COST-BASED FILTERING ALGORITHMS FOR A CAPACITATED LOT SIZING PROBLEM AND THE CONSTRAINED ARBORESCENCE PROBLEM

HOUNDJI VINASETAN RATHEIL

Thesis submitted in partial fulfillment of the requirements for the degree of:

- *Doctor of Sciences of Engineering and Technology from UCL*
- *Doctor of Sciences of Engineering from UAC*

May 2017

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics (ICTEAM)
Louvain School of Engineering (LSE)
Université catholique de Louvain (UCL)
Belgium

Institut de Formation et de Recherche en Informatique (IFRI)
Ecole Doctorale des Sciences De l'Ingénieur (ED-SDI)
Université d'Abomey-Calavi (UAC)
Benin

Thesis Committee

Prof. Laurence A. Wolsey (supervisor)	IMMAQ, UCL, Belgium
Prof. Pierre Schaus (supervisor)	ICTEAM, UCL, Belgium
Prof. M. Norbert Hounkonnou (supervisor)	ICMPA, UAC, Benin
Prof. Jean-Charles Régin	I3S, UNS, France
Prof. Mathieu Van Vyve (secretary)	IMMAQ, UCL, Belgium
Prof. Yves Deville (chairperson)	ICTEAM, UCL, Belgium

To my daughter, my wife, my brothers, and my parents.

ABSTRACT

Constraint Programming (CP) is a paradigm derived from artificial intelligence, operational research, and algorithmics that can be used to solve combinatorial problems. CP solves problems by interleaving search (assign a value to an unassigned variable) and propagation. Constraint propagation aims at removing/filtering inconsistent values from the domains of the variables in order to reduce the search space of the problem. In this thesis, we develop filtering algorithms for two complex combinatorial optimization problems: a Capacitated Lot Sizing Problem (CLSP) and the Constrained Arborescence Problem (CAP). Each of these problems has many variants and practical applications.

The CLSP is the problem of finding an optimal production plan for single or multiple items while satisfying demands of clients and respecting resource restrictions. The CLSP finds important applications in production planning. In this thesis, we introduce a CLSP in CP. In many lot sizing and scheduling problems, in particular when the planning horizon is discrete and finite, there are stocking costs to be minimized. These costs depend on the time spent between the production of an order and its delivery. We focus on developing specialized filtering algorithms to handle the stocking cost part of a class of the CLSP. We propose the global optimization constraint `StockingCost` when the per-period stocking cost is the same for all orders; and its generalized version, the `IDStockingCost` constraint (ID stands for Item Dependent).

In this thesis, we also deal with a well-known problem in graph theory: the Minimum Weight Arborescence (MWA) problem. Consider a weighted directed graph in which we distinguish one vertex r as the root. An MWA rooted at r is a directed spanning tree rooted at r with minimum total weight. We focus on the CAP that requires one to

find an arborescence that satisfies some side constraints and that has minimum weight. The CAP has many real life applications in telecommunication networks, computer networks, transportation problems, scheduling problems, etc. After sensitivity analysis of the MWA, we introduce the CAP in CP. We propose a dedicated global optimization constraint to handle any variant of the CAP in CP: the `MinArborescence` constraint.

All the proposed filtering algorithms are analyzed theoretically and evaluated experimentally. The different experimental evaluations of these propagators against the state-of-the-art propagators show their respective efficiencies.

RÉSUMÉ

La programmation par contraintes - Constraint Programming (CP) - est un paradigme né de l'intelligence artificielle, de la recherche opérationnelle et de l'algorithmique qui peut être utilisé pour résoudre des problèmes combinatoires. CP résout les problèmes en alternant recherche (assigner une valeur à une variable non assignée) et propagation. La propagation en CP a pour objectif de supprimer (filtrer) les valeurs inconsistantes des domaines des variables en élaguant l'arbre de recherche du problème. Dans cette thèse, nous proposons des algorithmes de filtrage pour deux problèmes d'optimisation combinatoires complexes : un problème de dimensionnement de lots avec capacités - Capacitated Lot Sizing Problem (CLSP) - et le problème d'arborescence contraint - Constrained Arborescence Problem (CAP). Chacun de ces problèmes a beaucoup de variantes et d'applications pratiques.

Le CLSP consiste à trouver un plan de production optimal pour un ou plusieurs type(s) d'articles tout en satisfaisant les demandes des clients et en respectant les restrictions sur les ressources de production. Le CLSP a des applications importantes en planification de production. Dans cette thèse, nous introduisons le CLSP en CP. Dans beaucoup de problèmes de dimensionnement de lots et d'ordonnancement, en particulier lorsque l'horizon de planification est discret et fini, il y a des coûts de stockage à minimiser. Ces coûts dépendent du temps passé entre la production d'un article et sa livraison. Nous nous intéressons aux algorithmes de filtrage pour traiter les coûts de stockage intervenant dans une classe de problèmes de CLSP. Nous proposons la contrainte d'optimisation globale `StockingCost` lorsque les coûts de stockage sont les mêmes pour tous les articles et sa version généralisée, la contrainte `IDStockingCost` (ID pour *Item Dependent*).

Dans cette thèse, nous traitons également d'un problème bien connu en théorie de graphes : l'arborescence recouvrante de poids minimum - Minimum Weight Arborescence (MWA). Considérons un graphe orienté et pondéré dans lequel on distingue un noeud r comme étant la racine. Un MWA enraciné en r est un arbre recouvrant orienté et enraciné en r de poids total minimum. Nous nous intéressons au CAP qui consiste à trouver une arborescence qui satisfait certaines contraintes et ayant le poids minimum. Le CAP a beaucoup d'applications pratiques dans les réseaux de télécommunications, réseaux informatiques, problèmes de transport, problèmes d'ordonnancement, etc. Après une analyse de sensibilité du MWA, nous introduisons le CAP en CP. Nous proposons une contrainte globale d'optimisation pour résoudre toutes les variantes du CAP en CP : la contrainte `MinArborescence`.

Tous les algorithmes de filtrage proposés sont analysés de façon théorique et évalués expérimentalement. Les différentes évaluations expérimentales de ces propagateurs par rapport aux propagateurs de l'état de l'art ont montré leurs efficacités respectives.

ACKNOWLEDGEMENTS

Even if there is only one author, this Ph.D. thesis could not have been achieved without implications, inputs, supports and encouragements of some persons.

First of all, I would like to deeply thank Laurence Wolsey who has believed in me and has given me the opportunity to do my master and Ph.D. thesis with him. He taught me to be more rigorous about my mathematical definitions, properties, propositions and proofs with his different inputs. Very warm thanks go to Pierre Schaus for his substantial inputs, support, implication and motivation during this thesis and also for our interesting discussions about proofs and beauty of algorithms. Special thanks to Prof. Norbert Hounkonnou for his implication and support during my four stays in Benin. Many thanks also to Yves Deville for his support since my master thesis. I am fortunate to have had 4 advisors from different generations and research interests and to have their different points of view.

This thesis would not have started without the personal implications of Profs. Marc Lobelle, Norbert Hounkonnou and Semiyou Adedjouma. Many thanks to them.

I would like to thank Profs. Eugene Ezin and Gaston Edah for putting me comfortable during my work stays at IFRI Institute, Benin.

I would like to thank all members of INGI staff (Vanessa, Sophie, etc.) as well as all members of IFRI (Dr Arnaud, Christian, Miranda, Ozias, Aimée, etc.).

I would like to thank all beninese Ph.D. students in INGI/IFRI for our interesting discussions and our mutual motivations in this adventure: Parfait, Emery, Yannick, Lionel, John.

I would like to thank all cool Becool members: François, Trung, John, Thanh, Hélène, Jean-Baptiste, Cyrille, Mickael, Sascha, Quentin, Steven, Renaud - for fun moments and interesting discussions in our offices, during conferences, etc. Special thanks to François for our fruitfull discussions that have re-activated my passion about algorithmic/programming contests.

My warm gratitude goes to my parents Cathérine and Saturnin for their presence, implication and support from my birth. I am what I am and where I am thanks to them. I would like to thank also my brothers Adonis, Romik and Rozen and my friends Patrick, Christian, Boris, Come, Maurice, Jean-Louis, Oscar, Pauline, etc. for their encouragements.

Last, but certainly not least, I am deeply grateful to two important women in my life: 1) my wife Erika for her continuous support, motivation (for about 10 years now and in particular during this thesis) and for her sacrifices for our family 2) my daughter Davina who came in the middle of this thesis and has motivated me more to improve the quality of all things I do because I have to be a model for her (and her future brothers/sisters).

Ratheil,
May 2017.

LIST OF TABLES

Table 1.1 Example: performance profile - measures 12
Table 1.2 Example: performance profile - performance ratios r_p^i 12
Table 1.3 Example: performance profile - cumulative performance $\rho_p(\tau)$ 12
Table 5.1 Average results on 100 instances with $T = 500$: `StockingCost`, *MinAss*, and *Basic* 70
Table 6.1 Average results on 100 instances with $T = 20$: *IDS*, *MinAss*, $MinAss_2$, and *Basic* 94
Table 6.2 Average results on 100 instances with $T = 500$: *IDS*, *MinAss*, and *Basic* 94
Table 6.3 Average results on 100 instances with $T = 500$: *StockingCost*, *IDS*, *MinAss*, and *Basic* 96
Table 7.1 Proportion of the reduced costs affected by Proposition 34 116
Table 7.2 Average results on 100 instances: MinArbo_IRC, MinArbo_RC, Arbo+*LB*, and Arbo 117
Table 7.3 Average results on 100 instances: MinArbo_IRC vs MinArbo_RC 118
Table 7.4 Results on 15 instances: MinArbo_IRC vs MinArbo_RC 119
Table A.1 Results on 100 PSP instances with *MinAss* based model and COS heuristic (Part 1) 130
Table A.2 Results on 100 PSP instances with *MinAss* based model and COS heuristic (Part 2) 131

Table B.1 Results on 100 RMWA instances with *Arbo* + *MinArbo_RC* based model and COS heuristic (Part 1) 134
Table B.2 Results on 100 RMWA instances with *Arbo* + *MinArbo_RC* based model and COS heuristic (Part 2) 135

LIST OF FIGURES

Figure 1.1 A feasible solution of the PSP instance of Example 1 . . . 3
Figure 1.2 An optimal solution of the PSP instance of Example 1 . . . 3
Figure 1.3 Graphe G_1 . . . 4
Figure 1.4 Graphe G_1 with $a_{i,j}$ and b_i . . . 6
Figure 1.5 Graphe G_1 with $a_{i,j}$ and b_i: a feasible solution of the RMWA problem associated . . . 6
Figure 1.6 Performance profile - Example . . . 13
Figure 3.1 A feasible solution of the PSP instance of Example 4 . . . 31
Figure 3.2 An optimal solution of the PSP instance of Example 4 . . . 32
Figure 4.1 Initial graph G_1 . . . 41
Figure 4.2 Computation of $A(G_1)^\star$: Phase 1, first iteration. $A_0 = \{(2,1),(4,2),(4,3),(2,4),(3,5)\}$. . . . 41
Figure 4.3 Computation of $A(G_1)^\star$: Phase 1, second iteration. $A_0 = \{(2,1),(4,2),(4,3),(2,4),(3,5),(0,4)\}$. 41
Figure 4.4 Computation of $A(G_1)^\star$: Phase 2. $A_0 = \{(2,1),(4,2),(4,3),(3,5),(0,4)\}$. $(2,4)$ is removed. 43
Figure 4.5 Graphe G_1 with $a_{i,j}$ and b_i . . . 46
Figure 4.6 Graphe G_1 with $a_{i,j}$ and b_i: a feasible solution of the RMWA problem associated . . . 46
Figure 5.1 An optimal assignment of the instance of Example 7 without capacity restrictions . . . 60
Figure 5.2 An optimal assignment of the instance of Example 7 . . . 61

Figure 5.3 An optimal assignment of the instance of Example 7 . . . 64
Figure 5.4 Evolution of $H^{opt}_{X_3 \leftarrow t}$ . . . 65
Figure 5.5 Performance profiles - Nodes: `StockingCost`, *MinAss*, and *Basic* . . . 71
Figure 5.6 Performance profiles - Time: `StockingCost`, *MinAss*, and *Basic* . . . 71
Figure 6.1 $\mathcal{P}^r$ without capacity restrictions . . . 80
Figure 6.2 An optimal assignment for $\mathcal{P}^r$ . . . 80
Figure 6.3 An optimal assignment for $\mathcal{P}^r$ without X_6 . . . 85
Figure 6.4 An optimal assignment for $\mathcal{P}^r$ without X_2 . . . 85
Figure 6.5 An optimal assignment for $\mathcal{P}^r$ without X_3 . . . 86
Figure 6.6 An optimal assignment for $\mathcal{P}^r$ without X_1 . . . 86
Figure 6.7 Performance profiles - Nodes: *IDS*, *MinAss*, and *Basic* . . . 95
Figure 6.8 Performance profiles - Time: *IDS*, *MinAss*, and *Basic* . . . 95
Figure 6.9 Performance profiles - Nodes: *StockingCost*, *IDS*, *MinAss*, and *Basic* . . . 97
Figure 6.10 Performance profiles - Time: *StockingCost*, *IDS*, *MinAss*, and *Basic* . . . 97
Figure 7.1 Initial graph G_1 . . . 102
Figure 7.2 $A(G_1)^\star$ . . . 103
Figure 7.3 `Arborescence` constraint filtering . . . 106
Figure 7.4 `Arborescence` constraint filtering - Example . . . 107
Figure 7.5 G_1 with $rc(5,3)$ and $irc(5,3)$ . . . 110
Figure 7.6 G_1 with $rc(1,2)$ and $irc(1,2)$ . . . 113
Figure 7.7 G_1 with $rc(1,4)$ and $irc(1,4)$ . . . 115
Figure 7.8 Performance profiles - Nodes: MinArbo_RC and MinArbo_IRC . . . 118
Figure 7.9 Performance profiles - Time: MinArbo_RC and MinArbo_IRC . . . 118

LIST OF ALGORITHMS

2.2.1 Principle of a `fix-point` algorithm 19

4.1.1 Computation of a minimum weight arborescence $A(G)^\star$ rooted at vertex r 42

5.3.1 `StockingCost`: Filtering of lower bound on H - BC($H^{\min}$) 57
5.4.1 `StockingCost`: Computation of v_i^{opt} for all i 60
5.4.2 `StockingCost`: Computation of $minfull[t]$ and $maxfull[t]$ 62
5.4.3 `StockingCost`: Bound consistent filtering of $X_i^{\min}$ 63
5.5.1 `StockingCost`: Complete filtering algorithm in $O(n)$ - Part 1 67
5.5.2 `StockingCost`: Complete filtering algorithm in $O(n)$ - Filtering 68

6.3.1 `IDStockingCost`: Filtering of lower bound on H with $(H^{opt})^r$ in $O(n \log n)$ 81
6.3.2 `IDStockingCost`: Filtering of lower bound on H with $(H^{opt})^r$ in $O(n \log n)$ - Part 2 82
6.4.1 `IDStockingCost`: Filtering of n date variables in $O(n)$ . . . 91

7.2.1 Class Arborescence 105
7.3.1 Computation of the improved reduced costs $irc(i,j), \forall (i,j) \in E$ in $O(|V|^2)$ 114

LIST OF ABBREVIATIONS

AP Assignment Problem

ATSP Asymmetric Traveling Salesman Problem

BC Bound Consistency

CAP Constrained Arborescence Problem

CLSP Capacitated Lot Sizing Problem

COP Constraint Optimization Problem

CP Constraint Programming

CSLP Continuous Setup Lot Sizing Problem

CSP Constraint Satisfaction Problem

DBDFS Discrepancy-Bounded Depth First Search

DC Domain Consistency

DDS Depth-Bounded Discrepancy Search

DFS Depth First Search

DLS Discrete Lot Sizing

DLSI Discrete Lot Sizing with variable Initial stock

DLSP Discrete Lot Sizing Problem

ELSP Economic Lot Sizing Problem

GAC Generalized Arc consistency

IDS Iterative Deepening Search

LDS Limited Discrepancy Search

LNS Large Neighborhood Search

LP Linear Programming

LS Lot Sizing

MDST Minimum Directed Spanning Tree

MIP Mixed Integer Programming

MST Minimum Spanning Tree

MWA Minimum Weight Arborescence

OR Operational Research

PLSP Proportional Lot Sizing and Scheduling Problem

PSP Pigment Sequencing Problem

RMWA Resource constrained Minimum Weight Arborescence

TSP Traveling Salesman Problem

UCL Université catholique de Louvain

WBST Weight-Bounded Spanning Tree

WW Wagner-Whitin

TABLE OF CONTENTS

Abstract V
Résumé VII
Acknowledgments XI
List of Tables XVI
List of Figures XVIII
List of Algorithms XIX
List of Abbreviations XXI
Table of Contents XXV
1 INTRODUCTION 1
1.1 Context . . . 1
1.2 Summary of the contributions . . . 7
1.3 Publications and other scientific realisations . . . 8
1.4 Methodology . . . 10
1.5 Outline . . . 13

I BACKGROUND 15
2 CONSTRAINT PROGRAMMING (CP) 17
2.1 Overview of Constraint Programming . . . 17
2.2 Propagators . . . 18
2.3 Global constraints . . . 19
2.3.1 The `minimumAssignment` constraint . . . 21
2.3.2 The `cost-gcc` constraint . . . 22
2.4 Search . . . 23
2.5 Cost-based propagator in the OscaR solver . . . 25

3 CAPACITATED LOT SIZING PROBLEM (CLSP) 27
3.1 Overview of the Lot Sizing problem . . . 27
3.1.1 Characteristics of the Lot Sizing problem . . . 27
3.1.2 Classification of the Lot Sizing problem . . . 28
3.2 Related works . . . 29
3.3 A variant of the CLSP . . . 31
3.3.1 A CP model for the PSP . . . 32
3.3.2 MIP formulation for the PSP . . . 33
4 CONSTRAINED ARBORESCENCE PROBLEM (CAP) 37
4.1 Overview of Minimum Weight Arborescence (MWA) . 37
4.1.1 Conditions for optimality of the MWA . . . 39
4.1.2 Computation of an MWA . . . 40
4.2 Related works . . . 43
4.3 A variant of the CAP . . . 45

II FILTERING ALGORITHMS FOR A CAPACITATED LOT SIZING PROBLEM 49
5 THE STOCKINGCOST CONSTRAINT 51
5.1 Introduction . . . 51
5.2 The StockingCost constraint . . . 52
5.2.1 Decomposing the constraint . . . 54
5.3 Filtering of the cost variable H . . . 55
5.4 Pruning the decision variables X_i . . . 58
5.5 A complete filtering algorithm in $O(n)$ . . . 64
5.6 Experimental results . . . 66
5.7 Conclusion . . . 70
6 THE ITEM DEPENDENT STOCKINGCOST CONSTRAINT 73
6.1 Introduction . . . 73
6.2 The Item Dependent StockingCost constraint . . . 74
6.2.1 Decomposing the constraint . . . 75
6.3 Filtering of the cost variable H . . . 76
6.4 Pruning the decision variables X_i . . . 84
6.5 Experimental results . . . 92
6.6 Conclusion . . . 98

III FILTERING ALGORITHMS FOR THE CONSTRAINED ARBORESCENCE PROBLEM 99
7 THE WEIGHTED ARBORESCENCE CONSTRAINT 101
7.1 Introduction . . . 101
7.2 The MinArborescence constraint . . . 103

7.2.1 Decomposing the constraint 104
7.3 Improved Reduced Costs 106
7.4 Experimental results 115
7.5 Conclusion 120

Conclusion and Perspectives 121

IV APPENDIX 127
A CP APPROACH RESULTS FOR THE PSP 129
B CP APPROACH RESULTS FOR THE RMWA PROBLEM 133
C CODING EXPERIENCE AND SOURCE CODE 137

BIBLIOGRAPHY 161

1 INTRODUCTION

This thesis aims to provide some global optimization constraints for solving the Capacitated Lot Sizing Problem (CLSP) and the Constrained Arborescence Problem (CAP) by Constraint Programming (CP).

1.1 CONTEXT

Constraint Programming (CP) is a paradigm that is effective in tackling some hard combinatorial problems[1] [RBW06, DSH90, Hen89]. One of the main processes in CP is propagation. Constraint propagation involves removing from the domain of variables, values that cannot appear in any consistent solution of the problem. However, as mentioned in [Foc+99], pure constraint propagation techniques have shown their limitations in dealing with objective functions. For optimization problems, it is interesting to have constraints that filter the variable representing the objective value and also the decision variables mainly based on some operational research (OR) rules. In optimization problems, the optimization constraints are constraints linked to (a part of) the objective function to efficiently filter inconsistent values from a cost-based reasoning. This kind of propagation is called cost-based propagation [FLM99b]. In CP, optimization constraints have been developed for propagation in a variety of optimization problems (see for example [FLM99b, DCP16, Ben+12, Ró2, BLPN12, CL97a, CL95, RWH99]).

1 A combinatorial problem requires to take discrete decisions that respect some constraints and, if required, optimize an objective function.

This thesis deals with filtering algorithms for two complex optimization problems: the CLSP and the CAP. Curiously, prior to our research (see [Hou13, HSW14, Hou+14, Hou+15, Hou+17b, Hou+17a]), these optimization problems have apparently not been tackled by CP.

CAPACITATED LOT SIZING PROBLEM. In production planning, one of the most important and difficult tasks is Lot Sizing (LS) [KGW03, AFT99]. A manufacturing firm's ability to compete in the market depends directly on decisions making in LS [KGW03]. The CLSP is a well-known version of LS. The CLSP requires one to determine a minimum cost (stocking cost and/or setup costs and/or changeover costs, etc.) production schedule to satisfy the demands for single or multiple items without exceeding machine capacities. In this thesis, we focus on the case in which 1) the planning horizon is discrete and finite, 2) the demand is deterministic that is it is known in advance, 3) the production of an order consumes one unit of capacity, and 4) an order is produced in one go i.e one cannot produce a fraction of an order.

In particular, we consider the multiple items capacitated lot sizing problem with sequence-dependent changeover costs called the Pigment Sequencing Problem (PSP) by [PW05]. For the PSP, there is a single machine with capacity limited to one unit per period. There are item-dependent stocking costs and sequence-dependent changeover costs: 1) the total stocking cost of an order is proportional to its stocking cost and the number of periods between its due date and the production period; 2) the changeover cost is induced when passing from the production of an item to another one. More precisely, consider n orders (from $m \leq n$ different items[2]) that have to be scheduled over a discrete time horizon of T periods on a machine that can produce one unit per period. Each order $p \in [1, \ldots, n]$ has a due date d_p and a stocking (storage) cost $h_{\mathcal{I}(p)} \geq 0$ (in which $\mathcal{I}(p) \in [1, \ldots, m]$ is the corresponding item of the order p). There is a changeover cost $q^{i,j} \geq 0$ between each pair of items (i, j) with $q^{i,i} = 0, \forall i \in [1, \ldots, m]$. Let $successor(p)$ be the order produced just after producing the order p. One wants to associate to each order p a production period $date(p) \in [1, \ldots, T]$ such that each order is produced on or before its due date ($date(p) \leq d_p, \forall p$), the capacity of the production is respected ($|\{p \mid date(p) = t\}| \leq 1, \forall t \in [1, \ldots, T]$), and the total stocking costs

2 item: order type.

and changeover costs $(\sum_p (d_p - date(p)) \cdot h_{\mathcal{I}(p)} + \sum_p q^{\mathcal{I}(p),\mathcal{I}(successor(p))})$ are minimized. A small instance of the PSP is described next.

Example 1. *Two types of orders (1 and 2) must be produced over the planning horizon* $[1,\ldots,5]$ *:* $m = 2$ *and* $T = 5$*. The stocking costs are respectively* $h_1 = 5$ *and* $h_2 = 2$ *for each item. The demands for item 1 are* $d^1_{t\in[1,\ldots,5]} = [0,1,0,1,0]$ *and for the second item are* $d^2_{t\in[1,\ldots,5]} = [0,0,1,0,1]$*. Thus the number of orders is* $n = 4$*, two for each item. The changeover costs are:* $q^{1,2} = 10$*,* $q^{2,1} = 5$ *and* $q^{1,1} = q^{2,2} = 0$*. The solution* $S_1 = [1,2,0,1,2]$ *(represented in* Figure 1.1*) is a feasible solution. This means that the item*

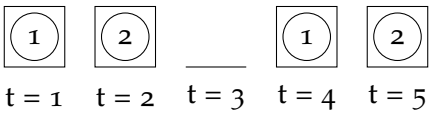


Figure 1.1: A feasible solution of the PSP instance of Example 1

1 *will be produced in periods* 1 *and* 4 *while the item* 2 *will be produced in periods* 2 *and* 5*. Period* 3 *is an idle period*[3]*. The cost associated to* S_1 *is* $C_{s_1} = h_1 + h_2 + q^{1,2} + q^{2,1} + q^{1,2} = 32$*. The optimal solution for this problem is* $S_2 = [2,1,0,1,2]$ *(represented in* Figure 1.2*) with cost* $C_{s_2} = 19$*.*

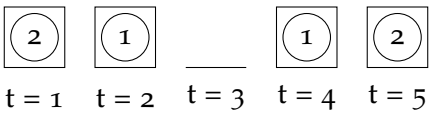


Figure 1.2: An optimal solution of the PSP instance of Example 1

In this thesis, we propose a global constraint called `StockingCost` to handle the stocking costs and fast filtering algorithms.

As concerns the changeover costs part of the PSP, it can be modeled as a classical Asymmetric Traveling Salesman Problem (ATSP) in which the cities to be visited represent the orders and the distances between them are the corresponding changeover costs. One can use some relaxations[4] to solve the ATSP in CP. A classical relaxation is the *assignment* relaxation that allows sub-tour(s) in a solution [FLM02]. In [Ben+12], Benchimol et al. use the *1-tree*[5] relaxation that removes the degree constraints (that enforce the degree of two at each vertex) to solve the TSP.

3 idle period: period in which there is no production.

4 the problem without some constraints.

5 A *1-tree* is defined as a spanning tree on the subgraph induced by the set of vertices $V \setminus \{1\}$, together with two distinct edges incident to node 1 [Ben+12].

The directed version of *1-tree* relaxation can also be used to solve the ATSP. The second part of this thesis is devoted to the problem associated to this relaxation.

CONSTRAINED ARBORESCENCE PROBLEM. In graph theory, the problem of finding a Minimum Spanning Tree (MST) [GH85] is one of the most well-known problems on undirected graphs. The corresponding version for directed graphs is the Minimum Directed Spanning Tree (MDST) or the Minimum Weight Arborescence (MWA) problem. An arborescence A rooted at a given vertex r is a directed spanning tree rooted at r. In other words, A is such that there is a path from r to every other vertex without cycle. An MWA is an arborescence of minimum cost. More formally, consider a weighted directed graph $G = (V, E)$ in which V is the vertex set and $E \subseteq \{(i, j) \mid i, j \in V\}$ is the edge set. A weight $w(i, j)$ is associated to each edge $(i, j) \in E$. Given a vertex r, the aim is to associate to each vertex $v \in V \setminus \{r\}$ exactly one vertex $p(v) \in V$ (with $(p(v), v) \in E$) such that, considering the sub-graph $A = (V, F)$ with $F = \{(p(v), v) \mid v \in V \setminus \{r\}\}$, there is no cycle and the total cost ($\sum_{v \in V \setminus \{r\}} w(p(v), v)$) is minimized. Figure 1.3 shows a weigthed directed graph (denoted G_1) and its MWA $A(G_1)^\star$ represented with dashed edges.

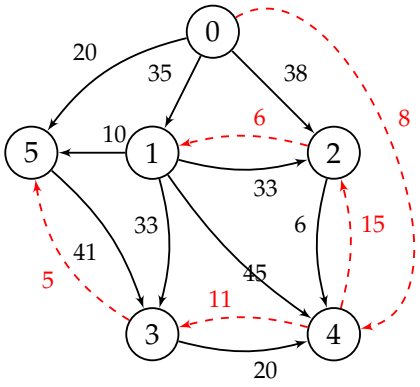


Figure 1.3: Graphe G_1

It is well known that graphs are good structures to model some real life problems. The MWA problem has many practical applications (for example in the design of distribution and telecommunication networks [FV97]). It is also a relaxation of the well-known ATSP problem [FT92].

Solving an MWA problem is relatively easy (it is solvable in polynomial time). In real world applications, there are generally other side constraints that make the problems much more difficult. Similarly to the constrained spanning tree [DK97a] problem on an undirected graph, the CAP is the problem that requires one to find an arborescence that satisfies other side constraints and is of minimum cost. To handle the CAP in CP, we propose an optimization constraint for this optimization problem denoted `MinArborescence`.

Concerning the MWA problem, a basic $O(|V|^2)$ optimization algorithm is available. This algorithm also provides a lower bound (Linear Programming (LP) reduced cost) on the cost of modifying the optimal solution by forcing an edge to be in the optimal solution. We propose an algorithm that runs also in $O(|V|^2)$ time to improve the LP reduced costs in some cases. We perform filtering on the basis of these different costs. The resulting global constraint is applied to the constrained arborescence problem in which the use of an edge $(i, j) \in E$ consumes a certain amount of resources $a_{i,j}$ and there is a limited resource b_i available at each vertex $i \in V$, the Resource constrained Minimum Weight Arborescence (RMWA). Formally, there is the following additional constraints: $\sum_{(i,j)\in\delta_i^+} a_{i,j} \cdot x_{i,j} \leq b_i, \forall i \in V$ in which δ_i^+ is the set of outgoing edges from i, $x_{i,j} = 1$ if $(i, j) \in A(G)$ and $x_{i,j} = 0$ otherwise.

Example 2. *Consider the graph G_1 with additional data concerning the RMWA problem. In* Figure 1.4*, the label of the edge (i, j) is $w(i, j)(a_{i,j})$ and the vertex i is followed by its resource b_i, $i(b_i)$. One can see that with the additional constraints $\sum_{(i,j)\in\delta_i^+} a_{i,j} \cdot x_{i,j} \leq b_i, \forall i \in V$ on this instance, the previous MWA (represented with dashed edges) is not valid because at the vertex 4, $a_{4,2} + a_{4,3} = 15 > b_4$. A feasible solution of this RMWA problem is represented in dashed in* Figure 1.5.

ABOUT THE STATE-OF-THE-ART APPROACHES. Both `StockingCost` and `MinArborescence` constraints can be decomposed into standard or existing constraints. The experimental comparisons show that our new filtering algorithms outperform the decompositions of each constraints. It is well known that CP is the combination of two key ingredients: CP = *Filtering* + *Search*. In this thesis we focused only on some filtering aspects of the stocking cost and arborescence problems. We use a naive search procedure to ease the filtering comparisons in most of our experimentation. We did not spend any effort on the search aspects. To give a chance to CP to be competitive

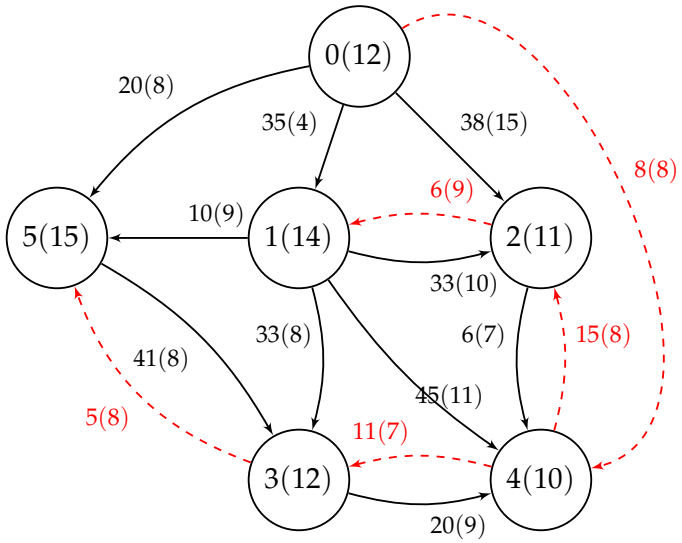


Figure 1.4: Graphe G_1 with $a_{i,j}$ and b_i

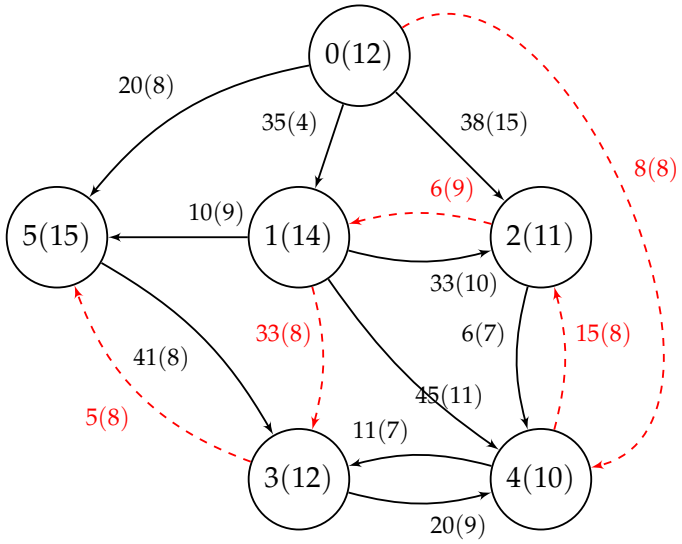


Figure 1.5: Graphe G_1 with $a_{i,j}$ and b_i: a feasible solution of the RMWA problem associated

with the state-of-the-art approaches (e.g. MIP [Wol98, WN99]) on the experimented problems, one should also develop the search aspects. For instance, use LNS [Sha98] to drive the search quickly toward good solutions or develop custom heuristics. Note that it is easier to model a problem in CP than other approaches and our new global constraints can be used in many variants of the problems.

1.2 SUMMARY OF THE CONTRIBUTIONS

The main contributions of this thesis are enumerated below for each of the two problems studied.

Capacitated Lot Sizing Problem

- The introduction of the CLSP in CP.
- The proposition of new models for the CLSP in CP.
- The definition of the `StockingCost` constraint for the CLSP when the orders to be produced have the same per period stocking cost.
- The introduction of new filtering algorithms for the `StockingCost` constraint.
- An implementation of these filtering algorithms available in the OscaR solver [Osc12] and an experimental evaluation of the propagator associated.
- The definition of the Item Dependent StockingCost constraint (denoted `IDStockingCost`) for the CLSP when the orders to be produced have different per period stocking costs. This constraint can be used to tackle any variant of the CLSP in which there are stocking costs as part of the objective function.
- The introduction of new filtering algorithms for the `IDStockingCost` constraint.
- An implementation of these filtering algorithms available in the OscaR solver and an experimental evaluation of the propagator associated.
- New benchmarks of a variant of the CLSP.

Constrained Arborescence Problem

- The introduction of the CAP in CP.
- The proposition of new models for the CAP in CP.
- The definition of the minimum arborescence constraint (denoted `MinArborescence`) to tackle the CAP in CP.
- New sensitivity analysis of the minimum weighted arborescence.
- The introduction of new filtering algorithms for the `MinArborescence` constraint.
- An implementation of these filtering algorithms available in the OscaR solver and an experimental evaluation of the propagator associated.
- New benchmarks of a variant of the CAP.

1.3 PUBLICATIONS AND OTHER SCIENTIFIC REALISATIONS

Each of our main results has been submitted to a refereed international conference/journal and we have presented some of these results at conferences. We have also made available some benchmarks about the CLSP and the CAP as well as toolkits (using the OscaR solver [Osc12]) to solve some variants of these problems by CP.

Full papers submitted to refereed international conferences/journal

- Vinasetan Ratheil Houndji, Pierre Schaus, Laurence Wolsey and Yves Deville. `The StockingCost Constraint`. Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, Springer International Publishing, 2014, Volume 8656, pp. 382-397.
- Vinasetan Ratheil Houndji, Pierre Schaus, Mahouton Norbert Hounkonnou and Laurence Wolsey. `The Weighted Arborescence Constraint`. Accepted. To appear in CPAIOR'2017 (Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming).

- Vinasetan Ratheil Houndji, Pierre Schaus and Laurence Wolsey. `The Item Dependent StockingCost Constraint`. Submitted to the journal Constraints. Under review.

National conferences and other scientific realisations

- Vinasetan Ratheil Houndji, Pierre Schaus and Laurence Wolsey. `CP Approach for the Multi-Item Capacitated Lot-Sizing Problem with Sequence-Dependent Changeover Costs`. 28th annual conference of the Belgian Operational Research Society, Mons, Belgium, 2014. pp. 145-146.

- Vinasetan Ratheil Houndji, Pierre Schaus, Laurence Wolsey and Yves Deville. `La contrainte globale StockingCost pour les problèmes de planification de production`. Journées Francophones de Programmation par Contraintes. Bordeaux, France, 2015. pp. 128-129.

- Vinasetan Ratheil Houndji, Pierre Schaus, Mahouton Norbert Hounkonnou and Laurence Wolsey. `La contrainte globale MinArborescence pour les problèmes d'arborescence de poids minimum`. Journées Francophones de Programmation par Contraintes. Bordeaux, France, 2017.

- Vinasetan Ratheil Houndji, Pierre Schaus, Laurence Wolsey and Yves Deville. `CSPLib Problem 058: Discrete Lot Sizing Problem. CSPLib`: A problem library for constraints. Available from `http://www.csplib.org/Problems/prob058`

- Vinasetan Ratheil Houndji and Pierre Schaus. `CP4PP: Constraint Programming for Production Planning`. Toolkit (using the OscaR solver as a dependency) and some instances on a variant of the CLSP. Available from `https://bitbucket.org/ratheilesse/cp4pp`

- Vinasetan Ratheil Houndji and Pierre Schaus. `CP4CAP: Constraint Programming for Constrained Arborescence Problem`. Toolkit (using the OscaR solver as a dependency) and some instances on a variant of the CAP. Available from `https://bitbucket.org/ratheilesse/cp4cap`

1.4 METHODOLOGY

We describe the main steps followed to develop and evaluate our new filtering algorithms for each of the new global constraints proposed in this thesis (`StockingCost`, `IDStockingCost` and `MinArborescence`).

Theoretical analysis

First, we formally define the given global constraint, clearly state when it is applicable and give some alternatives with the state-of-the-art CP constraints. Then we characterize the optimal solutions of the associated problem and perform sensitivity analysis by giving some relevant definitions, properties and propositions. Based on the latter, we propose some filtering rules and associated filtering algorithms. All non trivial properties/propositions are proved as well as the correctness of the algorithms. Finally, we analyze the time complexity of the proposed algorithms.

Experimental evaluation

Each of our new filtering algorithms is tested and evaluated experimentally. The implementations and tests have been realized within the OscaR open source solver [Osc12]. All experiments were conducted on a 2.4 GHz Intel core i5 processor using OS X 10.11.

1. Code testing.
   Each of our new filtering algorithms is compared wrt at least one alternative (with some state-of-the-art constraints) on small random instances with a static search heuristic. The number of solutions of each model must be the same, to check that our new filtering algorithms do not remove any consistent solutions. Of course, these tests do not guarantee the absence of bugs but they make us more confident about the correctness of the implementations.

2. Comparision wrt filtering and time.
   The evaluation of a given global constraint uses the methodology described in [VCLS15]: the search tree with a baseline model M is recorded and then the gains are computed by revisiting the search tree with stronger alternative filtering $M \cup \phi$.

Thus each model visits the same search space (in the same order of visited nodes) but the model $M \cup \phi$ skips the nodes removed by the additional filtering of ϕ. This approach allows us to use dynamic search heuristics without interfering with the filtering. The analysis is based on the arithmetic average number of nodes visited and the time needed to complete the recorded search space. We also compare the geometric average gain factors of each model wrt the baseline. In addition, to refine the analysis, we present some performance profiles.

For a given propagator, the performance profile [DM02] provides its cumulative performance wrt the best propagator on each instance. More formally, consider a set of propagators $\mathcal{F}$ and a set of instances $\mathcal{I}$. Let r_p^i be the performance ratio of the propagtor $p \in \mathcal{F}$ wrt to the best performance by any propagator in $\mathcal{F}$ on the instance i:

$$r_p^i = \frac{m_p^i}{\min\{m_f^i \mid f \in \mathcal{F}\}} \tag{1.1}$$

in which m_p^i is the measure of the performance metric (time or number of nodes visited) for the propagator p on the instance i. Then

$$\rho_p(\tau) = \frac{1}{|\mathcal{I}|} \cdot |\{i \in \mathcal{I} \mid r_p^i \leq \tau\}| \tag{1.2}$$

is the proportion of instances for which the propagator $p \in \mathcal{F}$ has a performance ratio r_p^i within a factor $\tau \in \mathbb{R}$ of the best possible ratio. In other words, for a point (x, y) on the performance profile, the value $(1 - y)$ gives the percentage of the instances where the given propagator is at least x times worse than the best propagator on each instance.

Example 3. *We want to compare three models M_1, M_2 and M_3 that have different propators on a same problem.* Table 1.1 *provides the different performance measures on 5 instances.*

One can deduce the different performance ratios by instance/-model using the formula 1.1 (see Table 1.2). Based on the different ratios provided by Table 1.2, Table 1.3 reports the different cumulative performances using the formula 1.2.

Figure 1.6 shows the coresponding performance profile. On this figure, one can see that M_3 is the best model for 60% of the instances, M_2 is the best for 40% of the instances and M_1 is the

Instance	M_1	M_2	M_3
1	12	8	**4**
2	**5**	**5**	6
3	18	6	**3**
4	10	**2**	8
5	21	8	**5**

Table 1.1: Example: performance profile - measures

Instance	M_1	M_2	M_3
1	3	2	**1**
2	**1**	**1**	1.2
3	6	2	**1**
4	5	**1**	4
5	4.2	1.6	**1**

Table 1.2: Example: performance profile - performance ratios r_p^i

τ	M_1	M_2	M_3
1	0.2	0.4	0.6
1.2	0.2	0.4	0.8
1.6	0.2	0.6	0.8
2	0.2	1	0.8
3	0.4	1	0.8
4	0.4	1	1
4.2	0.6	1	1
5	0.8	1	1
6	1	1	1

Table 1.3: Example: performance profile - cumulative performance $\rho_p(\tau)$

best for 20% of the instances. Clearly, M_1 is the worst model. M_1 is ≥ 3 times worse than the best model for 60% of the instances. Note that M_2 is never > 1.6 times worse than the other models and M_3 is ≤ 1.2 times worse than the other models for 80% of the instances.

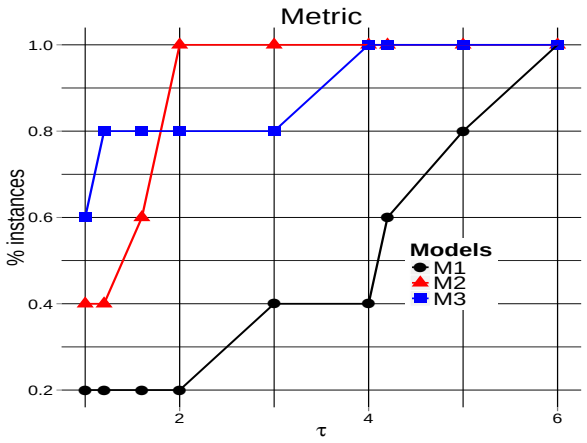


Figure 1.6: Performance profile - Example

1.5 OUTLINE

The remainder of this thesis is organized in three main parts. The first part (Part I) gives background and theoretical preliminaries about the topics addressed in this thesis. Part I contains three chapters: Chapter 2 focuses on CP, Chapter 3 focuses on the LS problem (and the CLSP) and Chapter 4 focuses on the MWA (and the CAP). The second part (Part II) regroups our contributions about the CLSP. Part II is divided into two chapters, each one is dedicated to some specialized filtering algorithms for the stocking costs arising in many variants of the CLSP. Chapter 5 introduces the `StockingCost` constraint for the CLSP when the per period stocking cost of all orders is the same and Chapter 6 describes the `Item Dependent StockingCost` constraint for the CLSP when the per period stocking costs can be different between orders. The last part (Part III), that contains a single chapter, is our contributions about the CAP. In Part III, Chapter 7 presents the optimization version of the arborescence constraint (denoted `MinArborescence`) for the CAP and new sensitivity analysis of the MWA. Finally, we conclude and give possible directions for future works.

Part I

BACKGROUND

2 CONSTRAINT PROGRAMMING (CP)

In this chapter, we give some preliminary notions about Constraint Programming (CP) and constraint propagation for optimization problems. The chapter is organized as follows: Section 2.1 gives an overview of CP; Section 2.2 defines propagation in CP; Section 2.3 shows the importance of the global constraints and cost-based filtering; Section 2.4 describes how search is performed in CP and Section 2.5 gives a skeleton of a cost-based propagator in the OscaR solver.

2.1 OVERVIEW OF CONSTRAINT PROGRAMMING

CP allows a declarative statement of the problem with a set of variables and a set of constraints between variables. Typically, CP involves finding a value for each one of a set of problem variables subject to constraints specifiying that some subsets of values cannot be used together [FM06]. By constraint, we mean a mathematical relation between the values that can be assigned to the variables involved. We can distinguish two classes of CP problems: Constraint Satisfaction Problems (CSP) and Constraint Optimisation Problems (COP). A CSP [FM06, MS98, Tsa95, Apt03] is a triple $< X, D, C >$ in which $X = \{X_1, X_2, \ldots, X_n\}$ is the set of n variables involved in the problem, $D = \{D_1, D_2, \ldots, D_n\}$ is the set of domains[1] of each variable X_i and $C = \{C_1, C_2, \ldots, C_m\}$ is the set of m constraints that should be respected in every solution. A solution of a CSP is an assignment of a value $v_i \in D_i$ to each variable $X_i \in X$ such that all constraints are satisfied. A COP is a CSP with an objective

1 The domain of a variable is the set of values that can be assigned to this variable.

function z. A numerical value $z(s)$ is associated to every consistent solution s of the problem and the goal is to find the best (minimum or maximum) consistent solution wrt the objective function.

2.2 PROPAGATORS

Consider a brute force algorithm that enumerates all the possible assignments of the variables with respect to their respective domains without any processing. In this case, the size of the search space of the given CSP is exponential in number of variables of the problem. CP reduces this size by performing some filterings based on the constraints and the associated propagators. Propagators are also known as filters implemented by some filtering algorithms and narrowing operators [Beng6]. A propagator is associated to a single constraint or a set of constraints. Note that different propagators (with different filtering algorithms) can be developed for the same (set of) constraint(s). The role of a propagator is to filter some (ideally all) inconsistent values from the domains of the variables. A couple variable/value (x, v) is inconsistent wrt a constraint c iff there is no solution with $x = v$ that respects the constraint c. For example, consider two variables $X_1 = \{2, 3, 5\}$ and $X_2 = \{1, 2, 4\}$ and the constraint $X_1 < X_2$. We can see that $X_1 = 5$, $X_2 = 1$ and $X_2 = 2$ are inconsistent wrt $X_1 < X_2$. After the filtering, the new domains of variables are respectively: $X_1 = \{2, 3\}$ and $X_2 = \{4\}$. A propagation engine, using a version of `fix-point` algorithm (see Algorithm 2.2.1), coordinates the execution of propagators in order to deliver constraint propagation for a collection of constraints [SCo6]. This process is repeated until a `fix-point` is reached i.e there are no more values that can be removed. We present, in Algorithm 2.2.1 [Sch15], the principle of a `fix-point` algorithm.

The amount of filtering of each constraint depends on the consistency level of the associated propagator. The main consistency in CP is Domain Consistency (DC) or Generalized Arc consistency (GAC) [Mac77]. If we individually consider each constraint, GAC is the best filtering that can be achieved. A propagator achieves GAC for a constraint iff all inconsistent values (wrt that constraint) are removed from the domains of the variables involved. Formally, let $vars(c)$ denote the set of variables involved in the constraint c. A propagator achieves GAC for a constraint iff $\forall X_i \in vars(c), \forall v \in D_i, \forall X_j \in vars(c) \setminus \{X_i\} : \exists v_j \in D_j$ such that c holds. For some constraints, GAC could be achieved in reasonable time. Unfortunately, this is not always the case. An-

Algorithm 2.2.1: Principle of a `fix-point` algorithm

```
repeat
    Select a constraint c.
    if c is consistent wrt the domains of variables then
        Apply filtering algorithms of c.
    end
    else
        Stop. There is no solution.
    end
until no value can be removed.
```

other well-known consistency level is Bound Consistency (BC). This latter, sometimes, offers a good trade-off between speed and filtering. For BC, the domain of each variable $X_i \in X$ is relaxed to $D_i = [\min\{v \in D_i\}, \ldots, \max\{v \in D_i\}]$. A propagator achieves BC for a constraint iff $\forall X_i \in vars(c), \forall v \in \{\min\{v \in D_i\}, \max\{v \in D_i\}\}, \forall X_j \in vars(c) \setminus \{X_i\} : \exists v_j \in [\min\{v \in D_j\}, \ldots, \max\{v \in D_j\}]$ such that c holds. There are some other consistency levels. We refer to [Beso6] for more information about the formal consistency levels and propagators in CP. It is worth noting that, some propagators use filtering algorithms whose consistency levels cannot easily be characterised, but that offer relatively good filtering.

2.3 GLOBAL CONSTRAINTS

A global constraint is a constraint on a non fixed number of variables. Such constraint is semantically redundant since the same relation can be expressed as the conjunction of several simpler constraints [HKo6]. A conjunction of simpler constraints that is semantically equivalent to a global constraint is often called decomposition of this global constraint. A global constraint is useful to better filter inconsistent values and/or to be more efficient in time than its different decompositions.

A well-known global constraint is the `alldifferent` constraint [Rég94, Pug98, LO+03, Hoe01]. The `alldifferent` constraint holds if and only if each variable involved in the constraint takes a different value. Consider the constraint $\texttt{alldifferent}(X_1, X_2, X_3)$ with $D_1 = \{1, 2\}$, $D_2 = \{1, 2\}$ and $D_1 = \{1, 2, 3\}$. An equivalent decomposition of this constraint is $(X_1 \neq X_2) \wedge (X_2 \neq X_3) \wedge (X_1 \neq X_3)$.

This set of three simple constraints can not filter anything by tacking each constraint individually. However the global constraint `alldifferent`(X_1, X_2, X_3) can see that $X_3 = 1$ and $X_3 = 2$ are inconsistent since $\{1, 2\}$ must be reserved for X_1 and X_2. The filtering for the `alldifferent` constraint is mainly based on *Hall interval* theorem [Hal35] (see for example [Pug98, LO+03]) or matching theory (see for example [Rég94, MT00]). Other variants of the `alldifferent` constraint exist (see for example [Ŕ99, CL97a, FLM99a, Bes+11]). A generalization of the `alldifferent` constraint is the `global cardinality` constraint (`gcc`) [Rég96, Qui+03, KT05a] that enforces the (minimal and/or maximal) number of occurences of each value.

We refer to [HK06, Ŕ11, BMR12] for more information about global constraints.

Concerning the optimization problems, it is interesting to design constraints that are linked to a part of the objective function. Such constraints are called optimization constraints. The propagators associated to these constraints are based on the different costs involved. The cost-based filtering algorithms are more efficient than pure CP filtering for optimization problems. In general, they use the following pieces of information [FLM99b]:

1. an optimal solution of a related relaxed (less constrained) problem;
2. the optimal value of this solution. This value is a lower bound on the original problem objective function. It is used to 1) check the consistency of the constraint and 2) filter the objective variable;
3. an optimistic evaluation of the cost increase if a value $v \in D_i$ is assigned to the variable X_i - to filter decision variables.

The global optimization constraints used for the experiments in this thesis are: the `minimumAssignment` constraint [Foc+99, CL97a], the `mincircuit` constraint [CL97b, CL97a, FLM02, Pes+98, Ben+12] and the `knapsack` constraint [FS02, Sel03, Tri03, DH80]. Below we describe the `minimumAssignment` constraint and the `cost-gcc` constraint, two global optimization constraints related to the new constraints proposed in this thesis.

2.3.1 *The* `minimumAssignment` *constraint*

Consider n variables in the set $V_1 = \{V_1^1, \ldots, V_1^n\}$ and n values in the set $V_2 = \{V_2^1, \ldots, V_2^n\}$ with a cost $c(i,j), \forall i,j \in [1,\ldots,n]$ associated to each pair (V_1^i, V_2^j). The Assignment Problem (AP) [DM97, CMT88] requires one to associate to each variable V_1^i a single value V_2^j such that the assigned values are different and the total cost of the assignment is minimal. The problem can be formulated as follows:

Formulation 1.

$$Z^{opt} = \min \sum_{(i,j)} c(i,j) \cdot x_{i,j} \tag{2.1}$$

$$(AP) \qquad \sum_{j \in V_2} x_{i,j} = 1, \forall i \in V_1 \tag{2.2}$$

$$\sum_{i \in V_1} x_{i,j} = 1, \forall j \in V_2 \tag{2.3}$$

$$x_{i,j} \in \{0,1\}, \forall (i,j) \tag{2.4}$$

in which the decision variable $x_{i,j} = 1$ when the value V_2^j is associated to the variable V_1^i and $x_{i,j} = 0$ otherwise. The constraint 2.2 (resp. 2.3) ensures that one and only one value in V_2 (resp. in V_1) is associated to each V_1^i (resp. V_2^j).

The `minimumAssignment` constraint has the following signature: `minimumAssignment`$([x_1,\ldots,x_n], C, Z)$ in which $x_i, \forall i \in [1,\ldots,n]$ are the decision variables, C is a cost function associating to each $j \in D_i, \forall i$ the cost to assign the value j to the variable x_i and Z is an upper bound on the total cost of the assignment. This constraint holds if there is a valid assignment with cost $\leq Z$. Note that, for this constraint, the number m of possible values to assign must be greater than or equal to n: $m \geq n$.

The well-known Hungarian algorithm [Kuh10, CMT88] can solve the AP in $O(n^2 m)$. The optimal cost returned by the Hungarian algorithm allows one to check the consistency of the `minimumAssignment` constraint and filter the objective variable: $Z \geq Z^{opt}$. The Hungarian algorithm also provides the LP reduced costs $\bar{c}(i,j)$ for each pair (i,j) not in the optimal solution. The filtering of the variables can be carried out based on the reduced costs i.e if $Z^{opt} + \bar{c}(i,j) > Z$ then $X_i \leftarrow j$ is inconsistent. Note that the Hungarian algorithm allows each re-computation of the optimal solution for the AP, needed in the case of modification of one value in the cost matrix, to be efficiently obtained

in $O(nm)$ [FLM99b]. Then the `minimumAssignment` constraint can be implemented incrementally: the first filtering is achieved in $O(n^2m)$ and the different updates during the search can be done in $O(nm)$.

Assume that, in the optimal solution for the AP, we have: $X_i \leftarrow k$ and $X_l \leftarrow j$. If we assign j to X_i ($X_i \leftarrow j$) then l and k must be re-assigned As in [FLM99b], let PTH be a minimum augmenting path from l to k and $cost(PTH)$ be its cost. Thus the cost of the optimal solution for the AP in which $X_i \leftarrow j$ is $Z^{opt} + \bar{c}(i,j) + cost(PTH)$ [FLM99b]. Focacci et al., in [Foc+99], present a way to improve the LP reduced costs by computing a bound of $cost(PTH)$ in $O(n^2)$. However, the experimental results presented in [Foc+99, FLM99b] suggest that the filtering based on the improved reduced costs does not significantly reduce the search space wrt the one based on the LP reduced costs.

Further, Ducomman et al. [DCP16] show that one can obtain the exact reduced costs from shortest paths in the residual graph. These costs can be obtained by executing Johnson's algorithm [Cor+01a] for all pairs in $O(n^2m)$ using a Fibonacci heap to implement Dijkstra algorithm for shortest path computation.

2.3.2 *The* `cost-gcc` *constraint*

In [Ró2], Régin introduces the `cost-gcc` constraint, a generalization of the `minimumAssignment` constraint. This global optimization constraint has the following form: `cost-gcc`$(\{X_1,\ldots,X_n\},\{[l_1,u_1],\ldots,[l_d,u_d]\},z,w)$ in wich:

- X is the variable set;
- $[l_k,u_k]$ is an interval, $\forall k \in V = \{v_1,\ldots,v_d\}$ with V the set of possible values that the variables can take;
- z is an upper bound of the cost of the assignment;
- w is the cost associated to each assignment: $w(i,k)$ is the cost incurred when the value k is assigned to the variable X_i.

Note that the `cost-gcc` constraint is the `gcc` constraint with costs.

The `cost-gcc` constraint holds when the number of times a variable takes the value $k \in V$ is in the interval $[l_k, u_k]$ and the total cost of the assignment is less than z:

$$|\{i \mid X_i = k\}| \geq l_k, \forall k \in V \tag{2.5}$$

$$|\{i \mid X_i = k\}| \leq u_k, \forall k \in V \tag{2.6}$$

$$\sum_{i,k} (X_i = k) \cdot w(i,k) \leq z \tag{2.7}$$

Consistency checking for this constraint can be achieved based on finding a minimum cost flow in a special directed graph called *value network* of `cost-gcc` (see [Ró2]). One can check the consistency for the `cost-gcc` constraint in $O(n \cdot S(m, n+d, \gamma))$ in which m is the number of arcs in the *value network* and $S(m, n+d, \gamma)$ is the complexity of the search for shortest paths from a node to every node in a graph with m arcs and $n+d$ nodes with a maximal cost γ [Ró2]. Régin, in [Ró2], presents an algorithm to achieve GAC for the `cost-gcc` constraint in $O(\Delta \cdot S(m, n+d, \gamma))$ (with $\Delta = \min\{n, d\}$) and also studies the incremental aspects of the algorithm. To the best of our knowledge, the arc-consistent `cost-gcc` constraint has never been implemented in a CP solver.

2.4 SEARCH

In general, the propagation is not sufficient to find a solution of a CSP. Actually, after a `fix-point` is reached, there are three possible cases:

1. a domain of at least one variable is empty: the corresponding CSP has no solution;
2. each variable has a single consistent value in its domain: a solution is found;
3. at least one variable has at least two values in its domain. Then the current CSP is simplified by splitting a domain of a variable. This is called branching.

A CSP is solved by decomposing the problem into smaller CSP until all the subproblems reach the case 1 (the problem does not have a solution) or one subproblem reaches the case 2. Note that when a branching decision is taken, propagation is applied on every new subproblem (until a `fix-point` is reached). The solutions are found

by interleaving propagation and branching decisions. This is why CSP is also defined as *Filtering* + *Search*. Search defines how the search space is explored with the alternatives done by the branching decisions. The classical branching decision consists to: first select the variable (with the domain to split) and then choose how to split. This is called variable-value heuristic.

It is interesting to design a specialized search heuristic (for branching decisions) based on the context of the problem in order to quickly have a consistent (and good for an optimization problem) solution. We refer to [Bee06] for more information about branching strategies and search. To traverse the entire search tree, CP solver mainly uses Depth First Search (DFS). DFS explores the tree from the left of the tree to the right and backtracks to the next unexplored branch when the current small CSP has no solution. Note that, when backtrack occurs, some data structures must restore their respective previous values. This state restoration can be achieved by 1) copying, 2) trailing (information to reconstruct the state is stored in a trail prior to change it), or 3) recomputing the state from scratch [SC06]. Most of the CP solvers are trailing-based. A variable that can restore its domain is called reversible. We refer to [SC06] for more information about state restoration. Alternative exploration strategies (all based on DFS) are: Iterative Deepening Search (IDS) [Kor85], Limited Discrepancy Search (LDS) [HG95], Depth-Bounded Discrepancy Search (DDS) [Wal97], Discrepancy-Bounded Depth First Search (DBDFS) [BP00], etc.

For the optimization problems, the common approach is a constraint based version of branch and bound introduced in [Hen89]. The principle is to dynamically (during the search) add the constraint $z < z(s)$ (for a minimization problem) in which $z(s)$ is the cost of the last consistent solution found s. By doing this, the CP solver does not explore the part of the tree in which it is sure that the costs of the consistent solutions inside these parts are $\geq z(s)$. The cost of the next consistent solution found after s is strictly better than $z(s)$ and the last solution found when the search is completed is an optimal solution. On the other hand, an optimization constraint can also help to provide a good lower bound on the objective value in order to quickly prove optimality. Other techniques to solve optimization problems in CP consist of iterating on possible values of z from its lower (or upper) bound or achieving a dichotomic search (see for example [BLPN12]).

For hard problems, the search space is very large and it is difficult to perform a complete search. Some local search techniques exist in CP

to search for solutions with costs close to optimal even without a proof of optimality. The interested reader may refer to [HM05, HS04, HT06, Sha98] for more information about local searches in CP.

2.5 COST-BASED PROPAGATOR IN THE OSCAR SOLVER

There are many CP solvers: OR Tool [Goo], Choco [PFL15], Gecode [Gec05], Ilog [J. 94], CHIP [Din+88], OscaR [Osc12], etc. We refer to [Cps] for a more complete list of the CP solvers. The OscaR solver [Osc12] is an open source toolkit for solving OR problems mainly based on Scala [Sca]. For our implementations and experiments, we use OscaR mainly because: 1) it is open source 2) it has an implementation of most of the state-of-the-art constraints needed for our experiments 3) the CP part of Oscar is maintained at UCL 4) its syntax is intuitive and easy to understand for final users. We refer to [Sch15] for a quick introduction to OscaR.

To implement a new propagator in OscaR, one needs to define two main procedures [Sch15]:

- *setup*(): setup the constraint. This procedure allows to register the constraint to potential modifications of the domains of the variables in its scope. A first consistency check and propagation can be done;

- *propagate*(). This procedure is called if a variable x has asked to do so with, for example, one of these procedures:
  - *callPropagateWhenBoundsChange* (when the lower bound or the upper bound of the domain of x changes);
  - *callPropagateWhenDomainChanges* (when the domain of x changes);
  - *callPropagateWhenBind* (when x is bound that means that there is a single value in the domain of x).

Listing 2.1 is a skeleton of a cost-based propagator in OscaR that describes to the solver how and when to process the different filtering events. First, we state in *setup* that the propagator should be triggered each time a decision variable changes one of its bounds (minimum or maximum) and also each time the cost variable changes one of its bounds. Then, in *propagate*, one sets how to filter the different variables: 1) filter the cost variable after the computation of an optimal solution for a relaxed problem 2) filter the different decision variables

X_i after the computation of a lower bound of the new cost when X_i is forced to take a value $v \in D_i$; $\forall i, v$.

Listing 2.1: Skeleton of a cost-based propagator in OscaR

```
import oscar.cp._
...

// X is the array of n decision variables and C is the cost of the objective part concerned
class ConstraintExample(val X: Array[CPIntVar], val C: CPIntVar) extends Constraint(X(0).store, "ConstraintExample") {

    //Preprocessing and checking the pre-conditions

  override def setup(l: CPPropagStrength): Unit = {
    X.foreach(_.callPropagateWhenBoundsChange(this))
    C.callPropagateWhenBoundsChange(this)
    propagate()
  }

  override def propagate(): Unit = {

    // Compute the optimal solution of the relaxed problem associated and its cost Copt.
    C.updateMin(Copt)

    // Compute an optimistic evaluation of the new cost to be paid if a value v in D(i) is assigned to variable X(i) and let newC(i)(v) be this cost.
    for (i <- 1 to n) {
        for(v <- D(i)){
            if (newC(i)(v) > C.max){
                X(i).removeValue(v)
            }
        }
    }
  }
}
```

3

CAPACITATED LOT SIZING PROBLEM (CLSP)

In this chapter, we give some preliminary notions about the lot sizing problem and some works related to this problem. In Section 3.1, the lot sizing problem is defined and some elements to classify its different versions are given. Then in Section 3.2, some works related to the lot sizing problem are presented. Finally, in Section 3.3, the pigment sequencing problem (a variant of the capacitated lot sizing problem) is described.

3.1 OVERVIEW OF THE LOT SIZING PROBLEM

Production planning considers the best use of production resources in order to satisfy production goals over a certain period named the planning horizon [KGW03]. The Lot Sizing (LS) problem is one of the most important problems in production planning [KGW03, AFT99]. There are many versions of the LS problem depending on their characteristics. These problems require one to determine a minimum cost (production costs and/or storage/stocking costs and/or setup costs, etc.) production schedule to satisfy the demands for single or multiple items while respecting side restrictions. We give below some characteristics of the LS problems and one of their classifications.

3.1.1 *Characteristics of the* Lot Sizing *problem*

The next properties affect directly classifying and modeling of the LS problems [KGW03]:

- Planning horizon: it is the time interval in which the production of items will take place. It may be finite or infinite / discrete or continious.
- Capacity or resource constraints: the LS problem may be capacitated or uncapacitated. If there are no restrictions on resources or production capacity, the problem is an uncapacitated problem. Otherwise, it is a capacitated problem.
- Number of items: the LS problem is a single item problem if there is only one type of order to be produced. Otherwise, it is a multiple items problem.
- Demand: demand is static if its value does not change over time and dynamic otherwise. If demand is known in advance, it is deterministic. Otherwise, it is probabilistic.
- Setup: when the production of an item induces a cost (resp. time) to prepare the machine, this cost (resp. time) is called setup cost (resp. setup time). This cost/time is independent of the amount of the item unit that is produced. It is changeover cost (resp. time) when the production of an item in a given period induces a cost (resp. time) that depends on the item produced in the previous period.
- Number of levels: the LS problem is multi-level problem if some items are used in the production of other items. The problem is a single-level if the production of an item is not dependent on the production of others.
- Shortage: there is shortage when it is not possible to satisfy all the demands. When it is allowed to satisfy some demands after their deadlines, there is backlogging. There are lost sales when it is allowed to not satisfy all the demands.

3.1.2 *Classification of the* Lot Sizing *problem*

We present, in this section, the classification of the LS problem for single item proposed in [Wol02, PW05], based on the following three variables: *PROB - CAPA - VAR*.
The variable *PROB* can take one of the following values [Wol02, PW05]:

- LS (Lot Sizing): this is the basic LS problem. There is a planning horizon of T periods and, in each period, there is a production capacity c_t and a demand to be satisfied d_t. The cost to be minimized includes 1) unit production cost p_t and a fixed set-up cost f_t if production takes place in t for each $t \in [1, \ldots, T]$, and 2) cost h_t per unit of stock at the end of the period $t \in [1, \ldots, T]$;
- WW (Wagner-Whitin): this is the basic LS problem, except that the unit production cost per period p_t and the per period stocking cost h_t satisfy $h_t + p_t \geq p_{t+1}$ for each $t \in [1, \ldots, T-1]$. This means that it is better to produce as late as possible;
- DLS (Discrete Lot Sizing): this is the basic LS problem except that there is either no production or production at full capacity c_t in each period $t \in [1, \ldots, T]$;
- DLSI (Discrete Lot Sizing with variable Initial stock): this is the DLS problem with an initial stock.

The second variable $CAPA$ is about the resource (or capacity) restrictions and can take one of the following values:

- C (capacitated): the capacities $c_t, \forall t \in [1, \ldots, T]$ may vary over time;
- CC (constant capacitated): the capacities are the same over time. $c_t = c_1, \forall t \in [1, \ldots, T]$;
- U (uncapacitated): there is no restriction on the capacity of production. $c_t = \infty, \forall t \in [1, \ldots, T]$.

Finally, the third variable treats other extensions of the problem such as B (Backlogging), SC (Start-up Cost), ST (Start-up Time), SS (Safety Stocks), etc. [Wol02, PW05].

3.2 RELATED WORKS

Historically, the paper "How many parts to make at once ?" [Har13] (published in 1913) can be considered as the first that introduces the LS problem. The research in this domain was intensified from Wagner and Within paper [WW58] and Manne paper [Man58] in 1958. In this thesis, we focus on the Capacitated Lot Sizing Problem (CLSP).

There are many other variants such as the Economic Lot Sizing Problem (ELSP) [GS97], the Proportional Lot Sizing and Scheduling Problem (PLSP [DH95], the Continuous Setup Lot Sizing Problem (CSLP) [DK97c]. We refer to [DK97b, JD06, UP10, Cop+16, BRG87] for general reviews about the LS problem.

A CLSP is a LS problem in which there are resource restrictions (capacity) for each period of production over a discrete and finite planning horizon. It is a single level, dynamic demand production planning problem without backlogging. The CLSP is known to be NP-Hard [CT90, BY82, FLK80] (for single and multiple items). Its special case, the Discrete Lot Sizing Problem (DLSP), is also NP-Hard [JD98, Bru00].

The exact solving solution approaches mainly consist of adding some strong valid inequalities (cut generation technique) or reformulating the problem (see for example [GH01, BW01, BRW84, LMV89, Gic08]). Since most of these problems are NP-Hard, many heuristics or relaxation based approaches have also been proposed (see for example [Hin95, CFL94, Dia+03, AFT99, Tri87, KR82]). We refer to [KGW03] to have details about the operational research approaches proposed in the litterature for the CLSP.

Related works in CP

Surprisingly, to the best of our knowledge, the LS problem have so far been ignored by CP before we started our research. However, one can use some alternatives with the state-of-the-art CP constraints. For each global constraint introduced in this thesis, we state the possible alternatives in the corresponding chapter. For example, for our global constraints `StockingCost` (see Chapter 5) and `IDStockingCost` (see Chapter 6), the `cost-gcc` constraint and the `minimumAssignment` constraint (described in Section 2.3) are good state-of-the-art CP alternatives.

After our paper on the `StockingCost` constraint [Hou+14], German et al. [Ger+15] have proposed the `LotSizing` constraint for a single item problem where the different costs depend on the period of production of the order. The authors compute lower bounds on the setup cost, the variable production cost and the stocking cost and filter the decision variables wrt these lower bounds. Unfortunately, they do not compare the filtering of the `LotSizing` constraint wrt the filtering provided by the `minimumAssignment` (or `cost-gcc`) constraint.

3.3 A VARIANT OF THE CLSP

This section describes the variant of the CLSP used for the experiments in this thesis and its integer programming formulations described in [Fle94, PW05, Hou+]: the Pigment Sequencing Problem (PSP). This is a multiple items, single machine problem with discrete planning horizon and production capacity limited to one unit per period. The demand is dynamic and deterministic. There are item-dependent stocking costs and sequence-dependent changeover costs. The total stocking cost of an order is proportional to its stocking cost (ie the stocking cost $h_i \geq 0$ of the corresponding item i) and the number of periods between its due date and the production period. The changeover cost $q^{i,j} \geq 0$ is induced when passing from the production of item i to another one j with $q^{i,i} = 0, \forall i$. The objective is to assign a production period for each order respecting its due date and the machine capacity constraint so as to minimize the sum of stocking costs and changeover costs. Note that, without loss of generality, demands can be normalized such that $d_t^i \in \{0, 1\}$ and then each single item has classification DLS-CC-SC (see Section 3.1.2). We recall the example given in Introduction (Chapter 1).

Example 4. *Two types of order (1 and 2) must be produced over the planning horizon* $[1, \ldots, 5]$. *The stocking costs are respectively* $h_1 = 5$ *and* $h_2 = 2$ *for each type of order (item). The demands for item 1 are* $d^1_{t\in[1,\ldots,5]} = [0, 1, 0, 1, 0]$ *and for the second item are* $d^2_{t\in[1,\ldots,5]} = [0, 0, 1, 0, 1]$. *Thus the number of orders is* $n = 4$. *The changeover costs are:* $q^{1,2} = 10$, $q^{2,1} = 5$ *and* $q^{1,1} = q^{2,2} = 0$. *The solution* $S_1 = [1, 2, 0, 1, 2]$ *(represented in* Figure 3.1*) is a feasible solution. This means that the item* 1 *will be produced in periods* 1 *and*

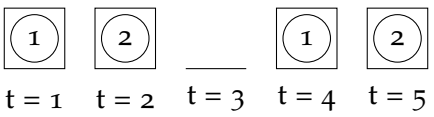


Figure 3.1: A feasible solution of the PSP instance of Example 4

4 *while the item* 2 *will be produced in periods* 2 *and* 5. *Period* 3 *is an idle period, a period in which there is no production. The cost associated to* S_1 *is* $C_{s_1} = h_1 + h_2 + q^{1,2} + q^{2,1} + q^{1,2} = 32$. *The optimal solution for this problem is* $S_2 = [2, 1, 0, 1, 2]$ *(represented in* Figure 3.2*) with cost* $C_{s_2} = 19$.

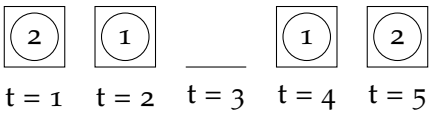


Figure 3.2: An optimal solution of the PSP instance of Example 4

3.3.1 *A CP model for the PSP*

This section presents a basic CP model for the PSP. We uniquely identify each order. The aim is to associate to each of these orders a period that respects the due date of the order. Let us denote by $date(p) \in [1, \ldots, T], \forall p \in [1, \ldots, n]$ the period in which the order p is produced, $dueDate(p) \in [1, \ldots, T]$ the due date of the order p, and $\mathcal{I}(p)$ the corresponding item of the order p. To enforce the feasibility of a solution, the main constraints are the following:

$$date(p) \leq duedate(p), \forall p \tag{3.1}$$

$$\texttt{alldifferent(date)} \tag{3.2}$$

in which:

- Equation 3.1: each order must be satisfied before its due date;
- Equation 3.2: since the capacity of the machine is limited to one, each order must be produced at different period.

One can model the changeover costs part of the problem as a ATSP. Each order represents a city to be visited and the changeover costs are the distance between two cities. To this end, we add an artificial order $n+1$ such that $date(n+1) = T+1$ with $q^{\mathcal{I}(p),\mathcal{I}(n+1)} = q^{\mathcal{I}(n+1),\mathcal{I}(p)} = 0, \forall p \in [1, \ldots, n]$. Denote by $successor(p), \forall p \in [1 \ldots n]$ the order produced just after producing the order p. The following additional constraints can then be added:

$$\texttt{circuit(successor)} \tag{3.3}$$

$$date(p) < date(successor(p)), \forall p \in [1, \ldots, n] \tag{3.4}$$

in which:

- Equation 3.3: it ensures the existence of an Hamiltonian circuit (see [Pes+98]);

- Equation 3.4: the order p must be produced before its successors.

Finally, the objective is to minimize the stocking costs and the changeover costs:

$$\sum_{p}(dueDate(p) - date(p)) \cdot h_{\mathcal{I}(p)} + \sum_{p} q^{\mathcal{I}(p),\mathcal{I}(successor(p))}$$

We refer to Section 5.6 to see a stronger CP model for the PSP.

3.3.2 *MIP formulation for the PSP*

Formulation 2 is a basic integer linear programming formulation of the PSP with m items to be produced over T periods with no initial stocks [PW05].

Formulation 2.

$$\min \sum_{i,j,t} q^{i,j}\chi_t^{i,j} + \sum_{i,t} h_i s_t^i \tag{3.5}$$

$$s_0^i = 0, \ \forall i \tag{3.6}$$

$$x_t^i + s_{t-1}^i = d_t^i + s_t^i, \forall i, t \tag{3.7}$$

$$(PSP_1) \qquad x_t^i \leq y_t^i, \ \forall i, t \tag{3.8}$$

$$\sum_i y_t^i = 1, \forall t \tag{3.9}$$

$$\chi_t^{i,j} \geq y_{t-1}^i + y_t^j - 1, \forall i, j, t \tag{3.10}$$

$$x_t^i, y_t^i, \chi_t^{i,j} \in \{0,1\}, s_t^i \in \mathrm{N}, \forall i, j \in [1,\ldots,m], \forall t \in [1,\ldots,T] \tag{3.11}$$

in which:

- x_t^i: binary variable of production. $x_t^i = 1$ if the item i is produced in period t and $x_t^i = 0$ otherwise;
- y_t^i: binary variable of setup. $y_t^i = 1$ if the machine is prepared for production of the item i in period t and $y_t^i = 0$ otherwise;
- s_t^i: the number of items i stocked at the end of period $t, \forall t \in [1,\ldots,T]$;

- $\chi_t^{i,j}$: binary variable of changeover. $\chi_t^{i,j} = 1$ if the machine is set-up for item j in period t and was set-up for item i in period $t-1$; $\chi_t^{i,j} = 0$ otherwise;

and:

- Equation 3.5: objective function - minimization of the sum of stocking and changeover costs;
- Equation 3.6: there is no initial stock;
- Equation 3.7: the demand satisfaction in each period. This is a flow balance constraint between the number of items produced, stocked and delivered;
- Equation 3.8: setup forcing constraint (y_t^i must be equal to 1 if an item i is produced in period t);
- Equation 3.9: machine capacity production restrictions. Here we use equality constraints (instead of $\sum_i y_t^i \leq 1, \forall t$) in order to compute the changeover costs;
- Equation 3.10: changeover variables definition. Actually, if $y_{t-1}^i = 1$ and $y_t^j = 1$ then $\chi_t^{i,j}$ must be equal to 1. Otherwise $\chi_t^{i,j} = 0$ thanks to the objective function.

The constraints modeling the changeover variables (constraints 3.10) can be replaced by the constraints 3.16 as in the next formulation [PW05].

Formulation 3.

$$
\begin{aligned}
& & \text{Equations } 3.5, 3.6, 3.7, 3.8, 3.9 & & (3.12)\\
& & \sum_i \chi_t^{i,j} = y_t^j, \forall j, t & & (3.13)\\
& (PSP_2) & \sum_j \chi_t^{i,j} = y_{t-1}^i, \forall i, t & & (3.14)\\
& & \sum_i y_0^i = 1 & & (3.15)\\
& & y_t^j - z_t^j = y_{t-1}^j - w_{t-1}^j = \chi_t^{jj}, \forall j, t & & (3.16)
\end{aligned}
$$

in which:

- z_t^i: binary variable of start-up. $z_t^i = 1$ if there is a *setup* for an item i in period t but not in period $t-1$. $z_t^i = 0$ otherwise;
- w_t^i: binary variable of switch-off. $w_t^i = 1$ if there is a *setup* for an item i in period t but not in period $t+1$. $w_t^i = 0$ otherwise.

A tighter reformulation (Formulation 4) is obtained by adding some DLS-CC-SC valid inequalities for each item i [PW05].

Formulation 4.

Formulation 3 (3.17)

$$(PSP_3) \quad s_{t-1}^i + \sum_{u=t}^{t+p-1} y_u^i + \sum_{u=t+1}^{t+p-1} (d_{ul}^i - (t+p-u)) z_u^i + \sum_{u=t+p}^{l} d_{ul}^i z_u^i \geq p \tag{3.18}$$

in which:

- d_{ul}^i is the sum of demands in the interval $[u, l]$ for the item i: $d_{ul}^i = \sum_{k=u}^{l} d_k^i$;
- $p = d_{tl}^i = \sum_{k=t}^{l} d_k^i$;
- Equation 3.18: valid inequalities on the interval $[t, l] \subseteq [1, \ldots, T]$ $\forall i \in [1, \ldots, n]$.

We refer to [PW05] for details about these valid inequalities and the different formulations. Formulation 4 can be considered as the state-of-the-art of exact method for the PSP. The interested reader may refer to [PW05, Hou13] to see some experimental results for these MIP formulations. Furthermore, Ceschia et al. [CDGS16] report the solution of some instances of the PSP using a simulated annealing approach.

4

CONSTRAINED ARBORESCENCE PROBLEM (CAP)

In this chapter, we give some preliminary notions about the Minimum Weight Arborescence (MWA) and some works related to the Constrained Arborescence Problem (CAP). In Section 4.1, the MWA problem is defined and an algorithm to compute an MWA is given. Then in Section 4.2, some works related to the CAP are presented. Finally, in Section 4.3, the resource constrained minimum weight arborescence problem (a variant of the CAP) is described.

4.1 OVERVIEW OF MINIMUM WEIGHT ARBORESCENCE (MWA)

An arborescence A rooted at r is a spanning tree [GH85] if we ignore the direction of edges and there is a directed path in A from r to each other vertex [Way13]. A Minimum Directed Spanning Tree (MDST) or Minimum Weight Arborescence (MWA) $A(G)^\star$ of the graph G is an arborescence of minimum total cost. Let us formally define the MWA problem. Consider a directed graph $G = (V, E)$, in which $V = \{v_1, v_2, \ldots, v_n\}$ is the vertex set and $E \subseteq \{(i, j) \mid i, j \in V\}$ is the edge set. Each edge $(i, j) \in E$ has a weight $w(i, j)$ and one vertex $r \in V$ is identified as the root. Without loss of generality, any edge entering the root r can be removed.

To formulate the MWA problem, let us use the following notations.

Definition 1. *Consider a subset of vertices $S \subseteq V$. Let δ_S^{in} be the set of edges entering S: $\delta_S^{in} = \{(i, j) \in E \mid (i \in V \setminus S) \wedge (j \in S)\}$. For a vertex $k \in V$, δ_k^{in} is the set of edges that enter in k.*

Definition 2. *Let V' be the set of vertices without the root r: $V' = V \setminus \{r\}$.*

The MWA problem can be formulated as follows [FV97]:

Formulation 5.

$$w(A(G)^\star) = \min \sum_{(i,j)\in E} w(i,j) \cdot x_{i,j} \tag{4.1}$$

$$(MWA) \qquad \sum_{(i,j)\in\delta_j^{in}} x_{i,j} = 1, \forall j \in V' \tag{4.2}$$

$$\sum_{(i,j)\in\delta_S^{in}} x_{i,j} \geq 1, \forall S \subseteq V' : |S| \geq 2 \tag{4.3}$$

$$x_{i,j} \in \{0,1\}, \forall (i,j) \in E \tag{4.4}$$

in which $x_{i,j} = 1$ if the edge (i,j) is in the optimal arborescence $A(G)^\star$ and $x_{i,j} = 0$ otherwise. The first group of constraints imposes that exactly one edge enters in each vertex $j \in V'$ and the constraints (4.3) enforce the existence of a path from the root r to all other vertices. As in [FT93], we assume that $w(i,i) = \infty, \forall i \in V$ and $w(i,j) > 0, \forall (i,j) \in E$ without loss of generality. In this case, following [Edm67], the constraints (4.4) can be relaxed to $x_{i,j} \geq 0, \forall i,j$(4.5) and the constraints (4.2) become redundant. This leads to Formulation 6 for the MWA problem.

Formulation 6.

$$w(A(G)^\star) = \min \sum_{(i,j)\in E} w(i,j) \cdot x_{i,j} \tag{4.6}$$

$$(MWA) \qquad \sum_{(i,j)\in\delta_S^{in}} x_{i,j} \geq 1, \forall S \subseteq V' : |S| \geq 2 \tag{4.7}$$

$$x_{i,j} \geq 0, \forall (i,j) \in E \tag{4.8}$$

Algorithms to compute an MWA $A(G)^\star$ of a given graph G were proposed independently by Chu and Liu ([CL65]), Edmonds ([Edm67]) and Bock ([Boc71]). A basic implementation of that algorithm is in $O(|V||E|)$. The associated algorithm is often called Edmonds' algorithm. An $O(\min\{|V|^2, |E|\log|V|\})$ implementation of the Edmonds' algorithm is proposed by [Tar77]. More sophisticated implementations exist (see for example [Gab+86, Men+04]). Fischetti and Toth ([FT93]) propose an $O(|V|^2)$ implementation to compute an MWA and also the associated linear programming reduced costs. We rely on this algorithm for filtering the `MinArborescence` constraint.

An MWA has two important properties that are used to construct it [KT05b].

Proposition 1. *A subgraph $A = (V, F)$ of the graph $G = (V, E)$ is an arborescence rooted at the vertex r if and only if A has no cycle, and for each vertex $v \neq r$, there is exactly one edge in F that enters v.*

Proposition 2. *For each $v \neq r$, select the cheapest edge entering v (breaking ties arbitrarily), and let $F^\star$ be this set of $|V| - 1$ edges. If $(V, F^\star)$ is an arborescence, then it is a MWA; otherwise, $w(V, F^\star)$ is a lower bound on the MWA.*

4.1.1 *Conditions for optimality of the* MWA

The LP dual problem $\mathcal{D}_{\text{MWA}}$ of MWA is [FT93]:

Formulation 7.

$$\max \sum_{S \subseteq V'} u_S \tag{4.9}$$

$$(\mathcal{D}_{MWA}) \quad w(i,j) - \sum_{(i,j) \in \delta_S^{in}, \forall S \subseteq V'} u_S \geq 0, \forall (i,j) \in E \tag{4.10}$$

$$u_S \geq 0, \forall S \subseteq V' \tag{4.11}$$

in which u_S is the dual variable associated to the subset of vertices $S \subseteq V'$.

Let $rc(i,j)$ be the LP reduced cost associated to the edge (i,j). The necessary and sufficient conditions for the optimality of MWA (with primal solution $x_{i,j}^\star$) and $\mathcal{D}_{\text{MWA}}$ (with dual solution $u_S^\star$) are [FT93]:

1. primal solution $x_{i,j}^\star$ satisfies the constraints (4.7) and (4.8);
2. $u_S^\star \geq 0$ for each $S \subseteq V'$;
3. reduced cost $rc(i,j) = w(i,j) - \sum_{(i,j) \in \delta_S^{in}, \forall S \subseteq V'} u_S^\star \geq 0$ for each $(i,j) \in E$;
4. $rc(i,j) = 0$ for each $(i,j) \in E$ such that $x_{i,j}^\star > 0$;
5. $\sum_{(i,j) \in \delta_S^{in}} x_{i,j}^\star = 1$ for each $S \subseteq V'$ such that $u_S^\star > 0$.

4.1.2 *Computation of an* MWA

Based on Proposition 1 and Proposition 2, an MWA can be computed in a greedy way. We sketch below the two-phase algorithm for computing an MWA $A(G)^\star$ for a graph G rooted at a given vertex r.

- Phase 1:
  1. select the minimum cost edge $min1[k] = \arg\min_{(v,k)\in\delta_k^{in}} w(v,k)$ entering each $k \in V'$ and let $F^\star$ be this set of $|V|-1$ edges.
  2. if $F^\star$ is an arborescence then we have an MWA. STOP.
  3. there exists a subset of vertices forming a cycle C. Then we contract all vertices in C into a single vertex c and consider the reduced costs $w^{\mathcal{M}}(i,j) = w(i,j) - min1[j], \forall (i,j)$. We denote by $G^{\mathcal{M}} = (V^{\mathcal{M}}, E^{\mathcal{M}})$ the contracted graph in which $V^{\mathcal{M}} = V \cup \{c\} \setminus \{k \mid k \in C\}$.
     Go to 1 with the new graph $G^{\mathcal{M}}$.

  In case some contraction(s) occurred in Phase 1, a second phase is needed to break the cycle in each contracted vertex.

- Phase 2: consider a cycle C with the selected entering edge $(a,b) \in E$. Remove all edges entering b except (a,b). This procedure ensures that the cycle is broken and each vertex involved is now connected to the rest of the graph.

For example, consider the graph $G_1 = (V_1, E_1)$ in Figure 4.1 with the vertex 0 as root. Figures 4.2, 4.3 (Phase 1 with the cycle $S_1 = \{2,4\}$) and Figure 4.4 (Phase 2 with the edge $(0,4)$ selected for the cycle S_1) show the different steps needed to construct $A(G_1)^\star$.

Algorithm 4.1.1 [FT93] computes an MWA $A(G)^\star$ for a graph G rooted at a given vertex r and also gives the different dual values $u_S^\star$. One can see the differents steps of the algorithm for the graph G_1 on the Figures 4.2, 4.3, 4.4.

Note that the optimality condition 5 implies that for each $S \subseteq V'$, $u_S^\star > 0 \implies \sum_{(i,j)\in\delta_S^{in}} x_{i,j}^\star = 1$ and $\sum_{(i,j)\in\delta_S^{in}} x_{i,j}^\star > 1 \implies u_S^\star = 0$ (because $u_S^\star \geq 0$). The different values of dual variables $u_S, \forall S \subseteq V'$ are obtained during the execution of Edmonds algorithm. Actually, for each vertex $k \in V : u_k^\star = \arg\min_{(v,k)\in\delta_k^{in}} w(v,k)$ (line 13 of Algo-

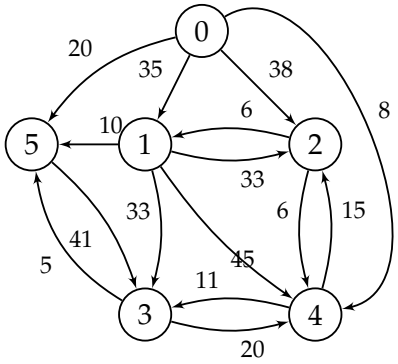


Figure 4.1: Initial graph G_1

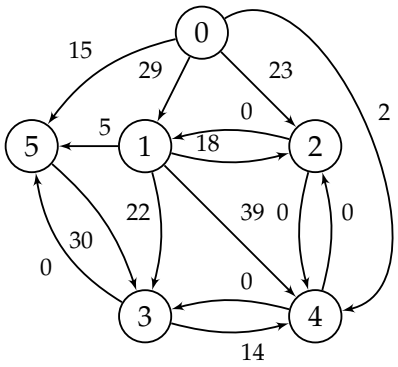


Figure 4.2: Computation of $A(G_1)^\star$: Phase 1, first iteration.
$A_0 = \{(2,1),(4,2),(4,3),(2,4),(3,5)\}$.

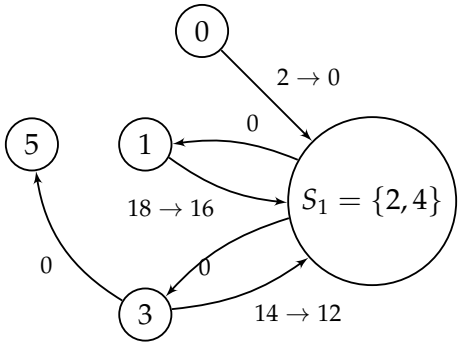


Figure 4.3: Computation of $A(G_1)^\star$: Phase 1, second iteration.
$A_0 = \{(2,1),(4,2),(4,3),(2,4),(3,5),(0,4)\}$.

Algorithm 4.1.1: Computation of a minimum weight arborescence $A(G)^\star$ rooted at vertex r

Input: $G = (V, E)$; $r \in V$; $w(e), \forall e \in E$

// all primal and dual variables $x^\star_{i,j}$ and $u^\star_S$ are assumed to be zero
foreach each edge $(i, j) \in E$ **do**
  $rc(i, j) \leftarrow w(i, j)$
end
$A_0 \leftarrow \emptyset$; $h \leftarrow 0$
// Phase 1:
while $G_0 = (V, A_0)$ is not r-connected **do**
  // A graph is r-connected iff there is a path from the vertex r to each other vertex v in V'.
  $h \leftarrow h + 1$
  Find any strong component S_h of G_0 such that $r \notin S_h$ and $A_0 \cap \delta^{in}_{S_h} = \emptyset$
  // If $|S_h| > 1$, then S_h is a directed cycle
  Determine the edge (i_h, j_h) in $\delta^{in}_{S_h}$ such that $rc(i_h, j_h) \leq rc(e), \forall e \in \delta^{in}_{S_h}$
  $u^\star_{S_h} \leftarrow rc(i_h, j_h)$ // dual variable associated to S_h
  $x^\star_{i_h, j_h} \leftarrow 1$;
  $A_0 \leftarrow A_0 \cup \{i_h, j_h\}$
  foreach each edge $(i, j) \in \delta^{in}_{S_h}$ **do**
    $rc(i, j) \leftarrow rc(i, j) - u^\star_{S_h}$
  end
end
// Phase 2:
$t \leftarrow h$
while $t \geq 1$ **do**
  // Extend A_0 to an arborescence by letting all but one edge of each strong component S
  if $x^\star_{i_t, j_t} = 1$ **then**
    foreach each $q < t$ such that $j_t \in S_q$ **do**
      $x^\star_{i_q, j_q} \leftarrow 0$
      $A_0 \leftarrow A_0 \setminus \{(i_q, j_q)\}$
    end
  end
  $t \leftarrow t - 1$
end

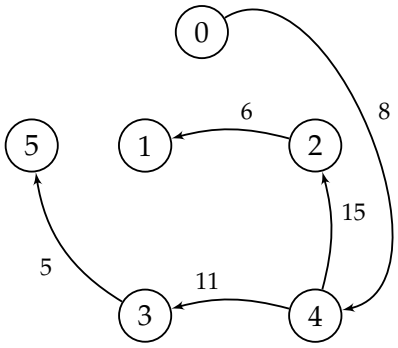


Figure 4.4: Computation of $A(G_1)^\star$: Phase 2.
$A_0 = \{(2,1),(4,2),(4,3),(3,5),(0,4)\}$. $(2,4)$ is removed.

rithm 4.1.1). If a subset $S \subseteq V' : |S| \geq 2$ is a strong component[1], then $u_S^\star$ is the minimum reduced cost of edges in δ_S^{in}. Only these subsets can have $u_S^\star > 0$. All other subsets ($S \subseteq V' : |S| \geq 2$ and S is not a strong component) have $u_S^\star = 0$. Thus there are $O(|V|)$ subsets $S \subseteq V' : |S| \geq 2$ that can have $u_S^\star > 0$. A straightforward algorithm can compute $rc(i,j) = w(i,j) - \sum_{(i,j)\in\delta_S^{in}, \forall S\subseteq V'} u_S^\star, \forall (i,j)$ in $O(|V|^3)$ by considering, for each edge, the $O(|V|)$ subsets that are directed cycles. In [FT93], Fischetti and Toth propose an $O(|V|^2)$ algorithm to compute $rc(i,j), \forall (i,j) \in E$.

4.2 RELATED WORKS

The constrained Minimum Spanning Tree (MST) problem requires one to find a spanning tree that respects some constraints and has minimum weight. The constrained MST problem is much more studied than the CAP. Deo and Kumar [DK97a] present 29 variants of this problem such as the diameter-constrained MST [Ach+92], the degree-constrained MST [NH80], the min-degree MST [AT10], the capacitated MST [CL73, PV84], etc.

As concerns the directed graphs, to the best of our knowledge, the only variant of the CAP studied in the literature is the Resource constrained Minimum-Weight Arborescence (RMWA) problem [Ros86,

1 A strong component of a graph G is a maximal (with respect to set inclusion) vertex set $S \subseteq V$ such that (i) $|S| = 1$ or (ii) for each pair of distinct vertices i and j in S, at least one path exists in G from vertex i to vertex j [FT93].

GR90, FV97]. The RMWA problem requires one to find an arborescence that respects the resource constraints on each vertex and has the minimum cost. Guignard and Rosenwein [GR90] introduce the problem and propose a Lagrangian decomposition approach. Later on, Fischetti and Vigo [FV97] propose a branch and cut algorithm by using some known classes of valid inequalities for the Asymmetric Travelling Salesman Problem (ATSP) and new heuristic procedures.

Related works in CP

Many articles focus on filtering of the MST constraints on undirected weighted graphs. Dooms and Katriel [DK07] propose filtering algorithms for the Weight-Bounded Spanning Tree WBST(G, T, I, W) constraint, which is defined on undirected graph variables G and T, a scalar variable I and a vector of scalar variables W. This constraint enforces that T is a spanning tree of G with total weight less than I, considering the edge weights W. Following [Rég08], the filtering algorithms for WBST introduced by [DK07] are rather difficult to understand and to implement. In [Rég08], Régin develops easier to implement consistency checking and filtering algorithms for this constraint. More interestingly, he presents an incremental version of the algorithm. Further, Régin et al. [Rég+10] extend this latter to also take into account mandatory edges. On the other hand, Una et al. [Una+16] use learning to accelerate the search for the WBST constraint and carry out numerical experiments on the diameter-constrained MST.

For directed graphs, Lorca [Lor10] proposes some constraints on trees and forests. In particular, the `tree` constraint [BFL05, Lor10] is about anti-arborescence on directed graph. By considering a set of vertices (called resource), the `tree` constraint partitions the vertices of a given graph into a set of disjoint anti-arborescences such that each anti-arborescence points to a resource vertex. The authors propose a GAC filtering algorithm in $O(|E||V|)$ based on the computation of strong articulation points of the graph. Further, Lorca and Fages [LJG11] revisit the `tree` constraint and introduce a new GAC filtering algorithm for this constraint in $O(|E| + |V|)$.

4.3 A VARIANT OF THE CAP

This section presents the variant of the CAP used for the experiments in this thesis: the Resource constrained Minimum Weight Arborescence (RMWA) problem. In this problem, each vertex i has a limited available resource b_i and the usage of an edge (i,j) consumes some resources $a_{i,j}$ from the vertex i.

Formulation 8 is a basic formulation for the RMWA problem:

Formulation 8.

$$w(A(G)^\star) = \min \sum_{(i,j)\in E} w(i,j)\cdot x_{i,j} \tag{4.12}$$

$$\sum_{(i,j)\in\delta_j^{in}} x_{i,j} = 1, \forall j \in V' \tag{4.13}$$

(*RMWA*)

$$\sum_{(i,j)\in\delta_S^{in}} x_{i,j} \geq 1, \forall S \subseteq V' : |S| \geq 2 \tag{4.14}$$

$$\sum_{(i,j)\in\delta_i^{+}} a_{i,j}\cdot x_{i,j} \leq b_i, \forall i \in V \tag{4.15}$$

$$x_{i,j} \in \{0,1\}, \forall (i,j) \in E \tag{4.16}$$

in which δ_i^+ is the set of outgoing edges from i, $a_{i,j}$ is the amount of resource that the edge (i,j) uses and b_i is the resource available at vertex i. The constraint (4.15) is about the resource restrictions on the vertices. The other three constraints together with the objective function are the classical ones that describe an MWA. We refer to [FV97] for details about MIP approaches to this problem.

We recall the example given in Introduction (Chapter 1).

Example 5. *Consider the graph G_1 with additional data concerning the RMWA problem. In* Figure 4.5*, the label of the edge (i,j) is $w(i,j)(a_{i,j})$ and the vertex i is followed by its resource b_i, $i(b_i)$. One can see that with the additional constraints $\sum_{(i,j)\in\delta_i^+} a_{i,j}\cdot x_{i,j} \leq b_i, \forall i \in V$ on this instance, the previous MWA (represented with dashed edges) is not valid because at the vertex 4, $a_{4,2} + a_{4,3} = 15 > b_4$. A feasible solution of this RMWA problem is represented in dashed in* Figure 4.6*.*

The RMWA problem has many practical applications in the design of distribution and telecommunication networks [FV97]: the root vertex is a centralized supplier and the other vertices are associated with customers (or intermediate distribution facilities). A limited amount of

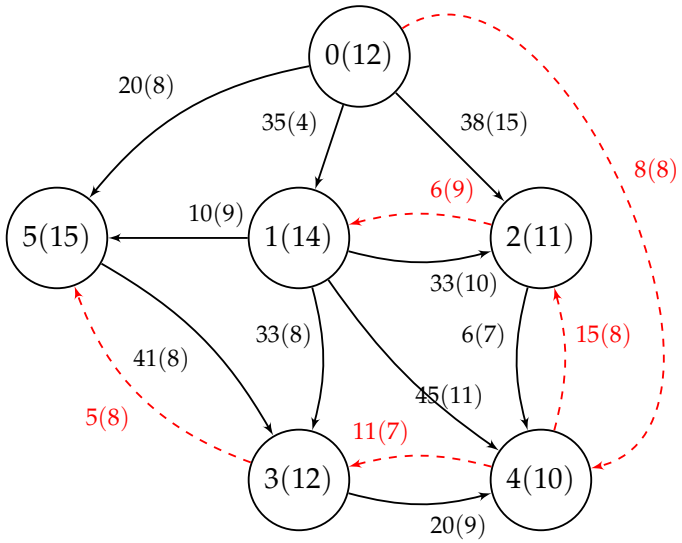


Figure 4.5: Graphe G_1 with $a_{i,j}$ and b_i

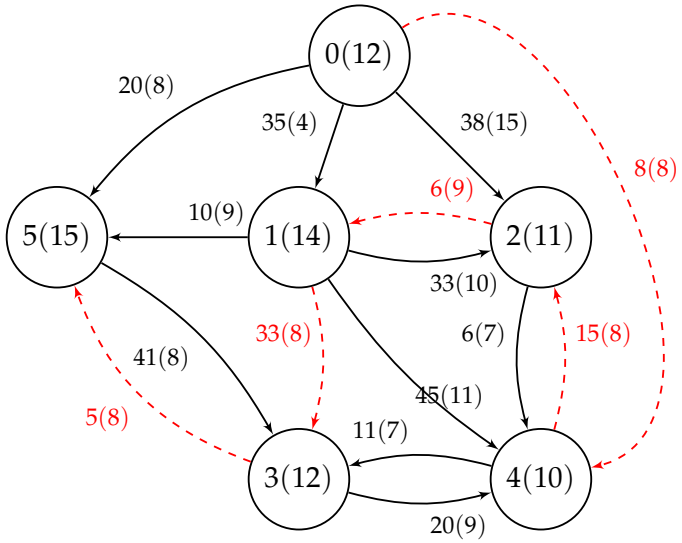


Figure 4.6: Graphe G_1 with $a_{i,j}$ and b_i: a feasible solution of the RMWA problem associated

resource is available at each vertex. The use of a directed link has a cost and consumes a given amount of resource. One wants an arborescence that respects the resource restrictions at minimum cost.

Part II

FILTERING ALGORITHMS FOR A CAPACITATED LOT SIZING PROBLEM

5 THE STOCKINGCOST CONSTRAINT

We refer to Chapter 3 for an overview of the Lot Sizing (LS) problem. Many LS problems call for the minimization of stocking/storage costs. This chapter introduces a new global constraint, denoted `StockingCost`, that holds when each order is produced on or before its due date, the production capacity is respected, and the total stocking cost is less than a given value. Here, we consider that all orders have the same per-period stocking cost. We propose a linear time algorithm to achieve bound consistency on the `StockingCost` constraint. On a variant of the Capacitated Lot Sizing Problem (CLSP), we demonstrate experimentally the pruning and time efficiency of our algorithm compared to other state-of-the-art approaches.

5.1 INTRODUCTION

There are often stocking costs to be minimized in the CLSP. These costs depend on the time spent between the production of an order and its delivery (due date). To handle such LS problems in CP, we propose an efficient bound consistent filtering algorithm for the $\texttt{StockingCost}([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$ constraint that requires each order i (represented by the variable X_i) to be produced by its due date d_i, the capacity c of the machine to be respected, and H is an upper bound on the stocking cost.

First, we characterize an optimal solution of the problem associated to the `StockingCost` constraint. Then we demonstrate that a greedy algorithm is able to efficiently compute the optimal cost H^{opt} and an

optimal solution. Since H^{opt} is the optimal cost, it allows achieving bound consistent filtering of the objective variable H. After this step, we analyze the evolution of $H^{opt}_{X_i \leftarrow v}$ that is the optimal cost when the variable X_i is forced to take the value v. We show that, since the stocking cost is the same for all orders, the evolution of $H^{opt}_{X_i \leftarrow v}$ is monotone. This property allows us to efficiently achieve bound consistent filtering for decision variables X_i.

This chapter is organized as follows: Section 5.2 defines the `StockingCost` constraint and shows how one can achieve pruning with the state-of-the-art approaches; Section 5.3 describes some algorithms to achieve bound consistency for the total stocking costs H and for the orders $i, \forall i \in [1, \ldots, n]$; Section 5.5 presents a complete $O(n)$ filtering algorithm to achieve bound consistency for all variables; Section 5.6 provides some experimental results on the PSP (that is described in Section 3.3) and Section 5.7 concludes.

5.2 THE STOCKINGCOST CONSTRAINT

The `StockingCost` constraint is able to tackle the stocking costs that arise in some variants of the CLSP when the per-period stocking cost is the same for all orders. In this section, we state when the `StockingCost` constraint is applicable and provide some alternatives with other CP constraints. Consider a set $i = 1, \ldots, n$ of orders over a planning horizon $[1, \ldots, T]$[1]. Each order i requires one unit of production capacity and has a due date $d_i \in [1, \ldots, T]$. The production capacity of the machine is c units per period t. The aim is to produce each order by its due date at latest without exceeding the machine capacity and to minimize the sum of the stocking costs of the orders. Without loss of generality, we assume that the per-period stocking cost for each order is one.

The `StockingCost` constraint has the following form:

$$\texttt{StockingCost}([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$$

in which:

- n is the total number of orders;
- each order i is represented by the variable X_i (with $i \in [1, \ldots, n]$) that is the date of production of the order i on the machine;

1 In typical applications of this constraint, assuming that c_t is $O(1)$, the number of orders n is on the order of the horizon T: $n \sim O(T)$.

- the integer d_i is the due-date for order i (with $i \in [1, \ldots, n]$);
- the integer c is the maximum number of orders the machine can produce during one period (capacity);
- if an item is produced before its due date, then it must be stocked. The variable H is an upper bound on the total number of periods that all the orders are in stock.

The StockingCost constraint holds when each order is produced before its due date ($X_i \leq d_i$), the capacity of the machine is respected (i.e. no more than c variables X_i have the same value), and H is an upper bound on the total stocking cost ($\sum_i (d_i - X_i) \leq H$).

Definition 3. *Each variable has a finite domain. We denote by $X_i^{\min}$ and $H^{\min}$ (resp. $X_i^{\max}$ and $H^{\max}$) the minimal (resp. maximal) value in the domain of variable X_i and H. We also let $T = (\max_i(X_i^{\max}) - \min_i(X_i^{\min}))$.*

We know that the objective of a filtering algorithm is to remove values that do not participate in any solution of the constraint. Here, we are interested in achieving bound-consistency for the `StockingCost` constraint. This consistency level generally offers a good trade-off between speed and filtering power. Formally, the bound-consistency definitions for the `StockingCost` constraint are stated below.

Definition 4. *Given a domain $\mathcal{D}$ of variables X_i and H, the constraint* `StockingCost`$([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$ *is* bound consistent *with respect to $\mathcal{D}$ iff:*

- *BC($X_i^{\min}$) $(1 \leq i \leq n)$. Let $x_i = X_i^{\min}$; there exist $x_j \in [X_j^{\min}, \ldots, X_j^{\max}]$ $(1 \leq j \leq n, i \neq j)$ and $h = H^{\max}$ such that* `StockingCost`$([x_1, \ldots, x_n], [d_1, \ldots, d_n], h, c)$ *holds;*
- *BC($X_i^{\max}$) $(1 \leq i \leq n)$. Let $x_i = X_i^{\max}$; there exist $x_j \in [X_j^{\min}, \ldots, X_j^{\max}]$ $(1 \leq j \leq n, i \neq j)$ and $h = H^{\max}$ such that* `StockingCost`$([x_1, \ldots, x_n], [d_1, \ldots, d_n], h, c)$ *holds;*
- *BC($H^{\min}$). Let $h = H^{\min}$; there exist $x_i \in [X_i^{\min}, \ldots, X_i^{\max}] (1 \leq i \leq n)$ such that* `StockingCost`$([x_1, \ldots, x_n], [d_1, \ldots, d_n], h, c)$ *holds.*

Without loss of generality, in the rest of this chapter, we assume that $X_i^{\max} \leq d_i, \forall i$. We give below how one can achieve filtering for the `StockingCost` constraint with other existing constraints.

5.2.1 *Decomposing the constraint*

It is classical to decompose a global constraint into a conjunction of simpler constraints, and applying the filtering algorithms available on the simpler constraints.

A first decomposition of the constraint $\texttt{StockingCost}([x_1,\ldots,x_n],[d_1,\ldots,d_n],h,c)$ is the following:

$$|\{i \mid X_i = t\}| \leq c, \forall t \tag{5.1}$$

$$\sum_i (d_i - X_i) \leq H \tag{5.2}$$

Assuming that the filtering algorithms for each of the separate constraints achieve bound consistency, the above decomposition (with $T+1$ constraints) does not achieve bound consistency of the `StockingCost` constraint, as illustrated in the following example.

Example 6. *Consider the following instance* $\texttt{StockingCost}([X_1 \in [1,\ldots,2], X_2 \in [1,\ldots,2]],[d_1 = 2, d_2 = 2], H \in [0,\ldots,2], c = 1)$. *The naive decomposition is not able to increase the lower bound on* H *because the computation of* H *gives* $(2 - X_1) + (2 - X_2) = [0,\ldots,1] + [0,\ldots,1] = [0,\ldots,2]$. *The problem is that it implicitly assumes that both orders can be placed at the due date but this is not possible because of the capacity* 1 *of the machine. The lower bound of* H *should be set to* 1. *It corresponds to one order produced in period* 1 *and the other in period* 2.

Other decompositions can be proposed to improve the filtering of the naive decomposition.

A first improvement is to use the global cardinality constraint (`gcc`) [Rég96, Qui+03] to model the capacity requirement of the machine imposing that no value should occur more than c times. The `gcc` constraint can efficiently replace T constraints of equation 5.1 in the basic decomposition. Bound consistency on the `gcc` constraint can be obtained in $O(n)$ plus the time for sorting the n variables. This consistency level offers a good time vs filtering tradeoff for interval domains [Hoe01]. However, the `gcc` constraint together with equation 5.2 do not achieve bound consistency of the `StockingCost` constraint.

A second possible improvement is to use a cost-based global cardinality constraint (`cost-gcc`) [Ró2]. In the `cost-gcc`, the cost of the arc (X_i, v) is equal to $+\infty$ if $v > d_i$ and $d_i - v$ otherwise. The `cost-gcc`

provides more pruning than equations 5.1 and 5.2 in the basic decomposition. As mentioned in Section 2.3.2, $O(n \cdot S(m, n+d, \gamma))$ time (with n the number of variables, d is the size of the domains, m is the number arcs and $S(m, n+d, \gamma)$ is the complexity of the search for shortest paths from a node to every node in a graph with m arcs and $n+d$ nodes with a maximal cost γ) is required to check the consistency of the `cost-gcc` constraint. For the `StockingCost`, there can be up to $n \cdot T$ arcs. Hence[2] the final complexity to obtain arc-consistency[3] on the `cost-gcc` used to model `StockingCost` can be up to $O(T^3) \approx O(n^3)$. Note that for $c = 1$, one can use the `minimumAssignment` constraint with a filtering based on LP reduced costs [Foc+99] or exact reduced costs [DCP16] (see Section 2.3.1).

Next sections provide new scalable filtering algorithms to achieve the bound consistent filtering for the `StockingCost` constraint.

5.3 FILTERING OF THE COST VARIABLE H

In this section, we show how to filter the lower bound on the cost variable H in $O(n)$ plus the time for sorting the n variables. Let $\mathcal{P}$ denote the problem of computing the optimal lower-bound for H:

$$
\begin{aligned}
& H^{opt}(\mathcal{P}) = \min \sum_i (d_i - X_i) \\
& X_i^{\min} \leq X_i \leq X_i^{\max}, \forall i \\
& |\{i \mid X_i = t\}| \leq c, \forall t
\end{aligned}
$$

First, let us characterize an optimal solution. For a given assignment, since the per-period stocking costs for all orders are the same, any permutation of orders does not change the cost of this assignment. Hence two solutions, that have the same sorted sequences of values, have the same cost. Denote by $\bar{X}$ a valid assignment vector in which $\bar{X}_i$ is the value (period) taken by X_i, $\forall i \in [1, \ldots, n]$.

Property 3. *For two valid assignments/solutions $\bar{X}$ (with $H(\bar{X}) = \sum_i (d_i - \bar{X}_i)$) and $\hat{X}$ (with $H(\hat{X}) = \sum_i (d_i - \hat{X}_i)$), if the sorted sequences of values of these solutions are the same, then $H(\bar{X}) = H(\hat{X})$.*

2 using a Fibonacci heap to implement Dijkstra algorithm for shortest path computation
3 without considering incremental aspects.

Note that, in an optimal solution, all orders should be produced as late as possible. In other words, if in a valid solution of $\mathcal{P}$, there is an available period for production in period t and there is an order that can be assigned to t but is assigned to $t' < t$ then that solution is not optimal.

Definition 5. *Considering a valid assignment $\bar{X}$ and a period t, the boolean value $t.full$ indicates whether this period is used at maximal capacity or not: $t.full \equiv |\{i \mid \bar{X}_i = t\}| = c$.*

Proposition 4. *Consider a valid assignment $\bar{X}$ wrt $\mathcal{P}$. This assignment is optimal iff $\forall i \in [1, \ldots, n], \nexists t : (\bar{X}_i < t) \wedge (\bar{X}_i^{\max} \geq t) \wedge (\neg t.full)$.*

Proof. First, let assume that $\bar{X}$ does not respect the condition for optimality of Proposition 4. This means that $\exists X_k \wedge \exists t : (\bar{X}_k < t) \wedge (X_k^{\max} \geq t) \wedge (\neg t.full)$. In this case, by moving X_k from $\bar{X}_k$ to t, we obtain a valid solution better than $\bar{X}$. Thus the condition is a necessary condition for the optimality of $\mathcal{P}$.

Now consider two solutions $\bar{X}$ and $\hat{X}$ such that $\bar{X}$ is non-optimal and $\hat{X}$ is optimal. Then there must exist a latest period t that is idle in $\bar{X}$ and active in $\hat{X}$. Thus $\bar{X}$ does not satisfy the condition of the Proposition 4.

□

To respect the condition for optimality of Proposition 4, the unique set of periods used by the optimal solutions of $\mathcal{P}$ can be obtained from right to left by considering orders decreasingly according to their $X_i^{\max}$. Algorithm 5.3.1 computes the optimal value $H^{opt}(\mathcal{P})$ in $O(n \log n)$ and detects infeasibility if the problem is not feasible. This algorithm greedily schedules the production of the orders by non-increasing due date. A current time line t is decreased and at each step, all the orders such that $X_i^{\max} = t$ are stored into a priority queue (heap) to be scheduled next. Note that each order is added/removed exactly once in the heap and the heap is popped at each iteration (line 11). The orders with largest $X_i^{\min}$ must be scheduled first until no more orders can be scheduled in period t or the maximum capacity c is reached.

Let $\mathcal{P}^r$ denote the same problem with relaxed lower bounds of X_i:

Algorithm 5.3.1: `StockingCost`: Filtering of lower bound on H - BC($H^{\min}$)

Input: $X = [X_1, \ldots, X_n]$ such that $X_i \leq d_i$ and sorted $(X_i^{\max} > X_{i+1}^{\max})$

```
1  H^opt ← 0 // total minimum stocking cost
2  t ← X_1^max // current period
3  slack ← c // current slack at this period
4  i ← 1
5  heap ← {} // priority queue sorting orders in decreasing X_i^min
6  while i ≤ n do
7      while i ≤ n ∧ X_i^max = t do
8          heap ← heap ∪ {i}
9          i ← i + 1
10     end
11     while heap.size > 0 do
12         // we virtually produce the order j in period t
13         j ← heap.popFirst
14         slack ← slack − 1
15         H^opt ← H^opt + (d_j − t)
16         // Invariant: each order produced so far respects the
               condition for optimality (Proposition 4) and its
               domain
17         if t < X_i^min then
18             the constraint is not feasible
19         end
20         if slack = 0 then
21             t ← t − 1
22             while i ≤ n ∧ X_i^max = t do
23                 heap ← heap ∪ {i}
24                 i ← i + 1
25             end
26             slack ← c
27         end
28     end
29     if i ≤ n then
30         t ← X_i^max
31     end
32 end
33 H^min ← max(H^min, H^opt)
```

$$H^{opt}(\mathcal{P}^r) = \min \sum_i (d_i - X_i)$$
$$X_i \leq X_i^{\max}, \forall i$$
$$|\{i \mid X_i = t\}| \leq c, \forall t$$

The condition for optimality of $\mathcal{P}$ (see Proposition 4) is also valid for the relaxed problem $\mathcal{P}^r$. Since this condition for optimality does not depend on $X_i^{\min}$, all the optimal solutions of $\mathcal{P}$ and $\mathcal{P}^r$ use the same set of periods. Thus, from Property 3, we have the following property.

Property 5. *If the problem $\mathcal{P}$ is feasible (i.e. the* `gcc` *constraint is feasible), then $H^{opt}(\mathcal{P}) = H^{opt}(\mathcal{P}^r)$.*

Observe that, if we use a simple queue instead of a priority queue in Algorithm 5.3.1, one may virtually assign orders to periods $t < X_i^{\min}$ and the feasibility test is not valid anymore, but the algorithm terminates with the same ordered sequence of periods used in the final solution. The complexity of the algorithm without priority queue is $O(n)$ instead of $O(n \log n)$. The greedy Algorithm 5.3.1 is able to compute the best lower bound $H^{opt}(\mathcal{P}^r)$ (in the following we drop problem argument since optimal values are the same) and filters the lower bound of H if possible: $H^{\min} \geq H^{opt}(\mathcal{P}^r)$.

5.4 PRUNING THE DECISION VARIABLES X_i

In this section, we show how we filter the decision variables X_i. The idea is to study the evolution of $H^{opt}_{X_i \leftarrow v}$ that is the optimal cost when the variable X_i is forced to take the value v. Since we assume the `gcc` constraint is already bound-consistent and thus feasible, only the cost argument may cause a filtering of lower-bounds $X_i^{\min}$. In the rest of the chapter, we implicitly assumed relaxed domains $[-\infty, \ldots, X_i^{\max}]$ with $X_i^{max} \leq d_i, \forall i \in [1, \ldots, n]$.

In this section, the aim is to perform some sensitivity analysis from an optimal solution of $\mathcal{P}$. Based on this, we achieve the bound consistent filtering of the decision variables.

Definition 6. *Let $H^{opt}_{X_i \leftarrow v}$ denote the optimal lower bound in a situation where X_i is forced to take the value $v \leq X_i^{\max}$.*

Clearly, v must be removed from the domain of X_i if $H^{opt}_{X_i \leftarrow v} > H^{\max}$. An interesting question is: what is the minimum value v for X_i such that $H^{opt}_{X_i \leftarrow v} = H^{opt}$?

Definition 7. *Let v_i^{opt} denote the minimum value such that $H^{opt}_{X_i \leftarrow v} = H^{opt}$. We have $v_i^{opt} = \min\{v \leq X_i^{\max} \mid H^{opt}_{X_i \leftarrow v} = H^{opt}\}$.*

The next property gives a lower bound on the evolution of H^{opt} when a variable X_i is forced to take a value $v < v_i^{opt}$. By the definition of v_i^{opt}, H^{opt} will increase by at least $(v_i^{opt} - v)$ if X_i is forced to take a value $v < v_i^{opt}$.

Property 6. *For $v < v_i^{opt}$, we have $H^{opt}_{X_i \leftarrow v} \geq H^{opt} + (v_i^{opt} - v)$.*

After the propagation of $H^{\min}$, one may still have some slack between the upper and the lower bound $H^{\max} - H^{\min}$. Since v_i^{opt} is the minimum value such that $H^{opt}_{X_i \leftarrow v} = H^{opt}$, we can use the lower bound of Property 6 to filter X_i as follows:

$$X_i^{\min} \leftarrow \max(X_i^{\min}, v_i^{opt} - (H^{\max} - H^{\min}))$$

In the following we show that the lower-bound of Property 6 can be improved and that we can actually predict the exact evolution of $H^{opt}_{X_i \leftarrow v}$ for an arbitrary value $v < v_i^{opt}$. A valuable information to this end is the number of orders scheduled at a given period t in an optimal solution:

Definition 8. *In an optimal solution $\bar{X}$ (i.e. $H(\bar{X}) = H^{opt}$), let*

$$count[t] = |\{i \mid \bar{X}_i = t\}|.$$

Algorithm 5.4.1 computes $v_i^{opt}, \forall i$ and $count[t], \forall t$ in linear time $O(T)$. The first step of the algorithm is to initialize $count[t]$ as the number of variables with upper bound equal to t. This can be done in linear time assuming the time horizon of size $(\max_i\{X_i^{\max}\} - \min_i\{X_i^{\min}\})$ is in $O(n)$. We can initialize an array $count$ of the size of the horizon and increment the entry $count[X_i^{\max}]$ of the array in $O(1)$ for each variable X_i.

The idea of Algorithm 5.4.1 is to use a Disjoint-Set T (also called union-find) data structure [Cor+01b] making it possible to have efficient operations for $T.Union(S_1, S_2)$, grouping two disjoint sets into a same set, and $T.Find(v)$ returning a "representative" of the set containing v. It is easy to extend a disjoint-set data structure with operations

$T.min(v)/T.max(v)$ returning the minimum/maximum value of the set containing value v. As detailed in the invariant of the algorithm, periods are grouped into a set S such that if $X_i^{\max} \in S$ then $v_i^{opt} = \min S$.

Algorithm 5.4.1: `StockingCost`: Computation of v_i^{opt} for all i

```
1  Initialize count as an array such that count[t] = |{i | X_i^max = t}|
2  Create a disjoint set data structure T with the integers
   t ∈ [min_i{X_i^min}, max_i{X_i^max}]
3  t ← max_i{X_i^max}
4  repeat
5      while count[t] > c do
6          count[t − 1] ← count[t − 1] + count[t] − c
7          count[t] ← c
8          T.Union(t − 1, t)
9          t ← t − 1
10     end
11     // Invariant: v_i^opt = t, ∀i ∈ {i | t ≤ X_i^max ≤ T.max(T.find(t))}
12     t ← t − 1
13 until t ≤ min_i{X_i^min}
14 // if count[min_i{X_i^min}] > c then the constraint is infeasible
15 ∀i : v_i^opt = T.min(T.find(X_i^max))
```

Example 7. *Consider the following instance* `StockingCost`$([X_1 \in [1,\ldots,3], X_2 \in [1,\ldots,6], X_3 \in [1,\ldots,7], X_4 \in [1,\ldots,7], X_5 \in [1,\ldots,8]], [d_1 = 3, d_2 = 6, d_3 = 7, d_4 = 7, d_5 = 8], H \in [0..4], c = 1)$. *At the beginning of the algorithm,* $count = [0,0,1,0,0,1,2,1]$ *and* $T = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\}\}$. Figure 5.1 *shows a corresponding representation of the orders.*

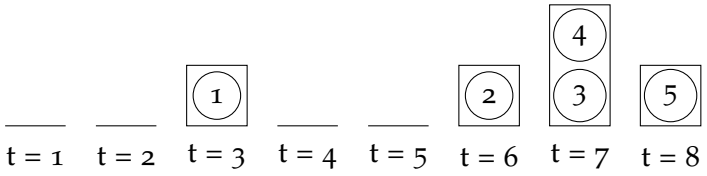


Figure 5.1: An optimal assignment of the instance of Example 7 without capacity restrictions

After the main loop of the algorithm, count $= [0,0,1,0,1,1,1,1]$ *(see a representation of the orders in* Figure 5.2*) and* $T = \{\{1\},\{2\},\{3\},\{4\},\{5,6,7\},\{8\}\}$. *Thus* $v^{opt}_{X_1} = 3$, $v^{opt}_{X_2} = v^{opt}_{X_3} = v^{opt}_{X_4} = 5$ *and* $v^{opt}_{X_5} = 8$.

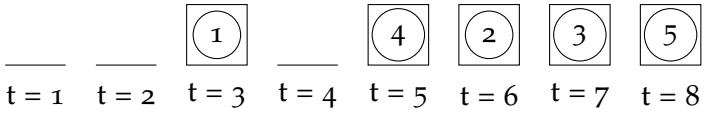


Figure 5.2: An optimal assignment of the instance of Example 7

Property 6 gives a lower bound on the evolution of the optimal stocking cost when assigning variable X_i to v. Unfortunately, this lower bound is not optimal. One can be convinced easily for instance that with $c = 1$, if $v < v^{opt}_i$ is assigned X_i, it virtually imposes to move to left at least all variables X_j such that $\{X_j \mid X^{\max}_j = v\}$. This suggests for c=1, the following improved lower bound for $H^{opt}_{X_i \leftarrow v}$:

$$H^{opt}_{X_i \leftarrow v} \geq H^{opt} + (v^{opt}_i - v) + |\{j \mid X^{\max}_j = v\}| \tag{5.3}$$

Example 8 illustrates that this lower bound is still not optimal. It is not sufficient just to consider the set $\{j \mid X^{\max}_j = v\}$ since more variables could be impacted.

Example 8. *Consider the following instance* `StockingCost`$([X_1 \in [1,\ldots,5], X_2 \in [1,\ldots,4], X_3 \in [1,\ldots,4]], [d_1 = 5, d_2 = 4, d_3 = 4], H \in [0,\ldots,10], c = 1)$ *with* $H^{opt} = 1$ *and* $v^{opt}_{X_1} = 5$. *For* $v = 4$, $H^{opt}_{X_1 \leftarrow 4} \geq H^{opt} + (v^{opt}_{X_1} - v) + |\{j \mid X^{\max}_j = v\}| = 1 + (5-4) + 2 = 4$. *Here,* $H^{opt}_{X_1 \leftarrow 4}$ *is really* 4. *For* $v = 3$, $H^{opt}_{X_1 \leftarrow 3} \geq H^{opt} + (v^{opt}_{X_1} - v) + |\{j \mid X^{\max}_j = v\}| = 1 + (5-3) + 0 = 3$ *but here* $H^{opt}_{X_1 \leftarrow 3} = 4$.

By Definition 5, a period t is *full* if it is using all its capacity $count[t] = c$. Note that the maximum number of full periods is reached when all orders are such that $X^{\max}_i = \max\{X^{\max}_i\}$. In this case, the number of full periods is $\lfloor \frac{n}{c} \rfloor$.

Property 7. *There are at most* $\lfloor \frac{n}{c} \rfloor$ *full periods.*

Let us use the next two definitions to evaluate the exact increase in cost of $H^{opt}_{X_i \leftarrow v}$ and to filter the decision variables.

Definition 9. $minfull[t]$ *is the latest period* $\leq t$ *which is not full. More exactly* $minfull[t] = \max\{t' \leq t \mid count[t'] < c\}$.

Definition 10. *$maxfull[t]$ is the earliest period $\geq t$ which is not full. More exactly $maxfull[t] = \min\{t' \geq t \mid count[t'] < c\}$.*

Property 8, stated below, gives the exact evolution of $H^{opt}_{X_i \leftarrow v}$ that allows the bound consistent filtering of $X_i^{\min}$.

Property 8. $\forall v < v_i^{opt}$, $H^{opt}_{X_i \leftarrow v} = H^{opt} + (v_i^{opt} - v) + (v - minfull[v]) = H^{opt} + v_i^{opt} - minfull[v]$.

Proof. The number of variables affected (that would need to be shifted by one to the left) by assigning $X_i \leftarrow v$ is equivalent to the impact caused by insertion of an artificial order with the domain $[-\infty, \ldots, v]$. This cost is $(v - minfull[v])$. Thus the exact impact of $X_i \leftarrow v$ is the number of variables affected by the move plus $(v_i^{opt} - v)$. □

Algorithm 5.4.2 computes $minfull[t], maxfull[t], \forall t$ in $O(T)$. This algorithm puts in the same set all consecutive periods t with $count[t] = c$ (ie full periods). Then it deduces $minfull[t]$ and $maxfull[t]$ for each period t.

Algorithm 5.4.2: `StockingCost`: Computation of $minfull[t]$ and $maxfull[t]$

```
1  Create a disjoint set data structure F with the integers
   t ∈ [min_i{X_i^min} − 1, max_i{X_i^max}]
2  t ← max_i{X_i^max}
3  repeat
4  |  if count[t] = c then
5  |  |  F.Union(t − 1, t)
6  |  end
7  |  t ← t − 1
8  until t < min_i{X_i^min}
9  ∀t : minfull[t] = F.min(F.find(t))
10 ∀t : |F.find(t)| > 1 : maxfull[t] = F.max(F.find(t)) + 1
11 ∀t : |F.find(t)| = 1 : maxfull[t] = F.max(F.find(t))
```

The next two properties are very important because if the new minimum for X_i falls on a full period, we can increase the lower bound further.

Property 9. *If a period t is full ($count[t] = c$) then $\forall i$:*

$$H^{opt}_{X_i \leftarrow t} = H^{opt}_{X_i \leftarrow t'},\ \forall t' \in [minfull[t], \ldots, maxfull[t][\text{ such that } t' < v_i^{opt}$$

Proof. Suppose that a period t is full. We know that $\forall t' \in [minfull[t], \ldots, maxfull[t][, \ minfull[t'] = minfull[t]$. Thus, $\forall t' \in [minfull[t], \ldots, maxfull[t][$ *such that* $t' < v_i^{opt}$, $H^{opt}_{X_i \leftarrow t'} = H^{opt} + v_i^{opt} - minfull[t'] = H^{opt} + v_i^{opt} - minfull[t] = H^{opt}_{X_i \leftarrow t}$. □

Property 10. *The function* $H^{opt}_{X_i \leftarrow t}$ *is monotone:* $H^{opt}_{X_i \leftarrow t} \geq H^{opt}_{X_i \leftarrow t'} \forall t < t' \leq v_i^{opt}$.

Proof. Consider t and t' such that: $t < t'$. From Property 10, if $t' \in [minfull[t], \ldots, maxfull[t][$ then $H^{opt}_{X_i \leftarrow t} = H^{opt}_{X_i \leftarrow t'}$. On the other hand, if $t' \geq maxfull[t]$ then $minfull[t'] \geq minfull[t]$. In this case, $H^{opt}_{X_i \leftarrow t} = H^{opt} + v_i^{opt} - minfull[t] \geq H^{opt} + v_i^{opt} - minfull[t'] = H^{opt}_{X_i \leftarrow t'}$. □

Now, we can filter the lower bound of each decision variable with Algorithm 5.4.3.

Proposition 11. Algorithm 5.4.3 *provides the bound consistent filtering rule for each* X_i *(*$BC(X_i^{\min})$*).*

Proof. We know that $newmin = v_i^{opt} - (H^{\max} - H^{\min})$ is a lower bound of $X_i^{\min}$ for each $i \in [1, \ldots, n]$. If the period $newmin$ is not full then there is a solution $\leq H^{\max}$ with $X_i = newmin$ and thus we have $BC(X_i^{\min})$. Otherwise, any solution with $X_i = newmin$ is $> H^{\max}$ (since at least one order in $newmin$ should be delay by one period). Also, from Property 9, any solution with $X_i = t$ such that $t \in [minfull[t], \ldots, maxfull[t][$ is $> H^{\max}$. Hence, from Property 10, the first available period that satifies the condition $\leq H^{\max}$ is $maxfull[newmin]$. □

Algorithm 5.4.3: `StockingCost`: Bound consistent filtering of $X_i^{\min}$

```
1 newmin ← v_i^opt − (H^max − H^min)
2 if count[newmin] = c then
3 |   newmin ← min{v_i^opt, maxfull[newmin]}
4 end
5 X_i^min ← max(X_i^min, newmin))
```

Example 9. *Consider the instance of* Example 7*:* `StockingCost`$([X_1 \in [1, \ldots, 3], X_2 \in [1, \ldots, 6], X_3 \in [1, \ldots, 7], X_4 \in [1, \ldots, 7], X_5 \in$

$[1,\ldots,8]], [d_1 = 3, d_2 = 6, d_3 = 7, d_4 = 7, d_5 = 8], H \in [0..4], c = 1)$. *We know that* $v^{opt}_{X_1} = 3$, $v^{opt}_{X_2} = v^{opt}_{X_3} = v^{opt}_{X_4} = 5$, $v^{opt}_{X_5} = 8$ *and* $count = [0,0,1,0,1,1,1,1]$. Figure 5.3 *shows a representation of an optimal solution of the associated problem.*

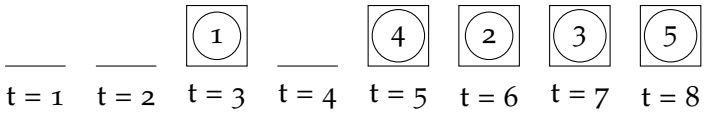


Figure 5.3: An optimal assignment of the instance of Example 7

After running Algorithm 5.3.1 *we have* $H^{opt} = 2$ *and thus* $H \in [2,\ldots,4]$. Algorithm 5.4.2 *gives* $F = \{\{0\},\{1\},\{2,3\},\{4,5,6,7,8\}\}$, $minfull = [1,2,2,4,4,4,4,4]$ *and* $maxfull = [1,2,4,4,9,9,9,9]$. Algorithm 5.4.3 *gives for:*

- X_1 : $newmin = 3 - 2 = 1$. $count[1] = 0$ *and* $X_1^{\min} = \max\{1,1\} = 1$;
- X_2, X_3, X_4 : $newmin = 5 - 2 = 3$. $count[3] = 1$, $newmin = \min\{4,5\} = 4$ *and* $X^{\min}_{j\in\{2,3,4\}} = \max\{1,4\} = 4$. Figure 5.4 *shows the evolution of* $H^{opt}_{X_3\leftarrow t}$. *Note that for* $t \in [1,\ldots,3]$, $H^{opt}_{X_3\leftarrow t} > H^{\max} = 4$.
- X_5 : $newmin = 8-2 = 6$. $count[6] = 1$, $newmin = \min\{8,9\} = 8$ *and* $X_5^{\min} = \max\{1,8\} = 8$.

Thus $X_1 \in [1,\ldots,3]$, $X_2 \in [4,\ldots,6]$, $X_3 \in [4,\ldots,7]$, $X_4 \in [4,\ldots,7]$ *and* $X_5 \in \{8\}$.

5.5 A COMPLETE FILTERING ALGORITHM IN $O(n)$

Algorithm 5.4.1 and Algorithm 5.4.2 for computing $v_i^{opt}, \forall i$ and $maxfull[t], \forall t$ presented so far have a complexity of $O(T)$. We describe, in this section, a complete self-contained version of the filtering for the `StockingCost` constraint running in $O(n)$ given a sorted version of the variables. Algorithm 5.5.1 keeps tracks of the orders in the same set (same v^{opt}) by maintaining two indexes j, k with the following properties:

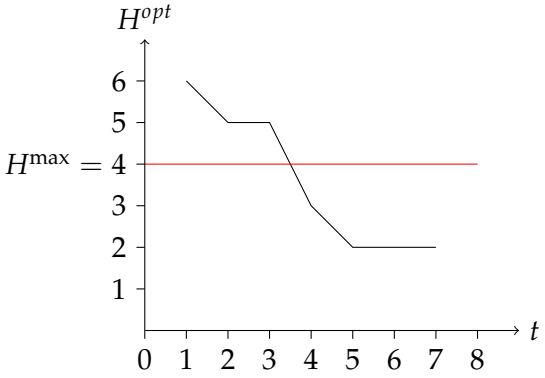


Figure 5.4: Evolution of $H^{opt}_{X_3 \leftarrow t}$

- After line 9, orders in $\{j, \ldots, i\}$ are the *open orders* $(i \mid X_i^{\max} \geq t)$ that still need to be placed into some periods in an optimal solution.
- After line 9, all the orders in $\{k, \ldots, i\}$ have the same v^{opt}. This value v^{opt} is only known when all the current remaining open orders can be placed into the current period. This is when the condition at line 14 is true.

Variable u keeps track of the $maxfull[t]$ potential value with $maxfull[t]$ implemented as a map with constant time insertion. Only periods t with $maxfull[t] > t$ are added to the map. Each time a full period t is discovered (at lines 21 and 33), one entry is added to the map. By Property 7 the number of entries added into the map is at most n.

Algorithm 5.5.2 just applies the filtering rules from Algorithm 5.4.3.

Implementation Details

Although Algorithm 5.5.1 is in $O(n)$, it requires the variables to be sorted. Since the filtering algorithms are called multiple times during the search process and only a small number of variables are modified between each call, simple sorting algorithms such as insertion or bubble sort are generally more efficient than classical sorting algorithms $O(n \log n)$.

The *map* can be a simple Hashmap but a simple implementation with two arrays of size T and a *magic number* incremented at each call

can be used to avoid computing hash functions and the map object creation/initialization at each call to the algorithm. One array contains the value for each key index in the map, and the other array contains magic numbers containing the value of the magic number at the insertion. An entry is present only if the value at corresponding index in the magic array is equal to the current magic number. Incrementing the magic number thus amounts to emptying the map in $O(1)$. The cost $O(T)$ at the map creation has to be paid only once and is thus amortized.

5.6 EXPERIMENTAL RESULTS

The experiments were conducted on instances of MI-DLS-CC-SC (Multiple Item - Discrete Lot Sizing - Constant Capacity - Setup Cost) problem: the PSP that is described in Section 3.3. Here the per-period stocking cost h is the same for all orders. We follow the methodology for experimental evaluation that is described in Section 1.4.

A CP Model

Let $date(p) \in [1, \ldots, T], \forall p \in [1, \ldots, n]$, represents the period in which the order p is satisfied. If *objStorage* is an upper bound on the total number of periods in which orders have to be held in stock, the stocking part can be modeled by the constraint:

$$\texttt{StockingCost}(date, dueDate, objStorage, 1)$$

Property 12. *There is no difference between two orders of the same item except for their due dates. Therefore given a feasible production schedule, if it is possible to swap the production periods of two orders involving the same item $(date(p_1), date(p_2)$ such that $item(p_1) = item(p_2))$, we obtain an identical solution with the same stocking cost.*

Based on Property 12, we remove such symmetries by adding precedence constraints on *date* variables involving by the same item:

$$date(p_1) < date(p_2), \forall (p_1, p_2) \in [1, \ldots, n] \times [1, \ldots, n] \text{ such that}$$

$$dueDate(p_1) < dueDate(p_2) \wedge item(p_1) = item(p_2)$$

Now the second part of the objective *objChangeover* concerning changeover costs has to be introduced in the model. This part is similar

Algorithm 5.5.1: `StockingCost`: Complete filtering algorithm in $O(n)$ - Part 1

Input: $X = [X_1, \ldots, X_n, X_{n+1}]$ sorted: $X_i^{\max} > X_{i+1}^{\max}$
$X_{n+1}^{\max} = -\infty$ `// artificial variable`

$H^{opt} \leftarrow 0$; $t \leftarrow X_1^{\max}$; $i \leftarrow 1$
$j \leftarrow 1$ `//` open orders $\{j, \ldots, i\}$ must be placed in some periods
$k \leftarrow 1$ `//` orders $\{k, \ldots, i\}$ have same v^{opt}
$u \leftarrow t + 1$
$maxfull \leftarrow map()$ `//` a map from int to int
while $i \leq n \vee j < i$ **do**
  while $i \leq n \wedge X_i^{\max} = t$ **do**
    $i \leftarrow i + 1$
  end
  `//` place at most c orders into period t
  for $i' \in [j, \ldots, \min(i-1, j+c-1)]$ **do**
    $H^{opt} \leftarrow H^{opt} + (d_{i'} - t)$
  end
  if $i - j \leq c$ **then** `//` all the open orders can be placed in t
    $full \leftarrow i - j = c$ `//` true if t is fill up completely
    $v_l^{opt} \leftarrow t, \forall l \in [k, \ldots, i)$
    $j \leftarrow i$
    $k \leftarrow i$
    if $full$ **then**
      `//` **Invariant (a):** $\forall t' \in [t, \ldots, u-1], count[t] = c$
      $maxfull[t] \leftarrow u$
      if $X_i^{\max} < t - 1$ **then**
        $u \leftarrow X_i^{\max} + 1$
      end
    end
    else
      $u \leftarrow X_i^{\max} + 1$
    end
    $t \leftarrow X_i^{\max}$
  end
  else `//` all open orders cannot be placed in t
    `//` **Invariant (b):** $\forall t' \in [t, \ldots, u-1], count[t] = c$
    $maxfull[t] \leftarrow u$
    $j \leftarrow j + c$ `//` place c orders into period t
    $t \leftarrow t - 1$
  end
end

Algorithm 5.5.2: `StockingCost`: Complete filtering algorithm in $O(n)$ - Filtering

```
1 H^min ← max(H^min, H^opt)
2 for i ∈ [1,...,n] do
3 |   newmin ← v_i^opt − (H^max − H^min)
4 |   if maxfull[t].hasKey(newmin) then
5 |   |   newmin ← min{v_i^opt, maxfull[newmin]}
6 |   end
7 |   X_i^min ← max(X_i^min, newmin))
8 end
```

to a *successor* CP model for the ATSP in which the cities to be visited represent the orders and the distances between them are the corresponding changeover costs. Let $successor(p), \forall p \in [1,\dots,n]$, define the order produced on the machine immediately after producing order p. We additionally create a dummy order $n+1$ to be produced after all the other orders. In the first step, a Hamiltonian circuit successor variable is imposed. This is achieved by using the classical *circuit* [Pes+98] constraint on successor variables for dynamic subtour filtering. The *date* and *successor* variables are linked with the element constraint by imposing that the production date of p is before the production date of its successors:

$$date(p) < date(successor(p)), \forall p \in [1,\dots,n]$$

As announced, the artificial production is scheduled at the end:

$$date(n+1) = T+1$$

Note that as with *date* variables, some symmetries can be broken. For two orders $n_1, n_2 \in [1,\dots,n]$ such that $dueDate(n_1) < dueDate(n_2)$ and $item(n_1) = item(n_2)$, we force that n_1 cannot be the successor of n_2 with $successor(n_2) \neq n_1$. Finally, a `minimumAssignment` constraint [FLM99b] is used on the *successor* variables and the changeover part of the objective *objChangeover*.

The objective to be minimized is simply the sum of stocking costs and changeover costs: $(objStorage \cdot h) + objChangeover$, in which h is the unit stocking cost.

Experimental results

In our experiments, we consider 100 random instances with:

- a planning horizon of 500 periods;
- the number of orders $n \in [490, \ldots, 500]$[4] with 10 different items. Note that in practice the number of orders is close to the number of periods;
- the changeover cost $q^{i,j} \in [10, \ldots, 50]$ for $i \neq j$ and $q^{i,i} = 0$ for each order i;
- the per-period stocking cost $h_i = 70$ for each order. We have tested different values of the per-period stocking cost and the results with this value seem representative[5].

All our source-code for the models, the global constraint, and the instances are available at [HSb, Hou+]. We compare the performance of the filtering algorithm due to `StockingCost`$(date, deadline, objStorage, 1)$ constraint with that achieved by the following two sets of constraints:

- *Basic*: the baseline model is obtained by decomposing `StockingCost` as:
  - `allDifferent(date)` using a forward checking filtering;
  - $\sum_p (dueDate(p) - date(p)) \cdot h \leq objStorage$.
- *MinAss*:
  - the `minimumAssignment` constraint with the LP reduced costs based filtering;
  - the `allDifferent` constraint with bound consistent filtering.

  Actually, after some experiments, this combination is one of the best (wrt filtering/time) state-of-the-art alternatives for the `StockingCost` constraint on the PSP.

4 At each period an item is randomly chosen and the corresponding demand is set to 1 with a certain probability. Then we only consider the feasible instances with $n \in [490, \ldots, 500]$.

5 Actually, more h_i is high, more the stocking part of the problem is important to solve the problem and more the best filtering based on the stocking costs is efficient. Here it is the filtering introduced in this chapter.

As search heuristic, we use the conflict ordering search (COS) [Gay+15] that performs well on the problem. The time limit to record the search tree by the baseline model is 60 seconds.

Table 5.1 shows the arithmetic average of the number of nodes and the time required for *Basic*, *MinAss*, and `StockingCost` respectively. Clearly, these results suggest that our `StockingCost` version offers a stronger and faster filtering than other decompositions. The `StockingCost` based model reduces the search tree by ≈ 1.9 wrt the `minimumAssignment` based model. Moreover, `StockingCost` is on average ≥ 8 times as fast as the other models. This is not surprising since filtering algorithm for `StockingCost` is in $O(n)$.

	`StockingCost`		*MinAss*		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	**8.81** 10^4	**7.4**	15.6 10^4	62.3	76.9 10^4	52.3
Av. gain factor	**11.4**	**8.8**	6.1	1.0	1.0	1.0

Table 5.1: Average results on 100 instances with $T = 500$: `StockingCost`, *MinAss*, and *Basic*

To further evaluate the different performances, Figure 5.5 and Figure 5.6 show the performance profiles (for `StockingCost`, *MinAss* and *Basic*) wrt the number of nodes visited and the time needed to complete the search respectively. We can see that:

- wrt nodes: for all the instances, `StockingCost` provides the best filtering. Note that for $\approx 30\%$ of instances, `StockingCost` is better than *MinAss* more than twice times;
- wrt time: `StockingCost` has also the best time. In particular, `StockingCost` is at least 5 times as fast as *MinAss* (and *Basic*) for $\approx 90\%$ of instances. Also `StockingCost` is at least 10 times as fast as *MinAss* (and *Basic*) for $\approx 30\%$ of instances.

5.7 CONCLUSION

We have introduced a new global constraint `StockingCost` to handle the stocking aspect of the LS problem when using CP. We have

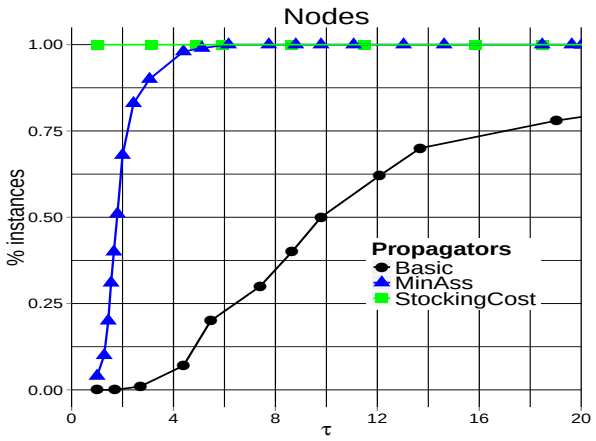


Figure 5.5: Performance profiles - Nodes: `StockingCost`, *MinAss*, and *Basic*

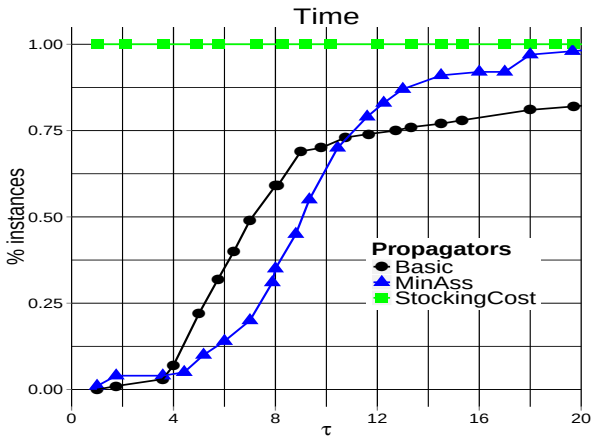


Figure 5.6: Performance profiles - Time: `StockingCost`, *MinAss*, and *Basic*

described an advanced filtering algorithm achieving bound consistency with a time complexity linear in the number of variables. The experimental results show the pruning and time efficiency of the `StockingCost` constraint on a variant of the CLSP compared to various decompositions of the constraint.

6 THE ITEM DEPENDENT STOCKINGCOST CONSTRAINT

In the previous chapter, we have introduced the global constraint `StockingCost` to compute the total number of time-slots between the production times and the due dates in a Capacitated Lot Sizing Problem (CLSP). This total number of periods can be converted into a stocking-cost by multiplying it by a unique per period stocking cost fixed parameter. Unfortunately, this constraint does not allow accurate computation of the stocking cost when the per period stocking cost is order dependent. We generalize the `StockingCost` constraint allowing a per period stocking cost that is potentially different for each order. We also allow the production capacity to vary over time. We propose an efficient filtering algorithm in $O(n \log n)$ with n is the number of orders to be produced. We experimentally demonstrate on a variant of the CLSP that our new filtering algorithm scales well and is competitive wrt the `StockingCost` constraint when the stocking cost is the same for all orders.

6.1 INTRODUCTION

The `StockingCost` constraint is well suited to compute the stocking cost when the per period stocking cost is the same for every order. Unfortunately, in many problems, it is order dependent since some order types (items) are more or less expensive to hold in stock. In that case, each order has the per period stocking cost of the corresponding item. To take these cases into account, we generalize the `StockingCost`

constraint. The new constraint is denoted `IDStockingCost` (ID stands for Item Dependent).

We use a similar approach than the one described in Chapter 5 to achieve filtering of the `IDStockingCost` constraint. However, here, to have an efficient algorithm we make a relaxation on the associated problem by only considering the due date and the capacity restrictions. This relaxation allows us to have a linearithmic algorithm to achieve a filtering of the `IDStockingCost` constraint. The filtering algorithm introduced does not achieve bound consistency but the experimental results show that it scales well.

This chapter is organized as follows: Section 6.2 gives the formal definition of the item dependent stockingCost constraint `IDStockingCost`; Section 6.3 describes a filtering algorithm for the cost variable based on a relaxed problem; Section 6.4 shows how to filter the decision variables based on an optimal solution of the relaxed problem; Section 6.5 presents some computational experiments on the PSP (that is described in Section 3.3) and Section 6.6 concludes.

6.2 THE ITEM DEPENDENT STOCKINGCOST CONSTRAINT

One has a time horizon T, a set $i = 1, \ldots, n$ of orders each with a due date $d_i \in [1, \ldots, T]$ and a per period stocking cost h_i. There is a machine which has c_t units of production capacity in period t. Producing an order in period t consumes one unit of capacity. The problem is to produce each order by its due date at latest without exceeding the machine capacity and to minimize the sum of the stocking costs of the orders.

The Item Dependent StockingCost Constraint `IDStockingCost` is the generalization of `StockingCost` constraint in which the orders have different stocking costs and the periods have different capacities. The `IDStockingCost`constraint takes the following form:

$$\texttt{IDStockingCost}([X_1, \ldots, X_n], [d_1, \ldots, d_n], [h_1, \ldots, h_n], H, [c_1, \ldots, c_T])$$

in which:

- n is the total number of orders to be produced;
- T is the total number of periods over the planning horizon $[1, \ldots, T]$;

- the variable X_i is the date of production of order i on the machine, $\forall i \in [1, \ldots, n]$;
- the integer d_i is the due-date for order i, $\forall i \in [1, \ldots, n]$, $d_i \leq T$;
- the integer $h_i \geq 0$ is the stocking cost for order i, $\forall i \in [1, \ldots, n]$;
- the integer $c_t \geq 0$ is the maximum number of orders the machine can produce during the period t (production capacity for t), $\forall t \in [1, \ldots, T]$;
- the variable H is an upper bound on the total stocking cost.

As in the previous chapter, without loss of generality, we assume that $X_i \leq d_i, \forall i$. We give below some possible decompositions of the `IDStockingCost` constraint.

6.2.1 *Decomposing the constraint*

The `IDStockingCost` constraint holds when:

$$|\{i \mid X_i = t\}| \leq c_t, \forall t \tag{6.1}$$

$$\sum_i (d_i - X_i) \cdot h_i \leq H \tag{6.2}$$

This decomposition (with $T + 1$ constraints) imposes that (6.1) the capacity of the machine is respected in every period t and (6.2) H is an upper bound on the total stocking cost.

As proposed in the previous chapter, the T constraints in equation (6.1) can be replaced by a global cardinality constraint `gcc`. Note that for $c_t = 1, \forall t \in [1, \ldots, T]$, the `gcc` constraint can be replaced by an `allDifferent` [Rég94, Pug98] constraint. An even stronger model is obtained by replacing the constraints in (6.1) and (6.2) by an arc-consistent `cost-gcc` [Ró2] constraint. For unary capacity one can use the `minimumAssignment` [FLM99b, DCP16] constraint with filtering based on reduced costs. The filtering algorithms for the `minimumAssignment` and `cost-gcc` execute in $O(T^3) \approx O(n^3)$. This chapter presents a fast filtering algorithm for `IDStockingCost` running in $O(n \log n)$.

In the rest of the chapter, without loss of generality, we assume that:

1. all orders $i \in [1, \ldots, n]$ are such that $h_i > 0$. If this is not the case, one can produce $n_0 = |\{i \mid h_i = 0\}|$ orders in the first n_0

periods and then consider the other orders over the planning horizon $[n_0 + 1, \ldots, T]$;

2. the gcc constraint is bound consistent. The gcc constraint is bound consistent means that for each $X_i, \forall v_i \in \{X_i^{\min}, X_i^{\max}\}$ and $\forall X_j \neq X_i : \exists v_j \in [X_j^{\min}, \ldots, X_j^{\max}]$ such that $\sum_k (v_k = t) \leq c_t, \forall t$. For example, consider three orders $X_1 \in [3,4]$, $X_2 \in [3,4]$, $X_3 \in [1,4]$ and $c_1 = 0$, $c_2 = c_3 = c_4 = 1$. We can see that X_3 can neither take the value 4 nor 3 because the interval $[3,4]$ must be reserved for X_1 and X_2. On the other hand, X_3 cannot take value 1 because $c_1 = 0$. Thus gcc is bound consistent if $X_1 \in [3,4]$, $X_2 \in [3,4]$ and $X_3 = \{2\}$.

6.3 FILTERING OF THE COST VARIABLE H

This section explains how we filter the lower bound on the cost variable H in $O(n \log n)$. This is performed by computing the optimal cost of a related relaxed problem.

The best lower bound H^{opt} of the global stocking costs variable H can be obtained by solving the following problem:

$$
\begin{aligned}
& H^{opt} = \min \sum_i (d_i - X_i) \cdot h_i \\
(\mathcal{P}) \quad & |\{i \mid X_i = t\}| \leq c_t, \forall t \\
& X_i \in D_i, \forall i
\end{aligned}
$$

in which D_i is the domain of the variable X_i ie the set of values that X_i can take.

The problem $\mathcal{P}$ can be solved with a max-flow min-cost algorithm on the bipartite graph linking orders and periods [Ró2]. Indeed the cost of assigning $X_i \leftarrow t$ can be computed as $(d_i - t) \cdot h_i$ if $t \in D_i$, $+\infty$ otherwise. For unit capacity, it is a minimum assignment problem that can be solved in $O(T^3)$ with the Hungarian algorithm. The costs on the arcs have the particularity to evolve in a convex way (linearly) along the values. But even in this case, we are not aware of a faster minimum assignment algorithm. Since our objective is to design a fast scalable filtering, some relaxations must be introduced.

Let $X_i^{\min}$ and $H^{\min}$ (resp. $X_i^{\max}$ and $H^{\max}$) denote the minimal (resp. maximal) value in the finite domain of variable X_i and H. The relaxation we make is to assume that X_i can take any value $\leq X_i^{\max}$

without holes: $D_i = [1, \ldots, X_i^{\max}]$. Our filtering algorithm is thus based on a relaxed problem in which the orders can be produced in any period before their minimum values (but not after their maximum values). Let $\mathcal{P}^r$ denote this new relaxed problem and $(H^{opt})^r$ denote its optimal value. $(H^{opt})^r$ gives a valid lower bound to possibly increase $H^{\min}$.

To solve this relaxation, one can compute $(H^{opt})^r$ in a greedy fashion assigning the production periods from the latest to the earliest. Clearly, the orders should be produced as late as possible (i.e. as close as possible to their due-date) in order to minimize their individual stocking cost. Unfortunately, the capacity constraints usually prevent us from assigning every X_i to its maximum value $X_i^{\max}$.

We below characterize an optimal solution of $\mathcal{P}^r$ after recalling the definition of full period.

Definition 11. *Let us denote by assPeriod a valid assignment vector in which assPeriod[i] is the value (period) taken by X_i. Considering a valid assignment and a period t, the boolean value $t.full$ indicates whether this period is used at maximal capacity or not: $t.full \equiv |\{i \mid assPeriod[i] = t\}| = c_t$.*

Property 13. *Consider a valid assignment assPeriod: assPeriod[i], $\forall i \in [1, \ldots, n]$ wrt $\mathcal{P}^r$. If this assignment is optimal, then*
(i) $\forall i \in [1, \ldots, n], \nexists t : (assPeriod[i] < t) \wedge (X_i^{\max} \geq t) \wedge (\neg t.full)$.

Proof. Let assume that *assPeriod* does not respect the criterion (i). This means that $\exists X_k \wedge \exists t : (assPeriod[k] < t) \wedge (X_k^{\max} \geq t) \wedge (\neg t.full)$. In that case, by moving X_k from $assPeriod[k]$ to t, we obtain a valid solution that is better than *assPeriod*. The improvement is: $(t - assPeriod[k]) \cdot h_k$. Thus the criterion (i) is a necessary condition for optimality of $\mathcal{P}^r$. □

Corollary 14. *Any optimal solution assPeriod uses the same set of periods: $\{assPeriod[k] : \forall k\}$ and this set is unique.*

The condition of Property 13 is not sufficient to ensure the optimality. Actually, if in a solution of $\mathcal{P}^r$ a valid permutation between two orders decreases the cost of that solution, then this latter is not optimal.

Property 15. *Consider a valid assignment assPeriod: assPeriod[i], $\forall i \in [1, \ldots, n]$ wrt $\mathcal{P}^r$. If this assignment is optimal, then*
(ii) $\nexists (k_1, k_2)$: $(assPeriod[k_1] < assPeriod[k_2]) \wedge (h_{k_1} > h_{k_2}) \wedge (X_{k_1}^{\max} \geq assPeriod[k_2])$.

Proof. Let assume that *assPeriod* does not respect the criterion (ii). That means that $\exists(X_{k_1}, X_{k_2}) : (assPeriod[k_1] < assPeriod[k_2]) \wedge (h_{k_1} > h_{k_2}) \wedge (X^{\max}_{k_1} \geq assPeriod[k_2])$. In that case, by swapping the orders k_1 and k_2, we obtain a valid solution that is better than *assPeriod*. The improvement is : $(assPeriod[k_2] - assPeriod[k_1]) \cdot h_{k_1} - (assPeriod[k_2] - assPeriod[k_1]) \cdot h_{k_2} > 0$. Thus the criterion (ii) is a necessary optimality condition. □

The next proposition states that the previous two necessary conditions are also sufficient for testing optimality to problem $\mathcal{P}^r$.

Proposition 16. *Consider a valid assignment assPeriod: $assPeriod[i], \forall i \in [1, \ldots, n]$ wrt $\mathcal{P}^r$. This assignment is optimal if and only if*
(i) $\forall i \in [1, \ldots, n], \nexists t : (assPeriod[i] < t) \wedge (X^{\max}_i \geq t) \wedge (\neg t.full)$
(ii) $\nexists(k_1, k_2)$*:* $(assPeriod[k_1] < assPeriod[k_2]) \wedge (h_{k_1} > h_{k_2}) \wedge (X^{\max}_{k_1} \geq assPeriod[k_2])$.

Proof. Without loss of generality, we assume that 1) All the orders have different stocking costs : $\forall(k_1, k_2) : h_{k_1} \neq h_{k_2}$. If this is not the case for two orders, we can increase the cost of one by an arbitrarily small value. 2) Unary capacity for all periods : $c_t = 1, \forall t$. The periods with zero capacity can simply be discarded and periods with capacities $c_t > 1$ can be replaced by c_t "artificial" unit periods. Of course the planning horizon changes. To reconstruct the solution of the initial problem, one can simply have a map that associates to each artificial period the corresponding period in the initial problem. 3) All the orders are sorted such that $assPeriod[i] > assPeriod[i+1]$.

We know that (i) and (ii) are necessary conditions for optimality. The objective is to prove that a solution that respects (i) and (ii) is unique and thus also optimal. From Corollary 13, we know that all optimal solutions use the same set of periods: $\{t_1, t_2, \ldots, t_n\}$ with $t_1 = \max_i\{X^{\max}_i\} > t_2 > \ldots > t_n$. Let $\mathcal{C}_1 = \{k \mid X^{\max}_k \geq t_1\}$ be the orders that could possibly be assigned to the first period t_1. To respect the property (ii), for the first period t_1, we must select the unique order $\text{argmax}_{k \in \mathcal{C}_1} h_k$. Now assume that periods $t_1 > t_2 > \ldots > t_i$ were successively assigned to orders $1, 2, \ldots, i$ and produced the unique partial solution that can be expanded to a solution for all the orders $1, \ldots, n$. We show that we have also a unique choice to expand the solution in period t_{i+1}. The order to select in period t_{i+1} is $\text{argmax}_{k \in \mathcal{C}_{i+1}}\{h_k\}$ with $\mathcal{C}_{i+1} = \{k : k > i \wedge X^{\max}_k \geq t_{i+1}\}$ is the set of orders that could possibly be assigned in period t_{i+1}. Indeed, selecting any other order would

lead to a violation of property (ii). Hence the final complete solution obtained is unique. □

Algorithm 6.3.1 considers orders sorted decreasingly according to their $X_i^{\max}$. A virtual sweep line decreases in period starting at $\max_i\{X_i^{\max}\}$. The sweep line collects in a priority queue all the orders that can be possibly scheduled in that period (such that $t \leq X_i^{\max}$). Each time it is decreased new orders can possibly enter into a priority queue (loop $12-14$). The priorities in the queue are the stocking costs h_i of the orders. A large cost h_i means that this order has a higher priority to be scheduled as late as possible (since t is decreasing). The variable *availableCapacity* represents the current remaining capacity in period t. It is initialized to the capacity c_t (line 10) and decreased by one each time an order is scheduled at t (line 19). An order is scheduled at lines $15-20$ by choosing the one with highest stocking cost from *ordersToSchedule*. The capacity and the cost are updated accordingly. The orders are scheduled at t until the capacity is reached (and then the current period is updated to the previous period with non null capacity) or the queue is empty (and then the algorithm jumps to the maximum value of the next order to be produced). This process is repeated until all orders have been scheduled.

At the end, $optPeriod[i], \forall i \in [1,\dots,n]$ is the optimal schedule showing the period assigned to the order i. We thus have at the end: $\sum_{i=1}^{n}(d_i - optPeriod[i]) \cdot h_i = (H^{opt})^r$ and $optOrders[t] = \{i \mid optPeriod[i] = t\}, \forall t \in [1,\dots,T]$ is the set of orders produced in period t.

Proposition 17. Algorithm 6.3.1 *computes an optimal solution of* $\mathcal{P}^r$ *in* $O(n \log n)$.

Proof. Algorithm 6.3.1 works as suggested in the proof of Proposition 16 and then Invariant (c) and Invariant (d) hold for each t from $\max_i\{X_i^{\max}\}$. Thus the solution returned by the algorithm 1) is feasible and 2) respects the properties (i) and (ii) of Proposition 16 and then is an optimal solution.
Complexity: the loop at lines $12-14$ that increments the order index i from 1 to n ensures that the main loop of the algorithm is executed $O(n)$ times. On the other hand, each order is pushed and popped exactly once in the queue *ordersToSchedule* in the main loop. Since *ordersToSchedule* is a priority queue, the global complexity is $O(n \log n)$. □

The next example shows the execution of Algorithm 6.3.1 on a small instance.

Example 10. *Consider the following instance:* `IDStockingCost`$([X_1 \in [1,\dots,4], X_2 \in [1,\dots,5], X_3 \in [1,\dots,4], X_4 \in [1,\dots,5], X_5 \in [1,\dots,8], X_6 \in [1,\dots,8]], [d_1 = 4, d_2 = 5, d_3 = 4, d_4 = 5, d_5 = 8, d_6 = 8], [h_1 = 3, h_2 = 10, h_3 = 4, h_4 = 2, h_5 = 2, h_6 = 4], H \in [0,\dots,34], c_1 = c_2 = c_4 = c_5 = c_6 = c_7 = c_8 = 1, c_3 = 0)$. *The main steps of the execution of* Algorithm 6.3.1 *are:*

- $t = 8$, *ordersToSchedule* $= \{5,6\}$ *and* $X_6 \leftarrow 8$. $(H^{opt})^r = 0$;
- $t = 7$, *ordersToSchedule* $= \{5\}$ *and* $X_5 \leftarrow 7$. $(H^{opt})^r = h_5 = 2$;
- $t = 5$, *ordersToSchedule* $= \{4,2\}$ *and* $X_2 \leftarrow 5$. $(H^{opt})^r = 2$;
- $t = 4$, *ordersToSchedule* $= \{4,1,3\}$ *and* $X_3 \leftarrow 4$. $(H^{opt})^r = 2$;
- $t = 3$, *ordersToSchedule* $= \{4,1\}$ $(c_3 = 0)$;
- $t = 2$, *ordersToSchedule* $= \{4,1\}$ *and* $X_1 \leftarrow 2$. $(H^{opt})^r = 2 + 2 \cdot h_1 = 8$;
- $t = 1$, *ordersToSchedule* $= \{4\}$ *and* $X_4 \leftarrow 1$. $(H^{opt})^r = 8 + 4 \cdot h_4 = 16$.

Then $H \in [16,\dots,34]$.

Figure 6.1 *(resp.* Figure 6.2*) shows the optimal period assignments for* $\mathcal{P}^r$ *without (resp. with) the capacity restrictions.*

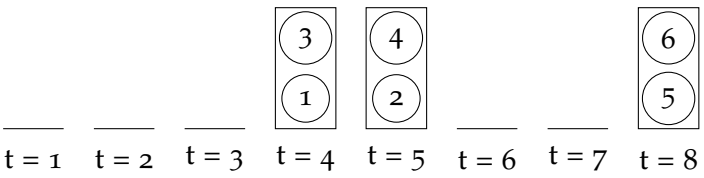


Figure 6.1: $\mathcal{P}^r$ without capacity restrictions

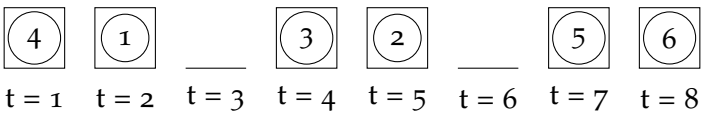


Figure 6.2: An optimal assignment for $\mathcal{P}^r$

Algorithm 6.3.1: `IDStockingCost`: Filtering of lower bound on H with $(H^{opt})^r$ in $O(n \log n)$

Input: $X = [X_1, \ldots, X_n]$ sorted ($X_i^{\max} \geq X_{i+1}^{\max}$)

```
1 (H^opt)^r ← 0 // total minimum stocking cost for P^r
2 optPeriod ← map() // optPeriod[i] is the period assigned to
    order i
3 // orders placed in t sorted top-down in non increasing h_i
4 ∀t : optOrders[t] ← stack()
5 ordersToSchedule ← priorityQueue() // priority=h_i
6 fullSetsStack ← stack() // stack of full sets
7 i ← 1
8 ... // see Algorithm 6.3.2
```

In an optimal solution of $\mathcal{P}^r$, a period t is full ($t.full$) if and if its capacity is reached: $t.full \equiv |optOrders[t]| = c_t$. Actually, an optimal solution of $\mathcal{P}^r$ is a sequence of full periods (obtained by scheduling orders as late as possible) separated by some non full periods. Let us formally define these sequences of production periods (after adapting the definitions of $minfull$ and $maxfull$ from Chapter 5). We call them full sets. These are used to filter the decision variables (see section 6.4).

Definition 12. *For a period t with $c_t > 0$, $minfull[t]$ is the largest period $\leq t$ such that all orders $X_k : X_k^{\max} \geq minfull[t]$ have $optPeriod[X_k] \geq minfull[t]$.*

Definition 13. *For a period t with $c_t > 0$, $maxfull[t]$ is the smallest period $\geq t$ such that all orders $X_k : X_k^{\max} > maxfull[t]$ have $optPeriod[X_k] > maxfull[t]$.*

For the instance in Example 10, $minfull[5] = minfull[4] = minfull[3] = minfull[2] = minfull[1] = 1$ and $maxfull[5] = maxfull[4] = maxfull[3] = maxfull[2] = maxfull[1] = 5$.

Definition 14. *An ordered set of periods $fs = \{M, \ldots, m\}$ (with $M > \ldots > m$) is a full set iff:*

$$(\forall t \in fs \setminus \{m\} : c_t > 0 \wedge t.full) \wedge (\forall t \in fs, maxfull[t] = M \wedge minfull[t] = m).$$

We consider that $minfull[fs] = m$ and $maxfull[fs] = M$.

For the instance in Example 10, there are two full sets: $\{8, 7\}$ and $\{5, 4, 2, 1\}$.

Algorithm 6.3.2: `IDStockingCost`: Filtering of lower bound on H with $(H^{opt})^r$ in $O(n \log n)$ - Part 2

```
1  ... // see Algorithm 6.3.1
2  i ← 1
3  while i ≤ n do
4      fullSet ← stack()
5      t ← X_i^max // current period
6      // Invariant (a): t is a maxfull period
7      availableCapacity ← c_t // available capa at t
8      repeat
9          while i ≤ n ∧ X_i^max = t do
10             ordersToSchedule.insert(i)
11             i ← i + 1
12         end
13         if availableCapacity > 0 then
14             j ← ordersToSchedule.delMax() // order with highest
                   cost
15             optPeriod[j] ← t
16             optOrders[t].push(j)
17             availableCapacity ← availableCapacity − 1
18             (H^opt)^r ← (H^opt)^r + (d_j − t) · h_j
19         end
20         else
21             // Invariant (b): t is a full period with c_t > 0
22             fullSet.push(t)
23             t ← previousPeriodWithNonNullCapa(t)
24             availableCapacity ← c_t
25         end
26         // Invariant (c): ∀i such that optPeriod[i] is defined:
               condition (i) of Proposition 16 holds
27         // Invariant (d): ∀k1, k2 such that optPeriod[k1] and
               optPeriod[k2] are defined: condition (ii) of Proposition
               16 holds
28     until ordersToSchedule.size > 0
29     // Invariant (e): t is a minfull period
30     fullSet.push(t)
31     // Invariant (f): fullSet is a full set
32     fullSetsStack.push(fullSet)
33 end
34 H^min ← max(H^min, (H^opt)^r)
```

Proposition 18. Algorithm 6.3.1 *computes* $fullSetsStack$*, a stack of all full sets of an optimal solution of* $\mathcal{P}^r$*.*

Proof. Invariant:

(a) and (e) - invariants for the maxfull and minfull periods.

Note that since gcc is bound consistent, $\forall i : c_{X_i^{\max}} > 0$. Consider the first iterations of the algorithm. At the beginning, $t_{\max} = \max_i\{X_i^{\max}\}$ is a maxfull period (by definition). Exit the loop $11 - 25$ means that all orders in $\{k \mid t_{\max} \geq X_k^{\max} \geq t\}$ (t is the current period) are already produced and the current period t is the closest period to $t_{\max}$ such that all orders in $\{k \mid t_{\max} \geq X_k^{\max} \geq t\}$ are produced: the current period t is then the minfull of all orders in $\{k \mid t_{\max} \geq X_k^{\max} \geq t\}$. Hence Invariant (a) and Invariant (e) hold for the first group of orders. The algorithm does the same process - when it starts at line 9 with another group of orders not yet produced - until all orders are produced. We know that: $\forall i : c_{X_i^{\max}} > 0$. Then, for each group of orders i.e. each time the algorithm comes back to line 9 (resp. line 26), the current t is a maxfull (resp. minfull).

(b) At line 22, t is a full period with $c_t > 0$.

At line 22, for the current period t: $availableCapacity = 0$ and at least one order is produced at t before. Thus $t.full$ and $c_t > 0$.

(f) $fullSet$ is a full set

This invariant holds because the invariants (a), (b) and (e) hold.

The algorithm starts from $\max_i\{X_i^{\max}\}$ and Invariant (f) holds until the last order is produced. Then the proposition is true. □

Implementation details

We consider each full set fs as a stack of periods such that $fs.first = minfull[fs]$ and $fs.last = maxfull[fs]$. The list of all full sets of an optimal solution is a stack of full sets: $fullSetsStack$; such that $fullSetsStack.first.firt \leq t, \forall t \in fullSetsStack$. Concretely, we implement a special data structure $StackOfStack$ that is a stack of stacks of integers with two arrays: the first one contains all integers involved in the $StackOfStack$ and the second one contains the cumulative size of different stacks pushed. The method $StackOfStack.push(i)$ pushes the integer i on the top of the current stack and the method $StackOfStack.pushStack()$ closes the current stack (if not empty) and starts a new empty one.

6.4 PRUNING THE DECISION VARIABLES X_i

Let $H^{opt}_{X_i \leftarrow v}$ (resp. $(H^{opt}_{X_i \leftarrow v})^r$) denote the optimal cost of $\mathcal{P}$ (resp. $\mathcal{P}^r$) in a situation in which X_i is forced to take the value v in an optimal solution of $\mathcal{P}$ (resp. $\mathcal{P}^r$), that is equivalent to adding the constraint $X_i = v$ to $\mathcal{P}$ (resp. $\mathcal{P}^r$). If $H^{opt}_{X_i \leftarrow v} > H^{\max}$, then the resulting problem is inconsistent and v can be safely removed from the domain of X_i. On the other hand, we know that $H^{opt}_{X_i \leftarrow v} \geq (H^{opt}_{X_i \leftarrow v})^r$. If $(H^{opt}_{X_i \leftarrow v})^r > H^{\max}$ (with $v < optPeriod[i]$), then v can be removed from the domain of X_i. To filter the decision variables X_i, the idea is to find a valid lower bound for $(H^{opt}_{X_i \leftarrow v})^r$ by performing some sensitivity analysis of the optimal solution of $\mathcal{P}^r$ returned by Algorithm 6.3.1.

If X_i is forced to take a value v ($v < optPeriod[i]$), it increases $(H^{opt})^r$ by at least $(optPeriod[i] - v) \cdot h_i$ but an additional production slot in $optPeriod[i]$ becomes free in the associated optimal solution. Consequently, the production of some orders can possibly be delayed and $(H^{opt})^r$ decreased. Formally,

Definition 15. *Let $newoptPeriod[j], \forall j \in [1, \ldots, n] \setminus \{i\}$ denote the new optimal assignment of periods when the order i is removed from its position $optPeriod[i]$.*

$gainCost[t]_i$ is the maximum cost decrease when a production scheduled in t is freed by removing an order i from $t = optPeriod[i]$ in an optimal solution of $\mathcal{P}^r$:

$$gainCost[t]_i = \sum_{j \in [1,\ldots,n] \setminus \{i\}} (newoptPeriod[j] - optPeriod[j]) \cdot h_j$$

Of course, $newoptPeriod[j], \forall j \in [1, \ldots, n] \setminus \{i\}$ must respect the two conditions for optimality of Proposition 16. It is worth noting that only full periods t_k with $c_{t_k} > 0$ can have $gainCost[t_k] > 0$. All others t_j have $gainCost[t_j] = 0$. Actually, a period t is not full means that there is at least one free place in t for production. Since these places are not used in the initial optimal assignment then they will not be used if another place in t is freed (see condition (i) of Proposition 16).

Example 11. *Consider the instance of Example 10 and its optimal solution represented in* Figure 6.2*:*

- *period 8: if the order 6 is removed from its optimal period 8, then $newoptPeriod[5] = 8$ (Figure 6.3) and $newoptPeriod[j] = optPeriod[j], \forall j \notin \{5, 6\}$.*
  $gainCost[8]_6 = h_5 = 2$;

- *period 7:* $newoptPeriod[j] = optPeriod[j], \forall j \neq 5$.
  $gainCost[7]_5 = 0$;
- *period 5: if the order* 2 *is removed from its optimal period* 5, *then* $newoptPeriod[4] = 5$ (Figure 6.4) *and* $newoptPeriod[3] = optPeriod[3]$,
  $newoptPeriod[1] = optPeriod[1]$ *because* $d_1, d_3 < 5$.
  $gainCost[5]_2 = (5-1) \cdot h_4 = 8$;
- *period 4:* $gainCost[4]_3 = 2 \cdot h_1 + h_4 = 8$ (Figure 6.5);
- *period 2:* $gainCost[2]_1 = h_4$ (Figure 6.6);
- *period 1:* $gainCost[1]_4 = 0$.

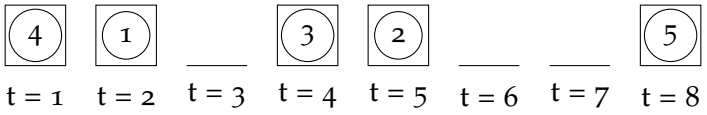


Figure 6.3: An optimal assignment for $\mathcal{P}^r$ without X_6

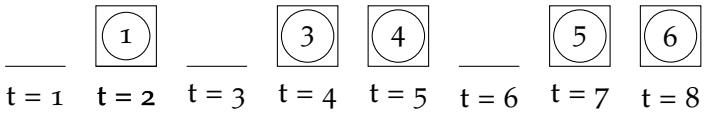


Figure 6.4: An optimal assignment for $\mathcal{P}^r$ without X_2

Intuitively, one can say that, if a place is freed in a full period t, only orders k that have $optPeriod[k] \leq t$ in the full set of t will eventually move. More precisely, for each full period t, let $left[t]$ be the set of orders such that: $left[t] = \{i \mid optPeriod[i] \in [minfull[t], \ldots, t[\}$.

Proposition 19. *If a full period* t *is no longer full due to the removal of order* i *(with* $optPeriod[i] = t$*) from* t, *then only orders in* $k \in left[t]$ *can have* $newoptPeriod[k] \neq optPeriod[k]$. *All other orders* j *have the same optimal period* $newoptPeriod[j] = optPeriod[j]$.

Proof. We run Algorithm 6.3.1 again with order i removed:

1. all orders k with $optPeriod[k] > maxfull[t]$ will conserve their respective optimal periods because $X_i^{\max} < optPeriod[k], \forall k$ and then order i is not taken into account (in the queue *ordersToSchedule*) for optimal assignment of periods $> maxfull[t]$;

2. all orders k with $t \leq optPeriod[k] \leq maxfull[t]$ will conserve their respective optimal periods. Actually, from Proposition 16, we know that for a given order k, $X_i^{\max} < optPeriod[k]$ or $h_k \geq h_i$. In these two cases, the presence/absence of X_i does not change the decision taken for orders k with their optimal periods in $[t, \ldots, maxfull[t]]$;

3. all orders k with $optPeriod[k] < minfull[t]$ will conserve their respective optimal periods because order i is not taken into account (in the queue *ordersToSchedule*) for optimal assignment of periods for all these orders.

□

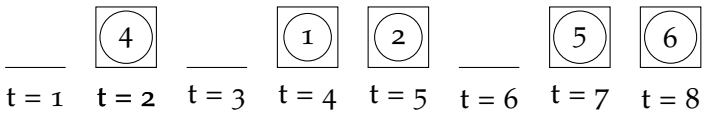


Figure 6.5: An optimal assignment for $\mathcal{P}^r$ without X_3

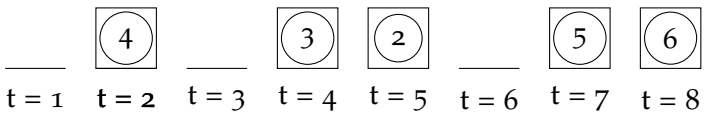


Figure 6.6: An optimal assignment for $\mathcal{P}^r$ without X_1

Property 20. *For a period t, $gainCost[t]_i$ does not depend on the order i in $optOrders[t]$ that is removed. We can simplify the notation from $gainCost[t]_i$ to $gainCost[t]$: $gainCost[t] = gainCost[t]_i, \forall i \in optOrders[t]$.*

Corollary 21. *For a full period t with $c_t > 0$:*

$$gainCost[t] = \sum_{j \in left[t]} (newoptPeriod[j] - optPeriod[j]) \cdot h_j$$

Observe that only orders k in $left[t]$ that have $X_k^{\max} \geq t$ can replace the order removed from t in the new optimal assignment. For each full period t, let $candidate[t]$ denote the set of orders $\in left[t]$ that can jump to the freed place in t when an order is removed from t. Formally, for a full period t, $candidate[t] = \{i \in left[t] \mid X_i^{\max} \geq t\}$. Let s_t denote the order that will replace the removed order in t: $s_t \in candidate[t] \wedge$

$newoptPeriod[s_t] = t$. For a given full period t with $c_t > 0$, $gainCost[t]$ depends on the order s_t and also depends on $gainCost[optPeriod[s_t]]$ since there is recursively another freed place created when s_t jumps to t.

We want to identify the order s_t that will take the freed place in t when an order is removed from t. This order must have the highest "potential *gainCost*" for period t among all other order in $candidate[t]$. More formally,

Definition 16. *Let $(gainCost[t])^k$ be the potential $gainCost[t]$ by assuming that it is the order $k \in candidate[t]$ that takes the freed place in t when an order is removed from t:*

$$(gainCost[t])^k = (t - optPeriod[k]) \cdot h_k + gainCost[optPeriod[k]]$$

The objective is to find the order $s_t \in candidate[t]$ with the following property: $(gainCost[t])^{s_t} \geq (gainCost[t])^k, \forall k \in candidate[t]$ and then $gainCost[t] = (gainCost[t])^{s_t}$.

For each full period t, let $toSelect[t]$ denote the set of orders in $candidate[t]$ that have the highest stocking cost: $toSelect[t] = \arg\max_{X_k \in candidate[t]} h_k$.

Proposition 22. *For a full period t with $candidate[t] \neq \emptyset$: $s_t \in toSelect[t]$.*

Proof. Consider a full period t such that $candidate[t] \neq \emptyset$ (and then $toSelect[t] \neq \emptyset$). If $s_t \notin toSelect[t]$, then $\exists k \in toSelect[t]$ such that $X_k^{\max} \geq t \wedge h_k > h_{s_t}$. This is not possible in an optimal solution (see condition (ii) of Proposition 16). □

Now we know that $s_t \in toSelect[t]$, but which one exactly must we select? The next proposition states that whatever the order s in $toSelect[t]$ that is chosen to take the freed place, we could compute $gainCost[t]$ from s. That means that all orders in $toSelect[t]$ have the same potential *gainCost*.

Proposition 23. $\forall s \in toSelect[t], (gainCost[t])^s = gainCost[t]$.

Proof. If $|toSelect[t]| = 1$, then the proposition is true. Now we assume that $|toSelect[t]| > 1$. Consider two orders k_1 and k_2 in $toSelect[t]$. From Proposition 16, only null capacity periods can appear between $optPeriod[k_1]$ and $optPeriod[k_2]$ because $k_1, k_2 \in \arg\max_{k \in candidate[t]} h_k$ (and $X_k^{\max} \geq t$). Since all orders $k \in toSelect[t]$ have the same stocking cost and $X_k^{\max} \geq t$, any pair of orders k_1 and k_2 in $toSelect[t]$ can swap their respective *optPeriod* without affecting the feasibility of the solution and the optimal cost. Thus the proposition is true. □

We can summarize the computation of $gainCost$ for each period in a full set.

Corollary 24. *Consider a full set* $\{M, \ldots, m\}$ *with* $m = minfull[t]$ *and* $M = maxfull[t]$, $\forall t \in \{M, \ldots, m\}$: $gainCost[m] = 0$ *and for all full period* $t \in \{M, \ldots, m\} \setminus \{m\}$ *from* m *to* M:
$gainCost[t] = (t - optPeriod[s]) \cdot h_s + gainCost[optPeriod[s]]$ *with* $s \in toSelect[t]$.

By assuming that $gainCost[t], \forall t$ are known, the next proposition gives a lower bound on $(H^{opt}_{X_i \leftarrow v})^r$.

Proposition 25.

$$(H^{opt}_{X_i \leftarrow v})^r \geq (H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]]$$

Proof. The cost $gainCost[optPeriod[i]]$ is the maximum decrease in cost when an order is removed from $optPeriod[i]$. We know that the cost $(optPeriod[i] - v) \cdot h_i$ is a lower bound on the increased cost when the order i is forced to take the value v because the capacity restriction can be violated for the period v. Thus $(H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]]$ is a lower bound on $(H^{opt}_{X_i \leftarrow v})^r$. □

From this lower bound on $(H^{opt}_{X_i \leftarrow v})^r$, we have the following filtering rule for variables $X_i, \forall i \in [1, \ldots, n]$.

Corollary 26. $\forall i \in [1, \ldots, n]$,

$$X_i^{\min} \geq optPeriod[i] - \lfloor \frac{H^{\max} - (H^{opt})^r + gainCost[optPeriod[i]]}{h_i} \rfloor$$

Proof. We know that v can be removed from the domain of X_i if $(H^{opt}_{X_i \leftarrow v})^r > H^{\max}$ and $(H^{opt}_{X_i \leftarrow v})^r \geq (H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]]$. The highest integer value v^* that respects the condition $(H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]] \leq H^{\max}$ is $v^* = optPeriod[i] - \lfloor \frac{H^{\max} - (H^{opt})^r + gainCost[optPeriod[i]]}{h_i} \rfloor$. □

Algorithm 6.4.1 computes $gainCost[t]$ for all full periods in $[1, \ldots, T]$ in chronogical order and filters the n decision variables. It uses the stack $orderToSelect$ that, after processing, contains an order in $toSelect[t]$ on top. At each step, the algorithm treats each full set (loop $5 - 15$) from their respective minfull periods thanks to $fullSetsStack$ computed in Algorithm 6.3.1. For a given full set, the algorithm pops

each full period (in chronological order) and computes its $gainCost[t]$ until the current full set is empty; in that case, it takes the next full set in $fullSetsStack$. Now let us focus on how $gainCost[t]$ is computed for each full period t. For a given full period t, the algorithm puts all orders in $left[t]$ into the stack $orderToSelect$ (lines $14-15$) in chronological order. For a given period t, for each pair of orders k_1 (with $X_{k_1}^{\max} \geq t$) and k_2 (with $X_{k_2}^{\max} \geq t$) in $orderToSelect$: if k_1 is above k_2, then $h_{k_1} \geq h_{k_2}$. The algorithm can safely remove orders k with $X_k^{\max} < t$ from the top of the stack (lines $7-8$) since these orders $k \notin candidate[t'], \forall t' \geq t$. After this operation, if the stack $orderToSelect$ is not empty, the order on top is an order in $toSelect[t]$ (Invariant (b) - see the proof of Proposition 27) and can be used to compute $gainCost[t]$ based on Corollary 24 (lines $12-13$). Note that for a period t if the stack is empty, then $toSelect[t] = \emptyset$ (ie $candidate[t] = \emptyset$) and $gainCost[t] = 0$.
Algorithm 6.4.1 filters each variable X_i based on the lower bound from Corollary 26 (lines $16-18$).

Proposition 27. Algorithm 6.4.1 *computes* $gainCost[t]$ *for all* $O(n)$ *full periods in linear time* $O(n)$.

Proof. Invariants:

(a) After line 6, $\forall t' \in [minfull[t], \ldots, t[$ with $c_{t'} > 0$: $gainCost[t']$ is defined.

For a given full set fs, the algorithm computes the different $gainCost$ of periods inside fs in increasing value of t from its $minfull$. Thus Invariant (a) holds.

(c) After line 15, $\forall k_1, k_2 \in \{k \mid X_k \in orderToSelect \wedge X_k^{\max} \geq t\}$: if X_{k_1} is above X_{k_2} in $orderToSelect$, then $h_{k_1} \geq h_{k_2}$.

From Proposition 16, we know that $\forall k_1, k_2$ such that $optPeriod[k_1] < optPeriod[k_2]$ we have $h_{k_1} \leq h_{k_2}$ or $((h_{k_1} > h_{k_2}) \wedge (X_{k_1}^{\max} < optPeriod[k_2]))$. The algorithm pushes orders into $orderToSelect$ from $minfull[t]$ to t. If we are in period $t' = optPeriod[k_2]$, then all orders k_1 pushed before are such that $(h_{k_1} \leq h_{k_2})$ or $((h_{k_1} > h_{k_2}) \wedge (X_{k_1}^{\max} < t'))$. Thus Invariant (c) holds.

(b) After line 8, $orderToSelect.first \in toSelect[t]$.

For a period t, the loop $14-15$ ensures that all orders i in $left[t]$ are pushed once in the stack. The loop $7-8$ removes orders that are on the top of stack such that $\{k \mid X_k^{\max} < t\}$. That operation ensures that the order s on the top of the stack can jump to the period t (i.e. $X_s^{\max} \geq t$). Since this order is on the top and Invariant (c) holds for the previous period processed, it has the highest stocking cost wrt

$\{k \in orderToSelect \wedge X_k^{\max} \geq t\}$ and then $s \in toSelect[t]$. Thus Invariant (b) holds.
Based on Invariant (a) and Invariant (b), the algorithm computes $gainCost[t]$ for each full period from Corollary 24.

Complexity: there are at most n full periods and then the main loop of the algorithm (lines $3-15$) is executed $O(n)$ times. Inside this loop, the loop $14-15$ that adds orders of the current period to *orderToSelect* is in $O(c)$ with $c = \max\{c_t \mid t \in fullSetsStack\}$. On the other hand, the orders removed from the stack *orderToSelect* by the loop at lines $7-8$ will never come back into the stack and then the complexity associated is globally in $O(n)$. Hence the global complexity of the computation $gainCost[t]$ for all full periods is $O(n)$. □

Example 12 shows an execution of Algorithm 6.4.1.

Example 12. *Let us run* Algorithm 6.4.1 *on the instance of Example 10. There are two full sets:* $fullSetsStack = \{\{8,7\},\{5,4,2,1\}\}$

1. $fullSet = \{5,4,2,1\}$, $orderToSelect \leftarrow \{\}$.
   - $t = 1$, $gainCost[1] = 0$ *and* $orderToSelect \leftarrow \{4\}$;
   - $t = 2$, $s = 4$, $gainCost[2] = gainCost[1] + (2-1) \cdot h_4 = 2$ *and* $orderToSelect \leftarrow \{4,1\}$;
   - $t = 4$, $s = 1$, $gainCost[4] = gainCost[2] + (4-2) \cdot h_1 = 8$ *and* $orderToSelect \leftarrow \{4,1,3\}$;
   - $t = 5$, *after line* 8 $orderToSelect \leftarrow \{4\}$, $s = 4$, $gainCost[5] = gainCost[1] + (5-1) \cdot h_4 = 8$ *and* $orderToSelect \leftarrow \{4,2\}$.
2. $fullSet = \{8,7\}$, $orderToSelect \leftarrow \{\}$.
   - $t = 7$, $gainCost[7] = 0$ *and* $orderToSelect \leftarrow \{5\}$;
   - $t = 8$, $s = 5$, $gainCost[8] = gainCost[7] + (8-7) \cdot h_5 = 2$ *and* $orderToSelect \leftarrow \{5,6\}$.

Now the filtering is achieved for each order. Consider the order 2: $v = optPeriod[2] - \lfloor \frac{H^{\max} + gainCost[optPeriod[2]] - (H^{opt})^r}{h_i} \rfloor = 5 - \lfloor \frac{34+8-16}{10} \rfloor = 3$ *and* $X_2^{\min} = 3$. *Since* $c_3 = 0$, $X_2^{\min} = 4$ *thanks to the* `gcc` *constraint.*

Algorithm 6.4.1: `IDStockingCost`: Filtering of n date variables in $O(n)$

Input: $optPeriod$, $optOrders$ and $fullSetsStack$ (computed in Algorithm 6.3.1)

$gainCost \leftarrow map()$ `// gainCost[t]=cost won if an order is removed in` t
$orderToSelect \leftarrow stack()$ `// items that could jump on current period`
while $fullSetsStack.notEmpty$ **do**
  $fullSet \leftarrow fullSetsStack.pop$
  while $fullSet.isNotEmpty$ **do**
    $t \leftarrow fullSet.pop$
    `//` **Invariant (a):** $\forall t' \in [minfull[t], \ldots, t[$ **with** $c_{t'} > 0$**:** $gainCost[t']$ **is defined**
    while $orderToSelect.isNotEmpty \wedge (X^{\max}_{orderToSelect.top} < t)$ **do**
      $orderToSelect.pop$
    end
    `//` **Invariant (b):** $orderToSelect.first \in toSelect[t]$
    if $orderToSelect.isEmpty$ **then**
      $gainCost[t] \leftarrow 0$
    end
    else
      $s \leftarrow orderToSelect.first$
      $gainCost[t] \leftarrow gainCost[optPeriod[s]] + (t - optPeriod[s]) \cdot h_s$
    end
    while $optOrders[t].isNotEmpty$ **do**
      $orderToSelect.push(optOrders[t].pop)$
    end
    `//` **Invariant (c):** $\forall k_1, k_2 \in \{k \in orderToSelect \wedge X^{\max}_k \geq t\}$**:** **if** k_1 **is above** k_2 **in** $orderToSelect$**, then** $h_{k_1} \geq h_{k_2}$
  end
end
for each order i **do**
  $v \leftarrow optPeriod[i] - \lfloor \frac{H^{\max} + gainCost[optPeriod[i]] - (H^{opt})^r}{h_i} \rfloor$
  $X^{\min}_i \leftarrow max\{X^{\min}, v\}$
end

Strengthening the filtering

During the search some orders are fixed by branching decisions or during the filtering in the fix-point calculation. The lower bound $(H^{opt})^r$ can be strengthened by preventing those fixed orders to move. This strengthening requires very little modification to our algorithm. First, the fixed orders are filtered out such that they are not considered by Algorithm 6.3.1 and Algorithm 6.4.1. A reversible [1] integer maintains the contributions of orders fixed to the objective. This value is denoted H^{fixed}. Also when an order is fixed in a period t, the corresponding capacity c_t - also a reversible integer - is decreased by one. The strengthened bound is then $(H^{opt})^r + H^{fixed}$. This bound is also used for the filtering of the X_i's.

6.5 EXPERIMENTAL RESULTS

The experiments were conducted on instances of the PSP described in Section 3.3. We use the same methodology and experimental settings as for the `StockingCost` constraint (see Section 5.6) but here the stocking costs per item are different. All our source-code for the models, the global constraint, and the instances are available at [HSb].

A CP model

The model used is a variant of that described in the previous chapter. Each order is uniquely identified. The decision variables are $date[p] \in [1,\ldots,T]$, $\forall p \in [1,\ldots,n]$. For the order p, $date[p]$ is the period for production of the order p. Note that $date[p]$ must repect its $dueDate[p]$: $date[p] \leq dueDate[p]$. Let $objStorage$ denote the total stocking cost: $objStorage = \sum_p (dueDate(p) - date(p)) \cdot h_p$ with $h_p = h_i$ is the stocking cost of the order p for an item i. The changeover costs are computed using a successor based model and the *circuit* [Pes+98] constraint. The changeover costs are aggregated into the variable $objChangeOver$. The total stocking cost variable $objChangeOver$ is computed using the constraint introduced in this paper:

$$\texttt{IDStockingCost}(date, dueDate, [h_1,\ldots,h_n], objStorage, [c_1,\ldots,c_T])$$

1 A reversible variable is a variable that can restore its domain when backtracks occur during the search.

with $c_t = 1, \forall t \in [1, \ldots, T]$. The overall objective to be minimized is $objStorage + objChangeover$.

Comparison on small instances

As first experiment, we consider 100 small random instances limited to 20 periods, 20 orders and 5 items with $q^{i,j} \in [10, \ldots, 50]$ for $i \neq j$, $q^{i,i} = 0$ and $h_i \in [10, \ldots, 50], \forall i$. We measure the gains over the *Basic* model using as filtering for `IDStockingCost`:

1. *IDS*: our filtering algorithms for `IDStockingCost`;
2. *MinAss*: the `minimumAssignment` constraint with the LP reduced costs based filtering + the `allDifferent` constraint with bound consistency filtering. As already mentioned, the `minimumAssignment` constraint is much more efficient when it is together with the `allDifferent` constraint (bound consistency filtering);
3. $MinAss_2$: the `minimumAssignment` constraint with exact reduced costs based filtering [DCP16] + `allDifferent` constraint with bound consistency filtering.

Table 6.1 shows the arithmetic average of the number of nodes and the time required for *Basic*, *MinAss*, $MinAss_2$ and *IDS* respectively. Table 6.1 also shows the geometric average gain factor (wrt *Basic*) for each propagator. Unsurprisingly $MinAss_2$ prunes the search trees the most but this improved filtering does not compensate for the time needed for the exact reduced costs. It is still at least 4 times (on average) slower than *MinAss* and *IDS*.

These results suggest that on small instances *MinAss* offers the best trade-off wrt filtering/time. Notice that *IDS* is competitive with *MinAss* in terms of computation time.

Comparison on large instances

The previous results showed that *MinAss* and *IDS* are competitive filtering for the `IDStockingCost` constraint on small instances. We now scale up the instance sizes to 500 periods (with the number of orders $n \in [490, \ldots, 500]$) and 10 different items. Again we consider 100 random instances with the following characteristics: $T = 500$,

	IDS		*MinAss*		$MinAss_2$		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	30.1 10^4	15.7	26.2 10^4	**13.2**	**24.9** 10^4	51.7	130 10^4	51.4
Av. gain factor	5.0	4.0	6.2	**4.9**	**6.7**	1.0	1.0	1.0

Table 6.1: Average results on 100 instances with $T = 20$: *IDS*, *MinAss*, $MinAss_2$, and *Basic*

$n \in [490, \ldots, 500]$, $nbItems = 10$, $q^{i,j} \in [10, \ldots, 50]$ for $i \neq j$, $q^{i,i} = 0$ and $h_i \in [50, \ldots, 100], \forall i$. Table 6.2 gives the average values for the number of nodes and computation time when replaying the search trees, plus the geometric average gain over the *Basic* approach. Clearly, the reported values suggest that *IDS* performs best, in particular wrt the computation time.

	IDS		*MinAss*		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	**6.68** 10^4	**5.8**	8.30 10^4	25.8	73.5 10^4	52.8
Av. gain factor	**12.7**	**10.0**	11.1	2.3	1.0	1.0

Table 6.2: Average results on 100 instances with $T = 500$: *IDS*, *MinAss*, and *Basic*

Figure 6.7 and Figure 6.8 show the performance profiles (for *IDS*, *MinAss* and *Basic*) wrt the number of nodes visited and the time needed to complete the search respectively. We can see that:

- wrt nodes: for $\approx 80\%$ of instances, *IDS* provides the best filtering;
- wrt time: *IDS* has the best time for all instances. Note that *IDS* is at least 4 times as fast as *MinAss* for $\approx 60\%$ of instances.

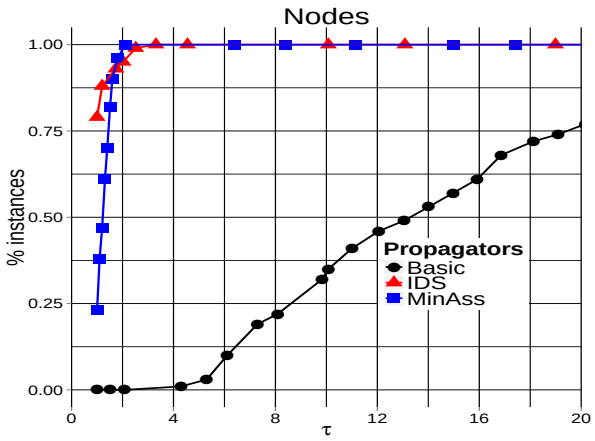


Figure 6.7: Performance profiles - Nodes: *IDS*, *MinAss*, and *Basic*

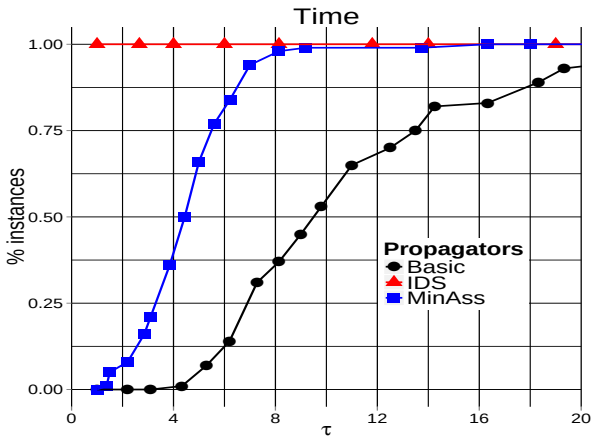


Figure 6.8: Performance profiles - Time: *IDS*, *MinAss*, and *Basic*

`IDStockingCost` *vs* `StockingCost`

The `IDStockingCost` constraint generalizes the `StockingCost` constraint. We now compare the performance of *IDS* with `StockingCost` on instances with equal stocking costs. We reuse the previous 100 instances generated with 500 demands and time periods, but using the same stocking cost for all the items.

As can be observed in Table 6.3, both *StockingCost* and *IDS* outperform *MinAss*. *MinAss* is at least 8 (on average) times slower than *IDS* and *StockingCost*. Note that, as established in Chapter 5, *StockingCost* offers a bound consistent filtering and is thus as expected the best propagator in this setting. However, the average values reported in Table 6.3 show that *IDS* is competitive wrt *StockingCost*. That is confirmed by the performance profiles presented in Figure 6.9 and Figure 6.10:

- wrt nodes: for $\approx 80\%$ of instances, *StockingCost* is only at most 1.1 times better than *IDS*;
- wrt time: for $\approx 80\%$ of instances, *IDS* has the best time even if it is ≥ 2 times worse than *StockingCost* for $\approx 5\%$ of instances.

	StockingCost		*IDS*		*MinAss*		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	**8.81** 10^4	7.4	9.36 10^4	**7.2**	15.6 10^4	62.3	76.9 10^4	52.3
Av. gain factor	**11.4**	**8.8**	10.0	8.3	6.1	1.0	1.0	1.0

Table 6.3: Average results on 100 instances with $T = 500$: *StockingCost*, *IDS*, *MinAss*, and *Basic*

About the specialized OR approach for the PSP

Pochet and Wolsey [PW05] report results solving instances with up to $T = 100$ using the specialized valid inequalities for DLS-CC-SC (see Section 3.3). Our CP model is able to solve and prove optimality of PSP instances with up to $T = 20$ (see Appendix A). Note that the part of the model concerning the changeover costs uses a `minimumAssignment` constraint that slows down the CP model for this problem. On the other

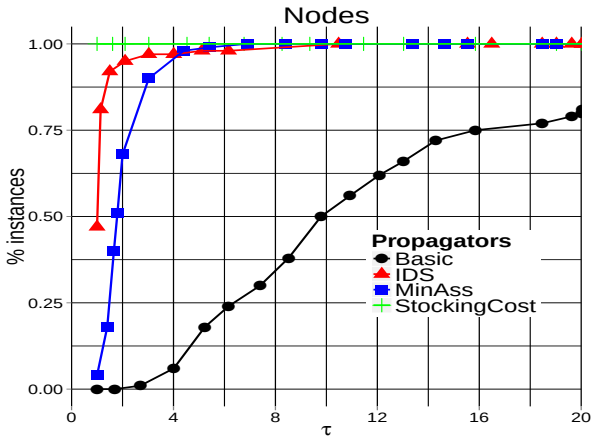


Figure 6.9: Performance profiles - Nodes: *StockingCost*, *IDS*, *MinAss*, and *Basic*

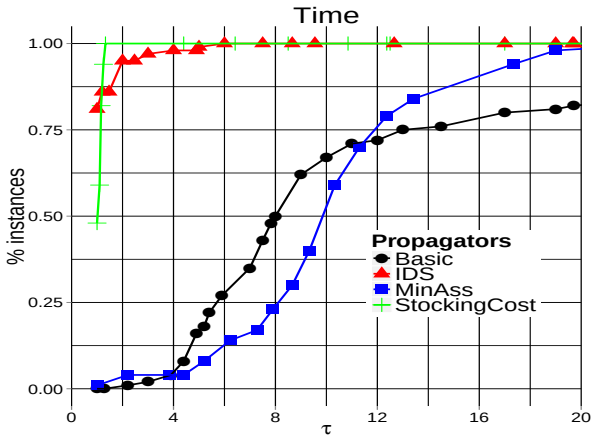


Figure 6.10: Performance profiles - Time: *StockingCost*, *IDS*, *MinAss*, and *Basic*

hand, the search aspects should be developed to give a chance to CP to be competitive with the state-of-the-art MIP approaches on the PSP.

6.6 CONCLUSION

We have introduced the `IDStockingCost` constraint, which is a generalization of the `StockingCost` constraint allowing for order dependent stocking costs and production capacity that may vary over time. We have proposed a scalable filtering algorithm for this constraint in $O(n \log n)$. Our experimentation on a variant of the CLSP shows that the filtering algorithm proposed: 1) scales well wrt the other state-of-the-art CP formulation based on minimum assignment and 2) can be used, on large instances, instead of the `StockingCost` constraint even when the stocking cost is the same for all items.

Part III

FILTERING ALGORITHMS FOR THE CONSTRAINED ARBORESCENCE PROBLEM

7 THE WEIGHTED ARBORESCENCE CONSTRAINT

We refer to Chapter 4 for an overview of Minimum Weight Arborescence (MWA) and the Constrained Arborescence Problem (CAP). In this chapter, we define the minimum weight arborescence constraint (denoted `MinArborescence`) to solve the CAP in CP. A filtering based on the LP reduced costs requires $O(|V|^2)$ in which $|V|$ is the number of vertices. We propose a procedure to strengthen the quality of the LP reduced costs in some cases, also running in $O(|V|^2)$. Computational results on a variant of the CAP show that the additional filtering provided by the constraint reduces the size of the search tree substantially.

7.1 INTRODUCTION

We address the CAP - that requires one to find an arborescence that satisfies other side constraints and is of minimum cost - and show how one can handle them in CP. Let us recall the definition of an MWA. Consider a weighted directed graph $G = (V, E)$ in which V is the vertex set and $E \subseteq \{(i,j) \mid i,j \in V\}$ is the edge set. A weight $w(i,j)$ is associated to each edge $(i,j) \in E$. Given a vertex r, the aim is to associate to each vertex $v \in V \setminus \{r\}$ exactly one vertex $p(v) \in V$ (with $(p(v),v) \in E$) such that, considering the sub-graph $A = (V,F)$ with $F = \{(p(v),v) \mid v \in V \setminus \{r\}\}$, there is no cycle and the total cost $(\sum_{v \in V \setminus \{r\}} w(p(v),v))$ is minimized. We focus on an optimization oriented constraint for MWA mentioned in [Foc+99][1].

1 The authors consider MWA as a relaxation of the Traveling Salesman Problem (TSP) and use LP reduced costs to filter inconsistent values.

The `MinArborescence` constraint holds if there is a set of edges that form an arborescence rooted at a given vertex with total cost less than the objective variable value K. This constraint can be handled as a classical optimization constraint. First, an optimal solution of the MWA problem associated is computed and its cost is used to check the consistency of the constraint and filter, if possible, the objective variable K. Since the LP reduced cost gives an optimistic evaluation of the cost increase by forcing an edge in the optimal solution, they can be used to filter the decision variables. Considering an MWA $A(G)^\star$, we show that the LP reduced cost of the edge (i, j) can be improved if there is a directed path $\mathcal{P}$ from j to i in $A(G)^\star$ such that each vertex in $\mathcal{P}$ is involved in at most one cycle[2] when computing $A(G)^\star$.

This chapter is organized as follows: Section 7.2 formally defines the optimization constraint for the MWA called `MinArborescence` and show how one can improve the filtering of its basic decomposition by avoiding the creation of cycles when some variables are fixed; Section 7.3 describes how the LP reduced costs can be improved in some cases after sensitivity analysis of an MWA; Section 7.4 shows some experimental results and Section 7.5 concludes.

In this chapter, we run our examples on the graph G_1 (Figure 7.1) from Chapter 4 and its MWA $A(G_1)^\star$ (Figure 7.2).

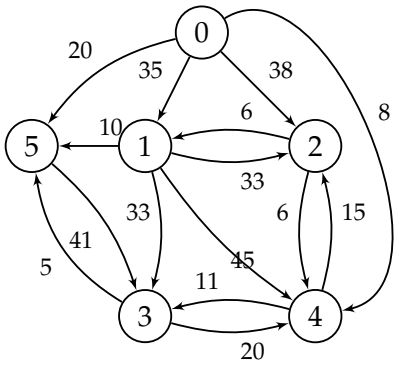


Figure 7.1: Initial graph G_1

2 See the computation of an MWA in Section 4.1.2.

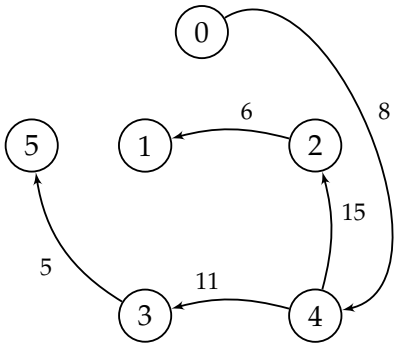


Figure 7.2: $A(G_1)^\star$

7.2 THE MINARBORESCENCE CONSTRAINT

We use the predecessor variable representation of a graph to define the `MinArborescence` constraint. The arborescence is modeled with one variable X_i for each vertex i of $G = (V, E)$ representing its predecessor. The initial domain of a variable X_i is thus the neighbors of i in G: $j \in D(X_i) \equiv (j, i) \in E$.

The constraint `MinArborescence(X, w, r, K)` holds if the set of edges $\{(X_i, i) \mid i \neq r\}$ is a valid arborescence[3] rooted at r with $\sum_{i \neq r} w(X_i, i) \leq K$.

The consistency of the constraint is achieved by computing an exact MWA $A(G)^\star$ rooted at r and verifying that $w(A(G)^\star) \leq K$. The value $w(A(G)^\star)$ is an exact lower bound for the variable K: $K \geq w(A(G)^\star)$. The filtering of the edges can be achieved based on the reduced costs. For a given edge $(i, j) \notin A(G)^\star$, if $w(A(G)^\star) + rc(i, j) > K$, then $X_i \leftarrow j$ is inconsistent. In Section 7.3, we propose a procedure to strengthen the quality of the LP reduced costs in $O(|V|^2)$ in some cases.

Observe that, for the `MinArborescence` constraint, it is not necessary to study the mandatory[4] edges as for the weighted spanning tree constraint. Actually, since exactly one edge entering each vertex in $V \setminus \{r\}$, if an edge (i, j) is mandatory then any of other edges (k, j) can not be part of a solution (i.e $w(A(G)^\star) + rc(k, j) > K$) and will be removed.

A GAC filtering for the `MinArborescence` constraint would require exact reduced costs, that to the best of our knowledge can only be

3 A set $\{(X_i, i) \mid i \neq r\}$ is an arborescence rooted at r if there is exactly one edge entering each $i \in V \setminus \{r\}$ and there is no cycle.

4 A mandatory edge is an edge that must be part in a solution [Ben+12].

obtained by recomputing the MWA from scratch in $O(|E||V|^2)$ which is in $O(|V|^4)$. The basic decomposition of the `MinArborescence` constraint does not scale well due to the exponential number of constraints in equation (4.3). We describe below a light constraint (called the `Arborescence` constraint) to have a scalable baseline model for experiments.

7.2.1 *Decomposing the constraint*

The constraint `Arborescence(X, r)` holds if the set of edges $\{(X_i, i) : \forall i \in V'\}$[5] is a valid arborescence rooted at r. We describe below an incremental forward checking like incremental filtering procedure for this constraint in Algorithm 7.2.1. The algorithm is inspired by the filtering described in [Pes+98] for enforcing a Hamiltonian circuit. During the search, the bound (fixed) variables [6] form a forest of arborescences. Eventually when all the variables are bound, a unique arborescence rooted at r is obtained. The filtering maintains for each node:

1. a reversible integer value $localRoot[i]$ defined as $localRoot[i] = i$ if X_i is not bound, otherwise it is recursively defined as $localRoot[X_i]$, and
2. a reversible set $leafNodes[i]$ that contains the leaf nodes of i in the forest formed by the bound variables.

The filtering prevents cycles by removing from any successor variable X_v all the values corresponding to its leaf nodes (i.e. the ones in the set $leafNodes[v]$). Algorithm 7.2.1 registers the filtering procedure to the bind events such that `bind(i)` is called whenever the variable X_i is bound. This `bind` procedure then finds the root lr of the (sub) arborescence at line 15. As illustrated in Figure 7.3, notice that j is not necessarily the root as vertex j might well have been the leaf node of another arborescence that is now being connected by the binding of X_i to value j. In this figure, a zigzag link means that there is a directed path between the two vertices involved while a dashed edge represent an edge removed by the filtering. The root lr then inherits all the leaf nodes of i. None of these leaf nodes is allowed to become a successor of lr, otherwise a cycle would be created.

5 $V' = V \setminus \{r\}$.
6 bound variable: variable with a single value in its domain.

Algorithm 7.2.1: Class Arborescence

```
1  X: array of |V| variables
2  localRoot: array of |V| reversible int
3  leafNodes: array of |V| reversible set of int
4  Method init()
5      X_r ← r                                          // self loop on r
6      foreach each vertex v ∈ V' do
7          leafNodes[v].insert(v)
8          X_v.removeValue(v)                            // no self loop
9          if X_v.isBound then
10             bind(v)
11         end
12         else
13             X_v.registerOnBindChanges()
14         end
15     end
16 Method bind(i: int)
17     j ← X_i                                  // edge (j, i) ∈ arborescence
18     lr ← localRoot[j]                                 // local root of j
19     foreach each v ∈ leafNodes[i] do
20         localRoot[v] ← lr
21         leafNodes[lr].insert(v)
22         X_lr.removeValue(v)
23     end
```

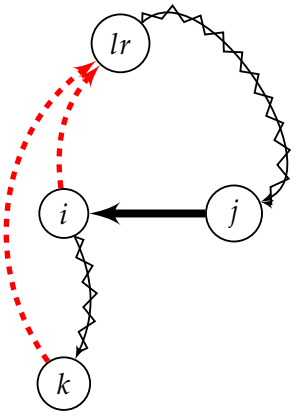


Figure 7.3: `Arborescence` constraint filtering

Example 13. *Consider the following partial assignment from the graph* G_1 (Figure 7.1)*:* $X_2 \leftarrow 1$, $X_5 \leftarrow 1$ *and* $X_3 \leftarrow 4$*. Then we have* $localRoot[2] = localRoot[5] = 1$, $leafNodes[1] = \{1,2,5\}$, $localRoot[3] = 4$ *and* $leafNodes[4] = \{3,4\}$*. These assignments are shown in* Figure 7.4*. Initially, the domain of the variable* X_4 *is* $X_4 \in \{0,1,2,3\}$*. Due to the assignment* $X_3 \leftarrow 4$*, the value* 3 *must be removed from the domain of* X_4 *to prevent the creation of the cycle* $\{3,4\}$*. Thus* $X_4 \in \{0,1,2\}$*. Assume that the value* 3 *is assigned to* X_1 *that means that the vertex* 3 *is the predecessor of the vertex* 1 *in the arborescence* ($X_1 \leftarrow 3$)*. Then all vertices in* $leafNodes[1]$ *have a new* $localRoot$ *that is the vertex* 4*:* $localRoot[1] = localRoot[2] = localRoot[5] = 4$*. It then follows that the vertices* 1 *and* 2 *must be removed from the domain of* X_4 *to prevent the creation of a cycle as illustrated in* Figure 7.4*. The dashed edges are the removed edges and the bold dashed edges are the edge removed by the assignment* $X_1 \leftarrow 3$*. Finally,* $X_4 \in \{0\}$.

The `MinArborescence(X, w, r, K)` can be decomposed as the `Arborescence(X, r)` constraint plus a sum over element constraints: $\sum_{i \neq r} w(X_i, i) \leq K$.

7.3 IMPROVED REDUCED COSTS

In the following, we write $A(G)^{\star}_{i \to j}$ for the MWA of the graph G when the edge (i,j) is forced to be in it. We know that the LP reduced cost $rc(i,j)$ gives a lower bound on the associated cost in-

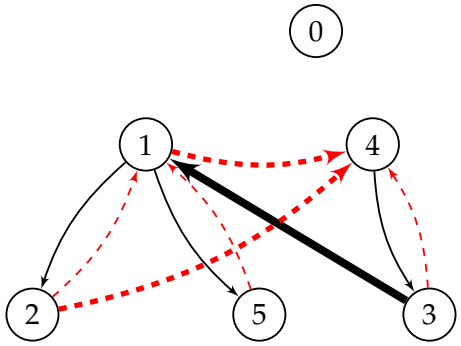


Figure 7.4: `Arborescence` constraint filtering - Example

crease: $w(A(G)^{\star}_{i\to j}) \geq w(A(G)^{\star}) + rc(i,j)$. However, this lower bound of $w(A(G)^{\star}_{i\to j})$ can be improved in some cases. In this section, we show that if there is a directed path $\mathcal{P}$ from j to i in $A(G)^{\star}$ such that each vertex in $\mathcal{P}$ is involved in at most one cycle when computing $A(G)^{\star}$ then $rc(i,j)$ can be improved. To more define the edges for which our improvement procedure is applicable, we denote by $parent[S], \forall S \subseteq V'$ the first directed cycle that includes the subset S found during the execution of the Edmonds' algorithm. We assume that $parent[S] = \emptyset$ if there is no such cycle and $parent[\emptyset] = \emptyset$. In the graph G_1, $parent[1] = parent[3] = parent[5] = \emptyset$ and $parent[2] = parent[4] = \{2,4\}$. Note that for the graph G_1, $parent[parent[k]] = \emptyset, \forall k \in V'$. Formally,

Definition 17. *Let $parent[S]$ be the smallest cycle strictly containing the subset $S \subseteq V'$: $parent[S]$ is the smallest subset $> |S|$ such that $\sum_{(i,j)\in\delta^{in}_{parent[S]}} x^{\star}_{i,j} = 1$ and $S \subset parent[S]$.*

The rest of this section provides some properties/propositions to improve $rc(i,j)$ if there exist a path $\mathcal{P}$ from j to i in $A(G)^{\star}$ such that $parent[parent[k]] = \emptyset, \forall k \in \mathcal{P}$. We first consider the simplest case that is $parent[k] = \emptyset, \forall k \in \mathcal{P}$.

The next three properties give some information to improve the LP reduced costs when all vertices involved do not have a parent.

Property 28. *Assume that there is a path $\mathcal{P} = (j, \ldots, i)$ from the vertex j to vertex i in $A(G)^{\star}$ such that $\forall k \in \mathcal{P} : parent[k] = \emptyset$. If the edge (i,j) is forced to be in the MWA then the cycle $c = \{k \mid k \in \mathcal{P}\}$ will be created during the execution of Edmonds' algorithm.*

Proof. $parent[k] = \emptyset$ means that $pred[k]$ is such that $w(pred[k],k) = \min\{w(v,k) \mid (v,k) \in \delta^{in}_{k}\}$ and then $pred[k]$ will be first selected by

Edmonds' algorithm $\forall k \in \mathcal{P} \setminus \{j\}$. On the other hand, if the edge (i,j) is forced into the MWA, it implies that all other edges entering j are removed. Consequently, the cycle $c = \{k \mid k \in \mathcal{P}\}$ will be created. □

Let us use the following notation to evaluate the improved reduced costs. Let $min1[k] = \arg\min_{(v,k)\in\delta_k^{in}} w(v,k)$ be the minimum cost edge entering the vertex k. If there is more than one edge with the smallest weight, we choose one of them arbitrarily. Also, let $min2[k] = \arg\min_{(v,k)\in\delta_k^{in}\wedge(v,k)\neq min1} w(v,k)$ be the second minimum cost edge entering k.

Definition 18. *For each vertex $k \in V$, let $bestTwoDiff[k]$ be the difference between the best two minimum costs of edges entering the vertex k: $\forall k \in V, bestTwoDiff[k] = w(min2[k]) - w(min1[k])$.*

For instance, in the graph G_1, $min1[5] = (3,5)$, $min2[5] = (1,5)$ and $bestTwoDiff[5] = 10 - 5 = 5$.

Property 29. *Consider the cycle $c = (i,\ldots,j)$ obtained by forcing the edge (i,j) such that $parent[k] = \emptyset, \forall k \in c$ (see* Property *28). The minimum cost increase if the cycle is broken/connected by the vertex $k' \in c$ is $bestTwoDiff[k']$.*

Proof. For a given $k' \in c$, $parent[k'] = \emptyset$ implies that the edge $(pred[k'],k') = min1[k']$. Then the cheapest way to break the cycle by k' is to use the edge with the second minimum cost $min2[k']$. Hence the minimum cost increase if the cycle is broken by the vertex k' is $w(min2[k']) - w(min1[k']) = bestTwoDiff[k']$. □

When $parent[j] = \emptyset$, the LP reduced costs $rc(i,j)$ are simple and can be easily interpreted. To express $rc(i,j)$ in such cases, let $pred[v], \forall v \in V'$, be the vertex in V such that the edge $(pred[v],v) \in A(G)^\star$: $x^\star_{pred[v],v} = 1$. For example, in the graph G_1, $pred[1] = 2$ and $pred[5] = 3$. Next property gives the LP reduced costs $rc(i,j)$ expression when $parent[j] = \emptyset$.

Property 30. *Consider a vertex $j \in V'$ such that $parent[j] = \emptyset$. For all $i \in V \setminus \{j\}$ with $(i,j) \notin A(G)^\star$: $rc(i,j) = w(i,j) - w(pred[j],j)$.*

Proof. We know that if $parent[j] = \emptyset$ then for each $S \subseteq V'$ with $j \in S$ and $|S| \geq 2$, $u^\star_S = 0$ (because none of them is a directed cycle). Then $rc(i,j) = w(i,j) - u^\star_j$. On the other hand, since $parent[j] = \emptyset$, then the edge $min1[j]$ is the one used to connect j in $A(G)^\star$. Hence $u^\star_j = w(min1[j]) = w(pred[j],j)$ and $rc(i,j) = w(i,j) - w(pred[j],j)$. □

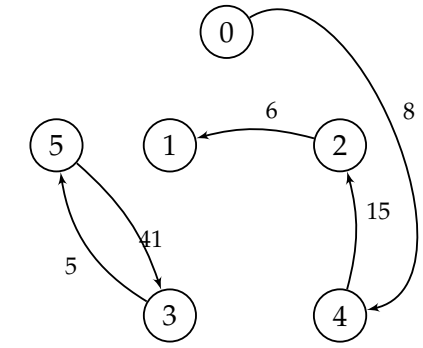
By considering an MWA $A(G)^\star$, the interpretation of $rc(i,j)$ when $parent[j] = \emptyset$ is that the edge (i,j) is forced into $A(G)^\star$ and the edge $(pred[j], j)$ that is used to connect j is removed. Intuitively, if this process induces a new cycle, this latter has to be (re)connected to the rest of the arborescence from a vertex in the cycle different from j. Proposition 31 established below gives a first improved reduced cost expression when a new cycle c is created by forcing (i,j) into the arborescence and $\forall k \in c : parent[k] = \emptyset$. Note that such new cycle will be created only if there is already a path in $A(G)^\star$ from the vertex j to the vertex i.

Proposition 31. *Assume that there is a path $\mathcal{P} = (j, \ldots, i)$ from the vertex j to vertex i in $A(G)^\star$ such that $\forall k \in \mathcal{P} : parent[k] = \emptyset$. Then $w(A(G)^\star_{i\to j}) \geq w(A(G)^\star) + rc(i,j) + \min_{k\in\mathcal{P}\setminus\{j\}}\{bestTwoDiff[k]\}$.*

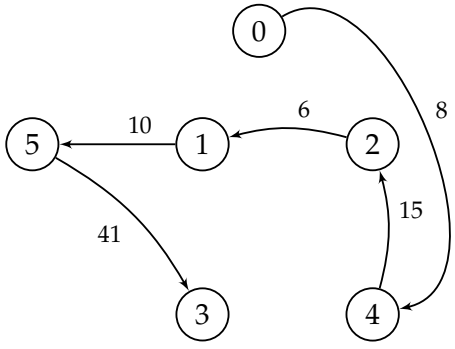
Proof. Without loss of generality, we assume that the cycle $c = \{k \mid k \in \mathcal{P}\}$ is the first one created by the Edmonds' algorithm (see Property 28). After this step, the new set of vertices is $V' = \{c\} \cup \{v \in V \mid v \notin c\}$. In $A(G)^\star$, the edges assigned to vertices in c do not influence the choice of edges for each vertex in $\{v \in V \mid v \notin c\}$ (because $parent[k] = \emptyset, \forall k \in c$). Thus $w(A(G)^\star_{i\to j}) \geq \sum_{k\in V\wedge k\notin c} w(pred[k], k) + w(c)$, in which $w(c)$ is the minimum sum of costs of edges when exactly one edge is assigned to each vertex in c without cycle. The cheapest way to connect all vertices in c such that each one has exactly one entering edge is to use all cheapest entering edges ($min1[k], \forall k \in c$). The cycle obtained must be broken in the cheapest way. To do so, the vertex used: 1) must be different from j (because (i,j) is already there) and 2) have to induce the minimal cost increase. Then, from Property 29, a lower bound of the minimum cost increase is $\min_{k\in\mathcal{P}\setminus\{j\}}\{bestTwoDiff[k]\}$. In addition, we have to add the cost of the forced edge (i,j) and remove the cost of the edge in $A(G)^\star$ that enters in j: $w(c) \geq \sum_{k\in c} w(pred[k],k) + \min_{k\in c\setminus\{j\}}\{bestTwoDiff[k]\} + w(i,j) - w(pred[j],j)$. We know that $\sum_{k\in c} w(pred[k],k) + \sum_{k\in V\wedge k\notin c} w(pred[k],k) = w(A(G)^\star)$ and $rc(i,j) = w(i,j) - w(pred[j],j)$ (see Property 30).
Thus $w(A(G)^\star_{i\to j}) \geq w(A(G)^\star) + rc(i,j) + \min_{k\in\mathcal{P}\setminus\{j\}}\{bestTwoDiff[k]\}$. □

Example 14. *Consider the graph G_1 presented in* Figure 7.1 *and its MWA (*Figure 7.2*). We want to force $(5,3)$ to be into the MWA:*

- $rc(5,3) = w(5,3) - w(4,3) = 41 - 11 = 30$. *This operation leads to the graph shown in* Figure 7.5 *(a). We can see that the new cycle created* $c = (5,3)$ *must be broken from a vertex different from* 3.
- $irc(5,3) = rc(5,3) + bestTwoDiff[5] = 30 + 5 = 35$. *The corresponding graph is shown in* Figure 7.5 *(b), that actually is the new* MWA.



(a) G_1 with $rc(5,3)$



(b) G_1 with $irc(5,3)$

Figure 7.5: G_1 with $rc(5,3)$ and $irc(5,3)$

Property 29, Property 30 and Proposition 31 can be generalized into Property 32, Property 33 and Proposition 34 below to include some vertices that have one parent.

Property 32. *Consider an ordered set of vertices* $(k_1, k_2, \ldots, k_n)$ *in* $A(G)^\star$ *such that* $c = (k_1, k_2, \ldots, k_n)$ *is a directed cycle connected (broken) by the*

vertex $k^\star$ and $parent[c] = \emptyset$. The minimum cost increase if the cycle is broken by another vertex $k' \in c \setminus \{k^\star\}$ is $\geq bestTwoDiff[k'] - u^\star_c$.

Proof. We know that $parent[c] = \emptyset$ implies that $u^\star_c = \min\{w(min2[k]) - w(min1[k]) \mid k \in c\} = w(min2[k^\star]) - w(min1[k^\star])$. Thus if the cycle is now connected by another vertex than $k^\star$, the edge $min1[k^\star]$ can be used instead of $min2[k^\star]$ and decreases the cost by $w(min2[k^\star]) - w(min1[k^\star]) = u^\star_c$. On the other hand, $\forall k_i \in c \setminus \{k^\star\}$, $(pred[k_i], k_i) = min1[k_i]$. It follows that the cheapest way to use k' to connect c is to use $min2[k']$. Hence a lower bound of the total cost induced is $min2[k'] - min1[k'] - u^\star_c = bestTwoDiff[k'] - u^\star_c$. □

The next property gives a lower bound of the LP reduced costs $rc(i,j)$ when $parent[parent[j]] = \emptyset$.

Property 33. *Consider a vertex $j \in V$ such that $parent[parent[j]] = \emptyset$. For all $i \in V \setminus \{j\}$ with $(i,j) \notin A(G)^\star$, $rc(i,j) \geq w(i,j) - w(pred[j], j) - u^\star_{parent_j}$.*

Proof. We know that if $parent[parent[k]] = \emptyset$, then among all S with $j \in S$ and $|S| \geq 2$, only $parent[k]$ can have $u^\star_S > 0$. Then $rc(i,j) = w(i,j) - u^\star_j - u^\star_{parent[j]}$. On the other hand, $u^\star_j = w(min1[j]) \leq w(pred[j], j)$. Thus $rc(i,j) \geq w(i,j) - w(pred[j], j) - u^\star_{parent_j}$. □

Now the improved reduced costs can be formulated as follows.

Proposition 34. *Assume that there is a path $\mathcal{P} = (j, \ldots, i)$ from the vertex j to vertex i in $A(G)^\star$ such that $\forall k \in \mathcal{P} : parent[parent[k]] = \emptyset$. Then $w(A(G)^\star_{i \to j}) \geq w(A(G)^\star) + rc(i,j) + \min_{k \in \mathcal{P} \setminus \{j\}} \{bestTwoDiff[k] - u^\star_{parent[k]}\}$.*

Proof. Note that if $\forall k \in \mathcal{P} : parent[k] = \emptyset$ (that implies that $u^\star_{parent[k]} = 0$), the formula is the same as the one of Proposition 31. Let Z denote the set of vertices in $\mathcal{P}$ and in all other cycles linked to $\mathcal{P}$. Formally, $Z = \{v \in V \mid v \in \mathcal{P}\} \cup \{k \in V \mid \exists v \in \mathcal{P} \wedge parent[k] = parent[v]\}$. We know that, in $A(G)^\star$, the edges assigned to vertices in Z do not influence the choice of edges for each vertex $k \in V \setminus Z$. Thus $w(A(G)^\star_{i \to j}) \geq \sum_{k \in V \setminus Z} w(pred[k], k) + w(Z)$ in which $w(Z)$ is the minimum sum of the costs of edges when we assign exactly one edge to each vertex in Z without cycle. The reasoning is close to the proof of Proposition 31. The differences here are:

1. $\exists k \in \mathcal{P} \setminus \{j\} : parent[k] \neq \emptyset$. Assume that we want to break the cycle by one vertex $v^\star$ in $\mathcal{P} \setminus \{j\}$. If the vertex $v^\star$ used is such that $parent[v^\star] = \emptyset$, then the minimum cost to pay is $\geq bestTwoDiff[v^\star]$ (here $u^\star_{parent[v^\star]} = 0$ because $parent[v^\star] = \emptyset$). If $v^\star$ is such that $parent[v^\star] \neq \emptyset \wedge parent[parent[v^\star]] = \emptyset$, then from Property 32, the cost to pay is $\geq bestTwoDiff[v^\star] - u^\star_{parent[v^\star]}$. By considering all vertices in $\mathcal{P} \setminus \{j\}$, the cost to pay is then $\geq \min_{k \in \mathcal{P} \setminus \{j\}} \{bestTwoDiff[k] - u^\star_{parent[k]}\}$.

2. the vertex j may have one parent. Let *connect* be the vertex that is used to connect the cycle $parent[j]$ in $A(G)^\star$.
   Case 1: $parent[i] \neq parent[j]$. If we force the edge (i,j), then the cycle $parent[j]$ should not be created because it is as if all edges entering j but (i,j) are removed. First, assume that $j \neq connect$. The edge in $parent[j]$ not in $A(G)^\star$ should be used and the cost won is $u^\star_{parent[j]}$ (as in the proof of Property 32). Thus a lower bound on the cost to break the cycle $parent[j]$ by j is: $w(i,j) - w(pred[j],j) - u^\star_{parent[j]}$. This lower bound is equal to $rc(i,j)$ because $w(pred[j],j) = w(min1[j])$. Now assume that $j = connect$. In this case $(pred[j],j) = min2[j]$ (because $parent[parent[j]] = \emptyset$). Using the edge (i,j) instead of $(pred[j],j)$ induces the cost $w(i,j) - w(min2[j]) = w(i,j) - w(min1[j]) - w(min2[j]) + w(min1[j) = w(i,j) - w(min1[j]) - u^\star_{parent[j]} = rc(i,j)$.
   Case 2: $parent[i] = parent[j] \neq \emptyset$. This means that the edge (i,j) is the one of the cycle that is not in the MWA $A(G)^\star$. In this case $rc(i,j) = 0$, and the new cycle created should be broken as described above (1.).

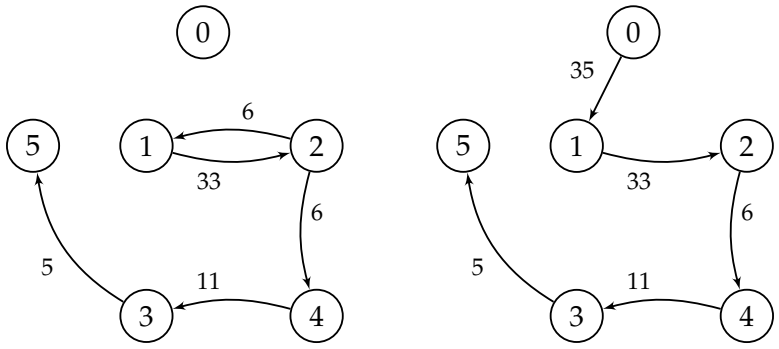
Hence a lower bound on $w(Z)$ is

$$\sum_{k \in Z} w(pred[k],k) + \min_{k \in \mathcal{P} \setminus \{j\}} \{bestTwoDiff[k] - u^\star_{parent[k]}\} + rc(i,j)$$

and $w(A(G)^\star_{i \to j}) \geq w(A(G)^\star) + \min_{k \in \mathcal{P} \setminus \{j\}} \{bestTwoDiff[k] - u^\star_{parent[k]}\} + rc(i,j)$. □

Note that $parent[parent[k]] = \emptyset$ if k is not in a cycle or k is in a cycle that is not contained in a larger cycle. The formula of Proposition 34 is available only if $\forall k \in \mathcal{P} : parent[parent[k]] = \emptyset$. Let $irc(i,j)$ denote the improved reduced cost of the edge (i,j): $irc(i,j) = rc(i,j) +$

$\max\{\min_{k\in\mathcal{P}\wedge k\neq j}\{bestTwoDiff[k] - u^{\star}_{parent[k]}\}, 0\}$ if the assumption of Proposition 34 is true and $irc(i,j) = rc(i,j)$ otherwise.



(a) G_1 with $rc(1,2)$ (b) G_1 with $irc(1,2)$

Figure 7.6: G_1 with $rc(1,2)$ and $irc(1,2)$

Example 15. *Consider the graph G_1 in* Figure 7.1 *and its* MWA *$A(G_1)^{\star}$ in* Figure 7.2. *For the contruction of $A(G_1)^{\star}$, the cycle $c_1 = \{2,4\}$ is created. We want to force into the* MWA *the edge:*

1. $(1,2)$: $rc(1,2) = w(1,2) - u^{\star}_2 - u^{\star}_{c_1} = w(1,2) - w(4,2) - (w(0,4) - w(2,4))$. $rc(1,2) = 16$. *The corresponding graph is presented in* Figure 7.6 *(a). Of course, the new cycle $(1,2)$ created must be broken from the vertex* 1. $irc(1,2) = rc(1,2) + (w(0,1) - w(2,1)) = 16 + 29 = 45$. *Actually, that is the exact reduced cost since the new graph obtained is an arborescence (see* Figure 7.6 *(b));*

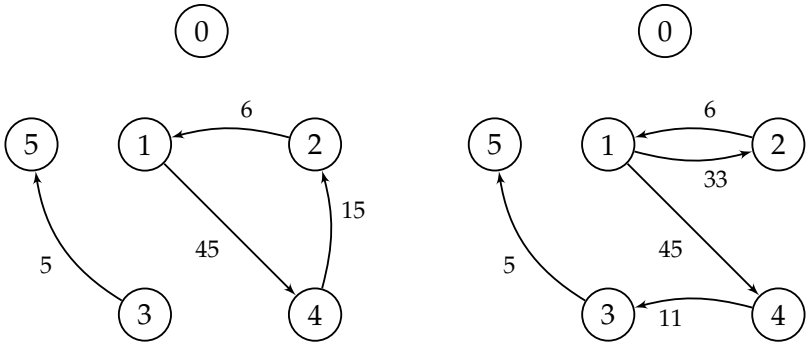
2. $(1,4)$: $rc(1,4) = w(1,4) - u^{\star}_4 - u^{\star}_{c_1} = w(1,4) - w(2,4) - (w(0,4) - w(2,4))$. $rc(1,4) = 37$. *The corresponding graph is presented in* Figure 7.7 *(a). But* $irc(1,4) = rc(1,4) + \min\{w(0,1) - w(2,1) = 29, w(1,2) - w(4,2) - u^{\star}_{c_1} = 16\}$. *Thus* $irc(1,4) = 37 + 16 = 53$. *Here (see* Figure 7.7 *(b)), the graph obtained is not an arborescence and $irc(1,4)$ is a lower bound.*

Algorithm 7.3.1 computes $irc(i,j), \forall(i,j)$ in $O(|V|^2)$. First, it initializes each $irc(i,j)$ to $rc(i,j)$, $\forall(i,j) \in E$. Then, for each edge (i,j) involved in the assumption of Proposition 34 (Invariant (a)), Algorithm 7.3.1 updates its $irc(i,j)$ according to the formula of Proposition 34.

Algorithm 7.3.1: Computation of the improved reduced costs $irc(i,j), \forall (i,j) \in E$ in $O(|V|^2)$

Input: $parent[k], \forall k \in V$; $pred[k], \forall k \in V$; $u^\star_{c_i}, \forall c_i \in \mathcal{C}$ and $bestTwoDiff[k], \forall k \in V$ that can be computed in $O(|V|^2)$
Output: $irc(i,j), \forall (i,j) \in E$

1 **foreach** each edge $(i,j) \in E$ **do**
2 $\quad irc(i,j) \leftarrow rc(i,j)$
3 **end**
4 **foreach** each vertex $i \in V$ **do**
5 $\quad$ **if** $parent[parent[i]] = \emptyset$ **then**
6 $\quad\quad min \leftarrow bestTwoDiff[i] - u^\star_{parent[i]}$
7 $\quad\quad j = pred[i]$
8 $\quad\quad$ **while** $(parent[parent[j]] = \emptyset) \wedge min > 0 \wedge j \neq r$ **do**
9 $\quad\quad\quad$ // **Invariant (a): there is a path $\mathcal{P}$ from j to i such that** $\forall k \in \mathcal{P} : parent[parent[k]] = \emptyset$
10 $\quad\quad\quad$ // **Invariant (b):** $min = \min_{k \in \mathcal{P} \setminus \{j\}} \{bestTwoDiff[k] - u_{parent[k]^\star}\}$
11 $\quad\quad\quad irc(i,j) \leftarrow irc(i,j) + min$
12 $\quad\quad\quad$ **if** $bestTwoDiff[j] - u_{parent[j]^\star} < min$ **then**
13 $\quad\quad\quad\quad min \leftarrow bestTwoDiff[j] - u^\star_{parent[j]}$
14 $\quad\quad\quad$ **end**
15 $\quad\quad\quad j = pred[j]$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 **end**



(a) G_1 with $rc(1,4)$ (b) G_1 with $irc(1,4)$

Figure 7.7: G_1 with $rc(1,4)$ and $irc(1,4)$

7.4 EXPERIMENTAL RESULTS

This section presents the experiments performed to evaluate the improved reduced costs and the filtering based on reduced costs. All our source-code for the models, the global constraint, and the instances are available at [HSa].

Proportion of the reduced costs that are improved

As a first experiment, we evaluate the proportion of reduced costs affected by Proposition 34. Therefore we randomly generated two classes of 100 instances with $w(i,j) \in [1,100]$ and different values of the number of vertices ($|V| = 20$, $|V| = 50$ and $|V| = 100$):

- *class*1: for each $i \in V$, $parent[parent[i]] = \emptyset$;
- *class*2: many vertices $i \in V$ are such that $parent[parent[i]] \neq \emptyset$.

The *class*1 was obtained by filtering out the random instances not satisfying the property $parent[parent[i]] = \emptyset, \forall i \in V$. In the following, we denote by $\lambda(i,j)$ the exact reduced cost associated to the edge $(i,j) \in E$ that is the cost increase if the (i,j) is forced to be $A(G)^\star$. This cost is obtained by computing an optimal solution with (i,j) as the single edge entering j and then subtract $w(A(G)^\star)$ from its optimal cost. Table 7.1 shows, for each class of instances (with respectively $|V| = 20$, $|V| = 50$ and $|V| = 100$), the proportion of instances of each class and for each group of instances: 1) the proportion of edges $(i,j) \in E$ such that

$rc(i,j) < \lambda(i,j)$; 2) the proportion of edges that have $irc(i,j) > rc(i,j)$; and 3) the proportion of edges such that $irc(i,j) = \lambda(i,j)$. Note that, for this benchmark, at least 37% of 300 instances are $class1$ instances and at least 45% of LP reduced costs (with $rc(i,j) < \lambda(i,j)$) are improved for these instances. Of course, the results are less interesting for the $class2$ instances.

	$\|V\| = 20$		$\|V\| = 50$		$\|V\| = 100$	
	Class1	Class2	Class1	Class2	Class1	Class2
%: instances of class$k$ ($k \in \{1,2\}$)	48	52	31	69	32	68
%: $rc(i,j) < \lambda(i,j)$	19.3	38.1	9.8	26.7	2.1	16.6
%: $irc(i,j) > rc(i,j)$	12.6	1.9	4.6	0.9	1.2	0.2
%: $irc(i,j) = \lambda(i,j)$	9.9	1.17	3.49	0.79	1.19	0.2

Table 7.1: Proportion of the reduced costs affected by Proposition 34

The benefit of the cost-based filtering

To test the `MinArborescence` constraint, experiments were conducted on an NP-Hard variant of the CAP: the RMWA that is described in Section 4.3. The RMWA can be modeled in CP with a `MinArborescence` constraint (or one of its decomposition) for the MWA part of problem and the `binaryKnapsack` constraint [FS02] together with `weightedSum` constraint for the resource constraints. We have randomly generated the different costs/weights as described in [GR90]: $a_{i,j} \in [10,25]$, $w(i,j) \in [5,25]$. To have more available edges to filter, we have used $b_i = 2 \cdot \lfloor \frac{\sum_{(i,j)\in\delta_i^+} a_{i,j}}{|\delta_i^+|} \rfloor$ (instead of $b_i = \lfloor \frac{\sum_{(i,j)\in\delta_i^+} a_{i,j}}{|\delta_i^+|} \rfloor$) and 75% graph density (each edge has a 25% probability to have an high value such that it will not be selected in an MWA).

In order to avoid the effect of the dynamic first fail heuristic interfering with the filtering, we use the approach described in Introduction (see Section 1.4). Here, the baseline is the decomposition model using the `Arborescence` constraint. This search tree is then replayed with the stronger reduced cost based filtering for `MinArborescence`. The recorded search tree for each instance corresponds to an exploration of 30 seconds. As an illustration for the results, Table 7.2 re-

ports the arithmetic average results (for 100 randomly instances with $|V| = 50$) for `MinArborescence` constraint with filtering respectively based on improved reduced costs (MinArbo_IRC), reduced costs (MinArbo_RC), the decomposition with `Arborescence` constraint (Arbo) and Arbo+filtering only based on lower bound of MWA (Arbo+LB). Table 7.2 also shows the geometric average gain factor (wrt Arbo) for each propagator. On average, the search space is divided by 566 with the reduced cost based filtering `MinArborescence` constraint (wrt Arbo) and by 166 with Arbo+LB. This demonstrates the benefits brought by the `MinArborescence` constraint introduced in this chapter.

	MinArbo_IRC		MinArbo_RC		Arbo+LB		Arbo	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	**14385**	**0**	**14385**	**0**	81239	1.4	6646748	28
Av. gain factor	**566**	**-**	**566**	**-**	166	-	1.0	1.0

Table 7.2: Average results on 100 instances: MinArbo_IRC, MinArbo_RC, Arbo+LB, and Arbo

MinArbo_RC vs MinArbo_IRC

To further differentiate the filtering of MinArbo_IRC, we now use MinArbo_RC as a baseline filtering for recording the search tree on another set of 100 randomly generated instances of *class*1 with $|V| = 50$. Figure 7.8 and Figure 7.9 show the corresponding performance profiles wrt the number of nodes visited and the time used respectively. For $\approx$ 30% of instances the search space is divided by at least 1.5 and for $\approx$ 7% the search space is divided by at least 4. About time performance profile, MinArbo_IRC is $\approx$ 1.5 times faster than MinArbo_RC for $\approx$ 20% of instances and is $\approx$ 2 times faster than MinArbo_RC for $\approx$ 10% of instances. Table 7.3 presents the average results on 100 instances. Unfortunately, as was expected, the average gain is limited as only $\approx$ 5% of LP reduced costs can be improved. We detail, in Table 7.4, the computational results on the first 15 instances. As shown in Table 7.4, MinArbo_IRC is really interesting only for some instances (2, 3, 9, 10).

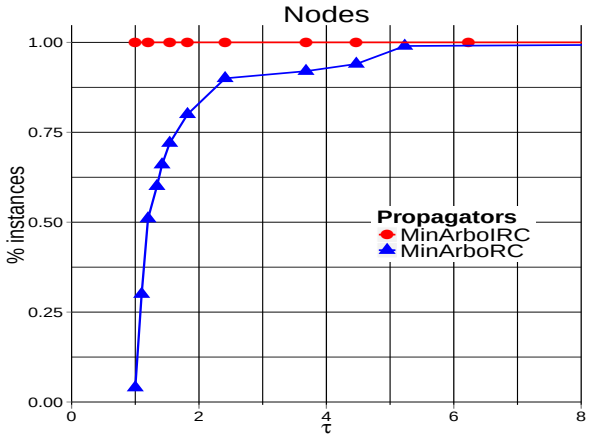


Figure 7.8: Performance profiles - Nodes: MinArbo_RC and MinArbo_IRC

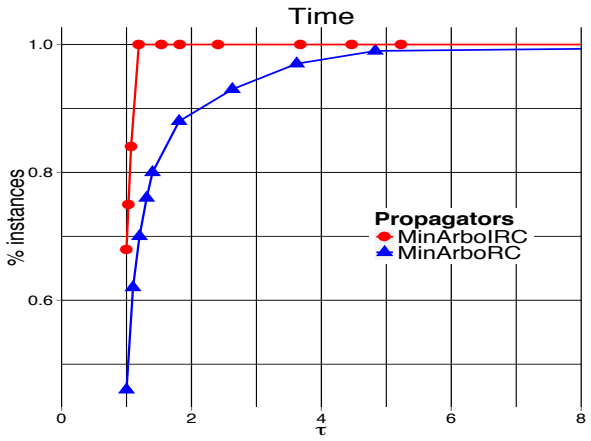


Figure 7.9: Performance profiles - Time: MinArbo_RC and MinArbo_IRC

	MinArbo_IRC		MinArbo_RC	
	Nodes	Time	Nodes	Time
Average (Av.)	**359922**	**24**	507555	28
Av. gain factor	**1.46**	**1.22**	1.0	1.0

Table 7.3: Average results on 100 instances: MinArbo_IRC vs MinArbo_RC

Instance	MinArbo_IRC		MinArbo_RC	
	Nodes	Time(s)	Nodes	Time
1	**392758**	29	416512	**27**
2	**135714**	**8**	644828	29
3	**235336**	**15**	499904	28
4	**359444**	**26**	494172	29
5	**326638**	32	335132	**29**
6	**394179**	**29**	406095	**29**
7	**306708**	**26**	419106	28
8	**532913**	29	569113	**28**
9	**111963**	**6**	553500	29
10	**155036**	**9**	811462	28
11	**295616**	**21**	539794	35
12	**256654**	29	302816	**28**
13	**301595**	**29**	333861	**29**
14	**345283**	**26**	480511	29
15	**344903**	32	346807	**29**

Table 7.4: Results on 15 instances: MinArbo_IRC vs MinArbo_RC

About the specialized OR approaches for the RMWA

Our CP model is able to solve and prove optimality of the RMWA instances with up to $|V| = 50$ (see Appendix B). Similar instances can be solved using the Lagrangian decomposition approach of [GR90]. The branch and cut algorithm of [FV97] reports results solving instances with up to $|V| = 500$. We believe this lack of performance of CP wrt to the branch and cut approach is due to the $|V|$ independent knapsack constraints inducing a weaker pruning. We should also develop some dedicated search heuristics to improve the CP performance on this problem.

7.5 CONCLUSION

We have defined the `MinArborescence` constraint based on the reduced costs to filter the edges. We have proposed an algorithm to improve the LP reduced costs of the minimum weighted arborescence in some cases. Finally, we have demonstrated experimentally the interest of improved reduced costs in some particular graphs and the efficiency of the cost-based filtering on the resource constrained minimum weight arborescence problem.

CONCLUSION AND PERSPECTIVES

CONCLUSION AND PERSPECTIVES

The research results presented in this thesis mainly revolve around global optimization constraints and cost-based filtering algorithms. We have introduced two new complex optimization problems in Constraint Programming (CP): a Capacitated Lot Sizing Problem (CLSP) and the Constrained Arborescence Problem (CAP)[7]. For each of these optimization problems, we have proposed some cost-based filtering algorithms to handle them in CP. We below summarize our main results and the possible future orientations.

CAPACITATED LOT SIZING PROBLEM

Lot Sizing (LS) is often an important aspect in production planning. We have focused on how to deal with the stocking costs that may arise in a CLSP. Actually, for the most of variants of the CLSP, the sum of stocking costs for each order forms part of the objective function to be minimized. First, we have introduced the `StockingCost` constraint. This constraint enforces that each order is produced on or before its due date, the production capacity of the machine is respected, and the total stocking cost is less than a given value. We have established some relevant properties/propositions about this constraint. Then we have developed an $O(n)$ (with n the number of orders) algorithm to solve an associated problem to optimality and to calculate the cost of modifying the optimal solution by forcing an order to be produced earlier than in the optimal solution. These costs allow to efficiently achieve bound consistent filtering for the `StockingCost` constraint. We have tested our

7 The CAP is also a sub-problem of some variants of the CLSP.

filtering algorithms on a variant of the CLSP: the Pigment Sequencing Problem (PSP). The experimental results show that our new filtering algorithm outperforms the formulation based on the `minimumAssignment` constraint[8]. Further, we have generalized the `StockingCost` constraint to take into account the order dependent per-period stocking costs and the production capacity that may vary over time. So we have proposed the `IDStockingCost` constraint. For this new global constraint, we have developed an $O(n \log n)$ filtering algorithm. This new filtering algorithm does not achieve bound consistency for the `IDStockingCost` constraint but works well in practice on large instances. Again, we have tested the filtering algorithm on the PSP with orders that may have different per-period stocking costs. The experimental results show that our new filtering algorithm scales well compared with the state-of-the-art CP propagators. Moreover, on large instances, it is competitive with the filtering algorithm introduced for the `StockingCost` constraint when all orders have the same per-period stocking cost.

We below state some possible future works for the CLSP in CP:

- Filtering: a possible perspective is to develop an efficient filtering algorithm with a stronger form of consistency for the `IDStockingCost` constraint. On the other hand, we have focused on the global optimization constraints for the stocking costs. In the CLSP, some other costs may be involved in the objective function. For example, in the PSP, there are the changeover costs together with the stocking costs to be minimized. To improve CP performance on the CLSP, it would be interesting to develop some optimization oriented propagators for the setup costs, the changeover costs, the production costs, etc. It could also be interesting to develop global filtering algorithms that combine several different costs.

- Search: we are mainly interested in filtering. Since CP is $filtering + search$, it would be interesting to develop some custom heuristics for these problems. One could also design generic heuristics linked to a global optimization constraint such as the `IDStockingCost` constraint (see for example heuristics based on counting good solutions introduced in [Pes16]). On the other hand, to solve big instances, one could try some local search approaches ([Sha98, HM05, HT06]).

8 To the best of our knowledge, this formulation is the best state-of-the-art CP formulation for the stocking costs part of the PSP.

- More experiments:
  - we have noticed that the global constraints introduced filter better when the stocking costs are high. It could be interesting to do more expermients to see the performance of CP with regard to other approaches by varying different parameters such as the number of periods, the density of the demands, the different costs involved, etc.
  - the experiments were conducted on the PSP. A possible perspective is to model and experiment other variants of the CLSP.

CONSTRAINED ARBORESCENCE PROBLEM

In this thesis, we have also dealt with a graph theory problem: the Minimum Weight Arborescence (MWA). The MWA can be used to model many real life problems (for example in telecommunication networks). We have introduced the CAP (that requires one to find an arborescence that satisfies some side constraints and that has the minimum weight) in CP. We have proposed an optimization constraint - the `MinArborescence` constraint - to tackle the CAP in CP. This constraint enforces that a set of edges is a valid arborescence rooted at a given vertex with total cost less than a given value. The filtering of the `MinArborescence` constraint can be performed with the LP reduced costs that can be obtained in $O(|V|^2)$ for all $O(|V|^2)$ edges (with V the set of vertices). We have developed an $O(|V|^2)$ algorithm to strengthen the quality of the LP reduced costs based on sensitivity analysis of MWA. Our experiments showed that our improved reduced costs are stronger than the LP ones when a few contracting cycles are needed to compute the MWA. Finally we have demonstrated experimentally the benefits of the cost-based filtering on a variant of the CAP: the resource constrained MWA problem.

As for the CLSP, the perspectives about the CAP can be classify as follows:

- Filtering: a good extension of this research is to develop some filtering algorithms wrt to the other constraints involved in the variants of this problem. For example about the resource constrained MWA problem, one could propose a global filtering algorithm for the $|V|$ independent knapsack contraints. On the other hand, the `MinArborescence` constraint would benefit from

incremental computation of the MWA and its reduced costs in order to reduce the $O(|V|^2)$ complexity.

- Search: it would be interesting to develop some dedicated heuristics for the CAP as well as generic ones wrt the `MinArborescence` constraint. On the other hand, applying some local search approaches on big instances could also be interesting.
- More experiments:
  - some interesting research questions are 1) how the different parameters (density of the graph, size of the graph, weights of the edges, etc.) influence the strength of the filtering based on reduced costs and the quality of the improved reduced costs ? 2) how much the reduced costs are improved ? 3) is it interesting to do more research to efficiently improve the quality of the reduced costs ?
  - we have mainly experimented the resource constrained MWA problem. As for the constrained spanning tree problem, one can think about other variants of the CAP such as the degree MWA problem, the diameter MWA problem, the weighted MWA problem, etc. In particular, we believe that CP can be effective on the degree MWA problem by using the `gcc` constraint [Rég96, Qui+03].

FINAL WORDS

To the best of our knowledge, our research on the CLSP and the CAP are the first in the CP community about these challenging optimization problems. We do hope that these results will trigger more research in the future to make CP more efficient on these interesting problems. This has already started for the CLSP with the German et al. paper [Ger+15]. On the other hand, research on search aspects should be conducted in order to compare the CP approach with the specialized approaches on these problems. Finally, an interesting research direction is to hybridize several OR approaches (including the CP approach) on the different variants of the CLSP and the CAP.

Part IV

APPENDIX

A

CP APPROACH RESULTS FOR THE PSP - INSTANCES SOLVED TO OPTIMALITY

We report in this appendix our best results on 100 instances (with $n = T = 20$) of the PSP solved to optimality. The results are obtained with the *MinAss* based model using COS heuristic [Gay+15]. Table A.1 and Table A.2 show on each instance the number of nodes (Nodes) and the time in seconds (Time) needed to complete the search. These tables also show the optimal cost for each instance. All the instances, the results, and the source codes are available at [HSb].

Instance	Nodes	Time(s)	OptCost	Instance	Nodes	Time(s)	OptCost
1	21012	9.14	1377	26	6990	1.575	1753
2	52214	7.292	1447	27	42580	5.837	1277
3	20928	2.946	1107	28	25774	3.287	1036
4	10666	1.784	1182	29	1148	0.211	880
5	1498	0.235	1471	30	1306	0.234	588
6	1182	0.186	1386	31	45798	6.812	2249
7	25978	4.068	1382	32	19314	3.009	1562
8	157560	25.352	3117	33	420	0.057	1104
9	1122	0.201	1315	34	1002	0.193	1108
10	3388	0.653	1952	35	105872	12.846	2655
11	10750	1.715	1202	36	918538	121.909	1493
12	25564	4.161	1135	37	41196	5.494	1840
13	2758	0.492	1026	38	6190	0.955	1113
14	6874	1.181	1363	39	9576	1.549	1744
15	6542	1.034	1430	40	16414	2.334	1193
16	15004	2.441	1145	41	1032	0.152	1189
17	18532	3.617	1367	42	8786	1.534	996
18	610	0.111	1315	43	1956	0.296	1995
19	10904	1.94	1490	44	10068	1.986	1339
20	22722	3.139	779	45	21280	3.465	1182
21	89702	11.177	2774	46	12404	2.154	1575
22	84592	10.432	1397	47	4396	0.743	1371
23	88708	15.039	1473	48	5376	0.899	1572
24	390	0.06	1308	49	16732	2.886	1882
25	2988	0.616	1810	50	18002	2.758	1405

Table A.1: Results on 100 PSP instances with *MinAss* based model and COS heuristic (Part 1)

Instance	Nodes	Time(s)	OptCost	Instance	Nodes	Time(s)	OptCost
51	8854	1.464	1397	76	10980	1.799	1484
52	9128	1.732	1531	77	3928	0.596	852
53	5602	0.936	1108	78	128864	16.187	1297
54	36322	5.602	1056	79	4530	0.947	1555
55	69128	9.09	1015	80	7668	1.331	1382
56	14100	2.152	2121	81	27902	4.254	1889
57	10446	1.782	919	82	29886	5.004	1947
58	16416	2.347	1384	83	4332	0.726	1681
59	147094	19.496	1490	84	2510	0.36	728
60	18940	2.895	1265	85	65888	9.404	2113
61	3330	0.711	977	86	2238	0.406	886
62	11040	1.6	872	87	9650	1.589	1152
63	1792	0.321	1481	88	22196	3.361	1353
64	10222	1.713	1869	89	43814	6.083	1390
65	39804	6.25	1781	90	177784	23.811	2449
66	7962	1.715	1571	91	17890	2.862	998
67	59610	8.501	1185	92	1440	0.19	1453
68	46692	6.979	1131	93	15702	2.279	1286
69	6486	1.223	1619	94	64326	11.726	1403
70	15208	2.447	1148	95	846	0.12	1618
71	12528	2.077	1024	96	15100	2.543	1409
72	17610	3.346	1555	97	34704	5.354	1290
73	76846	12.508	1104	98	2838	0.463	1403
74	2810	0.504	1053	99	17702	3.318	1237
75	6962	1.033	1021	100	19054	3.003	1040

Table A.2: Results on 100 PSP instances with *MinAss* based model and COS heuristic (Part 2)

B

CP APPROACH RESULTS FOR THE RMWA PROBLEM - INSTANCES SOLVED TO OPTIMALITY

We report in this appendix our best results on 100 instances (with $|V| = 50$) of the RMWA problem solved to optimality. The results are obtained with the *Arbo* + *MinArbo_RC* based model using COS heuristic [Gay+15]. Table B.1 and Table B.2 show on each instance the number of nodes (Nodes) and the time in seconds (Time) needed to complete the search. These tables also show the optimal cost for each instance. All the instances, the results, and the source codes are available at [HSa].

Instance	Nodes	Time(s)	OptCost	Instance	Nodes	Time(s)	OptCost
1	1308	7.417	254	26	474	0.392	255
2	1150	0.962	253	27	1918	1.199	257
3	2110	1.1	253	28	2448	1.543	262
4	454	0.348	260	29	6226	2.767	257
5	11898	6.457	265	30	1682	0.884	264
6	1166	0.745	260	31	1124	0.473	256
7	578	0.448	255	32	2642	1.313	260
8	814	0.647	259	33	870	0.525	254
9	634	0.528	259	34	946	0.51	253
10	380	0.215	253	35	960	0.717	264
11	2586	1.434	264	36	688	0.434	260
12	22002	8.319	259	37	634	0.565	256
13	1306	0.667	257	38	1056	0.57	258
14	774	0.583	259	39	2384	1.53	259
15	3840	1.822	264	40	1986	1.095	260
16	1430	0.848	265	41	826	0.443	257
17	12588	6.23	261	42	418	0.443	257
18	1194	0.661	261	43	14682	7.146	261
19	892	0.487	252	44	172	0.176	252
20	1798	1.18	259	45	694	0.582	258
21	736	0.544	257	46	156	0.205	255
22	434	0.485	260	47	1098	0.614	258
23	1862	0.991	255	48	666	0.498	256
24	662	0.454	257	49	530	0.517	256
25	1114	0.615	256	50	550	0.524	254

Table B.1: Results on 100 RMWA instances with *Arbo* + *MinArbo_RC* based model and COS heuristic (Part 1)

Instance	Nodes	Time(s)	OptCost	Instance	Nodes	Time(s)	OptCost
51	724	0.411	253	76	13222	6.101	265
52	1148	0.789	260	77	320	0.284	255
53	162	0.327	259	78	240	0.358	258
54	556	0.441	253	79	514	0.374	256
55	2088	1.22	258	80	312	0.374	257
56	784	0.509	256	81	600	0.466	255
57	2108	1.164	251	82	2178	1.014	254
58	402	0.441	255	83	420	0.424	258
59	474	0.456	260	84	3168	1.791	255
60	686	0.521	254	85	1014	0.673	259
61	846	0.574	259	86	1364	0.833	263
62	632	0.461	254	87	136	0.267	255
63	712	0.434	260	88	4844	2.469	263
64	2822	1.294	261	89	1906	1.114	258
65	2382	0.856	258	90	196	0.168	256
66	5900	3.451	258	91	7698	3.592	257
67	1596	0.81	257	92	3910	2.441	262
68	1660	1.17	258	93	1410	0.599	261
69	2790	1.367	254	94	538	0.388	251
70	760	0.481	255	95	2494	1.245	257
71	996	0.529	253	96	1152	0.627	262
72	1312	0.651	259	97	334	0.153	262
73	3262	1.745	257	98	244	0.159	257
74	2414	1.509	256	99	788	0.593	260
75	1290	0.742	263	100	1830	0.82	263

Table B.2: Results on 100 RMWA instances with *Arbo* + *MinArbo_RC* based model and COS heuristic (Part 2)

C

CODING EXPERIENCE AND SOURCE CODE

We provide in this appendix our implementations of the different filtering algorithms after a word about our coding experience. These implementations are available in the Oscar open source solver [Osc12].

CODING EXPERIENCE. We have spent a lot of time to implement the algorithms that use the main theoretical propositions contained in this thesis. At our beginnings, as soon as we have an idea that seems good, we are in a hurry to implement it to see if it works or not. We have quickly noticed that it is not always a good idea to precipitate ourselves on the implementation. Actually, sometimes, after a long time of coding and testing, we may observe that there is a bug not in the implementation but in the proposition used by the algorithm. Sometimes such bugs could easily be detected by trying to sketch a proof of the proposition. On the other hand, sometimes it is hard to prove a proposition. In these cases, a quick implementation of the idea and some intensively random tests with small instances make us more confident about the correctness of the proposition or allow us to adjust the proposition by, for example, taking into account some special cases. For the kind of research we have focused on, one need to have a good tradeoff between the time spent on the proofs of the propositions and the time spent on the implementation. Of course, it depends on how hard it is to 1) prove the proposition and 2) implement the algorithm.

The implementation task is not always easy when one wants an efficient and clear code. In particular, when the codes are developed for a big open source project like the OscaR solver, the quality requirements are high. In general, we obtain the final version after two, three

or more implementations of the algorithm. The first one is often easy to be sure that the proposition associated is well implemented. This is performed by the code testing procedure described in Section 1.4. After the first version, different improvements can be achieved (for example, with regards to the data structures used) until the current version is satisfactory enough. It is worth noting that, sometimes, the implementations help us to improve the algorithm itself or the theoretical propositions associated. The final version is generally obtained after some iterations over 1) the propositions, 2) the algorithms and 3) the implementations until satisfaction.

Listing C.1: The StokingCost constraint - filtering

```
/**
  * The StockingCost constraint holds when each item is produced before
  * its due date (X_i <= d_i), the capacity of the machine is respected
  * (i.e. no more than c variables X_i have the same value), and H
  * is an upper bound on the total stocking cost (sum_i(d_i - X_i) <= H).
  *
  * This constraint is useful for modeling
  * Production Planning Problem such as Lot Sizing Problems
  *
  * @param Y, the variable Y_i is the date of production of item i on the machine
  * @param d, the integer d_i is the due-date for item i
  * @param H, the variable H is an upper bound on the total number of slots all
       the items are need in stock.
  * @param c is the maximum number of items the machine can produce during one
      time slot (capacity). If an item is produced before its due date, then it
       must be stocked.
  *
  */
class StockingCost(val Y: Array[CPIntVar], val deadline: Array[Int], val H:
    CPIntVar, val c: Int) extends Constraint(Y(0).store, "StockingCost") {

  val allDiffBC = new AllDiffBC(Y)
  val n = Y.size
  var domMaxMax = Int.MinValue
  var domMinMin = Int.MaxValue
  var k = 0
  while (k < Y.size) {
    val m = Y(k).min
    val M = Y(k).max
    if (m < domMinMin) domMinMin = m
    if (M > domMaxMax) domMaxMax = M
    k += 1
  }
  val X = Array.tabulate(n)(Y(_))
  val Xmax = Array.fill(n+1)(Int.MinValue)
  val d = Array.fill(n)(0)
  val vopt = Array.fill(n)(0)

  override def setup(l: CPPropagStrength): Unit = {
    X.foreach(_.callPropagateWhenBoundsChange(this))
    H.callPropagateWhenBoundsChange(this)
    propagate()
  }

  //  --------------map  ---------------------
  var magic = 1
  val mapMagic = Array.fill(domMaxMax-domMinMin+1)(0)
  val map = Array.fill(domMaxMax-domMinMin+1)(0)

  def clearMap() {
    magic += 1
  }
  def index(k: Int) = k-domMinMin
```

```
50    def insert(k: Int,v: Int) {
51      map(index(k)) = v
52      mapMagic(index(k)) = magic
53    }
54    def hasKey(k: Int) = {
55      if (k < domMinMin) false
56      else mapMagic(index(k)) == magic
57    }
58    def get(k: Int) = map(index(k))
59
60    //  --------------incremental sort  ---------------------
61
62    val sortX = Array.tabulate(X.size)(i => i)
63
64    def sortIncremental() {
65      var nn = X.size
66      var i = 0
67      do {
68        var newn = 0
69        i = 1
70        while (i < nn) {
71          if (Y(sortX(i - 1)).max < Y(sortX(i)).max) {
72            val tmp = sortX(i - 1)
73            sortX(i - 1) = sortX(i)
74            sortX(i) = tmp
75            newn = i
76          }
77          i += 1
78        }
79        nn = newn
80      } while (nn > 0);
81      k = 0;
82      while (k < n) {
83        X(k) = Y(sortX(k))
84        Xmax(k) = X(k).max
85        d(k) = deadline(sortX(k))
86        k += 1
87      }
88    }
89
90    override def propagate(): Unit = {
91      allDiffBC.propagate()
92      sortIncremental()
93      var t = Xmax(0)
94      var i = 0
95      var j = 0 //open items {j, ... ,i\} must be placed in some slots
96      var k = 0 //items {k, ... ,i\} have same vOpt
97      var u = t+1
98      clearMap()
99      var Hopt = 0
100     var ind = 0
101     while (j < i || i < n) {
102       while (i < n && Xmax(i) == t) {
103         i += 1
```

```
104       }
105       // place at most c items into slot t
106       ind = j
107       while(ind <= (i-1).min(j+c-1)){ //update Hopt
108         Hopt += d(ind) - t
109         ind += 1
110       }
111
112       if (i - j <= c) { // all the open items can be placed in t
113       val full = (i-j) == c
114         ind = k
115         while(ind < i){
116           vopt(ind) = t
117           ind += 1
118         }
119         j = i
120         k = i
121         if (full) {
122           insert(t, u)
123           if (Xmax(i) < t-1) {
124             u = Xmax(i)+1
125           }
126         } else {
127           u = Xmax(i)+1
128         }
129         t = Xmax(i)
130       } else { // all the open items can not be placed in t
131         insert(t,u) //place c items into slot t
132         j += c
133         t -= 1
134       }
135     }
136     H.updateMin(Hopt)
137     val slack = H.max - H.min
138     i = 0
139     while (i < n) {
140       var newmin = vopt(i)  - slack
141       if (hasKey(newmin)) {
142         newmin = vopt(i).min(get(newmin))
143       }
144       X(i).updateMin(newmin)
145       i += 1
146     }
147   }
148 }
```

Listing C.2: The Item Dependent StokingCost constraint - filtering

```
1  /**
2    * The IDStockingCost constraint holds when each item is produced before
3    * its due date (X_i <= d_i), the capacity of the machine is respected
4    * (i.e. no more than c variables X_i have the same value), and H
5    * is an upper bound on the total stocking cost (sum_i((d_i - X_i) * h_i) <= H).
6    *
7    * This constraint is the generalization of StockingCost constraint to
8    * item dependent stocking cost and useful for modeling
9    * Production Planning Problem such as Lot Sizing Problems
10   *
11   * @param Y        , the variable Y_i is the date of production of item i on the
         machine
12   * @param deadline , the integer deadline_i is the due-date for item i
13   * @param h        , the integer h_i is the stocking cost for item i
14   * @param H        , the variable H is an upper bound on the total number of
         slots all the items are need in stock.
15   * @param cap      , the integer cap_t is the maximum number of items the
         machine can produce during one time slot t (capacity). If an item is
         produced before its due date, then it must be stocked.
16   *
17   */
18
19 class IDStockingCost(val Y: Array[CPIntVar], val deadline: Array[Int], val h:
       Array[Int], val H: CPIntVar, val cap: Array[Int]) extends Constraint(Y(0).
       store, "IDStockingCost") {
20
21   val allDiffBC = new AllDiffBC(Y)
22
23   val n = Y.size
24   val X = Array.tabulate(n)(Y(_))
25   val Xmax = Array.ofDim[Int](n)
26   val aux = Array.ofDim[Int](n + 1)
27   val runs = Array.ofDim[Int](n + 1)
28   var domMaxMax = Y.map(_.max).max
29   var domMinMin = Y.map(_.min).min
30   val c = Array.tabulate(domMaxMax + 1)(t => cap(t))
31   val optimalSlotTab = Array.fill(n + 1)(-1)
32   val optimalItemTab = Array.tabulate(domMaxMax + 1)(t => new
         ReversibleArrayStack[Int](s, c(t)))
33   val ordersToSchedule = new ArrayHeapInt(n)
34   val candidateTojump = new ArrayStackInt(n)
35   val fullSetsStack = new StackOfStackInt(2 * n)
36   val gainCostTab = Array.fill(domMaxMax + 1)(0)
37
38   // ---------------- sparse-set for isolating unbounded variables ---------
39
40   val capa = Array.tabulate(domMaxMax + 1)(t => new ReversibleInt(s, c(t)))
41   val fixedCost = new ReversibleInt(s, 0)
42   val nUnbound = new ReversibleInt(s, n)
43   val unBoundIdx = Array.tabulate(n)(i => i)
44
45   def processFixed(): Unit = {
46     var nUnboundTmp = nUnbound.value
```

```
47      var additionalFixedCost = 0
48      var i = nUnboundTmp
49      while (i > 0) {
50        i -= 1
51        val idx = unBoundIdx(i)
52        if (X(idx).isBound) {
53          // we decrease the capa
54          capa(X(idx).value).decr()
55          // compute the contribution of this item to the objective
56          additionalFixedCost += (deadline(idx) - X(idx).value) * h(idx)
57          // remove this item from the unbound ones
58          val tmp = unBoundIdx(nUnboundTmp - 1)
59          unBoundIdx(nUnboundTmp - 1) = idx
60          unBoundIdx(i) = tmp
61          nUnboundTmp -= 1
62        }
63      }
64      nUnbound.value = nUnboundTmp
65      fixedCost.value = fixedCost.value + additionalFixedCost
66    }
67
68    override def setup(l: CPPropagStrength): Unit = {
69      Y.foreach(_.callPropagateWhenBoundsChange(this))
70      H.callPropagateWhenBoundsChange(this)
71      propagate()
72    }
73
74    override def propagate(): Unit = {
75
76      allDiffBC.propagate()
77
78      // ----------- some preprocessing computation --------------------
79
80      processFixed()
81
82      val nU = nUnbound.value
83
84      //assert((0 until nU).forall(i => !X(unBoundIdx(i)).isBound))
85      //assert((nU + 1 until n).forall(i => X(unBoundIdx(i)).isBound))
86      //assert(fixedCost.value == (0 until n).filter(i => X(i).isBound).map(i => (
            deadline(i) - X(i).value) * h(i)).sum)
87
88      var i = nU
89      while (i > 0) {
90        i -= 1
91        Xmax(unBoundIdx(i)) = -X(unBoundIdx(i)).max
92      }
93      SortUtils.mergeSort(unBoundIdx, Xmax, 0, nU, aux, runs)
94
95      //assert((0 until nU - 1).forall(i => X(unBoundIdx(i)).max >= X(unBoundIdx(i
            + 1)).max))
96
97      // put the max as positive values again ... ;-)
98      i = nU
```

```
99      while (i > 0) {
100       i -= 1
101       Xmax(unBoundIdx(i)) = -Xmax(unBoundIdx(i))
102     }
103
104     // ----------- compute Hopt --------------------
105
106     var Hopt = 0
107     ordersToSchedule.clear()
108     fullSetsStack.reset()
109
110    // assert((0 to domMaxMax).forall(t => optimalItemTab(t).isEmpty))
111
112     var k = 0
113     while (k < nU) {
114       var t = Xmax(unBoundIdx(k))
115       var availableCapacity = capa(t).value
116       do {
117         while (k < nU && Xmax(unBoundIdx(k)) == t) {
118           val i = unBoundIdx(k)
119           ordersToSchedule.enqueue(-h(i), i)
120           k += 1
121         }
122         if (availableCapacity > 0) {
123           val currentInd = ordersToSchedule.dequeue
124           optimalSlotTab(currentInd) = t
125           optimalItemTab(t).push(currentInd)
126           availableCapacity = availableCapacity - 1
127           Hopt = Hopt + (deadline(currentInd) - t) * h(currentInd)
128         }
129         else {
130           fullSetsStack.push(t)
131           t = t - 1
132           while (capa(t).value == 0) t = t - 1
133           availableCapacity = capa(t).value
134         }
135       } while (ordersToSchedule.size > 0)
136       fullSetsStack.push(t)
137       fullSetsStack.pushStack()
138     }
139     val Hmin = Hopt + fixedCost.value
140
141     H.updateMin(Hmin)
142
143     // ----------- now compute the gain costs ----------------
144
145     val nFullSet = fullSetsStack.nStacks()
146     i = 0
147     while (i < nFullSet) {
148       /* size of current full set */
149       val fullSetSize = fullSetsStack.sizeTopStack()
150       candidateTojump.clear
151       var j = 0
152       while (j < fullSetSize) {
```

```
153         // set t to the next time slot of the current fullSet
154         val t = fullSetsStack.pop
155         // filter out candidate top candidate items that can not be placed in t
156         // such that the one that remains on top is the most costly one that can
                jump
157         while (!candidateTojump.isEmpty && Xmax((candidateTojump.top)) < t) {
158           candidateTojump.pop
159         }
160         if (candidateTojump.isEmpty) {
161           gainCostTab(t) = 0
162         } else {
163           // select one of the most costly item than can jump (i.e. the one on
                top of the stack)
164           val selected = candidateTojump.top
165           gainCostTab(t) = gainCostTab(optimalSlotTab(selected)) + (t -
                optimalSlotTab(selected)) * h(selected)
166         }
167         val s = optimalItemTab(t).size
168         // add the items placed in t in the candidateTojump
169         k = 0
170         while (k < s) {
171           candidateTojump.push(optimalItemTab(t).pop)
172           k += 1
173         }
174         j += 1
175       }
176       i += 1
177     }
178
179     //assert((0 to domMaxMax).forall(t => optimalItemTab(t).isEmpty))
180
181     //-------------- actual prunning the  X based on gain costs ----------
182
183     k = 0
184     while (k < nU) {
185       i = unBoundIdx(k)
186       val lb = optimalSlotTab(i) - (H.max + gainCostTab(optimalSlotTab(i)) -
            Hmin) / h(i)
187       X(i).updateMin(lb)
188       k += 1
189     }
190   }
191 }
```

Listing C.3: The StackOfStack data structure

```
1  /**
2    * Data structure to represent a stack of stack of integer values.
3    * For instance [[1,4,2],[5,8]] has two stacks.
4    * Its size is 5, the top-post stack has a size of 2.
5    * After one pop the status is [[1,4,2],[5]] and the top most stack has size 1
6    * After another pop the status is [[1,4,2]] and the top most stack has a size
         of 3
7    */
8  class StackOfStackInt(n: Int) {
9
10   if (n < 1) throw new IllegalArgumentException("n␣should␣be␣>␣0")
11
12   private[this] var mainStack: Array[Int] = Array.ofDim(n)
13   private[this] var cumulSizeOfStack: Array[Int] = Array.ofDim(n)
14   private[this] var indexMainStack = 0
15   private[this] var nStack = 0
16
17   def push(i: Int): Unit = {
18     if (indexMainStack == mainStack.length) grow()
19     mainStack(indexMainStack) = i
20     indexMainStack += 1
21   }
22
23   /**
24     * close the current stack (if not empty) and start a new empty one
25     */
26   def pushStack(): Unit = {
27     if (indexMainStack != 0 && cumulSizeOfStack(nStack) != indexMainStack) {
28       nStack += 1
29       cumulSizeOfStack(nStack) = indexMainStack
30     }
31   }
32
33   def isEmpty(): Boolean = {
34     indexMainStack == 0
35   }
36
37   def size(): Int = {
38     indexMainStack
39   }
40
41   /**
42     * Pop the top element of the top stack
43     * @return the value of the top element on the top stack
44     */
45   def pop(): Int = {
46     if (indexMainStack == 0) throw new NoSuchElementException("Stack␣empty")
47     else {
48       if (cumulSizeOfStack(nStack) == indexMainStack) nStack -= 1
49       indexMainStack -= 1
50       mainStack(indexMainStack)
51     }
52   }
```

```
53
54    /**
55      * @return The number of stacks that are stacked
56      */
57    def nStacks(): Int = {
58      nStack
59    }
60
61    /**
62      * @return The size of the top stack
63      */
64    def sizeTopStack(): Int = {
65      if (cumulSizeOfStack(nStack) == indexMainStack) {
66        cumulSizeOfStack(nStack) - cumulSizeOfStack(nStack - 1)
67      } else {
68        indexMainStack - cumulSizeOfStack(nStack)
69      }
70    }
71
72    def reset(): Unit = {
73      indexMainStack = 0
74      nStack = 0
75    }
76
77    private def grow(): Unit = {
78      val newStack = new Array[Int](indexMainStack * 2)
79      System.arraycopy(mainStack, 0, newStack, 0, indexMainStack)
80      mainStack = newStack
81
82      val newCumulSizeOfStack = new Array[Int](indexMainStack * 2)
83      System.arraycopy(cumulSizeOfStack, 0, newCumulSizeOfStack, 0, indexMainStack
           )
84      cumulSizeOfStack = newCumulSizeOfStack
85    }
86  }
```

Listing C.4: The MinArborescence constraint - filtering

```
/**
  * The minArborescence constraint is defined as:
  * minArborescence(preds, w, root, z) in which
  * preds(i) is the predecessor of the vertex i in the arborescence A(G) of the
      graph G found,
  * w is a cost function on edges of G,
  * root is a vertex and
  * z is an upper bound of the cost of arborescence of G rooted at the vertex r.
  * The constraint holds when there exists an arborescence A(G) rooted at the
      vertex r with w(A(G)) <= z.
  *
  * @param preds   , preds(i) is the predecessor of the vertex i in the
      arborescence
  * @param w       , w(i)(j) is the weight of the edge (i,j)
  * @param root    , the root vertex of the arborescence
  * @param z       , the variable z is an upper bound on the cost of the
      arborescence
  * @param withIRC , boolean variable: withIRC = true for filtering based on
  * improved reduced costs ; withIRC = false for filtering based on LP reduced
      costs
  *
  * O(n*n) in which n is the number of vertices
  *
  */

class MinArborescence(val preds: Array[CPIntVar], val w: Array[Array[Int]], val
     root: Int, val z: CPIntVar, val withIRC: Boolean = true) extends Constraint
     (preds(0).store, "Arborescence") {

  val arbo = new MinArborescence(w, root, true)
  val n = preds.length
  val M = z.max + 1
  val costMatrix = Array.tabulate(n, n)((i, j) => new ReversibleInt(s, if (!
      preds(j).hasValue(i)) M else w(i)(j)))

  override def setup(l: CPPropagStrength): Unit = {

    preds(root).assign(root)

    var k = 0
    while (k < n) {
      preds(k).updateMax(n - 1)
      k += 1
    }

    k = 0
    while (k < n) {
      if (!preds(k).isBound) {
        preds(k).callValRemoveIdxWhenValueIsRemoved(this, k)
        preds(k).callPropagateWhenDomainChanges(this)
      }
      k += 1
    }
```

```
46      if (!z.isBound) {
47        z.callPropagateWhenBoundsChange(this)
48      }
49      propagate()
50    }
51
52    override def valRemoveIdx(y: CPIntVar, k2: Int, k1: Int): Unit = {
53      costMatrix(k1)(k2).value = M
54    }
55
56    override def propagate(): Unit = {
57
58      val maxZ = z.max
59      val min = arbo.arbred(costMatrix, maxZ)
60      z.updateMin(min)
61
62      if (withIRC) arbo.computeImprovedRC()
63
64      val gap = maxZ - min
65      var k1 = 0
66      while (k1 < n) {
67        var k2 = 0
68        while (k2 < n) {
69          val gradient = if (withIRC) arbo.improvedRC(k1)(k2)
70          else arbo.rc(k1)(k2)
71
72          if (k2 != root && gradient > gap) {
73            preds(k2).removeValue(k1)
74          }
75          k2 += 1
76        }
77        k1 += 1
78      }
79    }
80  }
```

Listing C.5: Computation of an MWA

```
1  /**
2    * Computation of Minimum Weight Arborescence and LP reduced costs
3    * Based on O(n^2) fortran implementation of
4    * (Fischetti and Toth, 1993: AN EFFICIENT ALGORITHM FOR THE MIN-SUM
5    * ARBORESCENCE PROBLEM ON COMPLETE DIGRAPHS)
6    *
7    *
8    * @param tab    , tab(i)(j) is the weight of the edge (i,j)
9    * @param root   , is a vertex, the root of the Minimum Weight Arborescence
10   * @param rcflag , true if LP reduced costs are wanted, false otherwise.
11   *
12   */
13 class ArborWithoutBreakable(var tab: Array[Array[Int]], var root: Int, val
       rcflag: Boolean) {
14
15   val verbose = false
16
17   root += 1
18   val n = tab(0).length
19   var k1, k2 = 1
20   val c = Array.ofDim[Int](n + 1, n + 1)
21   val rc = Array.ofDim[Int](n + 1, n + 1)
22
23   val mm = 2 * n
24   val pred: Array[Int] = Array.ofDim(mm + 1)
25   val stack: Array[Int] = Array.ofDim(mm + 1)
26   val sv: Array[Int] = Array.ofDim(n + 1)
27   val shadow: Array[Int] = Array.ofDim(n + 1)
28   val lnext: Array[Int] = Array.ofDim(mm + 1)
29   val lpred: Array[Int] = Array.ofDim(mm + 1)
30   val arsel1: Array[Int] = Array.ofDim(mm + 1)
31   val arsel2: Array[Int] = Array.ofDim(mm + 1)
32   val parent: Array[Int] = Array.ofDim(mm + 1)
33   val u: Array[Int] = Array.ofDim(mm + 1)
34   val label: Array[Int] = Array.ofDim(mm + 1)
35   val line: Array[Int] = Array.ofDim(mm + 1)
36   val fson: Array[Int] = Array.ofDim(mm + 1)
37   val broth: Array[Int] = Array.ofDim(mm + 1)
38   val left: Array[Int] = Array.ofDim(mm + 1)
39   val right: Array[Int] = Array.ofDim(mm + 1)
40   val node: Array[Int] = Array.ofDim(mm + 1)
41   val delet = Array.fill(mm + 1)(false)
42   val reduc: Array[Int] = Array.ofDim(mm + 1)
43   val bestTwoDiff: Array[Int] = Array.ofDim(n + 1)
44   var cost, stage, v, v1, v2, f1, f2, sd1, lst, pr, nx, p, h, h1, r, delta, r2,
         min, i, j, len = 0
45   var np1, larsel1, larsel2, shdw, w, ll, linem, la, lb, l, minrow, mincol, l1,
         l2, rcst, lmin, sd2, sd = 0
46   var nLeft, m = n
47   var done = false
48   val inf = Int.MaxValue
49   val unvis = 0
50
```

```
  var k = 0
  var mainLoop = false

  def arbred(oC: Array[Array[ReversibleInt]], max: Int): Int = {
    //Intitialization
    var k1 = 1
    while (k1 <= n) {
      k2 = 1
      while (k2 <= n) {
        c(k1)(k2) = oC(k1 - 1)(k2 - 1).value
        rc(k1)(k2) = c(k1)(k2)
        k2 += 1
      }
      k1 += 1
    }
    k = 1
    while (k <= n) {
      label(k) = unvis
      parent(k) = mm
      rc(k)(k) = inf
      k += 1
    }
    init()

    nLeft = n
    m = n
    cost = 0
    len = 0
    stage = n
    label(root) = root
    insert(root)

    var loop1 = true
    do {
      var loop2 = true
      do {
        if (!mainLoop) {
          //20
          newst()
          if (done) {
            loop2 = false
          }
        }
        if (loop2) {
          var loop3 = true
          while (loop3) {
            //30
            if (!mainLoop) {
              v = stack(len)
              minarc()
              if (min > max) return inf
            }
            mainLoop = false
            //40
```

```
105             label(v) = stage
106             arsel1(v) = i
107             arsel2(v) = j
108             u(v) = min
109             cost = cost + min
110
111             if (label(v1) != unvis) {
112               loop3 = false
113             }
114             if (loop3) {
115               insert(v1)
116             }
117           }
118         }
119       } while (loop2 && label(v1) != unvis && label(v1) != stage)
120       if (done) {
121         loop1 = false
122       }
123       if (loop1) shrink()
124       if (nLeft == 1) {
125         loop1 = false
126       }
127       if (loop1) {
128         v = m
129         mainLoop = true
130       }
131     } while (loop1 && !done)
132
133     //60
134     arcarb()
135
136     if (rcflag) {
137       reduce()
138     }
139     k1 = 1
140     while (k1 <= n) {
141       k2 = 1
142       while (k2 <= n) {
143         rc(k1 - 1)(k2 - 1) = rc(k1)(k2)
144         improvedRC(k1-1)(k2-1) = rc(k1)(k2)
145         k2 += 1
146       }
147       k1 += 1
148     }
149     return cost
150   }
151
152   def arcarb() {
153 // Select the arcs of the optimal arborescence rooted at root
154     k = 1
155     while (k <= m) {
156       delet(k) = false
157       k += 1
158     }
```

```
    pred(root) = 0
    delet(root) = true
    k = m
    do {
      if (!delet(k)) {
        j = arsel2(k)
        pred(j) = arsel1(k)
        while (j != k) {
          delet(j) = true
          j = parent(j)
        }
      }
      k -= 1
    } while (k > 0)
  }

  def init() {
// Initialize the data structure defining the current reduced cost matrix
    k = 1
    while (k <= n) {
      line(k) = k
      sv(k) = k
      shadow(k) = 0
      lnext(k) = k + 1
      lpred(k + 1) = k
      k = k + 1
    }
    lnext(n + 1) = 1
    lpred(1) = n + 1
  }

  def insert(v: Int) {
//Insert the vertex v into the stack
    len = len + 1
    stack(len) = v
    np1 = n + 1
    lst = lpred(np1)
    if (v == lst) {
      return
    }
    pr = lpred(v)
    nx = lnext(v)
    lnext(pr) = nx
    lpred(nx) = pr
    lnext(lst) = v
    lpred(v) = lst
    lnext(v) = np1
    lpred(np1) = v
  }

  def minarc() {
//Find the minimum reduced-cost arc (v1,v2) entering vertex v2
    j = v
    l2 = line(v)
```

```
213       np1 = n + 1
214       l1 = lnext(np1)
215       min = rc(l1)(l2)
216       l = lnext(l1)
217       do {
218         if (rc(l)(l2) < min) {
219           min = rc(l)(l2)
220           l1 = l
221         }
222         l = lnext(l)
223       } while (l != np1)
224       v1 = sv(l1)
225       sd1 = shadow(l1)
226       i = l1
227       if (sd1 > 0) {
228         i = rc(sd1)(l2)
229       }
230     }
231
232     def newst() {
233   //Select an unlabelled vertex, stage, and initialize a new stage
234       done = true
235       while (label(stage) > 0) {
236         stage = stage - 1
237         if (stage <= 0) {
238           return
239         }
240       }
241
242       done = false
243       if (len != 0) {
244         v1 = stack(1)
245         l1 = line(v1)
246         v2 = stack(len)
247         l2 = line(v2)
248         np1 = n + 1
249         f1 = lnext(np1)
250         f2 = lpred(l1)
251         lnext(f2) = np1
252         lpred(np1) = f2
253         lnext(np1) = l1
254         lpred(l1) = np1
255         lnext(l2) = f1
256         lpred(f1) = l2
257       }
258       len = 1
259       stack(len) = stage
260     }
261
262     def reduce() {
263   //Compute the reduced-cost matrix rc
264       np1 = n + 1
265       v = np1
266       while (v <= m) {
```

```
    fson(v) = 0
    v += 1
  }
  fson(mm) = 0
  v = 1
  while (v <= m) {
    p = parent(v)
    if (fson(p) == 0) {
      fson(p) = v
      broth(v) = 0
    } else {
      broth(v) = fson(p)
      fson(p) = v
    }
    v += 1
  }
  broth(mm) = mm

  h = 0
  h1 = h + 1
  v = mm
  do {
    while (v > n) {
      left(v) = h1
      v = fson(v)
    }
    h = h1
    h1 += 1
    node(h) = v
    while (broth(v) == 0) {
      v = parent(v)
      right(v) = h
    }
    v = broth(v)
  } while (v != mm)
  u(root) = 0

  h = 1
  while (h <= n) {
    j = node(h)
    if (j == root) {
      i = 1
      while (i <= n) {
        rc(i)(j) = c(i)(j)
        i += 1
      }
    } else {
      rc(j)(j) = c(j)(j) //90
      l = h - 1
      r = h + 1
      delta = u(j)
      p = parent(j)
      var loop = true
      while (loop) {
```

```
321           l1 = left(p) //100
322           while (l >= l1) {
323             i = node(l)
324             rc(i)(j) = c(i)(j) - delta
325             l -= 1
326           }
327           r2 = right(p) //120
328           while (r <= r2) {
329             i = node(r)
330             rc(i)(j) = c(i)(j) - delta
331             r += 1
332           }
333           if (p == mm) {
334             loop = false
335           }
336           if (loop) {
337             delta = delta + u(p)
338             p = parent(p)
339           }
340         }
341       }
342       h += 1
343     }
344   }
345
346   def shrink() {
347 // Shrink the (super-) vertices stored in stack from vertex v downto vertex v1
348     if (len == nLeft && stack(0) == v1) {
349       nLeft = 1
350       return
351     }
352     m = m + 1
353     parent(m) = mm
354     v = stack(len)
355     larsel2 = line(v)
356     do {
357       w = stack(len)
358       len -= 1
359       parent(w) = m
360       nLeft -= 1
361       ll = line(w)
362       reduc(ll) = u(w)
363
364     } while (w != v1)
365     larsel1 = line(v1)
366
367     linem = larsel1
368     shdw = larsel2
369     np1 = n + 1
370     la = lnext(np1)
371     lb = lpred(larsel1)
372     min = inf
373     l = la
374
```

```
375     var loop = true
376     while (loop) {
377       //30
378       ll = larsel1
379       minrow = rc(l)(ll) - reduc(ll)
380       l1 = ll
381       mincol = rc(ll)(l)
382       l2 = ll
383       //40
384       do {
385         ll = lnext(ll)
386         rcst = rc(l)(ll) - reduc(ll)
387         if (rcst < minrow) {
388           minrow = rcst
389           l1 = ll
390         }
391         if (rc(ll)(l) < mincol) {
392           mincol = rc(ll)(l)
393           l2 = ll
394         }
395       } while (ll != larsel2)
396       //61
397       if (minrow < min) {
398         min = minrow
399         lmin = l
400       }
401       //70
402       rc(l)(linem) = minrow
403       rc(l)(shdw) = l1
404       sd1 = shadow(l1)
405       if (sd1 > 0) {
406         rc(l)(shdw) = rc(l)(sd1)
407       }
408       rc(linem)(l) = mincol
409       rc(shdw)(l) = l2
410       sd2 = shadow(l2)
411       if (sd2 > 0) {
412         rc(shdw)(l) = rc(sd2)(l)
413       }
414       sd = shadow(l)
415       if (sd > 0) {
416         rc(sd)(linem) = rc(sd)(l1)
417         rc(linem)(sd) = rc(l2)(sd)
418       }
419       //80
420       if (l == lb) {
421         loop = false
422       }
423       if (loop) {
424         l = lnext(l)
425       }
426     }
427     line(m) = linem
428     sv(linem) = m
```

```
    shadow(linem) = shdw
    rc(linem)(linem) = inf
    lnext(linem) = np1
    lpred(np1) = linem
    len += 1
    stack(len) = m
    nLeft += 1
    v1 = sv(lmin)
    j = rc(lmin)(shdw)
    sd = shadow(lmin)
    i = lmin
    if (sd > 0) {
      i = rc(sd)(linem)
    }
  }

  /*
   * Our improved rc
   */
  val improvedRC = Array.fill(n + 1, n + 1)(0)
  val bestDiffFromSrcUntilDest = Array.fill(n + 1, n + 1)(inf)
  val hasAtmostOneParent = Array.fill(n + 1)(false)

  def computeImprovedRC() {
    k1 = 1
    while (k1 <= n) {
      val pk1 = parent(k1)
      if (pk1 == mm || parent(pk1) == mm) hasAtmostOneParent(k1) = true
      k1 += 1
    }

    computeBestTwoDiff()

    // BestDiffBetween for v with at most one parent
    k1 = 1
    while (k1 <= n) {
      val pk1 = parent(k1)
      if (hasAtmostOneParent(k1)) {
        min = bestTwoDiff(k1) - u(pk1)
        k2 = pred(k1)
        while (min > 0 && hasAtmostOneParent(k2) && k2 != root) {
          bestDiffFromSrcUntilDest(k1)(k2) = min
          val currentBestDiff = bestTwoDiff(k2) - u(parent(k2))
          if (currentBestDiff < min) {
            min = currentBestDiff
          }
          k2 = pred(k2)
        }
      }
      k1 += 1
    }
    // ImprovedRC
    k1 = 0
    while (k1 < n) {
```

```
483       if (hasAtmostOneParent(k1+1)) {
484         k2 = 0
485         while (k2 < n) {
486           val additionalCost = if (bestDiffFromSrcUntilDest(k1 + 1)(k2 + 1) ==
                  inf) 0 else bestDiffFromSrcUntilDest(k1 + 1)(k2 + 1)
487           improvedRC(k1)(k2) += additionalCost
488           bestDiffFromSrcUntilDest(k1 + 1)(k2 + 1) = inf
489           k2 += 1
490         }
491       }
492       hasAtmostOneParent(k1 + 1) = false
493       k1 += 1
494     }
495   }
496
497   def computeBestTwoDiff(): Unit = {
498     // bestTwoDiff for each v in V
499     k1 = 1
500     while (k1 <= n) {
501       if (hasAtmostOneParent(k1)) {
502         var min1 = c(1)(k1)
503         var min2 = inf
504         k2 = 2
505         while (k2 <= n) {
506           val cK2K1 = c(k2)(k1)
507           if (cK2K1 < min1) {
508             val tmp = min1
509             min1 = cK2K1
510             min2 = tmp
511           } else if (cK2K1 < min2) {
512             min2 = cK2K1
513           }
514           k2 += 1
515           bestTwoDiff(k1) = min2 - min1
516         }
517       }
518       k1 += 1
519     }
520   }
521 }
```

BIBLIOGRAPHY

[Ach+92] N. R. Achuthan, L. Caccetta, P. Caccetta, and J. F. Geelen. "Algorithms for the minimum weight spanning tree with bounded diameter problem." In: *Optimization: techniques and applications* 1.2 (1992), pp. 297–304.

[AT10] İ. Akgün and B. Ç. Tansel. "Min-degree constrained minimum spanning tree problem: New formulation via Miller–Tucker–Zemlin constraints." In: *Computers & Operations Research* 37.1 (2010), pp. 72–82.

[Apt03] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[AFT99] V. A. Armentano, P. M. Franca, and F. M. B. de Toledo. "A network flow model for the capacitated lot-sizing problem." In: *Omega*. 2nd ser. 27 (1999), pp. 275–384.

[BRG87] H. C. Bahl, L. P. Ritzman, and J. N. D. Guupta. "Determining lot sizes and resource requirements: a review." In: *Operations Research*. 3rd ser. 35 (1987), pp. 329–345.

[BLPN12] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Vol. 39. Springer Science & Business Media, 2012.

[BRW84] I. Barany, T. J. V. Roy, and L. A. Wolsey. "Strong formulations for Multi-Item Capacitated Lot Sizing." In: *Management Science*. 10th ser. 30 (1984), pp. 1255–61.

[BP00] J. C. Beck and L. Perron. "Discrepancy-Bounded Depth First Search." In: *Second International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'2000*. 2000, pp. 133–147.

[Bee06] P. van Beek. "Handbook of Constraint Programming." In: Elseiver, 2006. Chap. 4: Backtracking Search Algorithms, pp. 85–134.

[BFL05] N. Beldiceanu, P. Flener, and X. Lorca. "The tree constraint." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2005, pp. 64–78.

[BMR12] N. Beldiceanu, C. Mats, and J.-X. Rampon. *Global Constraint Catalog*. Available from `http://sofdem.github.io/gccat/gccat/`. 2012.

[BW01] G. Belvaux and L. A. Wolsey. "Modelling practical lot-sizing problems as mixed integer programs." In: *Management Science* 47 (2001), pp. 724–738.

[Ben+12] P. Benchimol, W.-J. van Hoeve, J.-C. Régin, L.-M. Rousseau, and M. Rueher. "Improved Filtering for Weighted Circuit Constraints." In: *Constraints* 17.3 (2012), pp. 205–233.

[Ben96] F. Benhamou. "Heterogeneous Constraint Solving." In: *Fifth International Conference on Algebraic and Logic Programming (ALP96)*. 1996, pp. 62–76.

[Bes06] C. Bessiere. "Handbook of Constraint Programming." In: Elsevier, 2006. Chap. 3: Constraint Propagation, pp. 29–83.

[Bes+11] C. Bessiere, N. Narodytska, C.-G. Quimper, and T. Walsh. "The alldifferent constraint with precedences." In: *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 2011, pp. 36–52.

[BY82] G. R. Bitran and H. H. Yanasse. "Computational complexity of the capacitated lot size problem." In: *Management Science* 28 (1982), pp. 1174–1186.

[Boc71] F. Bock. "An algorithm to construct a minimum directed spanning tree in a directed network." In: *Developments in Operations Research* (1971), pp. 29–44.

[Bru00] W. Bruggeman. "The Discrete Lot Sizing and Scheduling Problem: complexity and modification for batch availability." In: *European Journal of Operational Research*. 3rd ser. 124 (2000), pp. 511–528.

[CMT88] G. Carpaneto, S. Martello, and P. Toth. "Algorithms and codes for the assignment problem." In: *Annals of operations research* 13.1 (1988), pp. 191–223.

[CL95] Y. Caseau and F. Laburthe. "Improving branch and bound for Jobshop scheduling with constraint propagation." In: *Combinatorics and Computer Science* 1120 (1995), pp. 129–149.

[CL97a] Y. Caseau and F. Laburthe. "Solving Various Weighted Matching Problems with Constraints." In: *Third International Conference on Principles and Practice of Constraint Programming (CP'97)*. 1997, pp. 17–31.

[CL97b] Y. Caseau and F. Laburthe. "Solving small TSPs with constraints." In: *14th international conference on Logic Programming (ICLP)*. 1997, pp. 316–330.

[Cps] *Catalog of Constraint Programming Solvers*. Available from http://openjvm.jvmhost.net/CPSolvers/.

[CDGS16] S. Ceschia, L. Di Gaspero, and A. Schaerf. *Optimization hub: Lot Sizing Problem*. Available from http://opthub.uniud.it. 2016.

[CL73] K. M. Chandy and T. Lo. "The capacitated minimum spanning tree." In: *Networks* 3.2 (1973), pp. 173–181.

[CT90] W. H. Chen and J. M. Thizy. "Analysis of relaxations for the multi-item capacitated lot-sizing problem." In: *Annals of Operations Research*. 1st ser. 26 (1990), pp. 29–72.

[CL65] Y. J. Chu and T. H. Liu. "On the shortest arborescence of a directed graph." In: *Sci. Sin., Ser. A* 14 (1965), pp. 1396–1400.

[CFL94] C. Chung, J. Flynn, and C. M. Lin. "An effective algorithm for the capacitated single item lot size problem." In: *European Journal of Operational Research*. 2nd ser. 75 (1994), pp. 427–440.

[Cop+16] K. Copil, M. Worbelauer, M. Meyr, and H. Tempelmeier. "Simultaneous lot sizing and scheduling problems: a classification and review of models." In: *OR Spectrum* (2016).

[Cor+01a] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to Algorithms (Third ed.)" In: The MIT press, 2001. Chap. 25: All-Pairs Shortest Paths.

[Cor+01b] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to Algorithms (Third ed.)" In: The MIT press, 2001. Chap. 21: Data structures for Disjoint Sets.

[DM97] M. Dell'Amico and S. Martello. "Linear assignment." In: *Annotated Bibliographies in Combinatorial Optimization* (1997), pp. 355–371.

[DH80] R. S. Demboand and P. L. Hammer. "A reduction algorithm for knapsack problems." In: *Methods of Operations Research* 36 (1980), pp. 49–60.

[DK97a] N. Deo and N. Kumar. "Computation of constrained spanning trees: A unified approach." In: *Network Optimization*. Springer, 1997, pp. 196–220.

[Dia+03] M. Diaby, H. C. Bahl, M. H. Karwan, and S. A. Zionts. "Lagrangean relaxation approach for very-large-scale capacitated lot-sizing." In: *Management Science*. 9th ser. 38 (2003), pp. 1329–1340.

[Din+88] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. "The constraint logic programming language CHIP." In: *International Conference on Fifth Generation Computer System*. 1988, pp. 693–702.

[DSH90] M. Dincbas, H. Simonis, and P. V. Hentenryck. "Solving large combinatorial problems in logic programming." In: *Journal of Logic Programming*. 1-2 8 (1990), pp. 75–93.

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

[DK07] G. Dooms and I. Katriel. "The not-too-heavy spanning tree constraint." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR 2007*. Springer. 2007, pp. 59–70.

[DK97b] A. Drexl and A. Kimms. "Lot sizing and scheduling, Survey and extensions." In: *European Journal of Operational Research* (1997), pp. 221–235.

[DH95] A. Drexl and K. Haase. "Proportional lot sizing and scheduling." In: *International Journal of Production Economics* 40.1 (1995), pp. 73–87.

[DK97c] A. Drexl and A. Kimms. "Lot sizing and scheduling: survey and extensions." In: *European Journal of Operational Research* 99.2 (1997), pp. 221–235.

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

[Edm67] J. Edmonds. "Optimum Branchings." In: *Journal of Research of the National Bureau for Standards* 71B(4) (1967), pp. 125–130.

[FS02] T. Fahle and M. Sellmann. "Cost based filtering for the constrained knapsack problem." In: *Annals of Operations Research* 115.1-4 (2002), pp. 73–93.

[FT92] M. Fischetti and P. Toth. "An additive bounding procedure for asymmetric travelling salesman problem." In: *Mathematical Programming* 53 (1992), pp. 173 –197.

[FT93] M. Fischetti and P. Toth. "An efficient algorithm for min-sum arborescence problem on complete digraphs." In: *Management Science* 9.3 (1993), pp. 1520–1536.

[FV97] M. Fischetti and D. Vigo. "A branch-and-cut algorithm for the resource-constrained minimum-weight arborescence problem." In: *Network*. 3rd ser. 29 (1997), pp. 55–67.

[Fle94] B. Fleischmann. "The Discrete Lot Sizing and Scheduling Problem with sequence dependent setup costs." In: *European Journal of Operational Research* 75 (1994), pp. 395–404.

[FLK80] M. Florain, J. K. Lenstra, and A. R. Kan. "Deterministic Production Planning: Algorithms and Complexity." In: *Management Science*. 7th ser. 26 (1980), pp. 669–679.

[FLM99a] F. Focacci, A. Lodi, and M. Milano. "Integration of CP and OR methods for matching problems." In: *First International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'99)*. 1999.

[FLM99b] F. Focacci, A. Lodi, and M. Milano. "Cost-based domain filtering." In: *Principles and Practice of Constraint Programming–CP'99*. Springer. 1999, pp. 189–203.

[FLM02] F. Focacci, A. Lodi, and M. Milano. "Embedding relaxations in global constraints for solving TSP and TSPTW." In: *Annals of Mathematics and Artificial Intelligence* 34 (2002), pp. 291–311.

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

[FM06] E. C. Freuder and A. K. Mackworth. "Handbook of Constraint Programming." In: Elseiver, 2006. Chap. 2: Constraint Satisfaction: An Emerging Paradigm, pp. 13–27.

[Gab+86] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs." In: *Combinatorica* 6.3 (1986), pp. 109–122.

[GS97] G. Gallego and D. X. Shaw. "Complexity of the ELSP with general cyclic schedules." In: *IIE transactions* 29.2 (1997), pp. 109–113.

[Gay+15] S. Gay, R. Hartert, C. Lecoutre, and P. Schaus. "Conflict ordering search for scheduling problems." In: *Principles and Practice of Constraint Programming - CP 2015*. Springer. 2015, pp. 144–148.

[Gec05] Gecode Team. *Gecode: Generic constraint development environment*. Available from http://www.gecode.org. 2005.

[Ger+15] G. German, H. Cambazard, J.-P. Gayon, and B. Penz. "Une contrainte globale pour le lot sizing." In: *Journée Francophone de Programmation par Contraintes - JFPC 2015*. 2015, pp. 118–127.

[GH01] S. M. T. F. Ghomi and S. S. Hashemin. "An analytical method for single level-constrained resources production problem with constant set-up cost." In: *Iranian Journal of Science and Technology* 26.B1 (2001), pp. 69–82.

[Gic08] C. Gicquel. "MIP models and exact methods for the Discrete Lot-sizing and Scheduling Problem with sequence-dependent changeover costs and times." PhD thesis. France: Ecole centrale Paris, 2008.

[Goo] Google ortools. Available from https://developers.google.com/optimization/.

[GH85] R. L. Graham and P. Hell. "On the history of the minimum spanning tree problem." In: *History of computing* 7 (1985), pp. 13–25.

[GR90] M. Guignard and M. B. Rosenwein. "An application of lagrangean decomposition to the resource-constrained minimum weighted arborescence problem." In: *Network*. 3rd ser. 20 (1990), pp. 345–359.

[Hal35] P. Hall. "On representatives of subsets." In: *J. London Math. Soc* 10.1 (1935), pp. 26–30.

[Har13] F. W. Harris. "How many parts to make at once." In: *Factory, The magazine of management* 10.2 (1913), pp. 135–136.

[HG95] W. D. Harvey and M. L. Ginsberg. "Limited discrepancy search." In: *14th international joint conference on Artificial intelligence*. Vol. 1. 1995, pp. 607–613.

[Hen89] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[HM05] P. V. Hentenryck and L. Michel. *Constraint-based local search*. The MIT Press, 2005.

[Hin95] K. S. Hindi. "Computationally efficient solution of the multi-item, capacitated lot-sizing problem." In: *Computers and Industrial Engineering*. 4th ser. 28 (1995), pp. 709–719.

[Hoe01] W.-J. van Hoeve. *The alldifferent Constraint: A Survey*. 2001.

[HK06] W.-J. van Hoevve and I. Katriel. "Handbook of Constraint Programming." In: Elsevier, 2006. Chap. 6: Global Constraints, pp. 169–208.

[HS04] H. H. Hoos and T. Stutzle. *Stochastic Local Search: Foundations and Applications*. The Morgan Kaufmann Series, 2004.

[HT06] H. H. Hoos and E. Tsang. "Handbook of Constraint Programming." In: Elseiver, 2006. Chap. 5: Local Search Methods, pp. 135–167.

[Hou13] V. R. Houndji. "Deux Problèmes de Planification de Production: Formulations et Résolution par Programmation en Nombres Entiers et par Programmation par Contraintes." MA thesis. Belgium: Louvain School of Engineering, 2013.

[HSa] V. R. Houndji and P. Schaus. *CP4CAP: Constraint Programming for Constrained Arborescence Problem (CAP). Toolkit using the OscaR solver as a dependency and some instances on a variant of the CAP.* Available from https://bitbucket.org/ratheilesse/cp4cap.

[HSb] V. R. Houndji and P. Schaus. *CP4PP: Constraint Programming for Production Planning. Toolkit using the OscaR solver as a dependency and some instances on a variant of the CLSP.* Available from https://bitbucket.org/ratheilesse/cp4pp.

[HSW14] V. R. Houndji, P. Schaus, and L. Wolsey. "CP Approach for the Multi-Item Capacitated Lot-Sizing Problem with Sequence-Dependent Changeover Costs." In: *Communication - 28th annual conference of the Belgian Operational Research Society - Orbel'28*. 2014, p. 145.

[Hou+] V. R. Houndji, P. Schaus, L. Wolsey, and Y. Deville. *CSPLib Problem 058: Discrete Lot Sizing Problem*. Ed. by C. Jefferson, I. Miguel, B. Hnich, T. Walsh, and I. P. Gent. Available from http://www.csplib.org/Problems/prob058.

[Hou+14] V. R. Houndji, P. Schaus, L. Wolsey, and Y. Deville. "The StockingCost Constraint." In: *Principles and Practice of Constraint Programming–CP'2014*. Springer. 2014, pp. 382–397.

[Hou+15] V. R. Houndji, P. Schaus, L. Wolsey, and Y. Deville. "La contrainte globale StockingCost pour les problèmes de planification de production." In: *Communication - Journées Francophones de Programmation par Contraintes, JFPC'2015*. 2015, pp. 128–129.

[Hou+17a] V. R. Houndji, P. Schaus, M. N. Hounkonnou, and L. Wolsey. "La contrainte globale MinArborescence pour les problèmes d'arborescence de poids minimum." In: *Communication - Journées Francophones de Programmation par Contraintes, JFPC'2017*. 2017.

[Hou+17b] V. R. Houndji, P. Schaus, M. N. Hounkonnou, and L. Wolsey. "The Weighted Arborescence Constraint." In: *Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'2017*. 2017.

[J. 94] J. F. Puget. *A C++ implementation of CLP*. Tech. rep. 1994.

[JD06] R. Jans and Z. Degraeve. "Modeling Industrial Lot Sizing Problems: A Review." In: *International Journal of Production Research* (2006).

[JD98] C. Jordan and A. Drexl. "Discrete Lot Sizing and Scheduling by batch sequencing." In: *Management Science*. 5th ser. 44 (1998), pp. 698–713.

[KGW03] B. Karimi, S. M. T. F. Ghomi, and J. Wilson. "The capacitated lot sizing problem: a review of models." In: *Omega, The international Journal of Management Science* (2003), pp. 365–378.

[KR82] R. Karni and Y. Roll. "A Heuristic Algorithm for the Multi-Item Lot-Sizing Problem with Capacity Constraints." In: *AIIE Transactions*. 4th ser. 14 (1982), pp. 249–56.

[KT05a] I. Katriel and S. Thiel. "Complete bound consistency for the global cardinality constraint." In: *Constraints* 10.3 (2005), pp. 191–217.

[KT05b] J. Kleinberg and E. Tardos. "Algorithm Design." In: Tsinghua University Press, 2005. Chap. 4.9: Minimum-Cost Arborescences: A multi-phase greedy algorithm.

[Kor85] R. E. Korf. "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search." In: *Artificial Intelligence* 27 (1985), pp. 97–109.

[Kuh10] H. W. Kuhn. "The hungarian method for the assignment problem." In: *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 29–47.

[LMV89] J. M. Y. Leung, T. L. Magnanti, and R. Vachani. "Facets and algorithms for capacitated lot sizing." In: *Mathematical Programming* 45 (1989), pp. 331–359.

[LO+03] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. "A fast and simple algorithm for bounds consistency of the alldifferent constraint." In: *International Joint Conference on Artificial Intelligence – IJCAI'03*. 2003, pp. 245–250.

[Lor10] X. Lorca. "Contraintes de Partitionnement de Graphe." PhD thesis. Université de Nantes, 2010.

[LJG11] X. Lorca and F. Jean-Guillaume. "Revisiting the tree Constraint." In: *Principles and Practice of Constraint Programming - CP 2011*. Vol. 6876. 2011, pp. 271–285.

[Mac77] A. K. Mackworth. "Consistency in networks of relation." In: *Artificial Intelligence*. 1st ser. 8 (1977), pp. 99–118.

[Man58] A. S. Manne. "Programming of Economic Lot Sizes." In: *Management Science*. 2nd ser. 4 (1958), pp. 115–135.

[MS98] K. Marriott and P. Stuckey. *Programming with constraints*. The MIT press, 1998.

[MT00] K. Mehlhorn and S. Thiel. "Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2000, pp. 306–319.

[Men+04] R. Mendelson, R. Tarjan, M. Thorup, and U. Zwick. "Melding Priority Queues." In: *Proceedings of SWAT 04* 3111.3 (2004), pp. 223–235.

[NH80] S. C. Narula and C. A. Ho. "Degree-constrained minimum spanning tree." In: *Computers & Operations Research* 7.4 (1980), pp. 239–249.

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

[PV84] C. H. Papadimitriou and U. V. Vazirani. "On two geometric problems related to the travelling salesman problem." In: *Journal of Algorithms* 5.2 (1984), pp. 231–246.

[Pes16] G. Pesant. "Counting-Based Search for Constraint Optimization Problems." In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016, pp. 3441–3447.

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

[PW05] Y. Pochet and L. Wolsey. *Production Planning by Mixed Integer Programming*. Springer, 2005.

[PFL15] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. Available from http://www.choco-solver.org. 2015.

[Pug98] J.-F. Puget. "A fast algorithm for the bound consistency of alldiff constraints." In: *Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*. 1998, pp. 359–366.

[Qui+03] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. "An efficient bounds consistency algorithm for the global cardinality constraint." In: *Principles and Practice of Constraint Programming–CP'2003*. Springer. 2003, pp. 600–614.

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

[Rég96] J.-C. Régin. "Generalized arc consistency for global cardinality constraint." In: *Proceedings of the thirteenth national conference on Artificial intelligence*. Vol. 1. AAAI Press. 1996, pp. 209–215.

[R99] J.-C. Régin. "The symmetric alldiff constraint." In: *16th International Joint Conference on Artificial Intelligence (IJCAI'99)*. 1999, pp. 420–425.

[R02] J.-C. Régin. "Cost-based arc consistency for global cardinality constraints." In: *Constraints* 7 (2002), pp. 387–405.

[Rég08] J.-C. Régin. "Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2008, pp. 233–247.

[Rí11] J.-C. Régin. "Hybrid Optimization." In: P. Van Hentenryck and M. Milano editors, 2011. Chap. Global constraint: a survey, pp. 63–134.

[Rég+10] J.-C. Régin, L.-M. Rousseau, M. Rueher, and W.-J. van Hoeve. "The weighted spanning tree constraint revisited." In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2010, pp. 287–291.

[RWH99] R. Rodosek, M. G. Wallace, and M. T. Hajian. "An exact constraint logic programming algorithm for the traveling salesman problem with time windows." In: *Annals of Operations Research* 86 (1999), pp. 63–87.

[Ros86] M. B. Rosenwein. "Design and application of solution methodologies to optimize problems in transportation logistics." PhD thesis. University of Pennsylvania, 1986.

[RBW06] F. Rossi, P. V. Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[Sca] Scala. *Scala programming language*. Available from `http://www.scala-lang.org`.

[Sch15] P. Schaus. *CP for the Impatient*. Available from `https://www.info.ucl.ac.be/pschaus/cp4impatient/`. 2015.

[Sel03] M. Sellmann. "Approximated consistency for knapsack constraints." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2003, pp. 679–693.

[Sha98] P. Shaw. "Using constraint programming and local search methods to solve vehicle routing problems." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1998, pp. 417–431.

[SC06] C. Shulte and M. Carlsson. "Handbook of Constraint Programming." In: Elseiver, 2006. Chap. 14: Finite Domain Constraint Programming Systems, pp. 495–526.

[Tar77] R. E. Tarjan. "Finding optimum branchings." In: *Networks* 7.3 (1977), pp. 25–35.

[Tri03] M. A. Trick. "A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints." In: *Annals of Operations Research* 118 (2003), pp. 73–60.

[Tri87] W. W. Trigerio. "A Dual-Cost Heuristic For The Capacitated Lot Sizing Problem." In: *IIE Transactions*. 1st ser. 19 (1987), pp. 67–72.

[Tsa95] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1995.

[UP10] H. Ullah and S. Parveen. "A Literature Review on Inventory Lot Sizing Problems." In: *Global Journal of Researches in Engineering* 10 (2010), pp. 21–36.

[Una+16] D. de Una, G. Gange, P. Schachte, and P. J. Stuckey. "Weighted Spanning Tree Constraint with Explanations." In: *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 2016, pp. 98–107.

[VCLS15] S. Van Cauwelaert, M. Lombardi, and P. Schaus. "Understanding the potential of propagators." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR 2015*. Springer. 2015, pp. 427–436.

[WW58] H. M. Wagner and T. M. Within. "Dynamic version of the economic lot size model." In: *Management Science* 50 (1958), pp. 89–96.

[Wal97] T. Walsh. "Depth-Bounded Discrepancy Search." In: *Fifteenth International Joint Conference on Artificial Intelligence, IJCAI'97*. 1997, pp. 1388–1393.

[Way13] K. Wayne. *Greedy Algorithms*. Available from `http://www.cs.princeton.edu/~wayne/kleinberg-tardos`. 2013.

[Wol98] L. A. Wolsey. *Integer programming*. Vol. 42. Wiley-Interscience, 1998.

[Wol02] L. A. Wolsey. "Solving multi-item lot-sizing problems with an MIP solver using classification and reformulation." In: *Management Science* 48.12 (2002), pp. 1587–1602.

[WN99] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.