignes 1 et 2) et d'autre part, on a les nouvelles règles de filtrage (lignes 3, 4 et 5).

Le premier cas d'explication est assez simple (i.e. pour AtMost et cardinalité). Il suffit de renvoyer l'ensemble des variables assignées à 1 pour expliquer les affections de type $[\![x = 0]\!]$; et l'ensemble des variables assignées à 0 pour expliquer les affections de type $[\![x = 1]\!]$. L'explication des échecs s'effectue par exemple à travers la règle de propagation :

$$\bigwedge_{\mathcal{D}(x_i)=\{1\}} [\![x_i = 1]\!] \Rightarrow \bot$$

pour les contraintes AtMost et

$$\bigwedge_{\mathcal{D}(x_i)=\{1\}} [\![x_i = 1]\!] \Rightarrow \bot \text{ ou } \bigwedge_{\mathcal{D}(x_i)=\{0\}} [\![x_i = 0]\!] \Rightarrow \bot$$

pour la contrainte de cardinalité.

Nous montrons dans la suite comment expliquer les nouvelles règles de filtrage de l'algorithme 4. Nous allons montrer dans un premier lieu comment expliquer l'échec provoqué à la ligne 5, puis comment utiliser cette dernière pour expliquer le filtrage des lignes 3, 4.

Explication pour l'échec L'explication de cet échec est calculé en deux étapes. D'abord on considère l'ensemble des affectations comme raison initiale. Dans un deuxième temps, on utilise une procédure linéaire qui permet de réduire la raison initiale en supprimant certaines affectations.

On va utiliser $max_{\mathcal{D}}(i)$ pour noter la cardinalité maximale des q sous-séquences contenant x_i au début de la i^{eme} itération de `leftmost`. Nous allons aussi mentionner le domaine sur les vecteur finaux $\overrightarrow{w}$ and L avec la notation $\overrightarrow{w}_{\mathcal{D}}$ and $L_{\mathcal{D}}$ respectivement.

Soit $[x_1, \ldots, x_n]$ une séquence de variables booléennes sujette à la contrainte AtMostSeqCard$(p, q, d, [x_1, \ldots, x_n])$. Nous associons tout domaine $\mathcal{D}$ pour $x_1 \ldots x_n$ à un domaine plus large $\widehat{\mathcal{D}}$ défini de la façon suivante :

$$\begin{array}{ll} \widehat{\mathcal{D}}(x_i) = \{0,1\} & \text{if } \mathcal{D}(x_i) = \{0\} \wedge max_{\mathcal{D}}(i) = p \\ \widehat{\mathcal{D}}(x_i) = \{0,1\} & \text{if } \mathcal{D}(x_i) = \{1\} \wedge max_{\mathcal{D}}(i) \neq p \\ \widehat{\mathcal{D}}(x_i) = \mathcal{D}(x_i) & \text{otherwise} \end{array}$$

Lemme C.5.1. $\overrightarrow{w}_{\widehat{\mathcal{D}}} = \overrightarrow{w}_{\mathcal{D}}$.

Théorème C.5.1. Si $\bigwedge_{\mathcal{D}(x_i)=\{1\}} [\![x_i = 1]\!] \wedge \bigwedge_{\mathcal{D}(x_i)=\{0\}} [\![x_i = 0]\!] \Rightarrow \bot$ est une explication valide de l'échec, alors $\bigwedge_{\widehat{\mathcal{D}}(x_i)=\{1\}} [\![x_i = 1]\!] \wedge \bigwedge_{\widehat{\mathcal{D}}(x_i)=\{0\}} [\![x_i = 0]\!] \Rightarrow \bot$ est également valide.

Le Théorème C.5.1 nous donne une procédure linéaire pour réduire l'explication naïve. En effet, il suffit de lancer `leftmost_count` une fois pour calculer les valeurs de $max_{\mathcal{D}}$, puis construire $\widehat{\mathcal{D}}$ pour avoir l'explication réduite.

Explications pour les règles de filtrage Nous utilisons un mécanisme assez simple pour expliquer les nouvelles règles de filtrage. Supposons que le filtrage $[\![x \neq v]\!]$ a eu lieu à la ligne 3 ou 4 de l'algorithme 4. Pour expliquer $[\![x \neq v]\!]$, on calcule l'explication $\Psi \Rightarrow \bot$ de l'échec provoqué si on assigne v à x puis renvoie $\Psi \setminus [\![x = v]\!]$ comme explication de $[\![x \neq v]\!]$.

C.5.1.2 Résultats expérimentaux

Nous comparons différents modèles hybrides avec le modèle PPC pour résoudre le problème de car-sequencing. Les instances sont organisées en 3 catégories : `sat[easy]` (74 instances satisfiables), `sat[hard]` (7 instances satisfiable) et `unsat` (28 instances non-satisfiable). Toutes les expériences ont été effectuées sur des processeurs Intel Xeon CPUs 2.67GHz sous Linux. Pour chaque instance, nous avons lancé 5 tests randomisés avec un temps limite de 20 minutes. Nous avons testé les modèles suivants :

- Mistral comme solveur hybride SAT/PPC avec la nouvelle explication de AtMostSeqCard. Nous avons testé les 4 heuristiques de branchement suivante :
  1. *hybrid (VSIDS)* avec VSIDS ;
  2. *hybrid (Slot)* utilise $\langle opt, mid, \delta, \varnothing \rangle$ comme heuristique (voir section C.3).
  3. *hybrid (Slot → VSIDS)* utilise d'abord *hybrid (Slot)* puis change après 100 redémarrages vers VSIDS.
  4. *hybrid (VSIDS → Slot)* l'inverse de *hybrid (Slot → VSIDS)*.
- Minisat-2.2.0 avec les trois modèles SAT CNF_A, CNF_S, CNF_{A+S} proposés dans [5].

- *PBO-clauses* : Un modèle pseudo-booléen implémenté dans MiniSat+ [49] basé sur un encodage SAT.
- *PBO-cutting planes* : Un deuxième modèle pseudo-booléen implémenté dans SAT4J [19] basée sur les coupes [45].
- *pure-CP* : Le modèle PPC sans apprentissage en utilisant AtMostSeqCard et la même heuristique que *hybrid (Slot)*.

La table C.3 présente un résumé pour ces tests. Pour chaque groupe d'instances, nous présentons le nombre total de tests réussis (*#suc*), le nombre d'échecs rencontrés (*fails*) et le temps CPU (*time*) en secondes. Nous montrons les statistiques des meilleures configurations (par rapport à *#suc*) en **gras**.

En considérant la moyenne générale entre les deux premiers groupes, la meilleure configuration est celle qui utilise l'apprentissage de clauses avec l'heuristique spécifique à ce problème. Cette étude montre que la propagation est très importante pour trouver rapidement des solutions en évitant les branches inutiles de l'arbre de recherche. Pour prouver l'insatisfiabilité, les modèles purement SAT sont de loin les meilleurs configurations. En ce qui concerne les modèles hybrides, on note qu'elles sont beaucoup plus performantes que le modèle PPC. Ce dernier n'est pas capable de trouver des preuves pour la moindre instance. Ces résultats montrent clairement que l'apprentissage de la clauses est le facteur le plus important pour prouver l'insatisfiabilité.

Table C.3: Apprentissage de clauses appliqué au Car-sequencing

Méthode	`sat[easy]` (74 × 5)			`sat[hard]` (7 × 5)			`unsat` (28 × 5)		
	#suc	avg fails	time	#suc	avg fails	time	#suc	avg fails	time
CNF_A	370	2073	1.71	28	337194	282.35	**85**	**249301**	**105.07**
CNF_S	370	1114	0.87	31	60956	56.49	65	220658	197.03
CNF_{A+S}	370	612	0.91	34	32711	36.52	77	190915	128.09
hybrid (VSIDS)	370	903	0.23	16	207211	286.32	35	177806	224.78
hybrid (VSIDS → Slot)	370	739	0.23	35	76256	64.52	37	204858	248.24
hybrid (Slot → VSIDS)	370	132	0.04	34	4568	2.50	37	234800	287.61
hybrid (Slot)	370	132	0.04	**35**	**6304**	**3.75**	23	174097	299.24
pure-CP	**370**	**43.06**	**0.03**	35	57966	16.25	0	-	-
PBO-clauses	277	538743	236.94	0	-	-	43	175990	106.92
PBO-cutting planes	272	2149	52.62	0	-	-	1	5031	53.38

C.5.2 Le problème de redondance de clauses et l'apport de la contrainte DomainFaithfulness

Nous décrivons d'abord la problématique de redondance liée à la génération retardée d'atomes. Dans cette approche, quand un atome $[\![x \leq u]\!]$ doit être généré, on ajoute les clauses $\neg[\![x \leq a]\!] \vee [\![x \leq u]\!]$ et $\neg[\![x \leq u]\!] \vee [\![x \leq b]\!]$ où a et b sont les bornes les plus proches de u telles que $a < u < b$ et toutes les deux ont déjà un atome généré. Après l'ajout de

ces clauses, $\neg[\![x \leq a]\!] \vee [\![x \leq b]\!]$ devient clairement une clause redondante. Nous montrons dans ce paragraphe comment éviter cette redondance.

Au lieu de générer des clauses pour encoder les différentes relations entre les atomes, nous proposons d'utiliser une nouvelle contrainte appelé DomainFaithfulness qui assure le même niveau de cohérence. Soit x une variable ayant comme domaine $\mathcal{D}(x) = [l, u]$, $[x_1, \ldots, x_n]$ une séquence de variables booléennes (générées d'une façon retardée) et $[v_1, \ldots, v_n]$ une séquence de valeurs tel que x_i est la variable qui représente $[\![x \leq v_i]\!]$. Nous définissons la contrainte DomainFaithfulness de la façon suivante :

Definition C.2. DomainFaithfulness$(x, [x_1, \ldots, x_n], [v_1, \ldots, v_n])$:

$$\forall i, x_i \leftrightarrow x \leq v_i$$

Pour chaque variable x, nous utilisons une contrainte DomainFaithfulness (notée par DomainFaithfulness(x)). Initialement, la contrainte DomainFaithfulness(x) ne concerne que la variable x. Par la suite, chaque fois qu'on génère un atome $x_i \Leftrightarrow [\![x \leq v_i]\!]$, on ajoute x_i à la contrainte et sa structure interne est mise à jour.

La contrainte DomainFaithfulness a un double rôle. D'abord, elle simule *UP* comme si les clauses de domaines étaient générées d'une façon proactive. Ensuite, elle assure une cohérence entre le domaine de la variable x et tout les atomes x_i. Pour cela, il suffit de changer la borne supérieure (respectivement inférieure) de x à v_i (respectivement $v_i + 1$) s'il existe $x_i \Leftrightarrow [\![x \leq v_i]\!]$ avec $\mathcal{D}(x_i) = \{1\}$ (respectivement $\mathcal{D}(x_i) = \{0\}$) et $v_i < max(x)$ (respectivement $v_i >= min(x)$).

C.5.3 Apprentissage dans les problèmes d'ordonnancement disjonctifs

Nous étudions dans cette partie l'apport de l'apprentissage de clauses pour la résolution des problèmes d'ordonnancement disjonctifs. Dans cette famille de problèmes, chaque ressource (ou machine) est caractérisée par une contrainte d'accès exclusif. En d'autres termes, deux tâches qui demandent la même machine ne peuvent pas s'exécuter en même temps. L'aspect filtrage de contraintes a été largement étudié dans la littérature (voir par exemple [10]). Nous nous intéressons ici plutôt à l'apprentissage de clauses qu'à la propagation. Pour cela, nous utilisons un modèle basé sur une décomposition en contraintes réifiées simples. Ce choix de modélisation a été l'objet d'une analyse détaillée dans [117, 71] où il a été montré que ce modèle est compétitif et parfois meilleur que des modèles utilisant des méthodes de propagation plus sophistiquées. Nous considérons le problème d'ordonnancement d'atelier de type «Job shop».

C.5.3.1 Modélisation

Nous considérons la définition d'un job comme un ensemble de tâches. Soit $n, m \in \mathbb{N}^*$, $\mathcal{J} = \{J_i \mid 1 \leq i \leq n\}$ un ensemble de jobs et $\mathcal{M} = \{M_k \mid 1 \leq k \leq m\}$ un ensemble de machines. Chaque job J_i est défini par m tâches $\{T_{ik} \mid 1 \leq k \leq m\}$ tel que T_{ik} nécessite la machine k. Inversement, chaque machine M_k est associée à n tâches $\{T_{ik} \mid 1 \leq i \leq n\}$. Chaque tâche T_{ik} a une durée de traitement p_{ik} au court de la quelle la machine M_k est allouée exclusivement au job i.

Soit t_{ik} la variable représentant le début de la tâche T_{ik}.

Pour tout $k \in [1, m]$, la contrainte de ressource unaire associée à la machine M_k peut être exprimée de la façon suivante : $\forall i \in [1, n], \forall j \in [1, n]$ *tel que* $i < j$

$$t_{ik} + p_{ik} \leq t_{jk} \vee t_{jk} + p_{jk} \leq t_{ik} \tag{C.1}$$

Nous utilisons une décomposition simple utilisant des contraintes réifiées avec $O(n^2)$ variables booléennes δ_{kij} par machine M_k liée avec les variables de tâches de la façon suivante : $\forall i \in [1, n], \forall j \in [1, n], i < j$

$$\delta_{kij} = \begin{cases} 0 & \Leftrightarrow \quad t_{ik} + p_{ik} \leq t_{jk} \\ 1 & \Leftrightarrow \quad t_{jk} + p_{jk} \leq t_{ik} \end{cases} \tag{C.2}$$

Dans ce qui suit, nous utilisons la notation Disjunctive(b, x, y, d_x, d_y) pour noter la contrainte exprimée dans C.2 instanciée à (δ_{kij} , t_{ik}, t_{jk}, p_{ik}, p_{jk}).

En plus des contraintes de type Disjunctive, ce problème nécessite pour chaque job un ordre total sur ses tâches. Nous allons donc supposer que T_{ik_j} est la j^{eme} tâche demandée par le job J_i. L'ordre des tâches pour chaque job est exprimée par des *contraintes de précédence*. Soit x, y deux variables et d un entier. La contrainte de précédence Precedence(x, y, d) est définie de la façon suivante : $x + d \leq y$.

Le problème de job shop à minimisation de makespan peut etre défini comme suit :

minimiser C_{max} tel que :

$$\begin{aligned} \forall i \in [1, n] : & \quad \text{Precedence}(t_{ik_m}, C_{max}, p_{ik_m}) \\ \forall k \in [1, m], \forall i \in [1, n], \forall j \in [1, n], i < j : & \quad \text{Disjunctive}(\delta_{kij}, t_{ik}, t_{jk}, p_{ik}, p_{jk}) \\ \forall i \in [1, n], \forall a \in [1, m-1] : & \quad \text{Precedence}(t_{ik_j}, t_{ik_{j+1}}, p_{ik_j}) \end{aligned} \tag{C.3}$$

Stratégie de recherche Nous utilisons la même stratégie de recherche proposée dans [71, 69, 70]. D'abord les décisions sont prises uniquement sur les variables de disjonction. Chaque décision va alors fixer un sens de précédence entre deux tâches partageant une même ressource. Le choix de variables sera celui de $taskDom/tw$ [69]. Le choix de valeur pour la variable δ_{kij} recommandée par $taskDom/tw$ est basée sur l'approche guidée par les solutions (initialement proposé dans [12]). L'exploration de l'arbre de recherche s'effectue en deux phases. Une phase de recherche binaire pour améliorer les bornes initiales du makespan C_{max} et une phase de "Branch and Bound" classique pour minimiser la borne supérieure de C_{max}.

Durant la première phase, trois bornes $\underline{C}_{max}, \underline{C}_{max}$ et $\bar{C}_{max}$ sont utilisées, représentant respectivement, la borne inférieure "prouvée", la borne inférieure "approximée" et la borne supérieure. Chaque itération correspond au problème de décision dont le but est de prouver l'existence d'un ordonnancement qui satisfait toutes les contraintes avec un makespan inférieur ou égal à $C_{max} = \left\lfloor \frac{\underline{C}_{max}+\bar{C}_{max}}{2} \right\rfloor$. Si l'insatisfiabilité de ce problème est prouvée, alors les valeurs $\underline{C}_{max}$ et $\underline{C}_{max}$ sont mises à jour, la nouvelle valeur pour à la fois $\underline{C}_{max}$ et $\underline{C}_{max}$ est $C_{max} + 1$. Si au contraire une solution est trouvée, alors la valeur de $\bar{C}_{max}$ devient C_{max}. Dans le but d'avoir une garantie sur la durée maximale de l'étape de dichotomie, on impose une limite de temps pour chaque résolution du problème de décision ci-dessus. Lorsque la limite de temps est dépassée, alors seulement la valeur de $\underline{C}_{max}$ est mise à jour et devient $C_{max} + 1$.

La phase de "Branch and Bound" commence alors avec les meilleurs bornes $[\underline{C}_{max}, \bar{C}_{max}]$ obtenues durant la recherche binaire.

C.5.3.2 Apprentissage de clauses

Nous avons implémenté, d'une manière similaire à [52], un solveur hybride SAT/PPC supportant la génération retardée d'explications et d'atomes. Cependant, nous proposons une nouvelle méthode d'analyse de conflit qui garantit un apprentissage de clauses portant uniquement sur les variables booléennes de disjonction.

Rappelons que les décisions prises durant la recherche concernent uniquement les variables booléennes de disjonctions. Ainsi, tout littéral de borne (i.e. de type $[\![x \leq v]\!]$ ou $[\![x \geq v]\!]$) qui apparaît dans l'analyse de conflit possède nécessairement une raison composée uniquement de littéraux portant sur les variables booléennes de disjonction. En exploitant cette propriété, nous proposons de remplacer chaque littéral de borne à générer par son explication. Ce processus est répété de manière itérative jusqu'à ce qu'il n'y ait plus de littéraux de bornes à expliquer.

L'avantage de cette méthode est que les clauses qu'on apprend ne contiennent que des variables booléennes de disjonction. Par conséquence, il n'y a aucune génération d'atomes lors de la recherche.

C.5.4 Résultats expérimentaux

Nous avons implémenté nos propositions au sein du solveur Mistral-2.0. Toutes les expériences ont été réalisées sur des processeurs Intel i7-4770 sous Ubuntu 12.04. Nous comparons le modèle précédent avec ou sans apprentissage de clauses. L'heuristique $taskDom/tw$ [69] est testée pour les deux modes (avec ou sans apprentissage). L'heuristique *vsids* [95] est également utilisée comme une autre heuristique de branchement avec les modèles hybrides. Nous utilisons un redémarrage géométrique [144] et une stratégie d'oubli de clauses basée sur la méthode «Size-Bounded Randomized» [75].

Différent modèles sont expérimentalement testés. Ils sont organisés de la façon suivante :

- Mistral($task$) : Le modèle PPC sans apprentissage en utilisant $taskDom/tw$ comme heuristique.
- Hybrid(θ, σ) : Le modèle hybride SAT/PPC avec :
  - $\theta \in \{vsids, task\}$ l'heuristique de branchement.
  - σ : égale à $disj$ si on utilise la nouvelle méthode d'analyse de conflit et $lazy$ si on utilise la génération retardée d'atomes via DomainFaithfulness.

Nous utilisons deux types de jeux de données largement étudiés dans la littérature : les instances de Lawrence [86] et les instances de Taillard [134]. Les dernières sont beaucoup plus difficiles et contiennent encore 32 instances ouvertes. Nous avons lancé toutes les instances avec 10 graines différentes pour le générateur de nombres aléatoires. Les expériences sont organisées en deux parties : une qui respecte la description que nous avons donnée de la stratégie de recherche et l'autre conçue spécifiquement pour améliorer les bornes inférieures pour les instances ouvertes de la littérature.

C.5.4.1 Minimisation du makespan

Nous utilisons le pourcentage moyen d'écart (à l'optimum) PRD comme mesure d'efficacité. Le PRD d'un modèle m pour une instance C est calculé avec la formule : $100 * \frac{C_m - C_{best}}{C_{best}}$, avec C_m le makespan minimal trouvé par le modèle m pour cette instance et C_{best} le meilleur makespan trouvé par tous les modèles pour l'instance C. Le PRD moyen permet d'agréger les valeurs d'objectif obtenues par un algorithme donné sur un

ensemble d'instance lorsque celles ci ne sont pas normalisées. La valeur minimale possible d'un PRD est 0 et signifie que le modèle renvoie toujours le meilleur makespan.

En ce qui concerne les instances de Lawrence, le modèle Hybrid($vsids, disj$) a le meilleur PRD avec une valeur de $0,01$ et le plus grand pourcentage de solutions optimales (92%). Le seul cas où le modèle sans apprentissage (i.e. Mistral($task$)) a un meilleur makespan est avec l'instance la27 mais sans pour autant avoir la meilleure moyenne pour cette instance. A titre de comparaison entre les différents modèles hybrides, nous avons observé essentiellement que, indépendamment de l'heuristique utilisée, l'apprentissage basé sur la nouvelle méthode (i.e. $disj$) est nettement meilleur que celui basé sur la génération retardée d'atomes.

Pour les instances de Taillard, le choix du mode d'apprentissage ne semble pas avoir un impact significatif bien que la nouvelle méthode d'apprentissage était légèrement meilleure selon le PRD. Le modèle PPC (i.e. Mistral($task$)) était complètement dominé par les modèles hybrides avec un PRD de 1.5474 face à une moyenne de 0.9487 pour les modèles Hybrid($vsids, \theta$) et une moyenne de 0.30185 pour les modèles Hybrid($task, \theta$). Selon le PRD globale moyen, les meilleures configurations pour ces instances sont celles qui utilisent l'heuristique $taskDom/tw$. Ces résultats ne confirment pas nos observations précédentes avec les instances de Lawrence. Nous proposons donc de classer les résultats selon la taille des instances. Nous utilisons le nombre de disjonctions comme mesure de la taille d'instance.

Dans la table C.4, chaque ligne résume des statistiques liées à un ensemble d'instances de même taille (i.e. nombre de disjonctions). On donne pour chaque modèle : la vitesse d'exploration en termes de noeuds explorés par seconde (Nodes/s) ; la taille moyenne des clauses (Size) ; et une métrique de performance M égale à <%O,T> (%O est le pourcentage d'optimalité et T est temps CPU moyen) avec l'ensemble d'instances tai-01-10 et le PRD moyen pour le reste d'instances. Les meilleurs valeurs de la métrique M sont mises en **gras**.

Table C.4: Job Shop : Statistiques

	Mistral($task$)				Hybrid($vsids, disj$)				Hybrid($vsids, lazy$)				Hybrid($task, disj$)				Hybrid($task, lazy$)			
	M		Nodes/S	Size	M		Nodes/S	Size	M		Nodes/S	Size	M		Nodes/S	Size	M		Nodes/S	Size
	%O	T			%O	T			%O	T			%O	T			%O	T		
tai01-10	90	599	8879	0	**90**	**488**	5820	11	86	1118	1140	18	90	593	4286	15	78	1330	1118	28
	PRD				PRD				PRD				PRD				PRD			
tai11-20	1.1287		6784	0	**0.2236**		3730.18	32.08	0.9065		406.70	45.36	0.3876		2549.06	42.23	0.7930		426.78	70.68
tai21-30	0.6368		3907.64	0	0.2612		2339.66	34.57	1.0504		327.92	50.13	**0.2088**		1656.24	45.70	0.5665		329.60	74.32
tai31-40	1.7568		3907.42	0	0.8185		2538.44	52.96	0.7786		411.74	67.85	0.5526		1493.04	76.86	**0.4251**		429.30	111.84
tai41-50	2.0680		2489.94	0	0.7728		1463.14	70.58	0.5424		312.76	87.64	**0.3398**		955.22	98.38	0.8340		309.36	139.34
tai51-60	2.1032		1845.90	0	2.2353		2586.70	57.43	2.3825		514.50	44.82	0.2560		1090	97.88	**0.1425**		540.38	87.12
tai61-70	3.0303		1377.26	0	2.6539		2049.90	64.75	2.4276		489.38	52.17	0.3708		859.52	125.51	**0.2557**		475.68	127.73

Les résultats de la table C.4 montrent clairement une corrélation entre la taille de l'instance et le choix de l'heuristique. En effet, *vsids* est le meilleur choix avec les instances de petites ou moyennes tailles (tai01-10 ou même Lawrence) alors que *task* devient de plus en plus efficace quand la taille de l'instance augmente. En particulier, avec les très grandes instances, ce sont les deux modèles Hybrid$(task, \sigma)$ qui dominent clairement les autres avec une meilleur performance pour le mode *lazy*.

En ce qui concerne les deux modes d'apprentissage, il est clair que le mode *lazy* ralentit considérablement la vitesse d'exploration. Par exemple, avec le groupe tai-11-20, Hybrid$(vsids, disj)$ explore 3730.18 noeuds par seconde alors que Hybrid$(vsids, lazy)$ explore 406.70 noeuds par seconde. Nous pensons que ce phénomène est causé par le temps mis pour propager les contraintes DomainFaithfulness qui ne sont pas présentes dans la nouvelle méthode d'analyse de conflit. De la même façon, nous avons remarqué qu'avec la nouvelle méthode d'analyse de conflit, l'heuristique *task* est constamment plus lente que *vsids*. Par exemple, la vitesse d'exploration passe de 3730.18 avec Hybrid$(vsids, disj)$ à 2549.06 avec Hybrid$(task, disj)$ pour le groupe d'instances tai-11-20.

Pour résumer, nous avons constaté expérimentalement que le modèle Hybrid$(vsids, disj)$ est le meilleur choix avec des instances de taille petite ou moyenne et que Hybrid $(task, lazy)$ est de loin la meilleure configuration avec les grandes instances. De plus, un branchement de type *taskDom/tw* semble plus efficace que *vsids* lorsque la taille de l'instance augmente. Finalement, la nouvelle méthode d'analyse de conflit que nous avons proposé constitue une véritable alternative d'apprentissage.

Nous nous intéressons maintenant à la possibilité d'améliorer les bornes inférieures des instances Taillard encore ouvertes. Nous avons donc légèrement changé le comportement de la recherche binaire pour favoriser l'amélioration de la borne inférieure. La seule différence consiste à considérer le problème de décision comme satisfiable dès qu'on atteint la limite de l'itération en cours. De cette façon, la recherche binaire va essayer d'améliorer le plus possible la borne inférieure courante.

Dans ces expérimentations, le modèle Hybrid$(vsids, disj)$ a dominé clairement les autres modèles. Toutes les meilleures bornes sont trouvées par cette configuration. Il est important de noter que ces valeurs améliorent nettement l'état de l'art pour quelques instances. Grâce à notre nouvelle méthode d'analyse de conflit, nous avons pu trouver de nouvelles bornes inférieures pour sept instances. La table C.5 montre pour chacune de ces instance la nouvelle borne (new) et l'antienne meilleure valeur (old).

TABLE C.5: Nouvelles bournes inférieures

tai13		tai21		tai23		tai25		tai26		tai29		tai30	
new	old	new	old	new	old	new	old	new	old	new	old	new	old
1305	1282	1613	1573	1514	1474	1543	1518	1561	1558	1573	1525	1508	1485

C.6 Conclusion

Nous avons contribué à la résolution des problèmes de séquencement et d'ordonnancement dans un contexte hybride SAT/PPC. Nous avons présenté une étude approfondie des heuristiques de branchement pour le problème de séquencement de chaîne d'assemblage de voitures. Dans un deuxième temps, nous avons étudié les mécanismes de propagation pour une classe de contraintes de séquencement à travers la conception de plusieurs algorithmes de filtrage. En particulier, nous avons proposées un algorithme de filtrage optimal, complet et efficace pour la contrainte AtMostSeqCard. Ensuite, nous avons développé un algorithme linéaire de génération d'explications réduites pour cette contrainte ce qui permet de l'utiliser dans un solveur hybride et bénéficier des avantages offerts par SAT. Finalement, nous avons proposé de nouvelles alternatives d'analyse de conflit pour les problèmes d'ordonnancement disjonctifs. Ces nouvelles méthodes ont permis d'améliorer l'état-de-l'art d'un nombre d'instances ouvertes de la littérature.

Différentes perspectives sont ouvertes à l'issue de ces travaux de recherche. D'abord, il est intéressant de voir l'apport de AtMostSeqCard avec d'autres types de problème de séquencement comme la planification d'horaires de travail. Ensuite, les nouvelles méthodes d'analyse de conflit que nous avons proposé peuvent facilement être adaptées avec d'autres problèmes de décision et d'optimisation combinatoire. Finalement, nous pensons que la relation entre décomposition de contraintes, propagation globale et apprentissage de clauses nécessite encore des recherches à la fois théoriques et expérimentales.

Index

Symbols

$C(\mathcal{S})$, 10
$G(N, E)$, 23
$\perp$, 10
ρ_j, 42
$max(x_i)$, 10
$min(x_i)$, 10
$\mathcal{D}$, 9
$\mathcal{D}(x)$, 9
$\mathcal{X}$, 9
$\mathcal{X}(C)$, 10
δ'_j, 57
δ_j, 42
$\leq_{Euc}$, 43
$\leq_{lex}$, 43
σ_j, 42
d_j^{opt}, 42
q_j/p_j, 42
set1, 45
set2, 45
set3, 45
set4, 45
set5, 46
AC, 16
BC, 16
CSP, 11
CDCL, 22
CNF, 21
CN, 11
DPLL, 22
VSIDS, 27
1-UIP, 25

A

Arc Consistency, 16
Arc Consistent, 16
arity, 10
assigned, 10
assignment, 16
atom, 21

B

backjump, 23
backjumping, 24
backward, 33
Berge cycle, 17
Boolean, 10
Bound consistency, 16
bound consistent, 16
bound support, 16

C

channeling, 17
clause, 21
clause database, 26
Confidence, 52
conflict analysis, 23
conflict clause, 22
Conflict Driven Clause Learning, 22
conflict part, 24
conflicting literals, 22
Conjunctive Normal Form, 21
consistent, 10
constraint, 10
constraint graph, 17
constraint network, 11
Constraint Satisfaction Problem, 11
constraint type, 11
cut, 24

D
Davis-Putnam-Logemann-Loveland, 22
decomposed, 17
direct encoding, 30
domain, 9
Domain Faithfulness, 31
domain of the variable, 9
Domination, 25

E
explain, 32
explain(p), 22
explanation, 32

F
fail domain, 10

G
generalized nogoods, 28
geometric, 20
Global Cardinality Constraint, 18
global constraint, 17
Global Sequencing Constraint, 18

H
heavy tailed, 20

I
Implication Graph, 23
incomparable, 15
inconsistent, 10
instantiation, 10

L
Lazy clause generation, 28
lazy generation, 34
LCG, 28
Learning, ix, 3
leftmost, 64
leftmost_count, 71
level(p), 22
lex_assignment, 57
light model, 108
literal, 21
local consistency, 15
Luby, 20

N
nogood, 22

O
order encoding, 30

P
PRD, 116
projection, 10
Propagation, ix, 2
propagation rule, 22, 32
propagator, 13
Pseudo-Boolean, 11

R
range, 10
rank(p), 22
reason part, 24
reason(p), 33
resolution, 22
Restarts, 20

S
satisfiable, 11
satisfying, 10
scope, 10
Search, ix, 2
sequence, 9
Significance, 52
singleton, 10
solution, 11
strictly stronger, 10
strictly weaker, 10
stronger, 10, 15
subsumes, 15
successful, 46
support, 16

T

`tuple`, 10

U

`UIP`, 25

`Unary Resource Constraint`, 108

`Unique Implication point`, 25

`Unit-propagation`, 22

`unsatisfiable`, 11

`UP`, 22

V

`valid`, 10

`Variable State Independent Decaying Sum`, 27

`violating`, 10

W

`weaker`, 10