Linear constraints and conflict-free learning for graphical models

Pierre Montalbano

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Pierre MONTALBANO

Le 15 décembre 2023

Contraintes linéaires et apprentissage sans conflit pour les modèles graphiques.

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
MIAT - Mathématiques et Informatique Appliquées Toulouse

Thèse dirigée par
Simon DE GIVRY et George Katsirelos

Jury

M. Jakob NORDSTRÖM, Rapporteur
M. Daniel LE BERRE, Rapporteur
Mme Charlotte TRUCHET, Examinatrice
M. Pierre SCHAUS, Examinateur
M. Martin COOPER, Examinateur
M. Simon DE GIVRY, Directeur de thèse
M. George KATSIRELOS, Co-directeur de thèse

To my father.
"Legends never die, they become a part of you"

Funding

This research was funded by the French “Agence Nationale de la Recherche” through grants EUR Bioeco ANR-18-EURE-0021 and ANITI ANR-19-P3IA-0004.

Remerciements

Je remercie chaleureusement toute l'équipe MIAT, ce laboratoire fut pour moi un lieu propice à mon épanouissement scientifique et personnel. Toute ma reconnaissance va à Simon et George qui m'ont accompagné pendant mes débuts sur le chemin de la recherche. J'ai beaucoup appris en travaillant avec eux, et j'espère que nos échanges leur ont également été utiles.

Je tiens également à remercier Martin Cooper, Daniel le Berre, Jakob Nordström, Pierre Schaus et Charlotte Truchet pour leur révision approfondie de ma thèse. Un grand merci à l'équipe ANITI, Thomas, Simon, George, Valentin, Samuel, Marianne, Sophie et les 2 David pour nos discussions très inspirantes et motivantes. Ma culture et mon ouverture scientifique ne seraient pas les mêmes sans les séminaires du vendredi organisés par Céline, Sandra et Meritxell, un grand merci à elles. Merci à Tomas Werner de m'avoir accueilli dans son laboratoire à Prague, cette collaboration fut très instructive.

Merci aux permanents, Sylvain, Nathalie, Élise, Nathalie, Raphaël, Philippe, Régis, Benjamin, Matthias, Claire, pour ne citer qu'eux, d'avoir partagé avec bienveillance leurs expériences et leurs avis sur le monde de la recherche.

Je remercie mes co-bureaux Khaoula, Aurélie et Fulya, pour leur bonne humeur et leurs conseils salvateurs !

Merci à Marianne, Samuel, Raphaël, Joséphine, Alexis, Julien, Julien, Prassana et toutes les personnes avec qui j'ai pu partager un repas, une partie de Tarot ou une pièce de puzzle dans la salle café. Sans ces petites pauses, cette thèse aurait été bien moins amusante ! Une pensée particulière à Hanna, sans qui ma tasse serait cassée et mon manuscrit non relié.

Merci à Valentin, ce fut un réel plaisir de partager avec toi ces trois ans de thèse. Tu m'as motivé à la fois sur le plan professionnel et sur le plan personnel, j'espère qu'on pourra de nouveau faire des séances de boxe/muscu!

Je ne pourrai jamais exprimer la gratitude que je ressens envers ma mère, mon père, mon frère, Marvin, Quentin, Alexis et Yann. Je suis tellement heureux d'avoir pu grandir et évoluer avec vous!

Enfin, un grand merci à Camille pour son amour et son soutien indéfectible depuis 4 ans.

Contents

Chapter 1

Introduction

Optimization plays a fundamental role in our everyday experience. We consistently face new challenges, going from vehicle routing, scheduling, recommender systems, or protein design to only cite them. Tackling those problems is difficult because they can involve a large number of variables with complex interactions. Most of them are out of reach of the human cognitive capacity, therefore, we rely on endlessly improving algorithms to find the best, or at least a good solution.

There exist different modeling and solving approaches, each of them shaped to solve a different range of problems. Among the two most common paradigms, we find on one hand *Contraint Programming*(CP), a framework based on logic and inference. On the other hand, Integer Linear Programming (ILP), specialized in linear interaction and employing advanced mathematics. Both ILP and CP offer complementary approaches to tackle complex optimization challenges. However, it is frequent that when a technique shows its efficiency in one paradigm, then researchers try to adapt it to the other paradigm. A notable example is Conflict Driven Clause Learning (CDCL). It was initially defined for Boolean satisfiability problem (SAT), for which it became a cornerstone of modern SAT solvers. Since then, derived approaches have been developed for MaxSAT, Pseudo Boolean optimization, Constraint Satisfaction Problem, or ILP. This strategy aims to learn constraints from the failure of the solver. The learned constraints will enhance the rest of the search by preventing the solver from making the same mistake again.

Graphical Models (GM) use graphs to encode complex relationships between decision variables, where nodes represent variables and (hyper)edges represent dependencies or correlations between them. GM provides a flexible framework to model different systems. For example, Cost Function Networks are undirected graphical models involving *local cost functions*. The task of finding the assignment minimizing the sum of all the local cost functions is known as the Weighted Constraint Satisfaction Problem (WCSP). This problem arises in various areas such as image analysis [Savchynskyy 2019] or bioinformatics [Allouche *et al.* 2014a]. The resolution of WCSP relies on *backtracking search* and *constraint propagation*. The solver searches an optimal solution by dividing the search space into smaller sub-problems. Constraint propagation techniques are used to provide a lower bound of the encountered sub-problems. Deriving strong lower bounds is crucial to avoid exploring unpromising regions.

In this thesis, we are interested in diversifying the range of instances modelable and solvable by a WCSP solver. We first show how to integrate linear constraints in a

Cost Function Network (CFN). They are expressive and compact constraints and are at the center of very efficient ILP solvers. Thus, handling linear constraints in WCSP solvers can significantly broaden their practical use. Secondly, we define Virtual Pair-Wise Consistency, a new soft local consistency enforcing strong bounds. Finally, guided by the success of conflict-based learning methods in multiple domains (such as SAT, Pseudo Boolean Optimization, or ILP), we design a new *conflict-free* learning mechanism. It aims to memorize through a linear constraint the lower bounds of the encountered sub-problems. If this sub-problem appears a second time in the search then propagating the previously learned constraint will help to obtain a strong lower bound. We show how such mechanism can be embedded in classic MILP solvers, before extending this to WCSP solvers.

This manuscript is organized as follows:

- Chapter 2 introduces the general concept behind several paradigms used for solving combinatorial problems. This includes Integer Linear Programming, Constraint Programming, Cost Function Networks, SAT, and Pseudo Boolean Optimization.

- Chapter 3 shows how linear constraints can be encoded and propagated in a CFN. We also provide an algorithm extending the soft local consistency Virtual Arc Consistency [Cooper *et al.* 2010](VAC) to handle linear constraints. Finally, we give the experimental results obtained on instances involving linear constraints. Most of the contribution of this chapter has been published in CPAIOR 2022 [Montalbano *et al.* 2022].

- Chapter 4 exploits the dual encoding of a CFN and VAC to enforce a newly defined soft local consistency: Virtual Pair-Wise Consistency. Experimental results show the benefit of such a strategy on several benchmarks. This contribution is a collaborative work with Tomas Werner from the Czech Technical University of Prague and has been published in CPAIOR 2023 [Montalbano *et al.* 2023].

- Chapter 5 introduces a conflict-free learning mechanism for memorizing bounds. We show how to embed this approach in classic MILP solvers, before extending this to WCSP solvers. We give some preliminary results obtained with this approach. We are currently preparing a submission presenting this contribution.

Chapter 2

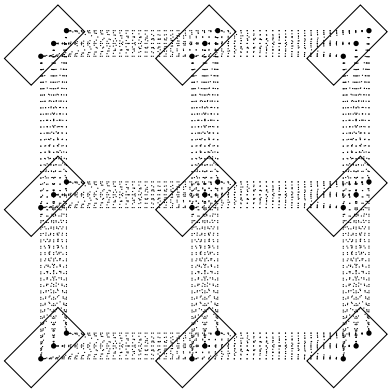
Optimization in Graphical Models

Contents

2.1 Graphical models

Graphical models (GMs) define a family of mathematical models providing a succinct representation for problems with complex relationships between variables. They have a wide range of applications such as bio-informatics, communication theory, statistical physics, computer vision, signal processing, information retrieval, and machine learning [Maathuis *et al.* 2018, Savchynskyy 2019]. A discrete GM is defined by a set of variables and a finite set of 'small' functions involving only a restricted number of variables. Those functions model interaction between variables. A binary associative and commutative operator is defined to combine the functions together and obtain joint multivariate functions. With this flexible definition, GMs can be used to model a large variety of well-known frameworks answering different tasks, going from satisfaction problems to probabilistic models [Cooper *et al.* 2020]. Some examples are: constraint networks [Rossi *et al.* 2006], propositional logic[Biere *et al.* 2021], generalized additive independence models [Bacchus & Grove 2013], Markov Random Fields [Kindermann & Snell 1980, Koller & Friedman 2009], Bayesian networks [Koller & Friedman 2009], Possibilistic and Fuzzy Constraint Networks [Dubois *et al.* 1993]. A GM can be depicted as a hypergraph where the vertices correspond to values/labels of the variables and the hyperedges

Figure 2.1: Example of a graphical model with a grid structure. The rectangle corresponds to variables and the nodes inside to labels. An edge depicts a relation between 2 labels.



to functions. Frequently, variables are represented as a square/bubble circling the values. When all the functions are pairwise (applies to 2 variables maximum), this hypergraph defines a simple graph (this explains the name graphical model) (see figure 2.1). In this thesis, we are mainly interested in graphical models like Cost Function Networks. This is an additive model where the local functions are *cost functions*. The task of finding the minimum cost assignment is known as the Weighted Constraint Satisfaction Problem (WCSP). We begin by introducing the various components related to WCSP solving.

2.2 Integer Linear Programming

Integer Linear Programming (ILP) is a powerful mathematical optimization technique used to solve a wide range of decision-making problems. The objective is to find optimal integer values for a set of decision variables while satisfying a system of linear constraints. An *objective function* embodies the goal to be maximized or minimized. ILP focuses specifically on linear relationships among discrete decision variables. In this thesis, we won't give an exhaustive tour of all theories and strategies developed over the years around the ILP paradigm. However, we recall some basic knowledge that finds a strong connection with different contributions of this thesis. We consider ILP problems in their canonical minimization form defined by equations (2.1)-(2.3). Where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and $x \in \mathbb{Z}$. Equation (2.1) defines the objective function we aim to minimize. Equations (2.2) define the linear constraints restricting the decision variables. Finally, (2.3) enforces that all the variables take an integer value.

$$\min c^T x \tag{2.1}$$

$$Ax \geq b \tag{2.2}$$

$$x \in \mathbb{Z} \tag{2.3}$$

Every ILP can be written in this canonical form. For example, we can transform a maximization problem into a minimization problem by multiplying the objective function by -1. Similarly, equality and $\leq$ constraints can always be transformed to $\geq$ constraints.

Example 2.1. *We use the Knapsack Problem with Conflict Graph (KPCG) as a running example. First introduced by Yamada et al. [Yamada et al. 2002], this problem is a variation of the well-known Knapsack Problem (KP) [Pisinger & Toth 1998], where some items are incompatible with some others. More formally, we describe the problem as a knapsack of capacity C and a collection of n items written $\boldsymbol{X} = \{x_1, \ldots, x_n\}$. Each item is described by a positive profit p_i and a positive weight w_i. The profits define the objective function to maximize while a linear capacity $\leq$ constraint restricts the possible combination of weights. Additionally, there is an undirected conflict graph $G = (V, E)$. In this graph, each vertex $i \in V$ represents an item, and an edge $(i, j) \in E$ indicates that items i and j cannot be packed together. In the interest of having a consistent notation throughout the paper, we write the KPCG as a minimization problem with $\geq$ constraints. We transform the maximization problem to a minimization problem by multiplying the objective function by -1. Similarly, the $\leq$ constraints are multiplied by -1 to obtain $\geq$ constraints. Finally, for every Boolean variable x, we introduce a Boolean variable $\bar{x}$ taking the opposite value of x, this corresponds to the constraint $x + \bar{x} = 1$. We can use this constraint to transform any negative coefficient $-\alpha x$ to $\alpha\bar{x} - \alpha$. This transformation preserves the equivalence of the problem.*
Here is an example of a KPCG with 7 items, with profits $\{4, 9, 4, 3, 5, 7, 7\}$ and weights $\{3, 5, 3, 3, 5, 5, 5\}$, the capacity is 10. The pairs of conflicts are:
$\{x_1, x_2\}$,$\{x_1, x_3\}$,$\{x_2, x_3\}$,$\{x_4, x_5\}$.
We also allow constraints enforcing that 2 items must have the same behavior (here, $x_6 = x_7$).

$$\min 4x_1 + 9x_2 + 4x_3 + 3x_4 + 5x_5 + 7x_6 + 7x_7 \tag{2.4a}$$
$$s.t$$
$$3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7 \geq 10 \tag{2.4b}$$
$$-x_1 - x_2 \geq -1 \tag{2.4c}$$
$$-x_1 - x_3 \geq -1 \tag{2.4d}$$
$$-x_2 - x_3 \geq -1 \tag{2.4e}$$
$$-x_4 - x_5 \geq -1 \tag{2.4f}$$
$$x_6 - x_7 \geq 0 \tag{2.4g}$$
$$x_7 - x_6 \geq 0 \tag{2.4h}$$
$$x_i \in \{0, 1\}, \quad i = 1, \ldots, 7 \tag{2.4i}$$

The optimal solution is $x_6 = 1, x_7 = 1$ (or $x_2 = x_5 = 1$), and all other variables are 0, with objective value 14. ■

Solving an ILP is an NP-Hard task [Schrijver 1998]. A classic approach is to

use a *branch and bound* (B&B) strategy [Little *et al.* 1963]. It enumerates all the possible solutions by using a rooted tree to represent the search space. Each node corresponds to a different sub-problem. A *branching variable* is chosen, and the children of a node are defined by restricting the domain of the branching variable. Thus, leading to a smaller sub-problem. We associate to each node an *upper* and a *lower* bound (UB and LB). The upper bound represents an upper limit on the objective value, it typically corresponds to the best-known solution. While the lower bound is an estimation of the best possible solution that we could discover by visiting the subtree. Most methods to compute lower bounds rely on *relaxation* of constraints. The idea is to remove or weaken some constraints to make the solving easier. The optimal objective value of the relaxed problem is necessarily lower or equal to the optimal objective value of the ILP. The most used relaxation is the *linear relaxation* and consists of relaxing all the integrality constraints. This new problem is a Linear Program (LP).

$$\min\{c^T x | Ax \geq b, x \in \mathbb{R}^+\} \tag{2.5}$$

The optimal solution of an LP can be found using the *simplex algorithm* [Murty 1983]. While the Simplex algorithm exhibits exponential theoretical complexity, it can in practice demonstrate polynomial average case complexity [Schrijver 1998, Spielman & Teng 2004].

Example 2.2. *The LP of example* (2.4a)-(2.4i) *is:*

$$\min 4x_1 + 9x_2 + 4x_3 + 3x_4 + 5x_5 + 7x_6 + 7x_7 \tag{2.6a}$$
$$s.t$$
$$3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7 \geq 10 \tag{2.6b}$$
$$-x_1 - x_2 \geq -1 \tag{2.6c}$$
$$-x_1 - x_3 \geq -1 \tag{2.6d}$$
$$-x_2 - x_3 \geq -1 \tag{2.6e}$$
$$-x_4 - x_5 \geq -1 \tag{2.6f}$$
$$x_6 - x_7 \geq 0 \tag{2.6g}$$
$$x_7 - x_6 \geq 0 \tag{2.6h}$$
$$x_i \in [0,1], \quad i = 1, \ldots, 7 \tag{2.6i}$$

The optimal solution is $x_3 = 1, x_5 = 1, x_6 = 0.2, x_7 = 0.2$ *all other variables are 0 with objective value 11.8.* ■

We can use the optimal relaxed solution to obtain an LB at each node, if LB $\geq$ UB then we don't need to visit the subtree. If the optimal relaxed solution corresponds to a feasible solution of the ILP, then we update the UB. A common strategy to choose the next decision variable in the (B&B) algorithm is to select a variable having a fractional value in the optimal relaxed solution. For example, if $x_6 = 0.2$ in the optimal relaxed solution then we can create 2 branches, one with $x_6 \leq 0$ and

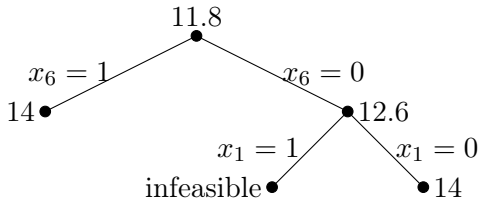


Figure 2.2: Example of a search tree developed to solve ILP (2.9a)- (2.6i)

the other with $x_6 \geq 1$.

Example 2.3. *We consider the KPCG* (2.4a)-(2.4i) *and its LP relaxation* (2.9a)-(2.6i). *The first optimal fractional solution is* $x_3 = 1, x_5 = 1, x_6 = 0.2, x_7 = 0.2$ *with objective value 11.8.* x_6 *is a fractional variable, we choose it as the first decision variable. If we set* $x_6 = 1$ *then the new optimal fractional solution is* $x_6 = 1, x_7 = 1$ *with objective value 14. It corresponds to an integer solution, we update the upper bound, UB=14. We don't need to pursue the search in this branch.*
If we set $x_6 = 0$*, then the optimal relaxed solution is* $x_1 = 0.6$*,* $x_2 = 0.4$*,* $x_3 = 0.4$*,* $x_5 = 1$ *with objective value 12.6. We continue the search and set* $x_6 = 0, x_1 = 1$*. The resulting problem is infeasible. We continue and set* $x_6 = 0, x_1 = 0$*. The optimal fractional solution is* $x_2 = 1, x_5 = 1$ *with objective value 14. This ends the search, the optimal objective value of* (2.4a) *is 14. Figure 2.2 represents the search tree, each node is associated with a lower bound, and each edge to an assignment.* ■

An additional feature for efficient MIP solving is *cut generation*. Cuts are additional linear constraints that are added to a linear programming relaxation to tighten the bounds and reduce the feasible solution space of the relaxation to get closer to the integer solution space. Those cuts are typically found after solving the linear relaxation. Learning cuts during a branching algorithm corresponds to a *branch and cuts* procedure. In this context, a cut is global if it is valid at every node of the search tree, while it is local if it is true only for a subtree. Cuts were introduced by Gomory in 1960 [Gomory 1960]. He has shown that an ILP can be solved by sequentially adding cuts to the linear relaxation, this defined the *cutting plane algorithm*. Later, Balas et al. introduced how to learn globally valid Gomory cuts inside a branch and cuts procedure [Balas *et al.* 1996]. This led to a significant improvement in the results and since then modern MIP solvers have employed several cutting-plane algorithms. We refer the reader to [Marchand *et al.* 2002] for more information on cuts in MIP.

Dual and Reduced costs

The dual of an LP P ((2.5)) is defined by the LP:

$$\max\{b^T y | A^T y \leq c, y \in \mathrm{R}^+\} \tag{2.7}$$

The dual introduces one dual variable for each primal constraint and one dual constraint for each primal variable. Analyzing the dual can provide useful information on the primal (P), in particular, if P is feasible then its dual is also feasible. Moreover, the objective value of any dual solution gives a lower bound of the primal. When the primal and dual solutions are both optimal, they coincide, ensuring the optimality of the problem.
Given a solution $\mathbf{y}$ of the dual, the reduced cost of x_i is defined by the slack of its corresponding dual constraint: $rc^{\mathbf{y}}(x_i) = c_i - A_i^T \mathbf{y}_i$. If $\mathbf{y}$ is optimal, then the reduced cost of a variable can be seen as the amount by which we must decrease the coefficient c_i of x_i in the objective function to obtain $x_i > 0$ in the optimal solution. Equivalently it can be interpreted as a lower bound on the difference of objective value between any feasible solution with $x_i > 0$ and the optimal solution.

Example 2.4. *The dual of LP* (2.9a)-(2.6i)*, involves 14 variables* $\{y_1, \ldots, y_{14}\}$ *(one for each constraint and one for each upper bound constraint* $x_i \leq 1$*) and 7 constraints.*

$$\begin{aligned}
\max 10y_1 - y_2 - y_3 - y_4 - y_5 + y_8 + y_9 + y_{10} + y_{11} + y_{12} + y_{13} + y_{14} \\
3y_1 - y_2 - y_3 + y_8 \leq 4 \\
5y_1 - y_2 - y_4 + y_9 \leq 9 \\
3y_1 - y_3 - y_4 + y_{10} \leq 4 \\
3y_1 - y_5 + y_{11} \leq 3 \\
5y_1 - y_5 + y_{12} \leq 5 \\
5y_1 + y_6 - y_7 + y_{13} \leq 7 \\
5y_1 - y_6 + y_7 + y_{14} \leq 7 \\
y_i \geq 0, \quad i = 1, \ldots, 7 \\
y_i \leq 0, \quad i = 8, \ldots, 14
\end{aligned}$$

The optimal dual solution $\mathbf{y}$ *is* $\mathbf{y}_1 = 1.4, \mathbf{y}_3 = 0.2, \mathbf{y}_5 = 1.2, \mathbf{y}_{12} = -0.8$ *with objective value* 11.8*. The reduced costs are:*

- $rc^{\mathbf{y}}(x_1) = 4 - 3 \times 1.4 - 0.2 = 0$
- $rc^{\mathbf{y}}(x_2) = 9 - 5 \times 1.4 = 2$
- $rc^{\mathbf{y}}(x_3) = 4 - 3 \times 1.4 - 0.2 = 0$
- $rc^{\mathbf{y}}(x_4) = 3 - 3 \times 1.4 - 1.2 = 0$
- $rc^{\mathbf{y}}(x_5) = 5 - 5 \times 1.4 - 1.2 - 0.8 = 0$
- $rc^{\mathbf{y}}(x_6) = 7 - 5 \times 1.4 = 0$
- $rc^{\mathbf{y}}(x_7) = 7 - 5 \times 1.4 = 0$

We can observe that the non-zero variables of primal optimal solutions $x_3 = 1, x_5 = 1, x_6 = 0.2, x_7 = 0.2$ *and* $x_1 = 1, x_5 = 1, x_6 = 0.2, x_7 = 0.2$ *have all reduced costs 0.* ■

2.3 Constraint Programming

Constraint programming (CP) [Rossi *et al.* 2006] is a powerful paradigm that enables the modeling and solving of complex combinatorial problems. In CP, problems are represented as a set of variables, domains, and constraints, such a triplet defines a *Constraint Network* (CN). Variables represent the unknowns or decision variables in the problem, while domains define the possible values that these variables can take. Constraints capture the relationships, conditions, and rules that must be satisfied by the variables. The objective is to find solutions that satisfy all the constraints, this defines the *Constraint Satisfaction Problem* (CSP). Both ILP and CP offer complementary approaches to tackle complex optimization challenges, with ILP excelling in problems where linear relationships prevail, while CP provides flexibility in modeling various constraints and discrete decision spaces. It's important to note that the picture is way more complicated when discussing actual solving performance. Many other parameters can impact the efficiency of the solving, and ILP/CP can perform poorly or well on problems where we expected the opposite.

One of the notable advantages of CP is its ability to handle *global constraints*[1]. They are high-level, reusable, and generic constraints that capture common patterns or complex relationships between variables. They provide a powerful tool to model efficiently and concisely different kinds of problems (see example 2.5). Moreover, a solver can take advantage of their particular structure to reduce the search space and enable a more efficient solving.

A very well-known global constraint is the AllDifferent constraint, which enforces that all the variables in its scope must take a different value. In this thesis, we will mainly focus on *linear constraints* as defined in ILP.

Example 2.5. *We can model the famous Sudoku puzzle as a CSP. As a reminder, a Sudoku is a* 9×9 *grid decomposed into 9* 3×3 *sub-grids. The objective is to fill the 81 cells with digits from* $1 - 9$ *without introducing the same digits twice within the same row, column, or sub-grid. In the CP model, the variables correspond to the cells, let* x_{ij} *correspond to the cell located at row* i *and column* j*. The domains of each variable are the 9 digits* $\boldsymbol{D} = \{1, \dots, 9\}$*. Let* X_k *designed the cells of the sub-grids* k*, with* $k \in \{1, \dots 9\}$*. The constraints enforce the rules of the CSP, for each row* i*, column* j*, and sub-grid* k*, we have the following binary constraints:*

$$\begin{aligned} CR_i : x_{ij} \neq x_{ij'} \quad & \forall i, j', j < j' \in \{1, \dots, 9\} \\ CC_j : x_{ij} \neq x_{i'j} \quad & \forall i, j, i < i' \in \{1, \dots, 9\} \\ CS_k : x_{ij} \neq x_{i'j'} \quad & \forall x_{ij}, x_{i'j'} \in X_k, i \neq i', j \neq j', k \in \{1, \dots, 9\} \end{aligned}$$

[1] Here is a catalog of global constraints. https://sofdem.github.io/gccat/gccat/titlepage.html

Equivalently we can express the constraints using the global constraint AllDifferent, let XR_i, XC_j, XS_k *be the set of variables in each row* i*, column* j*, and subgrid* k*, respectively*

$$\begin{aligned} CR_i &: \mathit{AllDifferent}(XR_i) \quad i \in \{1, \ldots, 0\} \\ CC_j &: \mathit{AllDifferent}(XC_j) \quad j \in \{1, \ldots, 0\} \\ CS_k &: \mathit{AllDifferent}(XS_k) \quad k \in \{1, \ldots, 0\} \end{aligned}$$

This reduces the number of constraints from 972 to 27. ■

Additionally, CP supports constraint optimization, where objectives and preferences can be incorporated to find optimal or near-optimal solutions based on predefined criteria. The application of constraint programming has yielded significant advancements in various fields, including artificial intelligence, operations research, scheduling, supply chain management, graph algorithms, computer vision, and computational linguistics [Rossi *et al.* 2006]. Its ability to handle combinatorial problems with large solution spaces, complex constraints, and diverse problem structures makes it a valuable tool for tackling real-world challenges.

2.3.1 Constraint Satisfaction Problem

Before introducing Weighted Constraint Satisfaction Problem (WCSP) we define Constraint Satisfaction (CSP). We directly use a notation depicting that CSPs are a particular form of WCSPs. A Constraint Network (CN) is a triplet $\langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C} \rangle$ where :

- $\boldsymbol{X} = \{x_1, \ldots, x_n\}$ are variables.
- $\boldsymbol{D} = \{\boldsymbol{D}_1, \ldots, \boldsymbol{D}_n\}$ are the domains of the variables.
- $\boldsymbol{C}$ is a set of constraints.

Let $J \subseteq \boldsymbol{X}$ be a subset of variables, we denote by $\ell(J)$ the Cartesian product $\Pi_{i \in J} \boldsymbol{D}_i$ of the domains of the variables in J. An assignment (or a tuple) $\tau \in \ell(J)$ is a mapping from each $i \in J$ to a value $a \in \boldsymbol{D}_i$. If $J = \boldsymbol{X}$ then τ defines a *complete assignment*, otherwise it is a *partial assignment*. A partial assignment is infeasible if at least one of the constraints in $\boldsymbol{C}$ is not satisfied. By τ_i we denote the value of x_i in τ, τ_J corresponds to the values of the variables $J \in \boldsymbol{X}$. For a given CN, a solution to the corresponding Constraint Satisfaction Problem (CSP) is a complete assignment satisfying all the constraints in $\boldsymbol{C}$. A constraint $c_{\boldsymbol{S}} \in \boldsymbol{C}$ is defined by a pair $\langle \boldsymbol{S}, r_{\boldsymbol{S}} \rangle$ where $\boldsymbol{S} \subseteq \boldsymbol{X}$ is the *scope* and $r_{\boldsymbol{S}}$ is a function $r_{\boldsymbol{S}} : \ell(\boldsymbol{S}) \to \{0; \top\}$, the value 0 correspond to an allowed tuple while $\top$ to a forbidden tuple. The size of the scope is the *arity* of the constraint. Each constraint is either represented in *extension* or *intention*. A constraint represented in extension, also known as a table constraint, explicitly lists all the allowed or forbidden tuples. Only low arity constraints can be written in extension within a reasonable memory size limit because the number of

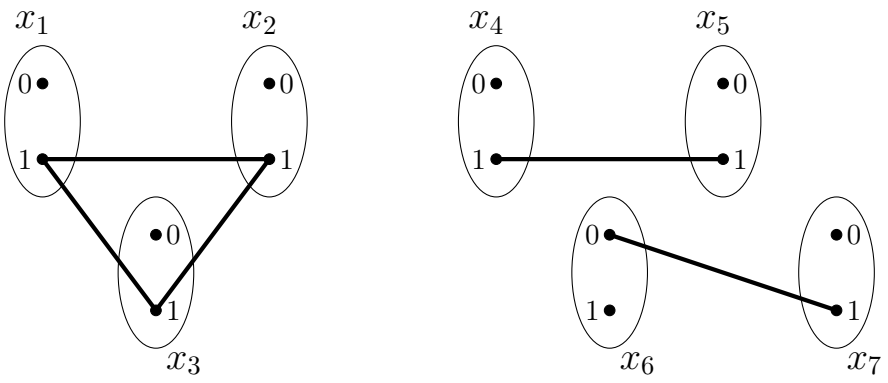


Figure 2.3: Graphical representation of the binary constraints. A thick edge corresponds to a forbidden tuple.

tuples grows exponentially with arity. A constraint given in intention, is defined by a function or a logical expression that specifies the relationship between the variables, for example, global constraints are typically given in intention.

Example 2.6. *We can model KPCG instance (2.4a)-(2.4i) as a CN, the problem has 7 variables* $\boldsymbol{X} = x_1, \ldots, x_7$ *with domains* $\{0, 1\}$*. There are 6 constraints, one global constraint* $c_{1,2,3,4,5,6,7} : 3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7 \geq 10$*, and 5 binary constraints. When the number of tuples is not too large, each constraint can be represented as a table where only the allowed tuples are written:*

$c_{1,2}$	x_1	x_2	$c_{1,3}$	x_1	x_3	$c_{2,3}$	x_2	x_3	$c_{4,5}$	x_4	x_5	$c_{6,7}$	x_6	x_7
	1	*0*		*1*	*0*		*1*	*0*		*1*	*0*		*1*	*1*
	0	*1*		*0*	*1*		*0*	*1*		*0*	*1*		*0*	*0*
	0	*0*		*0*	*0*		*0*	*0*		*0*	*0*			

Graphs can provide a more visual representation, particularly when we have binary constraints. Each variable is represented by a bubble, each value is represented by a node, and an edge between 2 nodes indicates a forbidden tuple (see 2.3). Note that constraints $c_{1,2}, c_{1,3}$ *and* $c_{2,3}$ *can be contracted using the clique global constraint. Such a constraint enforces that at-most-one variable within the scope can take value 1. Here we have* $c_{1,2,3} : clique(x_1, x_2, x_3)$*. We present how to model the objective function in example 2.8.* ■

To solve a CSP, one can typically use a backtracking search combined with an algorithm removing the values that won't appear in any solution of the current search space. We refer to this technique as *pruning* the *inconsistent* values. Removing those values reduces the search space, in this sense it serves the same purpose as bounding techniques used in ILP. Unfortunately, determining whether a value is inconsistent is most of the time an NP-Hard task [Rossi *et al.* 2006]. The CP community tried (and is still trying) to find the best trade-off between time spent

looking for inconsistent values and numbers (or quality) of the pruned values.
A common approach to prune inconsistent values is to define *local consistency* algorithms. Those algorithms reason only on a subset of constraints and remove the values which are locally inconsistent. The most used one is *Generalized Arc Consistency* (GAC).

Definition 2.1 (Generalized Arc Consistency)**.** *Let $P = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C} \rangle$ be a CSP. A value $a \in \boldsymbol{D}_i$ is said to be generalized arc consistent (GAC) with respect to a constraint $c_{\boldsymbol{S}} \in \boldsymbol{C}$, if there exists at least one assignment $\tau \in \ell(\boldsymbol{S})$ with $\tau_i = a$ verifying $c_{\boldsymbol{S}}(\tau) = 0$. A variable is GAC with respect to a constraint $c_{\boldsymbol{S}} \in \boldsymbol{C}$, if all its values are GAC with respect to $c_{\boldsymbol{S}}$. A variable is said to be GAC with respect to the whole CSP P if it is GAC with respect to all its constraints in $\boldsymbol{C}$. A CSP P is said to be GAC if all its variables are GAC.*

GAC on a binary CSP is also known as *Arc Consistency* (AC).

Example 2.7. *Given the CSP given in example 2.6, we can observe that for each variable x_i there exists at least two allowed tuples in each constraint: one with $x_i = 1$ and one with $x_i = 0$. The CSP is GAC. If we add the constraint $x_1 = 1$ then there exists no tuple allowed by $c_{1,2}$ such that $x_1 = 1$ and $x_2 = 1$ hence we can delete the value $x_2 = 1$. The same goes for $x_3 = 1$. Once it's done, the CSP is GAC.* ■

The *arc consistent closure* of a CSP P (noted $AC(P)$) is the unique CSP that results from removing values from domains that violate the arc consistency property. If $AC(P)$ is empty then P is infeasible. This happens as soon as the domain of one variable becomes empty. We call this a *domain wipe-out*. Over the years, different approaches have been proposed to enforce GAC as efficiently as possible on table constraints [Yap *et al.* 2020]. Aside GAC, there exists different local consistency algorithms trying to find the best trade-off between strength and complexity of the propagation [Rossi *et al.* 2006]. One approach can perform well for one class of instances and can be ineffective for another one. In general, establishing GAC on a global constraint is NP-hard [Bessiere *et al.* 2004] and only partial filtering can be enforced [Rossi *et al.* 2006]. However, for some particular global constraints (like AllDifferent), an efficient dedicated algorithm enforcing GAC has been designed [Rossi *et al.* 2006].
The CSP can also be used to model optimization problems. An objective function is modeled by adding an extra variable *obj* and a constraint enforcing that *obj* must be equal to the function we try to minimize/maximize. The cost of any solution corresponds to the value of *obj*, therefore the domain of *obj* can be large as it must contain all possible values between the best-known lower and upper bounds. This is referred to as the Constraint Satisfaction Optimization Problem. An optimal solution can be obtained by solving a sequence of CSPs [Van Hentenryck *et al.* 1992, Rossi *et al.* 2006]. For example, for a minimization problem, each time the solver solves a CSP and finds a solution with *obj*=sol, it adds the constraint $obj < sol$. Solving the resulting CSP will give a new solution with a lower cost. The

solver will keep finding better solutions until it derives an infeasible CSP, in this case, we know that the last solution was optimal.

Example 2.8. *Following example 2.6, we can model an objective function by introducing a variable obj with domain* $[0, 39]$ *and a constraint* $obj = 4x_1 + 9x_2 + 4x_3 + 3x_4 + 5x_5 + 7x_6 + 7x_7$. ■

2.3.2 Weighted Constraint Satisfaction Problem

Weighted CSP (WCSP) is an extension of the CSP that incorporates costs associated with the constraints and variables assignments. Those costs can be represented by using soft constraints, also called *Cost Functions*. A cost function $c_{\boldsymbol{S}}$ is defined by a pair $\langle \boldsymbol{S}, c_{\boldsymbol{S}} \rangle$ where $\boldsymbol{S} \subseteq \boldsymbol{X}$ is the scope and $c_{\boldsymbol{S}}$ is a function $c_{\boldsymbol{S}} : \ell(\boldsymbol{S}) \to [0, \top]$, the cost $\top \in \mathrm{R}^+ \cup \{\infty\}$ is a special constant symbolizing infeasibility. Just like for CSP constraints, a cost function can be expressed in extension or in intention.
A Cost Function Network (CFN) is a quadruplet $\langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top \rangle$ where :

- $\boldsymbol{X} = \{x_1, \ldots, x_n\}$ are variables.
- $D = \{\boldsymbol{D}_1, \ldots, \boldsymbol{D}_n\}$ are the domains of the variables.
- $\boldsymbol{C}$ is a set of cost functions.

A cost function having arity 2 is a *binary* cost function. A cost function having arity 1 is a *unary* cost function. We define the cost function having an empty scope as $c_\varnothing$. This cost will count in every assignment and provides a natural lower bound because negative costs are not allowed in a CFN. In the following, we suppose that all the unary costs functions and $c_\varnothing$ are defined in $\boldsymbol{C}$ and $C^+ = \boldsymbol{C} \setminus \{c_\varnothing\}$. The cost of a complete assignment $\tau \in \ell(\boldsymbol{X})$ is $c_P(\tau) = \sum_{\boldsymbol{S} \in \boldsymbol{C}} c_{\boldsymbol{S}}(\tau_{\boldsymbol{S}})$. If the cost is $\geq \top$ then τ defines a forbidden assignment. We denote by e the number of distinct cost functions, n the number of WCSP variables, and d the maximum domain size. The WCSP asks to find a complete non-forbidden assignment having minimum cost.

Example 2.9. *Following example 2.6, we can add unary costs to model the KPCG problem (2.4a)-(2.4i). As for CN, low arity cost functions can be represented as tables. We add one extra column to indicate the cost associated with each tuple. High arity global constraints need a specific representation, see chapter 3 for linear constraints. We change the domains name from* $\{0, 1\}$ *to* $\{a, b\}$*, to avoid confusing values with costs.*

$c_{1,2}$	x_1	x_2	$cost$
	a	a	0
	a	b	0
	b	a	0
	b	b	$\top$

$c_{1,3}$	x_1	x_3	$cost$
	a	a	0
	a	b	0
	b	a	0
	b	b	$\top$

$c_{2,3}$	x_2	x_3	$cost$
	a	a	0
	a	b	0
	b	a	0
	b	b	$\top$

$c_{4,5}$	x_4	x_5	$cost$
	a	a	0
	a	b	0
	b	a	0
	b	b	$\top$

$c_{6,7}$	x_6	x_7	$cost$
	a	a	0
	a	b	$\top$
	b	a	$\top$
	b	b	0

x_1	$cost$	x_2	$cost$	x_3	$cost$	x_4	$cost$	x_5	$cost$	x_6	$cost$	x_7	$cost$
a	0	a	0	a	0	a	0	a	0	a	0	a	0
b	4	b	9	b	4	b	3	b	5	b	7	b	7

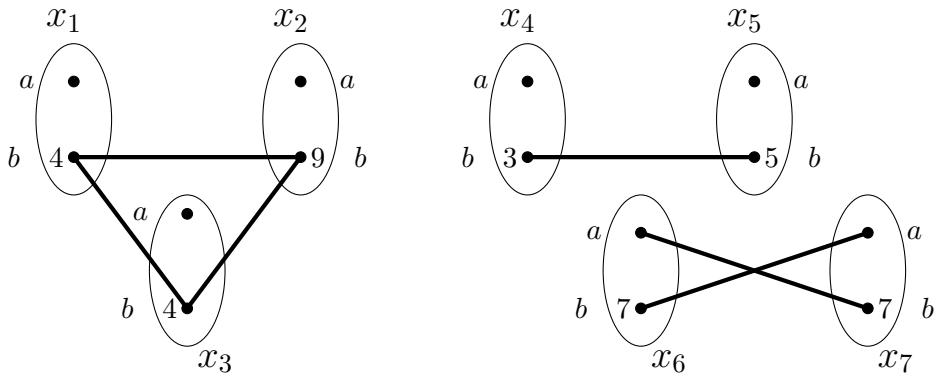


Figure 2.4: Graphical representation of the binary cost functions. A thick edge indicates a forbidden tuple.

We can also use a graph, the names of the values are written outside the bubble, and the non-zeros unary costs are inside. The non-zero costs are written on the edges (see figure 2.4).

■

CFN provides a flexible framework where costs can be used to represent various aspects, such as preference amongst the constraints/variables, and probabilities... Just like for CSP, the resolution of WCSP relies on backtracking search and constraint propagation. The *Soft Local Consistency* algorithms sequentially examine small subsets of cost functions. On top of removing the locally inconsistent values, it computes a lower bound which can be used to backtrack if it exceeds the upper bound. To find and enforce a lower bound, those algorithms rely on the notion of *equivalent WCSP*.

Definition 2.2 (Equivalent WCSP)**.** *Two WCSPs P, P' are equivalent if they have the same structure, i.e., the set of scopes and variables are identical. And $c_P(\tau) = c_{P'}(\tau)$ for all complete assignments $\tau \in \ell(\boldsymbol{X})$. However, the cost distribution inside individual cost functions might differ.*

When two WCSPs are equivalent, we also say they are a *reparametrization*, of each other. A reparametrization is better if $c_\varnothing$ is higher. All the reparametrizations presented here are based on local *Equivalence Preserving Transformations* (EPTs). An EPT corresponds to the movement of costs between two cost functions. Let $\boldsymbol{S} \subset \boldsymbol{S}'$ be two scopes with corresponding cost functions $c_{\boldsymbol{S}}, c_{\boldsymbol{S}'}$. Procedure MoveCost describes how to move a cost α from $c_{\boldsymbol{S}}$ to $c_{\boldsymbol{S}'}$ and keep the equivalence. Note that to keep a valid WCSP the costs need to stay in the range of possible costs, in particular, an EPT must not create any negative costs.

To see its correctness, observe that if τ is used in a complete assignment, then only exactly one extension of τ to $\boldsymbol{S}'$ must be used. Therefore, the sum of $c_{\boldsymbol{S}}$ and $c_{\boldsymbol{S}'}$ remains unaffected whether the cost α is attributed to τ in $c_{\boldsymbol{S}}$ or to all of its extensions τ' in $c_{\boldsymbol{S}'}$.

Procedure MoveCost($c_{\boldsymbol{S}}, c_{\boldsymbol{S}'}, \tau, \alpha$): Move α units of cost between the tuple τ of scope $\boldsymbol{S}$ and tuples τ' that extend τ in scope $\boldsymbol{S}'$

Data: Scopes $\boldsymbol{S} \subset \boldsymbol{S}'$
Data: $\tau \in \ell(\boldsymbol{S})$
Data: cost α to move
$c_{\boldsymbol{S}}(\tau) \leftarrow c_{\boldsymbol{S}}(\tau) + \alpha$;
foreach $\tau' \in \ell(\boldsymbol{S}') \mid \tau'_{\boldsymbol{S}} = \tau$ **do**
  $c_{\boldsymbol{S}'}(\tau') \leftarrow c_{\boldsymbol{S}'}(\tau') - \alpha$;

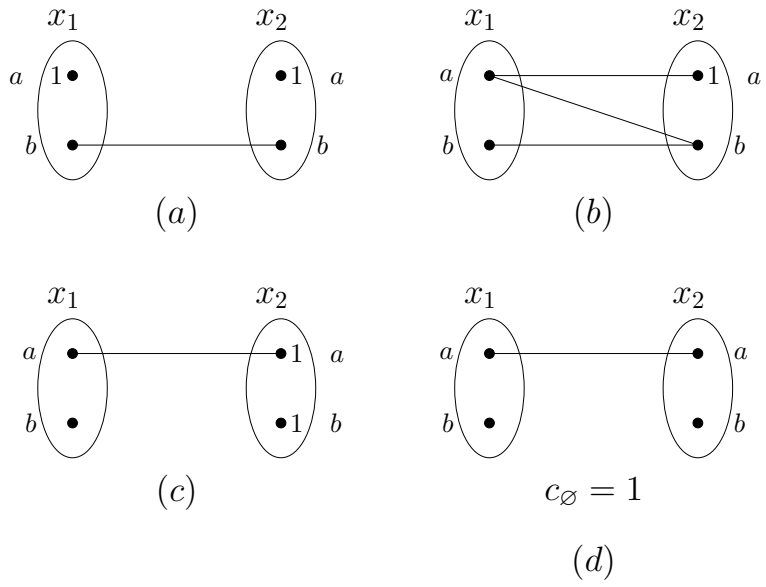


Figure 2.5: Graphical example of an extension $extend(c_1, (1, a), c_{1,2}, 1))$ (b), a projection $project(c_2, c_{1,2}, (2, b), 1)$ (c) and a unary projection $unaryProject(c_2, 1)$ (d). An edge corresponds to a cost of 1.

As a matter of terminology, when $\alpha > 0$, cost moves from the larger arity cost function $c_{\boldsymbol{S}'}$ to the smaller arity $c_{\boldsymbol{S}}$ and the move is called a *projection*, denoted $project(c_{\boldsymbol{S}}, c_{\boldsymbol{S}'}, \tau, \alpha)$.When $\alpha < 0$, cost moves to the larger arity cost function $c_{\boldsymbol{S}'}$ and the move is called an *extension*, denoted $extend(c_{\boldsymbol{S}}, \tau, c_{\boldsymbol{S}'}, -\alpha)$.When $\boldsymbol{S} = \emptyset$ and $|\boldsymbol{S}'| = 1$, with $\boldsymbol{S}' = \{i\}$, the move is called a *unary projection*, denoted $unaryProject(c_i, \alpha)$, equivalent to MoveCost$(c_\varnothing, c_i, \emptyset, \alpha)$. We never perform extensions from $c_\varnothing$, so it monotonically increases during the run of an algorithm and as we descend a branch of the search tree.

Example 2.10. *As an example, let's consider figure 2.5, we have 2 Booleans variables with domains $\{a, b\}$ and one binary constraint. A cost of 1 is moved from $x_1 = a$ to $c_{1,2}$ (fig b), as a consequence $c_1(a)$ is decreased and the cost of all the tuples using $x_1 = a$ in $c_{1,2}$ are increased:*

- $c_1(a) \to 0$
- $c_{1,2}(a,a) \to 1$
- $c_{1,2}(a,b) \to 1$

Observe that this operation is an extension ($extend(c_1,(1,a),c_{1,2},1)$*) and doesn't alter the cost of any assignment, both in (a) and (b)* $c_P((a,a)) = 2$, $c_P((a,b)) = 1$, $c_P((b,a)) = 1$, $c_P((b,b)) = 1$.
Figure 2.5 (c) displays a projection $project(c_2, c_{1,2}, (2,b), 1)$ *and (d) a unary projection* $unaryProject(c_2, 1)$. ■

If we consider only integer costs, finding an optimal reparametrization with the highest possible $c_\varnothing$ is an NP-Hard task [Cooper & Schiex 2004]. However, the optimal reparametrization allowing rational costs can be found from the optimal dual solution of the following linear relaxation of the WCSP [Cooper *et al.* 2010] called the *local polytope*. Also, the rational solution is in general higher, so this is a rare case where computationally easier problems give better results.

Local Polytope

$$\min Obj \stackrel{\text{def}}{=} c_\varnothing + \sum_{i \in \boldsymbol{X}, a \in \boldsymbol{D}_i} c_i(a) x_{ia} + \sum_{c_{\boldsymbol{S}} \in C^+, \tau \in \ell(\boldsymbol{S})} c_{\boldsymbol{S}}(\tau) x_{S:\tau} \tag{2.8a}$$

$$\text{s.t.} \quad \forall i \in \boldsymbol{X}, \quad \sum_{a \in \boldsymbol{D}_i} x_{ia} = 1 \tag{2.8b}$$

$$\forall c_{\boldsymbol{S}} \in C^+, i \in \boldsymbol{S}, a \in \boldsymbol{D}_i \Big(\sum_{\tau \in \ell(\boldsymbol{S}), \tau_i = a} x_{S:\tau} \Big) - x_{ia} = 0 \tag{2.8c}$$

The local polytope corresponds to the linear relaxation of the *tuple encoding* of the WCSP. Each value (i,a) is represented by a Boolean variable x_{ia} taking value 1 if $x_i = a$, similarly, tuple $\tau \in \ell(\boldsymbol{S})$ is represented by a Boolean variable $x_{\boldsymbol{S}:\tau}$ taking value 1 if τ is chosen. The objective function tries to minimize the sum of the cost functions (2.8a). While constraints (2.8b) enforce that each variable must be assigned to exactly one value. Finally, constraints (2.8c) enforce that the assignment of variables and tuples are compatible, also called marginal consistency constraints. Constraints enforcing that only one tuple should be selected for each scope are redundant with constraints (2.8c), therefore they are not written.

Example 2.11. *As an example we write parts of the local polytope corresponding to Example 2.9. We exclude the linear constraint because it has not yet been defined how*

we can model it in the local polytope, which is the subject of the following chapter.

$$\begin{aligned}
&\min 4x_{1b} + 9x_{2b} + 4x_{3b} + 3x_{4b} + 5x_{5b} + 7x_{6b} + 7x_{7b} \\
&+\top x_{12:bb} + \top x_{13:bb} + \top x_{23:bb} + \top x_{45:bb} + \top x_{67:ab} + \top x_{67:ba} && \text{(2.9a)} \\
&s.t \\
&x_{12:aa} + x_{12:ab} - x_{1a} = 0 && \text{(2.9b)} \\
&x_{12:ba} + x_{12:bb} - x_{1b} = 0 && \text{(2.9c)} \\
&x_{12:aa} + x_{12:ba} - x_{2a} = 0 && \text{(2.9d)} \\
&x_{12:ab} + x_{12:bb} - x_{2b} = 0 && \text{(2.9e)} \\
&x_{13:aa} + x_{13:ab} - x_{1a} = 0 && \text{(2.9f)} \\
&\dots \\
&x_{67:ab} + x_{67:bb} - x_{7b} = 0 && \text{(2.9g)} \\
&x_{ia} + x_{ib} = 1 \quad i = 1, \dots, 7 && \text{(2.9h)} \\
&x_{ia}, x_{ib} \geq 0, \quad i = 1, \dots, 7 && \text{(2.9i)}
\end{aligned}$$

■

In practice, solving this LP at each node of a search tree is prohibitively expensive and does not perform well [Hurley *et al.* 2016]. Indeed, solving LPs with this particular structure is as hard as solving any LPs [Prusa & Werner 2013] and the worst-case complexity of an exact LP algorithm is $O(N^{2.5})$ [Vaidya 1989], with $N \in O(ed + nd)$ for binary WCSPs, where e is the number of distinct binary cost functions, n is the number of WCSP variables and d is the maximum domain size. An alternative approach involves finding lower bounds of the local polytope by computing good feasible dual solutions. This led in parallel to Block-Coordinate Ascent (BCA) algorithms used in image analysis [Kolmogorov 2006, Werner 2007, Sontag *et al.* 2008, Komodakis *et al.* 2010, Sontag *et al.* 2012, Tourani *et al.* 2020] ([Savchynskyy 2019] surveys all those works) and *soft local consistencies* [Schiex 2000, Larrosa 2002a, de Givry *et al.* 2005, Zytnicki *et al.* 2009, Cooper *et al.* 2010] in constraint programming. In this thesis we are interested in the latter. Those methods have a strong connection with the dual problem of the local polytope:

Dual Local Polytope

$$\max \sum_{i \in \boldsymbol{X}} \pi_i \quad \text{s.t.} \tag{2.10a}$$

$$\forall i \in \boldsymbol{X}, a \in \boldsymbol{D}_i \quad \pi_i - \sum_{c_{\boldsymbol{S}} \in C^+, i \in \boldsymbol{S}} \varphi_{ia:\boldsymbol{S}} \leq c_i(a) \tag{2.10b}$$

$$\forall c_{\boldsymbol{S}} \in C^+, \tau \in \ell(\boldsymbol{S}) \quad \sum_{i \in \boldsymbol{S}, a \in \boldsymbol{D}_i, \tau_i = a} \varphi_{ia:\boldsymbol{S}} \leq c_{\boldsymbol{S}}(\tau) \tag{2.10c}$$

Where π_i is a dual variable corresponding to primal constraint (2.8b) and $\varphi_{ia:\boldsymbol{S}}$ corresponds to (2.8c). Given a dual feasible solution, we can modify the objective

function of the (Local Polytope) to obtain an equivalent problem but with an increased $c_\varnothing$. This modification can be obtained through a set of EPTs. A dual value π_i corresponds to cost move $unaryProject(x_i, \pi_i)$ and $\varphi_{ia:\boldsymbol{S}}$ corresponds to cost move MoveCost$(c_i, c_{\boldsymbol{S}}, (i,a), \varphi_{ia:\boldsymbol{S}})$. However, we need to relax the non-negativity condition on EPTs, meaning that one individual EPT might create a negative cost but the full set of EPTs must give a valid CFN. We can notice that the quantity $\pi_i - \sum_{c_{\boldsymbol{S}} \in C^+, i \in \boldsymbol{S}} \varphi_{ia:\boldsymbol{S}}$ (lhs of (2.10b)) is the sum of costs moved from/to $c_i(a)$, same goes for $\sum_{i \in \boldsymbol{S}, a \in \boldsymbol{D}_i} \varphi_{ia:\boldsymbol{S}}$ (lhs of (2.10c)) and $c_{\boldsymbol{S}}(\tau)$. It follows that given a dual solution of (Dual Local Polytope), a reparametrization can be extracted from the reduced costs $rc(x_{ia})$ and $rc(x_{\boldsymbol{S}:\tau})$ by setting $c_i(a)$ to $rc(x_{ia})$, $c_{\boldsymbol{S}}(\tau)$ to $rc(x_{\boldsymbol{S}:\tau})$ and $c_\varnothing$ to the optimum objective value (also observed in [Trösser *et al.* 2020]).

Example 2.12. *Following example 2.11, the dual local polytope of example 2.9 is:*

$$\max \sum_{i \in [1,7]} \pi_i \tag{2.11a}$$

$$s.t.$$

$$\Pi_1 - \varphi_{1a:12} - \varphi_{1a:13} \leq 0 \tag{2.11b}$$

$$\Pi_1 - \varphi_{1b:12} - \varphi_{1b:13} \leq 4 \tag{2.11c}$$

$$\dots \tag{2.11d}$$

$$\Pi_7 - \varphi_{7a:67} \leq 0 \tag{2.11e}$$

$$\Pi_7 - \varphi_{7b:67} \leq 7 \tag{2.11f}$$

$$\varphi_{1a:12} + \varphi_{2a:12} \leq 0 \tag{2.11g}$$

$$\varphi_{1a:12} + \varphi_{2b:12} \leq 0 \tag{2.11h}$$

$$\varphi_{1b:12} + \varphi_{2a:12} \leq 0 \tag{2.11i}$$

$$\varphi_{1b:12} + \varphi_{2b:12} \leq \top \tag{2.11j}$$

$$\dots$$

$$\varphi_{6b:67} + \varphi_{7a:67} \leq 0 \tag{2.11k}$$

$$\varphi_{6b:67} + \varphi_{7b:67} \leq \top \tag{2.11l}$$

In this example, Π_1 corresponds to costs moved from c_1 to $c_\varnothing$. If $\varphi_{1b:12}$ is negative it corresponds to costs moved from $c_1(b)$ to c_{12}, similarly $\varphi_{1b:13}$ corresponds to costs moved from $c_1(b)$ to c_{23}. We can see that equation 2.11c enforces that we do not move more costs from $c_1(b)$ than it is available. ■

2.4 Soft Local Consistency Algorithms

Similarly to local consistency for CSP [Rossi *et al.* 2006], soft local consistency (SLC) reasons on a local level by considering only a subset of cost functions. Those algorithms are designed to remove the locally inconsistent values and produce a local maximum corresponding to a dual feasible solution of the (Local Polytope).

Each dual feasible solution can be translated into a set of EPTs increasing $c_\varnothing$ by the cost of the dual solution. For clarity, we directly present the properties which define each SLC and the sequence enforcing those properties. Once again, we need to find a good trade-off between the strength of propagation (how much it increases $c_\varnothing$) and the complexity of its enforcement. We first introduce a simple and natural SLC: *node consistency*.

Definition 2.3 (Node Consistency (NC) [Larrosa 2002b]). *A WCSP $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top)$ is node consistent if for any variable $x_i \in \boldsymbol{X}$.*

1. $\forall a \in \boldsymbol{D}_i, c_i(a) + c_\varnothing < \top$.
2. $\exists a \in \boldsymbol{D}_i$ *such that* $c_i(a) = 0$.

Node consistency can be enforced by iterating over the unary cost functions, removing the values having a cost greater than $\top$ and applying $unaryProject$ when $\min_{a \in \boldsymbol{D}_i} c_i(a) > 0$. The complexity of this algorithm is $O(nd)$ in time and space [Larrosa 2002b]. If a domain becomes empty, then the WCSP is infeasible. Similarly, we can verify the existence of a zero-cost tuple in every cost function.

Definition 2.4 (∅-Inverse Consistency (∅IC) [Zytnicki *et al.* 2009]). *A WCSP P is ∅-Inverse Consistent if for every cost function $c_{\boldsymbol{S}} \in \boldsymbol{C}$ there exists a tuple $\tau \in \ell(\boldsymbol{S})$ such that $c_{\boldsymbol{S}}(\tau) = 0$.*

∅IC can be obtained by iterating over the cost functions $c_{\boldsymbol{S}} \in \boldsymbol{C}$ and applying $project(c_\varnothing, c_{\boldsymbol{S}}, \emptyset, \min_{\tau \in \ell(\boldsymbol{S})} c_{\boldsymbol{S}}(\tau))$ if $\min_{\tau \in \ell(\boldsymbol{S})} c_{\boldsymbol{S}}(\tau) > 0$. We strengthen the previous definition to take into account unary costs.

Definition 2.5 (Full ∅-Inverse Consistency ($F\emptyset IC$)). *A WCSP is Full ∅-Inverse Consistent if for every cost function $c_{\boldsymbol{S}} \in \boldsymbol{C}$ there exists $\tau \in \ell(\boldsymbol{S})$ such that $c_{\boldsymbol{S}}(\tau) + \sum_{x_j \in \boldsymbol{S}} c_j(\tau[x_j]) = 0$.*

F∅IC can be obtained by first extending the cost from the unary cost functions to $c_{\boldsymbol{S}}$ and then applying $project(c_\varnothing, c_{\boldsymbol{S}}, \emptyset, \min_{\tau \in \ell(\boldsymbol{S})} c_{\boldsymbol{S}}(\tau))$.
In practice, F∅IC is only employed on specific global constraints where enforcing a higher level of consistency is too demanding (see chapter 3). Otherwise, a good strategy to increase $c_\varnothing$ is to move costs from the higher arity constraint to the unary cost functions. For this purpose, we now introduce SLC based on *Arc Consistency* and the notion of *support*. Soft AC algorithms were particularly used to solve binary CFNs, most of them are defined only in this context. From now on, we suppose NC is enforced before employing the different Soft AC algorithms.

Definition 2.6 (Simple support). *Let $c_{x,y}$ be a cost function, a simple support of value $a \in \boldsymbol{D}_x$ for the function $c_{x,y}$ is a value $b \in \boldsymbol{D}_y$ such that $c_{x,y}(a, b) = 0$.*

Definition 2.7 (Soft Arc Consistency (SAC) [Cooper & Schiex 2004]). *A value (x, a) is Soft Arc Consistent if it has a simple support on every cost function $c_{\boldsymbol{S}} \in \boldsymbol{C}$ with $x \in \boldsymbol{S}$. A variable is SAC if all its values are SAC. A WCSP is SAC if all its variables are SAC. A WCSP is SAC* if it verifies SAC and NC.*

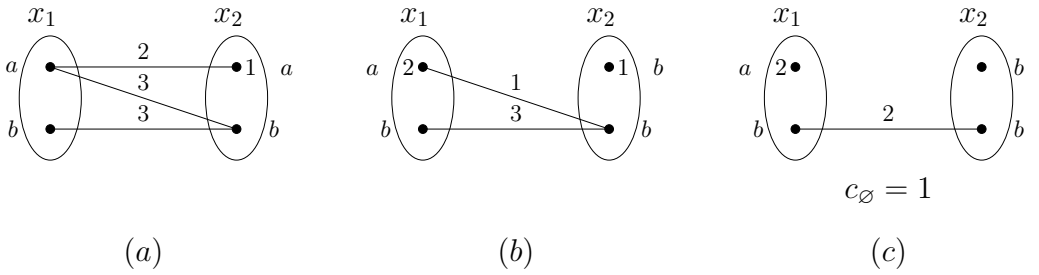


Figure 2.6: Establishing SAC on the CFN (a). First establish SAC on $x_1 = a$ (b) then on $x_2 = b$, enforcing node consistency on x_2 increase $c_\varnothing$ (c)

Soft Arc Consistency can be achieved by iterating over each pair of (value, cost function), if a value (x, a) has no support on $c_{\boldsymbol{S}}$ then it means that for every tuple $\tau \in \ell(\boldsymbol{S})$ with $\tau_x = a, c_{\boldsymbol{S}}(t) > 0$, therefore a positive cost can be moved to $c_x(a)$. After this cost move, NC might need to be re-enforced.

Definition 2.8 (Full Support)**.** *Let $c_{x,y}$ be a cost function, a full support of value $a \in \boldsymbol{D}_x$ for the function $c_{x,y}$ is a value $b \in \boldsymbol{D}_y$ such that $c_{x,y}(a, b) + c_y(b) = 0$.*

Definition 2.9 (Full Arc Consistency (FAC) [de Givry *et al.* 2005])**.** *A value (x, a) is Full Arc Consistent if it has a full support on every cost function $c_{\boldsymbol{S}} \in \boldsymbol{C}$ with $x \in \boldsymbol{S}$. A variable is FAC if all its values are FAC. A WCSP is FAC if all its variables are FAC. A WCSP is FAC* if it verifies FAC and NC.*

Clearly, if a WCSP is FAC then it is also SAC, hence FAC is strictly stronger than SAC. To apply FAC we need to find a full support for every value on every constraint, if we have a binary constraint $c_{x,y}$ and we want to find a full support for (x, a) we need to find $b \in \boldsymbol{D}_y$ s.t $c_{x,y}(a, b) = 0$ and $c_y(b) = 0$. The strategy is to first choose a value from $\boldsymbol{D}_y$ such that $c_{x,y}(a, b) \neq \top$, commonly, we choose the value (y, b) minimizing $c_y(b) + c_{x,y}(a, b)$. Then we transfer the unary cost $c_y(b)$ to the constraint $c_{x,y}$, it sets $c_y(b) = 0$. Then applying the EPT project from $c_{x,y}$ to (x, a) sets $c_{x,y}(a, b) = 0$.
Unfortunately, it is impossible to enforce FAC on every WCSP as shown in figure 2.7. Indeed, when finding a full support for value $x_1 = 1$ we break the full support of $x_2 = 1$, and vice versa when enforcing FAC on $x_2 = 1$. To avoid this issue, Larossa and Schiex [Larrosa & Schiex 2003] added an ordering to the variables to push the costs always in the same direction.

Definition 2.10 (Directional Arc Consistency (DAC) [Larrosa & Schiex 2003])**.** *(x_i, a) is Directed Arc Consistent if it has a full support for every cost function c_{ij} with $i < j$. A variable is DAC if all its values are DAC. A WCSP is DAC if all its variables are DAC. A WCSP is DAC* if it verifies DAC and NC.*

DAC aims to find full support for every variable, but only on a particular subset of cost functions determined by the ordering. Larrosa and Schiex proposed an

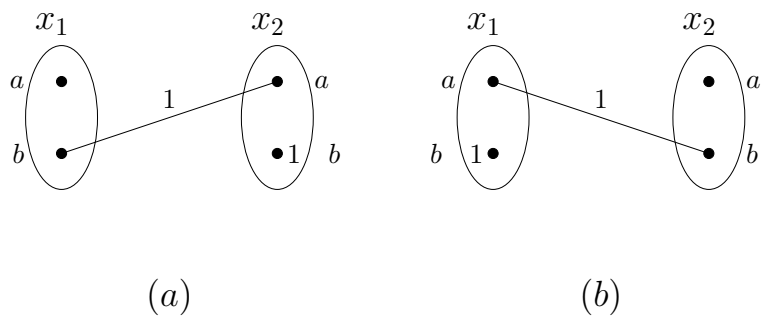


Figure 2.7: Case where establishing FAC is impossible

algorithm for a WCSP with binary cost functions [Larrosa & Schiex 2003], its complexity is $O(ed^2)$ in time and $O(ed)$ in space, e is the number of distinct binary cost functions, and d is the maximum domain size. They also propose a definition and an algorithm for a stronger consistency associating AC with DAC, it is Full Directed Arc Consistency :

Definition 2.11 (Full Directed Arc Consistency (FDAC) [Larrosa & Schiex 2003])**.** *A WCSP is FDAC if it verifies DAC and AC. It is FDAC* if it verifies FDAC and NC.*

The complexity of Larrosa and Schiex algorithm [Larrosa & Schiex 2003] is $O(end^3)$ in time and $O(ed)$ in space. Another way to get closer to FAC is described in [de Givry *et al.* 2005], it is Existential Arc Consistency.

Definition 2.12 (Existential Arc Consistency (EAC))**.** *A variable x is EAC if and only if it admits at least one value a such that $c_x(a) = 0$ and there exists a full support of (x, a) for every cost function c_{xy}. A WCSP is EAC if all its variables are EAC.*

In FAC we wanted to find full support for each value on all the cost functions, in EAC we just want to have at least one value with full support. This property can be associated with the precedent one, to define Existential Directed Arc Consistency.

Definition 2.13 (Existential Directed Arc Consistency (EDAC) [de Givry *et al.* 2005])**.** *A WCSP is EDAC if it verifies FDAC and EAC. It is EDAC* if it verifies EDAC and NC.*

The algorithm enforcing EDAC has a time complexity of $O(ed^2max(nd, \top))$ and $O(ed)$ in space. EDAC proposes a good trade-off between strength of propagation and complexity, and it is currently the default algorithm in the WCSP solver Toulbar2[2].

As mentioned before, Optimal Soft Arc Consistency (OSAC), can be obtained by solving the dual of the (Local Polytope). In order to enforce OSAC we need to

[2] https://github.com/toulbar2/toulbar2

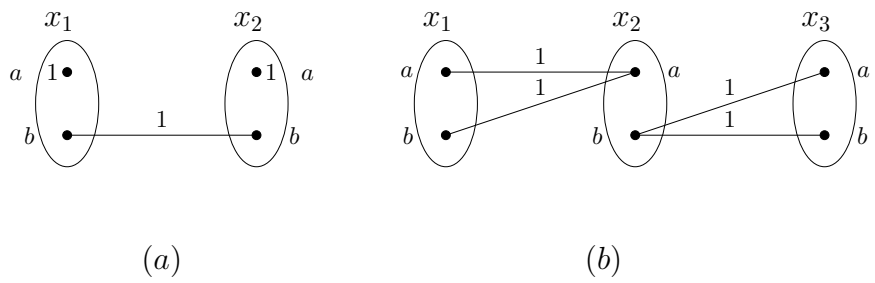


Figure 2.8: CFN (a) is SAC but is not $F\emptyset IC$, enforcing $F\emptyset IC$ increases$c_\varnothing$ by 1. CFN (b) is $F\emptyset IC$ but not SAC, enforcing SAC increases $c_\varnothing$ by 1.

relax the non-negativity condition on the EPTs, meaning that one individual EPT might create a negative cost, but the full sequence of EPTs must give a valid WCSP.

Definition 2.14 (Optimal Soft Arc Consistency (OSAC) [Cooper *et al.* 2010])**.** *A WCSP is optimal soft arc consistent if there exists no sequence of EPTs increasing* $c_\varnothing$*.*

In the next section, we will provide a detailed introduction to Virtual Arc Consistency (VAC), as it holds significant relevance with several contributions of this thesis. Finally, we study the relation between the different local consistencies.

Definition 2.15. *Let* ϕ *and* ψ *be two soft local consistencies. We say that* ϕ *implies* ψ *if all WCSP verifying* ϕ *also verifies* ψ*.*

For example FDAC implies DAC and AC. We can also compare the SLC according to the value of $c_\varnothing$. To fairly compare the SLC depending of an ordering like DAC or EDAC, we suppose they are always applied with the *best possible configuration*, meaning that there exists no ordering producing a better $c_\varnothing$.

Definition 2.16. *Let* ϕ *and* ψ *be two soft local consistencies. We say that* ϕ *is* $c_\varnothing$*-superior than* ϕ*, if for every WCSP where* ϕ *has been enforced with its best configuration, enforcing* ψ *doesn't increase* $c_\varnothing$*.*

For example, the author of [Cooper *et al.* 2010] has shown that VAC produces better bounds than EDAC, while there exists an instance verifying VAC but not EDAC. If ϕ is not $c_\varnothing$*-superior* to ψ and vice versa then we say that the two SLC are $c_\varnothing$*-incomparable*. For example, figure 2.8 depicts 2 CFNs one where $F\emptyset IC$ gives a better lower bound than SAC, and one where SAC gives a better lower bound than $F\emptyset IC$, therefore the two properties are $c_\varnothing$*-incomparable*. Figure 2.9 [Dehani 2014] (French) and table 2.1 recapitulate the strength of SLC algorithms and their complexity. Notice that soft AC-based algorithms are defined for binary CFNs while VAC and OSAC can be applied to CFNs with any arity, therefore for a fair comparison, we give the time complexity to enforce the different SLC on a binary WCSP only.

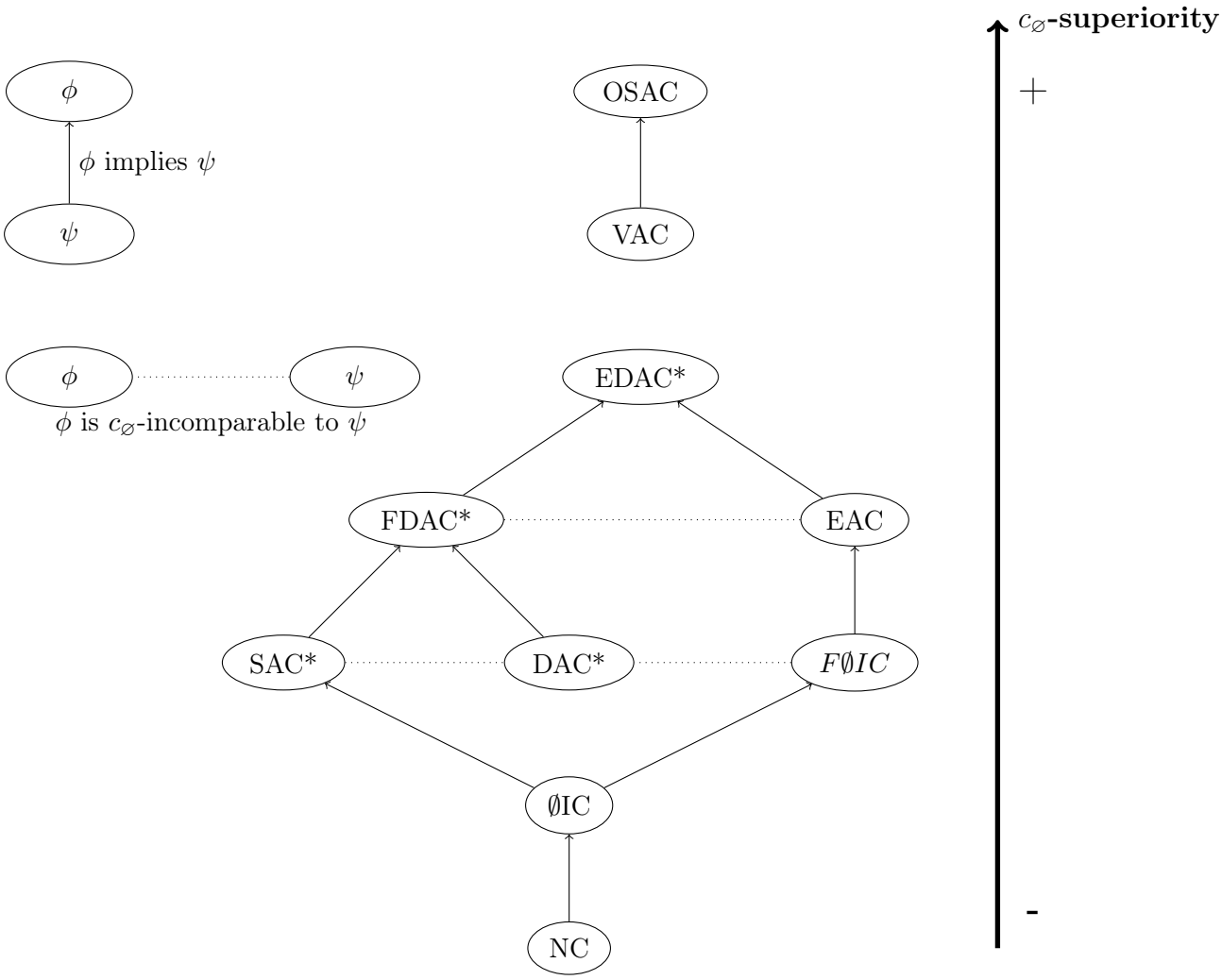


Figure 2.9: Relation between the different level of local consistency

In practice, OSAC is the algorithm giving the best lower bound, but its complexity doesn't make it usable in large problems. A viable strategy is to change the level of local consistency during the search, for example, using VAC only in prepossessing and then using EDAC during search has been proven worthwhile on different instances [Hurley *et al.* 2016].

Soft Global Cost Functions

Global constraints play a major role in the modeling flexibility of the CP paradigm. Research has been pursued to include those global constraints in CFN, either as hard global constraints, in this case, we want to deduce a lower bound when considering the global constraint plus the associated unary costs. Or as soft global constraints where a *cost measure* evaluates the cost of a given tuple depending on

Algorithm	Time Complexity
NC [Larrosa 2002b]	$O(nd)$
$\emptyset$IC [Zytnicki *et al.* 2009]	$O(ed^2)$
$F\emptyset IC$ [Montalbano *et al.* 2022]	$O(ed^2)$
SAC [Larrosa 2002b]	$O(n^2d^2 + ed^3)$
DAC [Larrosa & Schiex 2003]	$O(ed^2)$
FDAC [Larrosa & Schiex 2003]	$O(end^3)$
EDAC [de Givry *et al.* 2005]	$O(ed^2max(nd, \top))$
VAC_ε [Cooper *et al.* 2010]	$O(ed^2\top/\varepsilon)$
OSAC [Cooper *et al.* 2010]	$O(e^{4.5}d^{5.5}logM)$

Table 2.1: Time complexity to enforce different levels of local consistency on a binary WCSP. Where n is the number of variables, d the maximal domain size, e the number of binary cost functions, r the maximal arity and M the maximal finite cost.

how it violates the global constraint. Some global cost functions are decomposable in well-organized networks of cost functions of bounded arity [Allouche *et al.* 2012, Allouche *et al.* 2016], on which applying usual SAC algorithms is equivalent to a direct application on the original global cost function. Otherwise, depending on the global cost function a chosen level of soft local consistency can be efficiently enforced [Lee & Leung 2009, Lee & Leung 2012].

2.4.1 Virtual Arc Consistency

Virtual Arc Consistency produces high-quality bounds but is quite expensive to enforce. It relies on a particular CSP that can be derived from a WCSP instance.

Definition 2.17 (Bool(P) [Cooper *et al.* 2010])**.** *Let P $\langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top \rangle$ be a WCSP.* Bool(P) $\langle \boldsymbol{X}, \overline{\boldsymbol{D}}_i, \overline{\boldsymbol{C}} \rangle$ *is a CSP such that $\overline{\boldsymbol{D}}$ contains all the values $a \in \boldsymbol{D}_i$, $x_i \in \boldsymbol{X}$ with $c_i(a) = 0$. And $\overline{c}_{\boldsymbol{S}} = \langle \boldsymbol{S}, \overline{r}_{\boldsymbol{S}} \rangle \in \overline{C^+}$ if $c_{\boldsymbol{S}} \in \boldsymbol{C}$ and $\forall \tau \in \ell(\boldsymbol{S})\, \overline{r}_{\boldsymbol{S}}(\tau) = 0$ if $c_{\boldsymbol{S}}(\tau) = 0$, otherwise $\overline{r}_{\boldsymbol{S}}(\tau) = \top$.*

In simpler terms, Bool(P) allows exactly the tuples and values having a zero cost (see figure 2.10). Hence, we can observe that if Bool(P) is feasible it means the optimal cost of P is $c_\varnothing$, otherwise if Bool(P) is infeasible it means $c_\varnothing$ can be increased by a positive amount. Applying local consistency algorithms on Bool(P) to detect inconsistencies defines new levels of soft local consistency for P.

Definition 2.18 (Virtual Arc Consistency [Cooper *et al.* 2010])**.** *A WCSP P is virtual arc consistent if the (generalized) arc consistency closure of the CSP* Bool(P) *is non-empty.*

If the arc consistency closure is empty, it has been shown that it is possible to increase $c_\varnothing$ by a positive amount.

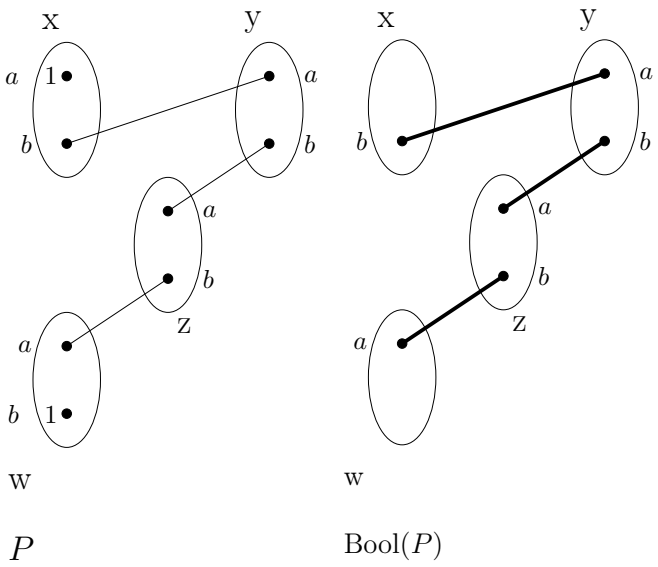


Figure 2.10: Example of P and Bool(P).

Theorem 2.1 ([Cooper *et al.* 2010]). *Let P be a WCSP such that $c_\varnothing < \top$. Then there exists a sequence of EPTs which when applied to P leads to an increase in $c_\emptyset$ if and only if the arc consistency closure of* Bool(P) *is empty.*

In practice, we need the sequence of AC operations proving the infeasibility of Bool(P) to increase $c_\varnothing$ in P. Hence, it is not wanted to completely solve the CSP defined by Bool(P), both because it is an NP-hard problem and because we could not use this information to increase $c_\varnothing$. The algorithm to enforce VAC can be decomposed into 3 phases:

1. Establish (G)AC on Bool(P). If no conflict occurred, then quit.
2. Given σ a sequence of arc consistency operations which led to a conflict, find a minimal subsequence of σ which provokes the conflict. Convert this subsequence into EPTs and produce the maximum achievable increase λ of $c_\varnothing$ while keeping all costs non-negative.
3. Apply the sequence of EPTs and go back to phase 1.

Running VAC on a small example will help have a better understanding. Phase 1 corresponds to establishing AC in figure 2.10 Right. We find directly that value $y = a$ has no support on $c_{x,y}$ and $z = b$ has no support on $c_{z,w}$, those values can be removed. Consequently, $y = b$ has no support on $c_{y,z}$ and a domain wipe-out occurs at variable y. Phase 2 trace-back those operations to derive a sequence of EPTs increasing $c_\varnothing$ by a cost λ (figure 2.11). We know the last EPT we want to perform is $unaryProject(y, \lambda)$, to perform this EPT values (y, a) and (y, b) *request* a cost λ. Therefore, we start the algorithm with a cost λ on $c_y(a), c_y(b)$ fig.2.11(a). Then we follow the operations made in Phase 1 to deduce from which cost function

we could take a cost λ to fulfill the requests of (y, a) and (y, b). The reason (y, a) has been removed is $c_{x,y}$, we deduce we can obtain $c_y(a) = \lambda$ by projecting a cost from $c_{x,y}$ to (y, a). To perform the projection, a cost λ is requested by all the tuples of $c_{x,y}$ verifying $y = a$, similarly, a cost λ is requested by all the tuples of $c_{y,z}$ verifying $y = b$ fig.2.11(b). Both $c_{x,y}(b, a)$ and $c_{y,z}(b, a)$ have a cost > 0 in P, their requests are already fulfilled and do not need to be traced-back. On the other hand, $c_{x,y}(a, a)$ and $c_{y,z}(b, b)$ have been removed because of $c_x(a)$ and $c_z(b)$, a cost λ is requested from (x, a) and (z, b) fig.2.11(c). The algorithm continues to trace back values removal necessary to explain the domain wipe-out fig.2.11 (d) and (e). Note that one value/tuple can participate in multiple value removal, in this case it will request a cost $k \times \lambda$, where k is the number of times the value participated in a value removal. Once the algorithm is done, every value/tuple in figure 2.11 (e) associated with a cost has a positive unary/binary cost in P. If a value (x_i, a) with $c_i(a) > 0$ requests $k\lambda$, then to keep non-negative cost while satisfying all the requests made to (x_i, a), we need $\lambda \leq \frac{c_i(a)}{k}$. We choose λ to be the maximal cost satisfying all the requests. Here we have $\lambda = 1$. Finally, phase 3 performs the actual sequence of EPTs to increase $c_\emptyset$ by λ (fig.2.12). Those 3 phases correspond to one iteration of VAC, we repeat those operations until Bool(P) is GAC.

- Initial $WCSP$ is in figure 2.12.(a)
- $extend(c_x, a, c_{x,y}, 1)$
- $extend(c_w, b, c_{z,w}, 1)$
- $project(c_y, c_{x,y}, a, 1)$
- $project(c_z, c_{z,w}, b, 1)$
- The obtained $WCSP$ is in figure 2.12.(b)
- $extend(c_z, b, c_{y,z}, 1)$
- $project(c_y, c_{y,z}, b, 1)$
- $unaryProject(y, 1)$
- Final $WCSP$ verifying VAC is in figure 2.12.(c).

VAC can be used to enforce tight bounds, however, the number of iterations made by VAC can be unbounded. Indeed, in phase 2 λ is obtained by dividing a cost by the number of requests, if this number is high then λ will be very small ($<$1). There exist cases where the algorithm will iteratively increase $c_\varnothing$ by a very small amount ($<$1) and never terminate. To avoid this situation, it is possible to add a threshold ε, if more than a given number of iterations does not improve $c_\varnothing$ by more than ε then VAC stops. To not discover the same conflict twice, whenever an increase of $c_\varnothing$ lower than ε is detected then the unary/binary cost responsible for fixing λ is ignored for the next iterations. This defines VAC$_\varepsilon$. A complementary heuristic is to also add

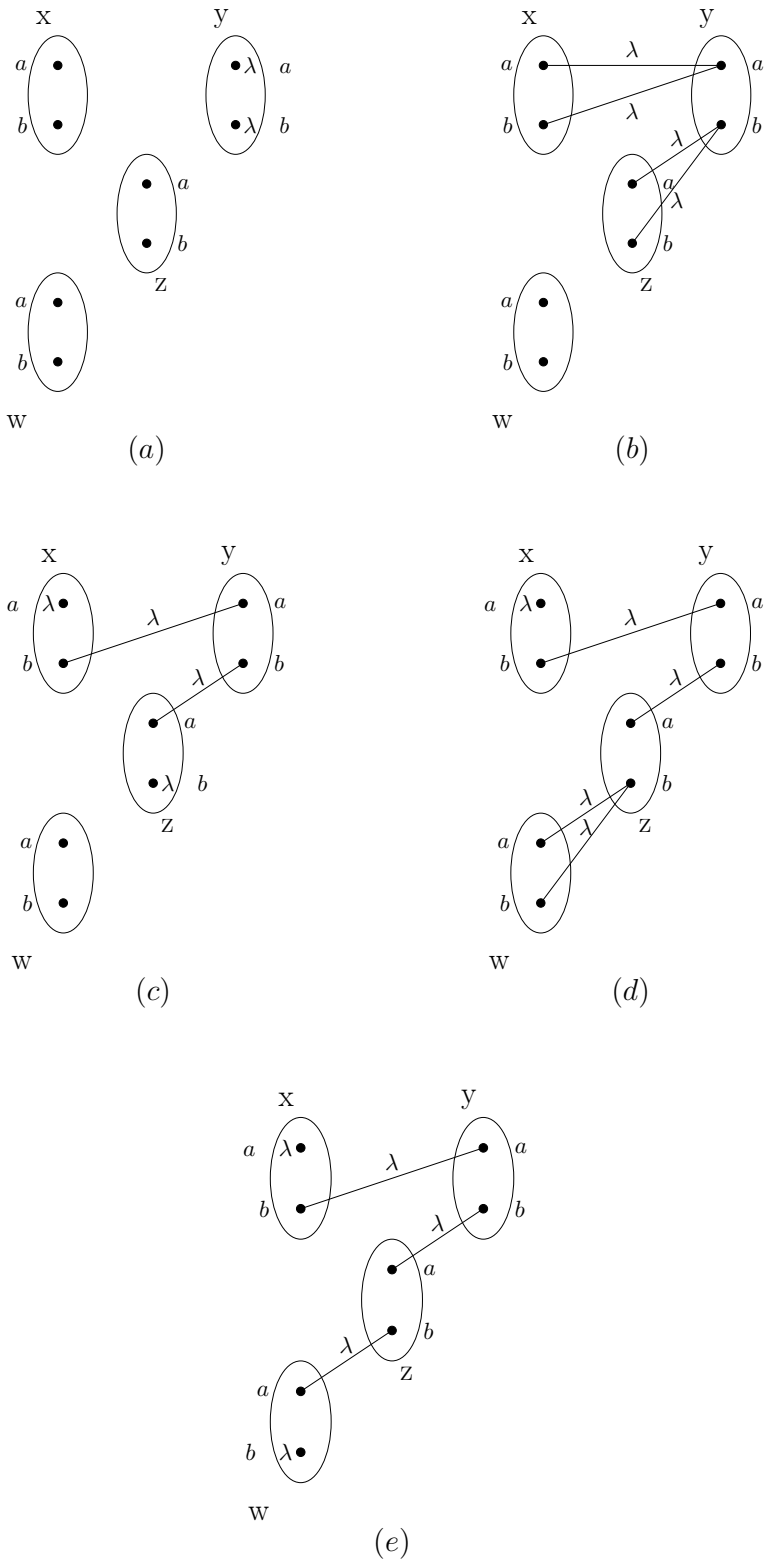


Figure 2.11: VAC phase 2

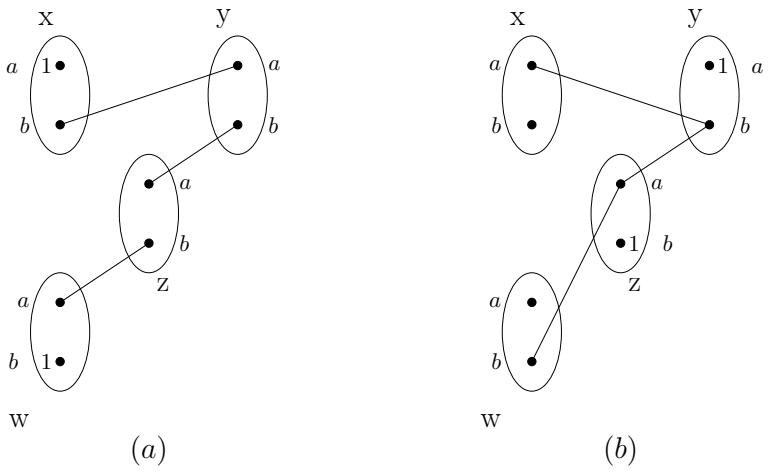


Figure 2.12: VAC phase 3

a threshold θ when creating Bool(P). Only the values/tuples with a cost greater than θ are considered forbidden in Bool(P), the others are allowed. If θ is high then if VAC discovers a conflict, it has more chance to be a large cost contribution. If θ is low then more tuples are forbidden in Bool(P) and VAC has more chance to discover a conflict but with a possibly small contribution. The heuristic consists of first applying VAC with a high θ in the hope of discovering large cost contributions and decreasing θ along the iterations.

2.5 SAT, Pseudo-Boolean Optimization, and Conflict-Based Learning

2.5.1 SAT

The Boolean satisfiability problem, commonly referred to as SAT, is fundamental and well-studied in computer science and artificial intelligence. It is defined by a set of Boolean variables, and a *formula*, where the formula is a conjunction of clauses, each clause is a disjunction of a *literals*, and a literal is either a variable itself (x) or the negation of a variable ($\overline{x}$). The objective is to find an assignment of the variables satisfying the formula (satisfying all the clauses), if one exists then the problem is said satisfiable else it is unsatisfiable.

Example 2.13. *We can model our KPCG instance (without objective function) using SAT . We first give the SAT modeling of the conflict graph:*

$$C_1 : \overline{x_1} \vee \overline{x_2}$$
$$C_2 : \overline{x_1} \vee \overline{x_3}$$
$$C_3 : \overline{x_2} \vee \overline{x_3}$$
$$C_4 : \overline{x_4} \vee \overline{x_5}$$
$$C_5 : x_6 \vee \overline{x_7}$$
$$C_6 : \overline{x_6} \vee x_7$$

Concerning the linear constraint, a direct encoding would need to explicitly give all the tuples forbidden by the linear constraints. However, there exist various ways to represent linear constraints in SAT: Binary Decision Diagrams, Multi-valued Decision Diagrams or sorting networks see [Abío & Stuckey 2014] for a survey. ■

The most basic approach to solve SAT is the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [Davis *et al.* 1962], it visits the search space using a rooted tree. It iteratively chooses a branching literal and assigns it to true, if all the clauses are satisfied then the obtained problem is satisfiable, the remaining variables can be assigned to true or false and the algorithm ends. If at least one clause is unsatisfiable, then the solver tries to assign the branching literal to false. Otherwise, the search continues by selecting a new branching literal. To enhance the efficiency of the search, the solver performs a simple inference rule called *Unit Propagation* (UP):

whenever a clause is not satisfied and contains only one non-assigned literal then this literal is assigned to true, we say that the literal has been propagated by UP. When a literal is propagated, another clause may become unit, and UP will deduce another assignment. This can be represented as a *trail*. The trail is a chronological record of all the literal assignments that have been made during the solving process. It keeps track of the order in which literals were assigned and their assigned values (true or false). For example,$\rho = \{x_1 = True, x_2 = False, x_3 = False\}$ indicates that x_1 is the first assigned literal followed by x_2 and x_3. The trail also records a reason motivating why the solver assigned each literal. It can be a decision made by DPLL, noted as 'd' in the trail, or a consequence of unit propagation, noted with the last propagated clause (see example 2.14). Each time a decision literal is assigned, it is associated with a decision level, the current decision level being the last known decision level. The trail allows the solver to know which literal has been assigned in which decision level. In particular, we will be interested in the literals assigned during the current decision level, which corresponds to all literals assigned after the last decision. Infeasibility is detected if UP leads to an empty clause.

Example 2.14. *Following example 2.13, if we assign $x_1 = True$ then x_2, x_3 must be set to false to satisfy constraints C_1, C_2. This partial assignment also verifies C_3. The corresponding trail would be $\rho = \{x_1 \overset{d}{=} True, x_2 \overset{C_1}{=} False, x_3 \overset{C_2}{=} False\}$* ■

Another approach to obtain a proof of unsatisfiability is to use the resolution proof system, which aims to associate the information embedded in two clauses to derive a new clause, this operation is repeated until an empty clause is derived. The resolution proof system is sound and complete, but it is not directly used in practice, however, it plays a major role in the well-known Conflict Directed Clause Learning (CDCL) algorithm [Marques-Silva & Sakallah 1999]. The resolution proof system relies on the resolution rule

$$\frac{v \vee \bigvee_{i=1}^{n} l_i \quad \overline{v} \vee \bigvee_{j=1}^{m} l'_j}{\bigvee_{i=1}^{n} l_i \vee \bigvee_{j=1}^{m} l'_j} \tag{2.12}$$

By iteratively applying the resolution rule, the proof system attempts to derive an empty clause, indicating a contradiction and hence proving the unsatisfiability of the original formula. CDCL is an extension of the DPLL algorithm, the idea is to learn a constraint after a *conflict* (when a clause is falsified). The learned clause should prevent the solver from discovering this same conflict again. In the context of SAT, when a conflict occurs, the solver starts *conflict analysis*. It identifies two clauses, the first one is the conflicting clause C_{conf}, which corresponds to the falsified clause under the current assignment. We know that a clause is falsified only if all its literals are falsified, as we are enforcing UP this is possible only if one or more literals in C_{conf} have been falsified at the current decision level (by enforcing UP on another clause or branching decision at this level). We can use the trail to trace back the last literal from C_{conf} to be falsified and which clause is the reason for this propagation, this gives the second clause, the reason clause C_{reason}. If v was the last falsified

literal then we know that v appears in C_{conf} and $\overline{v}$ appears in C_{reason}, so we can apply the resolution rule 2.12 on the two clauses. The result is a clause where v does not appear, this clause is still conflicting under the current assignment, we update C_{conf} and identify using the trail a new clause C_{reason} and apply the resolution rule. We repeat those operations until a stopping criterion, the most common one being *First Unique Implication Point* (1-UIP) [Marques-Silva & Sakallah 1999, Moskewicz *et al.* 2001, Audemard *et al.* 2008], basically the operation stops when the resulting clause contains at most one literal falsified at the current decision level.

Example 2.15. *Following example 2.13, we add two clauses to trigger a simple conflict:*

$$C_7 : x_2 \vee x_3 \vee x_4 \vee x_6 \vee x_7$$
$$C_8 : x_2 \vee x_3 \vee x_5 \vee x_6 \vee x_7$$

If we assign $x_1 = True$ then x_2, x_3 must be set to false to satisfy constraints C_1, C_2. If we then assign $x_6 = False$ then C_5 propagates $x_7 = False$, it follows that C_7, C_8 propagates $x_4 = True$, $x_5 = True$ and C_4 is conflicting. The corresponding trail would be $\rho = \{x_1 \overset{d}{=} True, x_2 \overset{C_1}{=} False, x_3 \overset{C_2}{=} False, x_6 \overset{d}{=} False, x_7 \overset{C_5}{=} False, x_4 \overset{C_7}{=} True, x_5 \overset{C_8}{=} True\}$. We set $C_{conflict} = C_4$, $C_{reason} = C_8$ and apply the resolution rule 2.12:

$$\frac{\overline{x_4} \vee \overline{x_5} \quad x_2 \vee x_3 \vee x_5 \vee x_6 \vee x_7}{C_9 : \overline{x_4} \vee x_2 \vee x_3 \vee x_6 \vee x_7}$$

Clause C_9 is conflicting under assignment $x_1 = True, x_2 = False, x_3 = False, x_6 = False, x_7 = False, x_4 = True$, we set $C_{conflict} = C_9$, $C_{reason} = C_7$ and apply the resolution rule 2.12:

$$\frac{\overline{x_4} \vee x_2 \vee x_3 \vee x_6 \vee x_7 \quad x_2 \vee x_3 \vee x_4 \vee x_6 \vee x_7}{C_{10} : \vee x_2 \vee x_3 \vee x_6 \vee x_7}$$

Clause C_{10} is conflicting under assignment $x_1 = True, x_2 = False, x_3 = False, x_6 = False, x_7 = False$, we set $C_{conflict} = C_{10}$, $C_{reason} = C_5$ and apply the resolution rule 2.12:

$$\frac{x_2 \vee x_3 \vee x_6 \vee x_7 \quad x_6 \vee \overline{x_7}}{C_{11} : x_2 \vee x_3 \vee x_6}$$

Clause C_{11} contains only one variable falsified at the current decision level (x_6) we stop the procedure. Observe that with this clause when assigning $x_1 = True$, the solver will automatically deduce $x_6 = True$ by unit propagation. ■

CDCL is now the most common approach to solving SAT problems. On top of this framework, many algorithms, heuristics, and efficient implementation techniques have been developed over the years, which led to the efficiency of current SAT solvers. For more information on SAT see [Biere *et al.* 2021].

2.5.2 Pseudo-Boolean Optimization

Pseudo-Boolean optimization (PBO), is a mathematical framework for solving optimization problems where the objective function and constraints are defined over a set of Boolean variables, which can take values of either 1 (true) or 0 (false). We use the same notation as in SAT, $\overline{x}$ corresponds to the negation of x, and we have the relation $x = 1 - \overline{x}$. PBO extends SAT by incorporating polynomial pseudo-Boolean constraints i.e. weighted combinations of products of Boolean variables and their negations. In this thesis, we restrict ourselves to linear PBO, where the constraints are exclusively weighted combinations of variables and their negation. PBO provides a more expressive framework than SAT; it is known that one PB constraint can represent an exponential number of clauses [Dixon *et al.* 2004]. Moreover, while both SAT and PBO have the potential to benefit from the cutting plane proof system [Gomory 1960], which is theoretically more powerful than the resolution proof system commonly used in SAT, it is in practice more challenging to take advantage of the power of cutting planes in SAT than in PBO due to the native presence of PB constraints in PBO. In particular, a subset of the cutting plane rules can be used to adapt the techniques developed in SAT.

Example 2.16. *Our KPCG instance has only Boolean variables hence its formulation is very similar to ILP, we simply convert it in the usual form with only* $\geq$ *constraints and positive coefficients.*

$$\begin{aligned}
\min\, & 4x_1 + 9x_2 + 4x_3 + 3x_4 + 5x_5 + 7x_6 + 7x_7 \\
& s.t \\
C_1 : & 3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7 \geq 10 \\
C_2 : & \overline{x_1} + \overline{x_2} \geq 1 \\
C_3 : & \overline{x_1} + \overline{x_3} \geq 1 \\
C_4 : & \overline{x_2} + \overline{x_3} \geq 1 \\
C_5 : & \overline{x_4} + \overline{x_5} \geq 1 \\
C_6 : & x_6 + \overline{x_7} \geq 1 \\
C_7 : & x_7 + \overline{x_6} \geq 1 \\
& x_i \in \{0, 1\}, \quad i = 1, \ldots, 7
\end{aligned}$$

■

There exist several ways to solve a PBO problem. One is to transform the pseudo-Boolean constraints into a SAT formula and run CDCL [Eén & Sörensson 2006, Gebser *et al.* 2012, Martins *et al.* 2014, Sakai & Nabeshima 2015]. Another way is to work natively on linear constraints [Chai & Kuehlmann 2003, Dixon & Ginsberg 2002, Elffers & Nordström 2018, Le Berre & Parrain 2010, Sheini & Sakallah 2006], in this case, they behave similarly to SAT solvers and notably CDCL has been extended to a pseudo-Boolean formulation and will be discussed in the next

section. The basic inference rule to *propagate* the constraints is *Domain Propagation*. For each constraint it is possible to detect whether a given literal needs to be set to true by looking at the *slack* of the constraint. The slack measures how far a current partial assignment is from falsifying a constraint. Given a partial assignment τ and a linear constraint $C_1 = \sum_{i=1}^{n} \alpha_i l_i \geq b$, the slack is:

$$slack(C_1, \tau) = \sum_{\substack{i=1 \\ l_i \text{not falsified by } \tau}}^{n} \alpha_i - b \tag{2.13}$$

If a literal l_i verifies $\alpha_i > slack(C_1, \tau)$, then l_i must be assigned to true as a negative slack corresponds to a conflicting constraint. When a literal is assigned, we need to verify the slack of all the constraints having the given literal in its scope. Once again, the different assignments can be represented as a trail, reporting sequentially when and why a literal has been assigned.

Example 2.17. *Following example 2.16, if we assign $x_1 = 1$, we obtain the following slack:*

- $slack(C_1, \{x_1 = 1\}) = 3 + 5 + 3 + 3 + 5 + 5 + 5 - 10 = 19$
- $slack(C_2, \{x_1 = 1\}) = 0$ *it follows that* $x_2 = 0$
- $slack(C_3, \{x_1 = 1\}) = 0$ *it follows that* $x_3 = 0$
- $slack(C_4, \{x_2 = 0, x_3 = 0\}) = 1$ *the constraint is not conflicting.*
- $slack(C_1, \{x_1 = 1, x_2 = 0, x_3 = 0\}) = 11$

The corresponding trail is $\rho = \{x_1 \overset{d}{=} 1, x_2 \overset{C_2}{=} 0, x_3 \overset{C_3}{=} 0\}$ ■

It is possible to simulate a similar behavior to CDCL in PBO, the conflicting constraint $C_{conflict}$ is a constraint having a negative slack, once again we use the trail to find the last propagated literals and derive a reason constraint C_{reason}. We will use the following rules to resolve this conflict.

$$\frac{\sum_{i=1}^{n} \alpha_i l_i \geq b}{\sum_{i=1}^{n} \min(\alpha_i, b) l_i \geq b} (saturation) \tag{2.14}$$

$$\frac{\alpha l + \sum_{i=1}^{n} \alpha_i l_i \geq b, \quad \beta \bar{l} + \sum_{i=1}^{n'} \beta l'_i \geq b' \quad \rho, \rho' \in \mathbf{N}^*, \rho\alpha = \rho'\beta}{\sum_{i=1}^{n} \rho\alpha_i l_i + \sum_{i=1}^{n'} \rho'\beta l'_i \geq \rho b + \rho' b' - \rho\alpha} (cancellation) \tag{2.15}$$

$$\frac{\alpha l + \sum_{i=1}^{n} \alpha_i l_i \geq b}{\sum_{i=1}^{n} \alpha_i l_i \geq b - \alpha} (weakening) \tag{2.16}$$

The saturation rule is always worth applying as it reduces the size of the coefficients and may help derive better bounds when we use an LP solver. The cancellation

rule serves the same purpose as the resolution rule in SAT. Unfortunately, in this context if we apply the cancellation rule on $C_{conflict}$ and C_{reason} we obtain a possibly non-conflicting constraint under the current assignment, meaning that learning this constraint will not prevent the solver from discovering this conflict again. Chai and Kuehlmann [Chai & Kuehlmann 2003] show that the weakening rule will help to obtain a conflicting constraint after the cancellation rule. The most basic strategy is to weaken one by one the satisfied or unassigned literals of C_{reason}, until the cancellation leads to a conflicting constraint, after each weakening the saturation rule can be applied. Finally, the resulting conflicting constraint becomes $C_{conflict}$ and the process is repeated.

Example 2.18. *Following example 2.16, suppose we assign $x_1 = 1$, it follows that $x_2 = 0, x_3 = 0$, the current trail is $\rho = \{x_1 \overset{d}{=} 1, x_2 \overset{C_2}{=} 0, x_3 \overset{C_3}{=} 0\}$. Suppose the next decision is $x_6 = 0$ then C_6 propagates $x_7 = 0$ it follows $slack(C_1, \{x_1 = 1, x_2 = 0, x_3 = 0, x_6 = 0, x_7 = 0\}) = 1$ enforcing that $x_4 = 1$, $x_5 = 1$, finally the slack of C_5 is negative and we reached a conflict. The trail is $\rho = \{x_1 \overset{d}{=} 1, x_2 \overset{C_2}{=} 0, x_3 \overset{C_3}{=} 0, x_6 \overset{d}{=} 0, x_7 \overset{C_6}{=} 0, x_4 \overset{C_1}{=} 1, x_5 \overset{C_1}{=} 1\}$ and we have $C_{conflict} = C_5$ and $C_{reason} = C_1$. We first apply the cancellation rule 2.15 on x_5*

$$\frac{5x_5 + 3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_6 + 5x_7 \geq 10 \quad \overline{x_5} + \overline{x_4} \geq 1}{C_8 : 3x_1 + 5x_2 + 3x_3 + 2\overline{x_4} + 5x_6 + 5x_7 \geq 7}$$

We observe that $slack(C_8, \{x_1 = 1, x_2 = 0, x_3 = 0, x_6 = 0, x_7 = 0, x_4 = 1\}) = -4 < 0$, the constraint is conflicting. We set $C_{conflict} = C_8$ and continue. The next variable in the trail is $x_4 = 1$, we set $C_{reason} = C_1$.

$$\frac{3x_1 + 5x_2 + 3x_3 + 2\overline{x_4} + 5x_6 + 5x_7 \geq 7 \quad 5x_5 + 3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_6 + 5x_7 \geq 10}{C_9 : 15x_1 + 25x_2 + 15x_3 + 10x_5 + 25x_6 + 25x_7 \geq 35}$$

When dividing by 5 we obtain $C_9 : 3x_1 + 5x_2 + 3x_3 + 2x_5 + 5x_6 + 5x_7 \geq 7$ We observe that $slack(C_9, \{x_1 = 1, x_2 = 0, x_3 = 0, x_6 = 0, x_7 = 0\}) = -2 < 0$, the constraint is conflicting. Next variable is x_7, we set $C_{reason} = C_6$

$$\frac{5x_7 + 3x_1 + 5x_2 + 3x_3 + 2x_5 + 5x_6 \geq 7 \quad \overline{x_7} + x_6 \geq 1}{C_9 : 3x_1 + 5x_2 + 3x_3 + 2x_5 + 10x_6 \geq 7}$$

We can apply the saturation rule 2.14 to obtain $C_{10} : 3x_1 + 5x_2 + 3x_3 + 2x_5 + 7x_6 \geq 7$. We observe that $slack(C_9, \{x_1 = 1, x_2 = 0, x_3 = 0, x_6 = 0\}) = -2 < 0$, the constraint is conflicting, only one variable falsified at the current decision variable (x_6), we can stop the conflict analysis. With this new constraint, if we assign $x_1 = 1$ and perform domain propagation, then the solver deduces $x_6 = 1$. ■

Other strategies and rules have been built around this algorithm, for example in Sat4j [3] the unassigned literals of C_{reason} are weakened first. Alternatively, Elffers

[3] http://www.sat4j.org/index.php

and Nordström [Elffers & Nordström 2018] use the division rule instead of the saturation rule and weaken all the non-falsified literals with a coefficient not divisible by the coefficient of the propagated literal.

$$\frac{\sum_{i=1}^{n} \alpha_i l_i \geq b}{\sum_{i=1}^{n} \lceil \alpha_i / c \rceil l_i \geq \lceil b/c \rceil} \text{(division by c)} \tag{2.17}$$

Example 2.19. *Coming back to example 2.18, we now want to derive a constraint using the division rule. Given the trail* $\rho = \{x_1 \stackrel{d}{=} 1, x_2 \stackrel{C_2}{=} 0, x_3 \stackrel{C_3}{=} 0, x_6 \stackrel{d}{=} 0, x_7 \stackrel{C_6}{=} 0, x_4 \stackrel{C_1}{=} 1, x_5 \stackrel{C_1}{=} 1\}$. *We first consider variable* x_5, *we have* $C_{conflict} = C_5$ *and* $C_{reason} = C_1$. *In* C_{reason} *the coefficient of* x_5 *is 5, we first weaken the variables in* C_{reason} *non falsified by* ρ *and with a coefficient not divisible by 5.*

$$C_{reason} \leftarrow 5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7 \geq 7$$

We then divide by 5, *using the division rule 2.17.*

$$\frac{5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7 \geq 7}{x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \geq 2}$$

The coefficient of $\overline{x_5}$ *in* $C_{conflict}$ *is already one. We proceed to the cancellation rule 2.15:*

$$\frac{x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \geq 2 \quad \overline{x_5} + \overline{x_4} \geq 1}{C_8 : x_2 + x_3 + x_6 + x_7 \geq 1}$$

We observe that $slack(C_8, \{x_1 = 1, x_2 = 0, x_3 = 0, x_6 = 0, x_7 = 0, x_4 = 1\}) = -1 < 0$, *the constraint is conflicting. We set* $C_{conflict} = C_8$ *and continue. The next variable in the trail is* $x_4 = 1$, *it doesn't appear in* C_8, *we continue. The next variable is* x_7, *we set* $C_{reason} = C_6$. *The coefficient of* x_7 *and* $\overline{x_7}$ *is one, we proceed to the resolution rule:*

$$\frac{x_2 + x_3 + x_6 + x_7 \geq 1 \quad \overline{x_7} + x_6 \geq 1}{C_9 : x_2 + x_3 + 2x_6 \geq 1}$$

We can apply the saturation rule 2.14 to obtain $C_9 : x_2 + x_3 + x_6 \geq 1$. *We observe that* $slack(C_9, \{x_1 = 1, x_2 = 0, x_3 = 0, x_6 = 0\}) = -1 < 0$, *the constraint is conflicting, only one variable falsified at the current decision variable appears* (x_6), *we can stop the conflict analysis. With this new constraint, if we assign* $x_1 = 1$ *and perform domain propagation, the solver deduces* $x_6 = 1$. ■

2.5.3 Other Conflict Driven Learning

Other paradigms also adopted a conflict-based learning mechanism. We present the main concepts behind some of them.

NoGood recording for CSP [Dechter 1990, Katsirelos & Bacchus 2005]. This method identifies and records specific sets of variable assignments, known as "no-goods" that are found to be infeasible or lead to contradictions during the search.

When a conflict occurs, a nogood is generated, capturing the conflicting variable assignments. These nogoods are stored and utilized to prevent the search algorithm from revisiting the same or similar states in the future

MaxCDCL for MaxSAT [Li *et al.* 2021]. MaxSAT is an optimization version of SAT where the objective is to find an assignment satisfying a set of *hard* clauses and falsifying the minimal number of *soft* clauses. Therefore, a conflict in MaxSAT can occur, if a hard clause is falsified or if more than UB soft clauses are falsified. MaxCDCL defines two different procedures to learn a clause from both conflicts.

MIP conflict analysis [Achterberg 2007, Witzig 2022]. In the context of an MIP problem solved by a branch and bound algorithm, where the LP relaxation of the problem is solved at each node. We distinguish two different conflicts. First, the infeasibility has been detected by domain propagation and a single constraint is conflicting. A procedure analogous to CDCL can be performed. The solver constructs a conflict graph describing the relation between the deduced bound changes due to propagation and branching decisions. This conflict graph can be analyzed to learn a new constraint. The second type of conflict occurs if the LP relaxation is infeasible. We suppose that an objective cutoff constraint forbids assignment with a higher objective value than the current UB. Therefore, the LP relaxation can be infeasible because its optimal objective value is higher than the UB. In this case, it is possible derive a dual solution and relies on the lemma of Farkas [Farkas 1902] to produce a proof of infeasibility:

Definition 2.19 (Farkas constraint [Achterberg 2007])**.** *Given an LP problem* $\min\{c^{\mathbf{T}}x | Ax \geq b, x \in \mathbb{R}^{+}\}$ *and a dual solution* y *then the Farkas constraint is defined as:*

$$\mathbf{y}^{\mathbf{T}}Ax \geq \mathbf{y}^{\mathbf{T}}b \tag{2.18}$$

A Farkas constraint can be modified, strengthened, and added to the pool of constraints. Those kinds of constraints are built exclusively to prune more values with domain propagation. Indeed, as they are directly derived from an aggregation of globally valid constraints, they will not help the LP solver to derive better bounds, thus they do not actually enter the LP relaxation. Those constraints can also be used as a starting point for the conflict graph analysis.

In more recent works, Witzig shows how to learn constraint from internal feasible nodes, this defines *conflict-free learning* [Witzig 2022]. At each node, it is possible to produce a Farkas constraint (also referred as dual proof constraint) from a dual solution. This constraint is not directly useful but it can be strengthened according to the information obtained by going deeper into the search tree. Once again, those constraints are used only to prune inconsistent values.

Example 2.20. *Following example 2.4, we had the dual solution* $\mathbf{y} = \{\mathbf{y}_1 = 1.4, \mathbf{y}_3 = 0.2, \mathbf{y}_5 = 1.2, \mathbf{y}_{12} = 0.8\}$ *with objective value* 11.8*. The Farkas constraint*

issued from $\mathbf{y}$ *is:*

$$\begin{aligned} & 1.4 \times (3x_1 + 5x_2 + 3x_3 + 3x_4 + 5x_5 + 5x_6 + 5x_7) \\ & +0.2 \times (-x_1 - x_3) + 1.2 \times (-x_4 - x_5) - 0.8x_5 \\ & \geq 1.4 \times 10 - 0.2 - 1.2 - 0.8 \\ \iff & 4x_1 + 7x_2 + 4x_3 + 3x_4 + 5x_5 + 7x_6 + 7x_7 \geq 11.8 \end{aligned}$$

■

Chapter 3

Graphical models and linear constraints

Contents

3.1 Representing and propagating linear constraints

Linear constraints are expressive and compact constraints providing a powerful tool for modeling and solving a wide range of optimization problems, including computer science, operations research, and artificial intelligence [Boros & Hammer 2002]. Unfortunately, representing linear constraints in extension introduces a number of tuples which is exponential in the arity of the constraint, hence the default way of representing constraints in CFNs is intractable for such constraints. One possibility would be to represent the linear constraints in intention and integrate an LP solver to handle them. However, solving an LP at each node could be very expensive as Hurley et al [Hurley *et al.* 2016] have shown that ILP solvers can be significantly slower than dedicated WCSP solvers. Moreover, the LP solver can be subject to

numerical instability. It is not wanted to introduce numerical instability into exact solvers that may work exclusively with integers to guarantee the optimality of their solution.

In this chapter, we introduce a method for representing in a CFN any linear constraints without introducing extra variables. We extend soft local consistency algorithms to handle those constraints. More precisely we focus on PB constraints $\sum_{i\in\boldsymbol{S}} w_i x_i \geq C$ along with a partition of its variables into sets $A_1, \ldots, A_k \in \boldsymbol{S}$, $\bigcup_{j=1}^{k} A_j = \boldsymbol{S}$. For each partition A_j a constraint enforces that only one variable can take value one: $\sum_{x_j\in A_j} x_j = 1$. Such constraint is known as an *Exactly One* (EO) constraint, while $\sum_{x_j\in A_j} x_j \leq 1$ is an *At Most One* (AMO) constraint. This formulation is more general than a single PB constraint. In particular, it allows us to extend PB constraints to multi-valued variables. Let $\boldsymbol{S}$ be a scope over a set of WCSP variables with arbitrary domains and w_{iv} the weight associated with value $v \in \boldsymbol{D}_i$. As we do in the local polytope Local Polytope, for each variable in $\boldsymbol{S}$, we use 0/1 variables x_{iv} which take value 1 if $x_i = v$ and 0 if $x_i \neq v$. The constraint $\sum_{i\in\boldsymbol{S},v\in\boldsymbol{D}_i} w_{iv}x_{iv} \geq C$ matches the pattern described above, with partitions $A_i = \{x_{iv} \mid v \in \boldsymbol{D}_i\}$. Without loss of generality, we suppose that the weights and the capacity are all positive. Finally, this formulation admits the case where there exists an AMO constraint over some partitions: we add another 0/1 variable in each such partition and give it weight 0, so that this partition now has an EO constraint. Note that all those 0/1 variables are used to represent the constraint but do not appear outside of it, and we never branch on those variables.

Example 3.1. *Suppose we have a WCSP with 2 variables $X = \{x_1, x_2\}$ with domains $\boldsymbol{D}_1 = \{1, 2, 3\}$ and $\boldsymbol{D}_2 = \{1, 2\}$, we can express a PB constraint using Boolean variables $x_{11}, x_{12}, x_{13}, x_{21}, x_{22}$.*

$$\begin{aligned} 4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} &\geq 40 \\ x_{11} + x_{12} + x_{13} &= 1 \\ x_{21} + x_{22} &= 1 \\ x_{11}, x_{12}, x_{13}, x_{21}, x_{22} &\in \{0, 1\} \end{aligned}$$

A solution of this problem could be $x_{13} = 1, x_{21} = 1$ and corresponds to $x_1 = 3, x_2 = 1$ in the WCSP. ■

Constraint representation.

We will focus here on $F\emptyset IC$ (definition 2.5 in section 2.4) as the soft local consistency we aim to enforce. But first, we need an appropriate encoding that can represent the state of a linear constraint after a series of cost moves (procedure MoveCost) between the unary cost functions and the constraint, without storing a cost for each of the exponentially (in the arity of the constraint) many tuples. Observe that initially, the cost of any given tuple starts at 0 for allowed tuples and $\top$ for tuples that violate the constraint. After some cost moves, the cost of each tuple is the sum

of costs that have been moved to or from the values it contains. Therefore, it can be expressed as a linear function. Let δ_{iv} be the total cost that has been moved between the constraint and the corresponding unary cost $c_i(v)$. A cost move from c_{iv} to the linear constraint increases δ_{iv}, while a cost move in the opposite direction decreases it. Hence, we can have negative δ costs. We represent by $\delta_\emptyset$ the cost moved from this constraint to $c_\varnothing$. This quantity is necessarily positive. Initially, no cost moves have been performed and all the δ costs are 0. After any sequence of EPTs, the cost of an assingment τ is defined by:

$$c_{\boldsymbol{S}}(\tau) = \begin{cases} \sum_{\tau_i=v}(\delta_{iv}) - \delta_\emptyset & \text{if } \tau \text{ satisfies the constraint} \\ \top & \text{otherwise} \end{cases} \tag{3.1}$$

We require that $c_{\boldsymbol{S}}(\tau) \geq 0$ for all τ to maintain the invariant that no negative costs are present in any part of a WCSP.

Example 3.2. *The PB constraint in example 3.1 is small enough to be represented as a table. This example aims to show how we use the δ costs to implicitly represent the PB constraint. We give the table representation of the PB constraint (c_{PB}), the values of the δ costs and the unary costs of x_1 and x_2.*

x_1	x_2	*Cost*
1	*1*	$\top$
2	*1*	$\top$
3	*1*	*0*
1	*2*	*0*
2	*2*	*0*
3	*2*	*0*

δ	*Cost*
δ_{11}	*0*
δ_{12}	*0*
δ_{13}	*0*
δ_{21}	*0*
δ_{22}	*0*
$\delta_\emptyset$	*0*

x_1	*Cost*
1	*40*
2	*55*
3	*85*

x_2	*Cost*
1	*47*
2	*95*

Suppose we do the following sequence of EPTs:

- $extend(c_1, (x_1, 3), c_{PB}, 30)$
- $extend(c_2, (x_2, 2), c_{PB}, 20)$
- $project(c_\varnothing, c_{PB}, \emptyset, 20)$
- $project(c_1, c_{PB}, (x_2, 2), 10)$

The updated costs are:

x_1	x_2	*Cost*
1	*1*	$\top$*-20-10=*$\top$
2	*1*	$\top$*-20-10=*$\top$
3	*1*	*30-20-10=0*
1	*2*	*20-20=0*
2	*2*	*20-20=0*
3	*2*	*30+20-20=30*

δ	*Cost*
δ_{11}	*0*
δ_{12}	*0*
δ_{13}	*30*
δ_{21}	*-10*
δ_{22}	*20*
$\delta_{\emptyset}$	*20*

x_1	*Cost*
1	*40*
2	*55*
3	*85-30=55*

x_2	*Cost*
1	*47+10=57*
2	*95-20=75*

We observe that the cost of each tuple is equal to the sum of the δ of the values it contains minus $\delta_{\emptyset}$. ■

From equation 3.1, the minimal cost assignment of a linear constraint $c_{\boldsymbol{S}}$ can be obtained by solving the following integer program:

$$\min \textstyle\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} \delta_{iv} x_{iv} - \delta_{\emptyset} \tag{3.2a}$$

s.t.

$$\textstyle\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} w_{iv} x_{iv} \geq C \tag{3.2b}$$

$$\textstyle\sum_{v \in \boldsymbol{D}_i} x_{iv} = 1, \qquad \forall i \in \boldsymbol{S} \tag{3.2c}$$

$$x_{iv} \in \{0,1\}, \qquad \forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i \tag{3.2d}$$

We call this $ILP_{\emptyset}(c_S)$. The main property of $ILP_{\emptyset}(c_S)$ is that $c_{\boldsymbol{S}}$ is $\emptyset IC$ if and only if $opt(ILP_{\emptyset}(c_S)) = 0$. Otherwise if $opt(ILP_{\emptyset}(c_S)) > 0$ we can move some costs to $c_{\varnothing}$: $project(c_{\varnothing}, c_{\boldsymbol{S}}, \emptyset, opt(ILP_{\emptyset}(c_S)))$.

However, to detect violations of $F\emptyset IC$, it is not enough to look at the cost of tuples of the constraint, as we must also take unary costs into account. Therefore, the propagator we design considers the problem with a modified objective:

$$\min \textstyle\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} (\delta_{iv} + c_i(v)) x_{iv} - \delta_{\emptyset} \tag{3.3a}$$

$$\textstyle\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} w_{iv} x_{iv} \geq C \tag{3.3b}$$

$$\textstyle\sum_{v \in \boldsymbol{D}_i} x_{iv} = 1, \qquad \forall i \in \boldsymbol{S} \tag{3.3c}$$

$$x_{iv} \in \{0,1\}, \qquad \forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i \tag{3.3d}$$

Let this problem be $ILP_{F\emptyset}$. A linear constraint $c_{\boldsymbol{S}}$ is $F\emptyset IC$ if and only if $opt(ILP_{F\emptyset}) = 0$. In the following, we write $p_{iv} = \delta_{iv} + c_i(v)$ for compactness, when it does not matter how much of the coefficient came from δ_{iv} and how much came from $c_i(v)$. In contrast with $ILP_{\emptyset}(c_S)$, if $opt(ILP_{F\emptyset}) > 0$, we cannot move $opt(ILP_{F\emptyset})$ units of cost to $c_{\varnothing}$. Instead, we first have to move some cost from unary cost functions into the constraint before we can project it to $c_{\varnothing}$. In this case, the composition of p_{iv} from δ_{iv} and $c_i(v)$ is significant.

Example 3.3. *Following example 3.2. After the cost moves, $ILP_\emptyset(c_S)$ is:*

$$\begin{aligned} \min\, & 30x_{13} - 10x_{21} + 20x_{22} - 20 \\ & s.t. \\ & 4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} \geq 40 \\ & \sum_{v \in \boldsymbol{D}_i} x_{iv} = 1 \quad \forall x_i \in \{x_1, x_2\} \\ & 0 \leq x_{iv} \leq 1 \quad \forall x_i \in \{x_1, x_2\}, v \in \boldsymbol{D}_i \end{aligned}$$

The optimal solution of this problem is 0, therefore the constraint is $\emptyset IC$. The objective function of $ILP_{F\emptyset}$ is:

$$\min 40x_{11} + 55x_{12} + (30 + 55)x_{13}(-10 + 57)x_{21} + (20 + 75)x_{22} - 20$$

The optimal solution is 112, therefore the constraint is not $F\emptyset IC$. ■

Unfortunately, $ILP_\emptyset(c_S)$ and $ILP_{F\emptyset}$ have the knapsack problem as a special case, hence it is NP-hard to determine whether a linear constraint is $\emptyset IC$ or $F\emptyset IC$. Therefore, we detect only a subset of cases where the constraint is not $F\emptyset IC$ by relaxing the integrality constraint (3.3d) and (3.2d) into $0 \leq x_{iv} \leq 1$ and solving the resulting linear programs, called $LP_\emptyset$ and $LP_{F\emptyset}$, respectively. This forgoes the guarantee that $opt(LP_{F\emptyset}) = 0$ if and only if the constraint is $F\emptyset IC$, and satisfies only the 'if' part. More simply, if $opt(LP_{F\emptyset}) > 0$ then the constraint is not $F\emptyset IC$, and similarly for $LP_\emptyset$ and $\emptyset IC$.

$LP_{F\emptyset}$ has a special structure. It is a Multiple-Choice Knapsack Problem (MCKP) [Pisinger & Toth 1998], or a *knapsack problem with special ordered sets* [Johnson & Padberg 1981]. These can be solved more efficiently than arbitrary LPs, a fact that we use in our propagator.

3.1.1 Solving the Knapsack LP

We obtain an optimal solution $\boldsymbol{x}^*$ of the primal $LP_{F\emptyset}$ by applying Pisinger's greedy algorithm [Pisinger & Toth 1998]. This gives an optimal solution $\boldsymbol{x}^*$ in time $O(N \log N)$ [1], with $N = |\boldsymbol{x}^*|$, such that either $\boldsymbol{x}^*$ has no fractional value or it has exactly two fractional values. In the latter case, the WCSP variable $k \in \boldsymbol{S}$, verifying $\exists s, s' \in \boldsymbol{D}_k$ such that $0 < x^*_{ks}, x^*_{ks'} < 1$, is called a *split class* and $x_{ks}, x_{ks'}$ are the *split variables*. We denote by $\mathbf{o} = \sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} p_{iv} x^*_{iv} - \delta_\emptyset$, the optimal cost of $LP_{F\emptyset}$. Consider now the dual of $LP_{F\emptyset}$:

$$\begin{aligned} \max\quad & C \times y_{cc} + \textstyle\sum_{i \in \boldsymbol{S}} y_i - \delta_\emptyset \\ s.t.\quad & \\ & y_{cc} \times w_{iv} + y_i \leq p_{iv} \qquad \forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i \\ & y_{cc} \geq 0 \end{aligned} \tag{3.4}$$

[1] The Dyer-Zemel algorithm [Dyer 1984, Zemel 1984] can compute a solution in $O(N)$ time, but we have not yet implemented it.

Where y_{cc} is the dual variable corresponding to the capacity constraint and y_i corresponds to the EO constraint of x_i. It is easy to compute the optimal dual solution from the optimal primal solution. If the optimal primal solution is fractional, let k be the split class, $x_{ks}, x_{ks'}$ the split variables. Otherwise if the optimal primal solution is integer then we set $k = -1$. For $i \neq k$, define the variable x_{is} as the variable used in the optimal solution, *i.e.*, $x^*_{is} = 1$. We define the dual solution $\mathbf{y}$ as:

$$y_{cc} = \begin{cases} \frac{p_{ks} - p_{ks'}}{w_{ks} - w_{ks'}} & \text{if } k \neq -1 \\ 0 & \text{if } k = -1 \end{cases} \tag{3.5a}$$

$$\forall i \in \boldsymbol{S} \quad y_i = p_{is} - y_{cc} \times w_{is} \tag{3.5b}$$

Lemma 3.1. *The dual solution* $\mathbf{y}$ *defined by 3.5a-3.5b is optimal.*

Proof. First, we show that $\mathbf{y}$ defines a feasible dual solution. The optimal primal solution is issued from Pisinger's algorithm, which is based on a particular ordering of the variables. First, for each variable $i \in \boldsymbol{S}$ the values $j \in \boldsymbol{D}_i$ are ordered by decreasing weight w_{ij}. Then for each consecutive value verifying $p_{ij-1} - p_{ij} > 0$ a *slope* $\frac{p_{ij-1} - p_{ij}}{w_{ij-1} - w_{ij}}$ is defined, the value of a slope is necessarily positive. Finally, the slopes are sorted in increasing order. In a greedy principle, the algorithm follows the ordering of the slopes to derive the optimal relaxed solution. The value of y_{cc} corresponds to the last considered slope, therefore $y_{cc} \geq 0$. Moreover, this choice for y_{cc} also proves that every dual constraint 3.4 is verified.
We now show that $\mathbf{y}$ has the same objective value as the optimal relaxed solution x^*. We omit the constant term $\delta_\emptyset$ in the proof.
The objective value of $\mathbf{y}$ is:

$$\begin{aligned} C \times y_{cc} + \sum_{i \in \boldsymbol{S}} y_i &= C \times y_{cc} + \sum_{i \in \boldsymbol{S}} p_{is} - y_{cc} \times w_{is} \\ &= y_{cc}(C - \sum_{i \in \boldsymbol{S}} w_{is}) + \sum_{i \in \boldsymbol{S}} p_{is} \end{aligned}$$

If x^* is an integer solution then $y_{cc} = 0$ and $\sum_{i \in \boldsymbol{S}} p_{is} = \sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} p_{iv} x^*_{iv}$. Hence, $\mathbf{y}$ is optimal.
Otherwise, if x^* is fractional. From Pisinger's algorithm, we know that $C = \sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} w_{iv} x^*_{iv}$, therefore:

$$\begin{aligned} C - \sum_{i \in \boldsymbol{S}} w_{is} &= (x^*_{ks} - 1) w_{ks} + x^*_{ks'} w_{ks'} \\ &= (x^*_{ks} - 1) w_{ks} + (1 - x^*_{ks}) w_{k's} \\ &= (w_{ks} - w_{ks'})(x^*_{ks} - 1) \end{aligned}$$

Finally:

$$\begin{aligned}
& y_{cc}(C - \sum_{i \in \boldsymbol{S}} w_{is}) + \sum_{i \in \boldsymbol{S}} p_{is} \\
= & \frac{p_{ks} - p_{ks'}}{w_{ks} - w_{ks'}}(w_{ks} - w_{ks'})(x^*_{ks} - 1) + \sum_{i \in \boldsymbol{S}} p_{is} \\
= & (p_{ks} - p_{ks'})(x^*_{ks} - 1) + \sum_{i \in \boldsymbol{S}} p_{is} \\
= & \sum_{i \in \boldsymbol{S} \setminus \{k\}} p_{is} + x^*_{ks} p_{ks} - x^*_{ks} p_{ks'} + p_{ks'} \\
= & \sum_{i \in \boldsymbol{S} \setminus \{k\}} p_{is} + x^*_{ks} p_{ks} + x^*_{ks'} p_{ks'} \\
= & \sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} p_{iv} x^*_{iv}
\end{aligned}$$

We deduce that $\mathbf{y}$ is the optimal dual solution. □

From the dual optimal solution $\mathbf{y}$, we compute the reduced cost $rc^{\mathbf{y}}(x_{iv})$ of every variable x_{iv}, i.e., the slack of the dual constraint that corresponds to x_{iv} (see 2.2). In this chapter, since $\mathbf{y}$ always unambiguously refers to the optimal dual solution, we omit $\mathbf{y}$ and write $rc(x_{iv})$.

The reduced cost of a variable x can be interpreted as the amount by which we must decrease the coefficient of x in the objective function in order to have $x > 0$ in the optimal solution. We explain later that this implies that we can project some cost to unary cost functions.

In the specific case of $LP_{F\emptyset}$, we have:

$$\begin{aligned}
rc(x_{ks}) = rc(x_{ks'}) &= 0 \\
rc(x_{is}) &= 0 \quad \forall i \in \boldsymbol{S} \setminus \{k\} \\
rc(x_{iv}) = p_{iv} - y_{cc} \times w_{iv} - y_i &\quad \forall i \in \boldsymbol{S}, v \neq s
\end{aligned}$$

Observation 3.1. *Consider the linear program $LP'_{F\emptyset}$ which is identical to $LP_{F\emptyset}$ but has $\forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i, \quad p'_{iv} = p_{iv} - rc(x_{iv})$. Then $opt(LP'_{F\emptyset}) = opt(LP_{F\emptyset})$.*

Proof. The optimal solution $\boldsymbol{x}^*$ of $LP_{F\emptyset}$ has the same cost o in $LP_{F\emptyset}$ and $LP'_{F\emptyset}$, as the coefficients of the variables that are greater than 0 are unchanged. The optimal dual solution $\mathbf{y}$ is also a dual solution of $LP'_{F\emptyset}$, as the slack in the dual of $LP_{F\emptyset}$ matches exactly the reduction in the right-hand side between $LP_{F\emptyset}$ and $LP'_{F\emptyset}$. Moreover, as the dual objective did not change, it has the same cost and matches the primal cost, so $opt(LP'_{F\emptyset}) = o = opt(LP_{F\emptyset})$. □

Example 3.4. *Consider the following problem:*

$$\begin{aligned} \min\, & 40x_{11} + 55x_{12} + 85x_{13} + 47x_{21} + 95x_{22} \\ & s.t. \\ & 4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} \geq 40 \\ & \sum_{v \in \boldsymbol{D}_i} x_{iv} = 1 \quad \forall x_i \in \{x_1, x_2\} \\ & 0 \leq x_{iv} \leq 1 \quad \forall x_i \in \{x_1, x_2\}, v \in \boldsymbol{D}_i \end{aligned}$$

Pisinger's algorithm gives the optimal primal solution $\boldsymbol{x}^* = \{0, 1, 0, \frac{7}{12}, \frac{5}{12}\}$ *with cost* $o = 55 + \frac{7}{12} \times 47 + \frac{5}{12} \times 95 = 122$.

We deduce the following dual optimal solution :

- $y_{cc} = 2$
- $y_1 = 55 - 2 \times 14 = 27$
- $y_2 = 47 - 2 \times 16 = 15$

The following reduced costs are obtained: $rc(x_{12}) = rc(x_{21}) = rc(x_{22}) = 0$ *and* $rc(x_{11}) = 5$, $rc(x_{13}) = 10$, *and we deduce that replacing the previous objective function by the following one does not change the cost of the optimal solution:*

$$\min 35x_{11} + 55x_{12} + 75x_{13} + 47x_{21} + 95x_{22}$$

We observe that the solution $\boldsymbol{x}^* = \{0, 1, 0, \frac{7}{12}, \frac{5}{12}\}$ *is still optimal.* ■

3.1.2 Propagation

As explained in section 2.4, soft consistency algorithms (except OSAC) solve the LP relaxation of a WCSP only approximately, and they are not confluent, meaning they may converge to different fixed points. The exact set of EPTs produced by a propagation algorithm may affect both the bound produced in the current run, but also the bounds produced in future runs.

Empirically, it appears that it is better to leave as much cost as possible in lower arity constraints: it is better to have costs in the nullary constraint $c_\varnothing$ than a unary constraint, it is better to have costs in unary constraints than binary constraints, and so on. A significant factor to consider is the sequence in which we propagate the constraints, this will be discussed later (see section 3.1.4).

Given a linear constraint and the associated unary costs, it is possible to increase the lower bound $c_\varnothing$ by at least $opt(LP_{F\emptyset})$. Since that is the primary consideration, we design our algorithm to always do that. Additionally, we aim to extend as little cost as possible from the unary cost functions in order to make $opt(LP_\emptyset) = o = opt(LP_{F\emptyset})$ and then project o to $c_\varnothing$.

If we move a cost $c_i(v) - rc(x_{iv})$ between the linear constraint and each unary cost function and value, then $opt(LP_\emptyset) = o$ and we can project o to $c_\varnothing$. Note that this

quantity can be positive or negative and its sign gives the direction of the cost move. If it is positive the cost goes from the unary cost function to the linear constraint, and the other direction if it is negative. As we have $c_i(v) - rc(x_{iv}) = (y_{cc} \times w_{iv} + y_i) - \delta_{iv}$, we thus obtain the EPTs performed by Procedure TransformPB.

Procedure TransformPB($c_{\boldsymbol{S}}, y_{cc}, y_i, o$)

Data: $c_{\boldsymbol{S}}$: PB constraint
Data: y_{cc}, y_i, o: optimal dual solution of $LP_{F\emptyset}$

for *all the variables* x_{iv} **do**
  $c_i(v) \leftarrow c_i(v) - y_{cc} \times w_{iv} - y_i + \delta_{iv}$;
  $\delta_{iv} \leftarrow y_{cc} \times w_{iv} + y_i$;
$c_\varnothing \leftarrow c_\varnothing + o$;
$\delta_\varnothing \leftarrow \delta_\varnothing + o$;

Theorem 3.1. *Algorithm TransformPB preserves equivalence.*

Proof. Recall that $p_{iv} = c_i(v) + \delta_{iv}$ and that $rc(x_{iv}) \geq 0$. If $c_i(v) - rc(x_{iv}) \geq 0$ then the cost move is an extension of less than $c_i(v)$, which is valid. If $c_i(v) - rc(x_{iv}) < 0$ then the cost move is a projection, by definition of the reduced cost, the cost of any solution $\boldsymbol{x}'$ with $x'_{iv} = 1$ is at least $o - c_i(v) + rc(x_{iv})$. This operation is also valid.

Finally, to check that our sequence of EPTs justifies the increase of $c_\varnothing$ by o, we compute the optimum of $LP_\emptyset$. From Observation 3.1, $opt(LP_\emptyset) = o$, which means we can project o to $c_\varnothing$ and increase $\delta_\varnothing$ to bring $opt(LP_\emptyset) = opt(LP_{F\emptyset}) = 0$. □

We can improve on this by observing that the optimal solution of $ILP_{F\emptyset}$ is necessary integral. Therefore, we can get closer to this optimal integer solution by increasing $c_\varnothing$ by $\lceil o \rceil$. In this case, it is also necessary to round up all cost moves. By rounding up, we can no longer rely on Observation 3.1, but it still holds that $opt(LP_\emptyset) = 0$. We also approach $\emptyset IC$ by verifying that for any value x_{jb}, the minimal cost tuple (not necessarily satisfying the capacity constraint (3.2b)) with $x_{jb} = 1$ has cost 0. The cost of this minimal tuple is equal to $\delta_{jb} + \min \sum_{i \in \boldsymbol{S} \setminus x_j, v \in \boldsymbol{D}_i} (\delta_{iv} + c_i(v)) x_{iv} - \delta_\varnothing$. If this cost is non-zero, we can project a positive cost to $c_j(b)$.

Procedure Propagate is the entry point to the propagator. It enforces domain consistency on the linear constraint, this can be done by computing the solution having the maximum achievable weight. If changing the value of one variable produces an infeasible solution then we can delete this value. Then it solves $LP_{F\emptyset}$, if there is more than one optimal solution, we prefer the one minimizing the reduced cost of the EAC support of each variable. Finally, it uses Procedure TransformPB, to perform cost moves.

Theorem 3.2. *Procedure Propagate runs in $O(nd \log nd)$ time where n is the number of WCSP variables involved and d the maximum domain size.*

Proof. Pisinger's algorithm dominates the complexity, as it runs in $O(N \log N)$, where N is the number of LP variables. In our case, $N = nd$, so it takes $O(nd \log nd)$

Procedure Propagate($c_{\boldsymbol{S}}$)

Data: $c_{\boldsymbol{S}}$: PB constraint with EO partitions
DomainConsistency($c_{\boldsymbol{S}}$) ;
(y_{cc}, y_i, o) = DualSolve($LP_{F\emptyset}$) ;
TransformPB($c_{\boldsymbol{S}}, y_{cc}, y_i, o$) ;

time. Domain consistency on a linear inequality can be performed in linear time. Finally, Procedure TransformPB iterates once over all variables and values and performs constant time operations on each. Hence, the total complexity is $O(nd \log nd)$. □

Example 3.5. *Returning to Example 3.4, where $c_{\boldsymbol{S}}$ is the PB constraint with EO partitions over two WCSP variables x_1 and x_2, we had the following reduced costs: $rc(x_{12}) = rc(x_{21}) = rc(x_{22}) = 0$, $rc(x_{11}) = 5$, $rc(x_{13}) = 10$, the optimal cost was 122. We deduce the following cost moves:*

- $extend(c_1, (1,1), c_{\boldsymbol{S}}, 40 - 5 = 35)$
- $extend(c_1, (1,2), c_{\boldsymbol{S}}, 55)$
- $extend(c_1, (1,3), c_{\boldsymbol{S}}, 85 - 10 = 75)$
- $extend(c_2, (2,1), c_{\boldsymbol{S}}, 47)$
- $extend(c_2, (2,2), c_{\boldsymbol{S}}, 95)$
- $project(c_\varnothing, c_{\boldsymbol{S}}, \emptyset, 122)$

It implies the resulting δ costs:

- $\delta_{11} = 35$
- $\delta_{12} = 55$
- $\delta_{13} = 75$
- $\delta_{21} = 47$
- $\delta_{22} = 95$
- $\delta_\varnothing = 122$

The unary costs after these operations are $c_1(2) = c_2(1) = c_2(2) = 0$, $c_1(1) = 5$, $c_1(3) = 10$. If we construct the table of possible assignments of $LP_\emptyset$ obtained after the extensions, we can see that:

- $c_{\boldsymbol{S}}(\{x_1 = 1, x_2 = 2\}) = \delta_{11} + \delta_{22} - \delta_\varnothing = 8$
- $c_{\boldsymbol{S}}(\{x_1 = 2, x_2 = 2\}) = \delta_{12} + \delta_{22} - \delta_\varnothing = 28$
- $c_{\boldsymbol{S}}(\{x_1 = 3, x_2 = 1\}) = \delta_{13} + \delta_{21} - \delta_\varnothing = 0$
- $c_{\boldsymbol{S}}(\{x_1 = 3, x_2 = 2\}) = \delta_{13} + \delta_{22} - \delta_\varnothing = 48$

All the other tuples don't satisfy the constraint. We observe that the optimal solution is 0*, hence our extensions justify the increase of* $c_\varnothing$.
Now assume that other EPTs outside the PB constraint have modified the unary costs:

- $c_1(1) \rightarrow c_1(1) + 16 = 21$
- $c_1(2) \rightarrow c_1(2) + 30 = 30$
- $c_1(3) \rightarrow c_1(3) - 9 = 1$

We want to compute a new lower bound for the PB constraint by solving $LP_{F\emptyset}$*:*

$$\min(21+35)x_{11} + (30+55)x_{12} + (75+1)x_{13} + (0+47)x_{21} + (0+95)x_{22} - 122$$
$$s.t.$$
$$4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} \geq 40$$
$$\sum_{v \in \boldsymbol{D}_i} x_{iv} = 1 \quad \forall x_i \in \{x_1, x_2\}$$
$$0 \leq x_{ij} \leq 1 \quad \forall x_i \in \{x_1, x_2\}, v \in \boldsymbol{D}_i$$

The optimal solution is $\boldsymbol{x}^* = \{0, 0, 1, 1, 0\}$ *and its cost is* $o = 76 + 47 - 122 = 1$. *We deduce the dual optimal solution* $y_{cc} = 1$, $y_1 = 52$, $y_2 = 31$ *with reduced costs* $rc(x_{11}) = rc(x_{13}) = rc(x_{21}) = 0$ *and* $rc(x_{12}) = 19$, $rc(x_{22}) = 24$. *We carry out the following cost moves:*

- $extend(c_1, (1,1), c_{\boldsymbol{S}}, 21)$
- $extend(c_1, (1,2), c_{\boldsymbol{S}}, 11)$
- $extend(c_1, (1,3), c_{\boldsymbol{S}}, 1)$
- $project(c_2, c_{\boldsymbol{S}}, (2,2), 24)$
- $project(c_\varnothing, c_{\boldsymbol{S}}, \emptyset, 1)$

with $\delta_{11} = 56$, $\delta_{12} = 66$, $\delta_{13} = 76$, $\delta_{21} = 47$, $\delta_{22} = 71$, $\delta_\varnothing = 123$. ■

3.1.3 $F\emptyset IC$ and dual solution of the local polytope

In chapter 2, we showed how the EPTs on table constraints can be matched to a dual solution of the Local Polytope. In this section, we extend this theoretical result to integrate the EPTs made by algorithm Propagate on linear constraints. We first extend the local polytope to include linear constraints. Let $P`\langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top \rangle$ be a WCSP, we denote by $\mathcal{W} \subseteq \boldsymbol{C}$ the set of linear constraints and $C^+ = \boldsymbol{C} \setminus \{c_\varnothing\}$. Let the LPs (PrimalLin), (DualLin) define the primal and dual problem of P. The linear constraints are defined by equation (3.6d), where $W \in \mathbb{R}_+^{m \times n}$, $b \in \mathbb{R}_+^m$. Without loss of generality, we suppose that all constraints are greater or equal with only positive coefficients in the rhs and lhs. By default, the scope of all the linear constraints is $\boldsymbol{X}$, but the matrix W may have zero entries and the effective scope of each individual linear constraint may be smaller. We denote by w_{ia}^k (resp $w_{\boldsymbol{S}:\tau}^k$) the coefficient of

value (i, a) (resp $\tau \in \ell(\boldsymbol{S})$) in the row k of W and the dual variable corresponding to row k is λ_k.

PrimalLin

$$\min Obj \stackrel{\text{def}}{=} c_\varnothing + \sum_{i \in \boldsymbol{X}, a \in \boldsymbol{D}_i} c_i(a) x_{ia} + \sum_{c_{\boldsymbol{S}} \in C^+ \setminus \mathcal{W}, \tau \in \ell(\boldsymbol{S})} c_{\boldsymbol{S}}(\tau) x_{\boldsymbol{S}:\tau} \tag{3.6a}$$

$$\text{s.t.} \quad \forall i \in \boldsymbol{X}, \quad \sum_{a \in \boldsymbol{D}_i} x_{ia} = 1 \tag{3.6b}$$

$$\forall c_{\boldsymbol{S}} \in C^+ \setminus \mathcal{W}, i \in \boldsymbol{S}, a \in \boldsymbol{D}_i \big(\sum_{\tau \in \ell(\boldsymbol{S}), \tau_i = a} x_{\boldsymbol{S}:\tau} \big) - x_{ia} = 0 \tag{3.6c}$$

$$Wx \geq b \tag{3.6d}$$

DualLin

$$\max \sum_{i \in \boldsymbol{X}} \pi_i + \sum_{k=1}^{m} \lambda_k b_k \quad \text{s.t.} \tag{3.7a}$$

$$\forall i \in \boldsymbol{X}, a \in \boldsymbol{D}_i \quad \sum_{k=1}^{m} w_{ia}^k \lambda_k + \pi_i - \sum_{c_{\boldsymbol{S}} \in C^+, i \in \boldsymbol{S}} \varphi_{ia:\boldsymbol{S}} \leq c_i(a) \tag{3.7b}$$

$$\forall c_{\boldsymbol{S}} \in C^+ \setminus \mathcal{W}, \tau \in \ell(\boldsymbol{S}) \quad \sum_{k=1}^{m} w_{\boldsymbol{S}:\tau}^k \lambda_k + \sum_{i \in \boldsymbol{S}, a \in \boldsymbol{D}_i, \tau_i = a} \varphi_{ia:\boldsymbol{S}} \leq c_{\boldsymbol{S}}(\tau) \tag{3.7c}$$

$$\forall 1 \leq k \leq m \quad \lambda_k \geq 0 \tag{3.7d}$$

We can obtain a feasible dual solution $\mathbf{y} = \{\Pi, \varphi, \Lambda\}$ (initialized to 0) by analyzing the sequence of EPTs made by the consistency algorithms. If an EPT implies a unary constraint c_i or a table constraint $c_{\boldsymbol{S}}$ we proceed as seen in Chapter 2:

- operation $unaryProject(c_i, \alpha)$ gives $\pi_i \leftarrow \pi_i + \alpha$
- operation MoveCost$(c_i, c_{\boldsymbol{S}}, \{x_i = a\}, \alpha)$ gives $\varphi_{ia:\boldsymbol{S}} \leftarrow \varphi_{ia:\boldsymbol{S}} + \alpha$

Otherwise if procedure Propagate is called on the linear constraint $c_{\boldsymbol{S}_k}$, then $\mathbf{y}$ depends only on the values of y_{cc} and y_i computed in the last iteration of TransformPB:

- $\lambda_k \leftarrow y_{cc}$
- $\pi_i \leftarrow \pi_i + y_i \ \forall i \in \boldsymbol{S}_k$

Lemma 3.2. *The dual solution* $\mathbf{y}$ *defines a feasible solution and its objective value matches the cost of* $c_\varnothing$.

Proof. From chapter 2, we know that when there are no linear constraints then $\pi_i - \sum_{c_{\boldsymbol{S}} \in C^+, i \in \boldsymbol{S}} \varphi_{ia:\boldsymbol{S}}$ corresponds to the quantity of cost moved to/from $c_i(a)$. We

show that $\sum_{k=1}^{m} w_{ia}^{k}\lambda_k + \pi_i - \sum_{c_{\boldsymbol{S}}\in C^+, i\in\boldsymbol{S}} \varphi_{ia:\boldsymbol{S}}$ (lhs of (3.7b)) also corresponds to the quantity of cost moved to/from $c_i(a)$ in the presence of linear constraints. Given a linear constraint $c_{\boldsymbol{S}_k}$, procedure TransformPB maintain that δ_{ia} corresponds to the quantity of cost moved between $c_i(a)$ and $c_{\boldsymbol{S}_k}$. We have $\delta_{ia} = y_{cc} \times w_{ia} + y_i$. Therefore, if we set $\lambda_k = y_{cc}$ and $\pi_i = \pi_i + y_i$, then the quantity δ_{ia} appears in the lhs of (3.7b). We can do this for every linear constraint. Hence, we show that the lhs of (3.7b) corresponds to the quantity of (possibly negative) cost moved from $c_i(a)$ to any constraints. A similar result can be obtained concerning the lhs of (3.7c) and $c_{\boldsymbol{S}}(\tau)$. Theorem 3.1 proved that the sequence of EPTs conducted during the propagation is valid, meaning that all costs remain positive, hence the sum of all the (possibly negative) costs moved from a value must be lower than its unary cost. We deduce that both (3.7b) and (3.7c) are verified by $\mathbf{y}$.
We prove that the objective value of $\mathbf{y}$ corresponds to the increase of $c_\varnothing$ by using a similar reasoning. We know that $\delta_\emptyset$ gives the quantity of cost moved from a linear constraint to $c_\varnothing$. Moreover, procedure TransformPB maintains the fact that the optimal objective value of $LP_\emptyset$ is 0, meaning that the dual optimal objective value of $LP_\emptyset$ is $\delta_\emptyset$. The optimal dual solution of $LP_\emptyset$ is computed during procedure TransformPB, its cost is $b_k \times y_{cc} + \sum_{x_i\in\boldsymbol{S}} y_i$. Setting $\lambda_k \leftarrow y_{cc}$ and $\pi_i \leftarrow \pi_i + y_i$ will increase the objective value of $\mathbf{y}$ by exactly $b_k \times y_{cc} + \sum_{x_i\in\boldsymbol{S}} y_i$. In the end, the cost of $\mathbf{y}$ is the sum of all the $\delta_\emptyset$ plus the costs of unary projections, which exactly matches the increase of $c_\varnothing$. □

We know how to produce a dual solution with cost $c_\varnothing$, however, the way we represent linear constraints fails to validate a convenient property verified with table constraints. If $\mathbf{y}$ is a dual solution partially obtained by algorithm Propagate and $\tilde{P}$ the resulting reparametrization of P, it does not hold that the dual solution $\tilde{\mathbf{y}} = 0$ of $\tilde{P}$ has a cost $c_\varnothing$. The reason is simple, with our representation the constant $-\delta_\emptyset$ appears in $ILP_\emptyset(c_S)$. Therefore, the dual solution $\tilde{\mathbf{y}} = 0$ will have a cost $c_\varnothing - \delta_\emptyset$. It signifies that with linear constraints it is not possible to capture the dual solution corresponding to the reparametrization transforming one internal node to another. This is shown in example 3.6. In practice, it has no incidence on the solving process but it will be noteworthy when we try to design a learning mechanism in chapter 5. We dig deeper into the analysis to concretely understand where this extra cost comes from and why it is necessary. In the linear constraint, the initial cost of any assignment is 0 if it satisfies the constraint and $\top$ otherwise. After some cost moves, the δ are modified according to a dual solution $\mathbf{y} = (y^*, y^*_{cc})$ and procedure TransformPB. The cost of any tuple satisfying the constraint is:

$$\sum_{i\in\boldsymbol{X}, j=\tau_i} \delta_{ij} - \delta_\emptyset \tag{3.8}$$

$$= \sum_{i\in\boldsymbol{X}, j=\tau_i} (y^*_{cc} w_{ij} + y^*_i) - y^*_{cc} \times C - \sum_{i\in\boldsymbol{X}} y^*_i \tag{3.9}$$

$$= y^*_{cc}\Big(\sum_{i\in\boldsymbol{X}, j=\tau_i} w_{ij} - C\Big) \tag{3.10}$$

If the tuple verifies the constraint then this cost is positive. Furthermore, we can see it is exactly equal to y^*_{cc} times the slack between the *weight* achieved by the tuple and the capacity. Where the weight of a tuple denotes the quantity $\sum_{i\in\boldsymbol{X},j=\tau_i} w_{ij}$. This is not a coincidence. Indeed, we find again this result by considering a knapsack constraint transformed into an equality constraint by adding an extra slack variable s. This corresponds to expressing the ILP $ILP_{F\emptyset}$ in its normal form:

$$\begin{aligned}
\min \quad & \textstyle\sum_{i\in\boldsymbol{S},v\in\boldsymbol{D}_i}(\delta_{iv}+c_i(v))x_{iv}-\delta_\emptyset && && \text{(3.11a)}\\
& \textstyle\sum_{i\in\boldsymbol{S},v\in\boldsymbol{D}_i} w_{iv}x_{iv}-s=C && && \text{(3.11b)}\\
& \textstyle\sum_{v\in\boldsymbol{D}_i} x_{iv}=1, && \forall i\in\boldsymbol{S} && \text{(3.11c)}\\
& x_{iv}\in\{0,1\}, && \forall i\in\boldsymbol{S}, v\in\boldsymbol{D}_i && \text{(3.11d)}\\
& s\geq 0 && && \text{(3.11e)}
\end{aligned}$$

Suppose a black box is returning an optimal dual solution $\mathbf{y}=(y^*,y^*_{cc})$. The procedure TransformPB gives the following δ costs:

$$\delta_{iv}=y^*_{cc}w_{ij}+y^*_i \tag{3.12}$$
$$\delta_s=-y_{cc} \tag{3.13}$$
$$\delta_\emptyset=y^*_{cc}\times C+\sum_{i\in\boldsymbol{X}} y^*_i \tag{3.14}$$

For any tuple τ satisfying the constraint, the value of the slack variable s is $\sum_{i\in\boldsymbol{X},j=\tau_i} w_{ij}-C$. Therefore, the cost of any assignment τ is:

$$\sum_{i\in\boldsymbol{X},j=\tau_i}\delta_{ij}+s\delta_s-\delta_\emptyset \tag{3.15}$$
$$=\sum_{i\in\boldsymbol{X},j=\tau_i}(y^*_{cc}w_{ij}+y^*_i)-y^*_{cc}(\sum_{i\in\boldsymbol{X},j=\tau_i} w_{ij}-C)-y^*_{cc}\times C-\sum_{i\in\boldsymbol{X}} y^*_i \tag{3.16}$$
$$=0 \tag{3.17}$$

All the tuples have a zero cost. In this situation, it would not be necessary to maintain the δ costs.
Consequently, in our representation, the redundant costs appearing in a reparametrization are directly connected to the reduced cost of a slack variable. Introducing this slack variable in the WCSP solver would allow us to remove completely the δ costs. However, those slack variables can have a very large domain and it can be practically challenging to model them in discrete optimizers. In our particular case, TOULBAR2 is not able to model them.

Example 3.6. *Let P be the WCSP with 4 variables with domain $\{a,b\}$ and 2 linear*

constraints:

$$\begin{aligned} \min\, & 2x_{1a} + 3x_{2a} + 4x_{3a} \\ & s.t \\ c_{123} : & 2x_{1a} + 2x_{2a} + 2x_{3a} \geq 3 \\ c_{134} : & x_{1b} + x_{3a} + x_{4a} \geq 2 \\ \forall i \in [1,\ldots,4] \quad & x_{ia} + x_{ib} = 1 \\ \forall i \in [1,\ldots,4] \quad & x_{ia} + x_{ib} \in \{0,1\} \end{aligned}$$

The dedicated propagator Propagate first propagates c_{123}. It considers an LP with constraints c_{123} and the EO constraints $x_{ia} + x_{ib} = 1, 1 \leq i \leq 3$. The dual optimal solution is $\lambda_{123} = 1.5, \pi_1 = -1$ and cost 3.5 (rounded to 4). It performs the following cost moves:

- $Extend(c_1, (1,a), c_{123}, 2)$
- $Project(c_1, c_{123}, (1,b), 1)$
- $Extend(c_2, (2,a), c_{123}, 3)$
- $Extend(c_3, (3,a), c_{123}, 3)$
- $Project(c_\varnothing, c_{123}, \emptyset, 4)$

The following LP gives the obtained reparametrization. We write as δ_{123} the δ costs stored in c_{123} . Those costs are invisible outside the propagation of c_{123}.

$$\begin{aligned} \min\, & x_{1b} + x_{3a} + 4 + [\delta_{123} : 2x_{1a} - x_{1b} + 3x_{2a} + 3x_{3a} - 4] \\ & s.t \\ c_{123} : & 2x_{1a} + 2x_{2a} + 2x_{3a} \geq 3 \\ c_{134} : & x_{1b} + x_{3a} + x_{4a} \geq 2 \\ \forall i \in [1,\ldots,4] \quad & x_{ia} + x_{ib} = 1 \\ \forall i \in [1,\ldots,4] \quad & x_{ia} + x_{ib} \in \{0,1\} \end{aligned}$$

The solver propagates c_{134}. The optimal dual solution is $\lambda_{134} = 1, \pi_4 = -1$ and cost 1. We deduce the EPTs :

- $Extend(c_1, (1,b), c_{134}, 1)$
- $Extend(c_3, (3,a), c_{134}, 1)$
- $Project(c_4, c_{134}, (4,b), 1)$
- $Project(c_\varnothing, c_{134}, \emptyset, 1)$

The resulting reparametrization is:

$$\begin{aligned} x_{4b} + 5 + [\delta_{134} : x_{1b} + x_{3a} - x_{4b} - 1] + [\delta_{123} : 2x_{1a} - x_{1b} + 3x_{2a} + 3x_{3a} - 4] \\ s.t \\ c_{123} : 2x_{1a} + 2x_{2a} + 2x_{3a} \geq 3 \\ c_{134} : x_{1b} + x_{3a} + x_{4a} \geq 2 \\ \forall i \in [1, \ldots, 4] \quad x_{ia} + x_{ib} = 1 \\ \forall i \in [1, \ldots, 4] \quad x_{ia} + x_{ib} \in \{0, 1\} \end{aligned}$$

The propagation stops here. The dual solution corresponding to this reparametrization is $\mathbf{y} = \{\lambda_{123} = 1.5, \lambda_{134} = 1, \pi_1 = -1, \pi_4 = -1\}$.
If we assign $x_1 = b$*, then* $c_1(a) = \top$*. We call this new problem* $\tilde{P}_{x_1=b}$*. We can notice that the zero dual solution of* $\tilde{P}_{x_1=b}$ *has a cost 5-4-1=0.*
Domain propagation on c_{123} *leads to* $x_2 = a, x_3 = a$*. All the variables of the constraint are assigned, propagating* c_{123} *gives an optimal dual solution* $\lambda_{123} = 0, \pi_1 = -1, \pi_3 = 3, \pi_2 = 3$ *with cost 1 (we have* $\delta_\emptyset = 4$*). The EPTs involving removed values are unnecessary, therefore the only deduced EPT is:*

- $Project(c_\varnothing, c_{123}, \emptyset, 1)$

Propagating c_{134} *gives dual optimal solution* $\lambda_{134} = 0, \pi_1 = 1, \pi_3 = 1$ *with cost 1. The deduced EPTs are:*

- $Extend(c_4, (4, b), c_{134}, 1)$
- $Project(c_\varnothing, c_{134}, \emptyset, 1)$

In the resulting reparametrization, only one tuple of c_{123} *is feasible and it has a 0 cost, therefore it is unnecessary to keep track of the* δ *costs. Similarly, for constraint* c_{134} *variables* x_{1b} *and* x_{3a} *are assigned we can directly add their* δ *costs to* $\delta_\emptyset$*. The removed values are associated with an arbitrarily large cost of 100.*

$$\begin{aligned} 100x_{1a} + 100x_{2b} + 100x_{3b} + 7 \\ s.t \\ c_{123} : 2x_{1a} + 2x_{2a} + 2x_{3a} \geq 3 \\ c_{134} : x_{1b} + x_{3a} + x_{4a} \geq 2 \\ \forall i \in [1, \ldots, 4] \quad x_{ia} + x_{ib} = 1 \\ \forall i \in [1, \ldots, 4] \quad x_{ia} + x_{ib} \in \{0, 1\} \end{aligned}$$

The propagation stops here. The dual solution corresponding to this reparametrization is $\tilde{\mathbf{y}} = \{\tilde{\lambda_{123}} = 0, \tilde{\lambda_{134}} = 0, \tilde{\pi}_1 = 0, \tilde{\pi}_2 = 3, \tilde{\pi}_3 = 4\}$ *and its cost is 7.*
We can see that the increase of LB at node $\tilde{P}$ *is 2 but dual solution* $\tilde{\mathbf{y}}$ *has a cost of 7. It doesn't correspond to the EPTs only performed at node* $\tilde{P}$*.* ■

3.1.4 Additional Considerations

We describe here the different implementation details or heuristics we tried/used to improve the main algorithm or to make it more compatible with TOULBAR2. The different heuristics were tested on a benchmark composed of 22 KPCG instances [Bettinelli *et al.* 2017] (500-1000 Boolean variables with graph density varying from 0.1 to 0.9, one linear constraint), 5 weighted capacity warehouse instances [Kratica *et al.* 2001](600 variables of maximum domain size 100, up to 90901 cost functions with 300 linear constraints), 12 multi-demand multi-dimensional knapsack instances (MDMKP) [Cappanera & Trubian 2005] (100 Boolean variables, 6 linear constraints), 24 instances from the OPT-SMALLINT-LIN 2016 pseudo-Boolean Competition:[2] area_delay, dt-problems, trarea_ac (250-26836 Boolean variables, 76-55586 linear constraints).

Ordering the constraints

The order in which we propagate the constraints has great leverage on the quality of the lower bound. The propagation process is triggered whenever the domain of a variable is modified or its cost is changed. In this case, all the constraints having this variable in its scope must be propagated, so we must decide which constraints must be propagated first. Constraint propagation is based on soft local consistency, depending on the previous propagation one cost can be inside or outside the reach of the local consistency algorithm. This is amplified by the fact that the algorithm Propagate is based on a relaxed optimal solution and might extend more cost than necessary to the linear constraint. Once a cost has been moved to a linear constraint, it becomes invisible for other soft local consistency algorithms. We tried several approaches to order the constraints:

- Random: Constraints are sorted randomly.
- Decreasing arity: Follows the arity.
- Increasing/Decreasing DAC Ordering: The DAC ordering of the variables is used by the soft local consistency EDAC (see definition 2.13 in chapter 2). We sort the constraint depending on the variable with the lowest DAC ordering within the scope.
- Decreasing Tightness: For linear constraints, it depends on how hard it is to satisfy the constraint (capacity divided by maximal achievable weight). For cost functions expressed in extension, it is the sum of the costs divided by the number of non-zero costs.
- Lagrangian: For linear constraints, it studies the *Lagrangean Relaxation* [Martello & Toth 1987]. This relaxation introduces a Lagrangian multiplier and it has been shown that the optimal value of the lagrangean multiplier is equal to the

[2] http://www.cril.univ-artois.fr/PB16/

Table 3.1: Comparison of the average solving time and average number of nodes for different ordering of the linear constraints. The number of instances for each class and the number of solved instances are written in parentheses.

	KPCG(21)	mdmkp(12)	OPT-SMALLINT(22)	warehouse(2)
Decreasing DAC	**413s** 1132228 (21)	**97s** 1454059 (12)	**361s** 1319537 (20)	1411s 177883 (2)
Arity	506s 1317228 (21)	**97s** 1454059 (12)	711s 4162782 (17)	1435s 241034 (2)
Random	501s 1750695 (19)	141s 2188354 (12)	867s 4692683 (14)	**1216s** 169077 (2)
Lagrangian	422s 1132228 (21)	139s 1950697 (12)	1104s 6023148 (11)	1732s 76074 (2)
Increasing DAC	687s 1610471 (19)	**99s** 1454059 (12)	1101s 6287224 (12)	1834s 241034 (2)
Tightness	489s 1141141 (21)	324s 4009239 (11)	1090s 60764183 (12)	1943s 72927 (1)

value of the dual variable y_{cc} at optimality. We sort the linear constraints by increasing lagrangean multiplier and follow the DAC ordering otherwise. We also tested with decreasing lagrangean multiplier but this led to slightly worse results and is not reported here.

Table 3.1 gives the results obtained for the different classes. One KPCG, 3 warehouses, and 2 OPT-SMALLINT instances are removed because unsolved by all the approaches. Overall the DAC decreasing ordering seems to be the most polyvalent. However, there is room for improvement as the random ordering is competitive and solves some instances significantly faster. For example, the instance *normalized-lo_16x16_008.opb.metafix.opb* is solved in 3.2s with the random ordering, 931s with increasing DAC and unsolved in 2000s with the other orderings. For the mdmkp benchmark which contains only 6 linear constraints, we can see that the ordering has a clear impact on the solving process. This difference is even greater when the instance is a combination of linear constraints and table constraints as in the OPT-SMALLINT instances. Finding the best ordering seems to be a difficult task, especially when we consider that other heuristics can influence the solving process and benefit or not from the constraint ordering. This may explain why the DAC-based ordering produces better results, as it is used in several places in TOULBAR2. In the following we always use the DAC-based ordering of the constraint.

Solving the MCKP exactly

We implemented a basic dynamic programming algorithm to solve the MCKP exactly (problem $ILP_{F\emptyset}$). The user can control in TOULBAR2 if he wants to use an exact solution with the option "$-kpdp = [integer]$" (-2: never, -1: only in preprocessing, 0: at every search node, $k > 0$: at every k search nodes). During the

propagation, the dynamic program is called to solve $ILP_{F\emptyset}$ and obtain an integer optimal solution with cost *opt*. Then to compute the exact reduced cost of a value (i, v) not used in the optimal solution $(x_{iv} = 0))$ [3], the solver fixes $x_{iv} = 1$ and solves again $ILP_{F\emptyset}$. The difference between this solution and *opt* is the reduced cost of x_{iv}. Note that the order of the variables has an impact on the computed reduced costs, here we choose to follow the DAC ordering. If the solvers alternate between optimal and relaxed solutions, then the solver doesn't perform any EPTs if the computed relaxed solution has a lower cost than the last integer solution.
The instances of the pseudo-Boolean competition mainly contain linear constraints associating a weight of 1 to the variables. In this context, Pisinger's algorithm directly derives an optimal integer solution, therefore, it is unnecessary to use an exact approach, so we exclude them from the benchmarks. First, as expected this approach is very effective on pure knapsack problems, as long as the coefficients are not too large. On 60 instances with 50-200 variables taken from [Pisinger 2005] 38 are solved within the 2000s when computing a relaxed optimal solution against 58 when using an optimal solution. The average solving time on the instances solved by both approaches is 38s when using the relaxed optimal solution and 1s when using the optimal one. On the other instances, we experimented computing a relaxed optimal solution (Default), an optimal solution at every node (kpdp=0), only during preprocessing (kpdp=-1), every 1000 (kpdp=1000) and every 10000 (kpdp=10000) nodes. The results are shown in table 3.2, 1 KPCG and 3 warehouses instances are removed because unsolved with all the approaches. Solving the MCKP exactly at each node is prohibitive, with only 6 instances solved versus 35 when not using an exact approach. We observe that for some instances it can drastically decrease the number of nodes visited (18199 nodes and $1,080$ seconds versus 3703662 nodes and 230 seconds on one KPCG instance). However, note that our choice of implementing linear constraint as $\geq$ constraints with positive coefficients impacts the efficiency of the dynamic programming when considering $\leq$ constraints. Indeed, if the constraint is $2x_1 + 7x_2 + 8x_3 + 9x_4 + 10x_5 \leq 10$ then the equivalent $\geq$ constraint is $-2x_1 - 7x_2 - 8x_3 - 9x_4 - 10x_5 \geq -10$. When we remove the negative coefficients we get $2\bar{x_1} + 7\bar{x_2} + 8\bar{x_3} + 9\bar{x_4} + 10\bar{x_5} \geq 26$. The capacity went from 10 to 26, this will impact the dynamic programming but not the Pisinger's algorithm. For the instance cited above the number of variables is 500 and the capacity went from 450 to 29709.
Surprisingly, the number of nodes does not decrease when computing an optimal solution occasionally. The exact approach will modify the distribution of the costs and the branching decision. In particular, the cost distribution is impacted by the computed reduced costs, in the exact approach it depends on an ordering heuristically chosen. While an ordering heuristic only appears briefly in the Pisinger's algorithm when two variables have similar profit/weight. Therefore, this approach can be more "fair" and does not wrongly mislead the solver.

[3] The values verifying $x_{iv} = 1)$ necessarily have a reduced cost of 0.

Table 3.2: Comparison of the average solving time and the average number of nodes for different settings of the kpdp parameter. The number of instances for each class and the number of solved instances are written in parentheses.

	KPCG(21)	mdmkp(12)	warehouse(2)
Default	**413s** (21) 1132228	**97s** (12) 624681	1411s (2) 72088
kpdp=0	1838s (5) 77053	2000s (0) 5012	1780s (1) 295435
kpdp=-1	496 (20) 1141347	211s (12) 1648119	1646s (1) 300100
kpdp=1000	738s (16) 1825849	524s (12) 1836749	**1319s** (2) 80849
kpdp=10000	571s (20) 1380769	245s (12) 1689858	2000s (0) 2199623

Weighted degree heuristic

By default, TOULBAR2 uses the weighted degree heuristic [Boussemart *et al.* 2004b] to define the variable ordering. This heuristic associates a weight to each variable, those weights are updated whenever the solver reaches a conflict. The solver looks at the last propagated constraint and identifies a set of variables explaining the conflict, then it increases the weight of the variables within this explanation. For a linear constraint $c_{\boldsymbol{S}}$ the solver first verifies if the constraint is satisfiable under the current assignment. If this is the case then the conflict appeared because the lower bound is greater than the upper bound, and the last increase was made by the linear constraint. It increases the weight of all the variables in the scope. Otherwise, if the constraint is unsatisfiable, we use a simple greedy algorithm to identify the explanation (based on [Hebrard & Siala 2017]). We compute $MaxWeight$ the maximal achievable weight with the current domains $\boldsymbol{D}_i \setminus A_i$, where A_i corresponds to the values removed from $\boldsymbol{D}_i$. Then, $MaxWeight = \sum_{i \in \boldsymbol{S}} \max_{a \in \boldsymbol{D}_i \setminus A_i} w_{ia}$. If the constraint failed we know $MaxWeight < capacity$. We verify for each variable x_i (following the reverse DAC order[4]). If $MaxWeight - \max_{a \in \boldsymbol{D}_i \setminus A_i} w_{ia} + \max_{a \in \boldsymbol{D}_i} w_{ia} \geq capacity$, then x_i is part of the explanation. Indeed, if the value with initially the largest weight had not been removed then there would be no conflict. Otherwise, we set $MaxWeight \leftarrow MaxWeight - \max_{a \in \boldsymbol{D}_i \setminus A_i} w_{ia} + \max_{a \in \boldsymbol{D}_i} w_{ia}$ and continue.
We also add a specific behavior when all the weights of the variables are equal, in this case: we do nothing. This is specifically designed for the instances of the OPT-SMALL-INT competition where some instances have a large number of cardinality constraints. In this context, we observed that the weighed degree heuristic was not efficient. A possible reason for this failure is that in those instances conflicts often appear on several constraints at the same time, however, we always follow the same constraint ordering and stop at the first conflict. As a consequence, the

[4]We also tried to follow the DAC order, but it led to worse results

Table 3.3: Comparison of the average solving time and average number of nodes for different weighted degree strategies for linear constraint. The number of instances for each class and the number of solved instances are written in parentheses.

	KPCG(21)	mdmkp(12)	OPT-SMALLINT(21)	warehouse(3)
No increases	534s 1398969 (20)	277s 5537327 (12)	**269s** 744727 (21)	1597s 169992 (2)
Increase All	**523s** 1402069 (20)	275s 5537327 (12)	815s 4387631 (14)	**1528s** 195749 (3)
Increase Explanation	413s 1132228 (21)	**97s** 1454059 (12)	283s 698432 (20)	1607s 176702 (2)

procedure repeatedly increases the weights of the same possibly unmeaningful variables/constraints.
We tried several approaches:

- Do nothing.
- Increase the weights of all the variables.
- Increase the weight of an explanation as defined above.

Table 3.3 gives the results obtained for the different classes, 1 KPCG, 2 warehouses, and 3 OPT-SMALLINT instances are removed because unsolved by all the approaches. We can observe that increasing all the weights or none of them produces similar results for KPCG and mdmkp, but increasing all the weights is significantly worse in OPT-SMALLINT instances. This justifies our choice of ignoring the constraint where all the weights are equal. Using the explanation produces clear better results for KPCG and mdmkp, it solves one less instance on warehouse and OPT-SMALLINT. We choose to keep this heuristic as the default heuristic.

3.2 VAC on linear constraints

VAC [Cooper *et al.* 2010] is a local consistency algorithm aiming to increase the lower bound of a WCSP P by working on a derived CSP: Bool(P). For every cost function in P, only the tuples and values having a zero cost are allowed in Bool(P) (see definition 2.17 in chapter 2). If Bool(P) is inconsistent then it means the lower bound of P can be increased. If the inconsistency of Bool(P) is detected by AC, then VAC has been designed to extract a sequence of EPTs to increase $c_\varnothing$. It has shown good performance when used to obtain a strong initial lower bound. But whereas VAC has been defined on any WCSP, it would need in our case to enforce GAC on linear constraints with assignment costs, which is NP-Hard. Moreover, the linear constraints are not represented in extension, thus it is not possible to explicitly list all the tuples having a non-zero cost. However, we show that we can use algorithm Propagate to detect a subset of the inconsistent tuples. Hence, we design a new version of VAC (VAC-lin), where GAC is applied to any non-linear

constraints while we use algorithm Propagate to propagate linear constraints. Just like the original VAC, VAC-lin can be decomposed in 3 phases.

1. Establish an incomplete GAC in Bool(P), where algorithm Propagate is applied to linear constraints. If no conflict occurred then we can't prove that P is not VAC.

2. Given σ a sequence of arc consistency operations which led to a conflict in Bool(P). Find a minimal subsequence of σ triggering the conflict, for linear constraints we analyze the reduced costs and use conflict explanations [Hebrard & Siala 2017] to identify the necessary AC operations to obtain a conflict. Similarly to VAC we convert this subsequence in SAC operations and produce the maximum achievable increase λ of $c_\emptyset$ while keeping all costs non-negative.

3. Apply the sequence of SAC operations and go back to phase 1.

The theorem behind VAC (theorem 2.1 in section 2.4.1) can be extended to show VAC-lin is sound.

Theorem 3.3. *Let P be a WCSP such that $c_\emptyset < \top$. If applying GAC on non-linear constraints and our dedicated algorithm on linear constraints leads to a conflict, then there exists a sequence of soft arc consistency operations which when applied to P leads to an increase in $c_\emptyset$.*

Proof. Any value removed by our algorithm would have been removed by applying GAC on linear constraints. Theorem 2.1 concludes the proof. □

VAC has been extended with a heuristic where we associate a threshold θ to Bool(P) [Cooper *et al.* 2010]. Only the tuples and values having a cost $< \theta$ are allowed in Bool(P). If θ is high then if VAC discovers a conflict, it has more chance to be a large cost contribution. If θ is low then more tuples are forbidden in Bool(P) and VAC has more chance to discover a conflict but with a possibly small contribution. The heuristic consists of first applying VAC with a high θ in the hope of discovering large cost contributions and decreasing θ along the iterations. We directly integrate this in VAC-lin.

3.2.1 VAC-lin subroutines

In this section, we present the different algorithms used to enforce VAC-lin. Algorithm 1 presents how the 3 phases are coordinated. It first calls the first phase `VAC-lin-Phase1`. If a conflict occurs it calls the second phase `VAC-lin-Phase2`. Finally, if $c_\varnothing$ can be increased by a positive cost, it calls the last phase `VAC-lin-Phase3` to increase $c_\varnothing$. We highlight the modifications needed to integrate linear constraints in VAC-lin. To obtain a more coherent algorithm we adapt the former algorithm of VAC to a new notation. We describe the different structures we need to enforce VAC-lin:

- θ is a threshold. It is a positive cost, when building Bool(P) only the costs greater than θ are considered forbidden. In practice, θ decreases over the iterations of VAC. This heuristics tends to find large cost improvements first.

- P is a queue. It contains pairs of (value,constraint) that need to be propagated in Bool(P). Initially, it contains every possible pair (line 5), then whenever a value is removed, all the constraints linked to this value need to be propagated again (line 9).

- Q is a queue. It contains the values removed when applying incomplete GAC on Bool(P). Q is built in Phase 1 and used in Phase 2.

- M is a Boolean function. It indicates for each value $(i, a) \in Q$ whether its removal is necessary to trigger a conflict in Bool(P).

- killer associates for each value $(i, a) \in Q$ a constraint $c_{\boldsymbol{S}}$ whose propagation removed value (i, a) and an explanation for this removal. An explanation is a set of values such that if those values are removed from $c_{\boldsymbol{S}}$ then $c_{\boldsymbol{S}}$ propagates the removal of (i, a). We want the explanation to be as small as possible. A minimal explanation corresponds to an explanation such that every value in the explanation is necessary to deduce the removal of (i, a).

- λ is a positive cost, it corresponds to a cost movable to $c_\varnothing$. λ is computed in Phase 2.

- k associates to a value $(i, a) \in Q$ the quantity of quantum λ requested by the value. This quantity depends on the number of times the value appears in an explanation.

- $k_{\boldsymbol{S}}$ depends of the cost function $c_{\boldsymbol{S}}$. It associates to a value (i, a) the quantity of quantum that needs to be transferred from (i, a) to $c_{\boldsymbol{S}}$.

Algorithm 1: VAC-lin Main

```
Initialize all k, k_S to 0, λ ← ⊤ ;
conflict, explanation ← VAC-lin-Phase1() ;
if (conflict = ∅) then return;
foreach (i, a) ∈ explanation do
    k(i, a) ← 1, M(i, a) ← true;
    if (c_i(a) ≠ 0) then M(i, a) ← false, λ ← min(λ, c_i(a)) ;
λ ← VAC-lin-Phase2() ;
if λ > 0 then
    VAC-lin-Phase3();
```

Phase 1

In the first phase, VAC-lin considers Bool(P) and applies GAC on non-linear constraints and an incomplete GAC on linear constraints. It ends when a conflict appears or no more values can be removed. Each propagator returns a cost (OPT), a set of removed values ($Killed$), and an explanation for the removals ($KillerSet$). Whenever a value (i, a) is removed because it has no support on constraint $c_{\boldsymbol{S}}$, this value is added to a queue Q (line 6) and it is recorded by updating the killer structure: killer$(i, a) = (c_{\boldsymbol{S}}, KillerSet)$ (line 7). The function `SimplePass`$(i, c_{\boldsymbol{S}})$ describes how GAC is enforced on variable x_i and a non-linear constraint $c_{\boldsymbol{S}}$. The procedure is similar to VAC, if a value (i, a) is not GAC then it is removed, a zero cost and an empty explanation are returned. They will be automatically updated in Phase 2.

`LinPass1`$(c_{\boldsymbol{S}})$) presents how to enforce an incomplete GAC on a linear constraint $c_{\boldsymbol{S}}$ using algorithm Propagate. We differentiate two situations, either the constraint is conflicting and we want to obtain an explanation before going to phase 2. Or the constraint is not conflicting and we verify if some values can be removed. The constraint can be conflicting if it is not feasible or if its optimal cost is $> \theta$ (none of its tuples are allowed in Bool(P)). If the linear constraint is not feasible then it computes a minimal explanation using conflict explanation [Hebrard & Siala 2017] (line 2) and goes to phase 2. Otherwise, it removes the values that are not domain consistent (line 1) and computes an optimal solution of $LP_\emptyset$ with the associated reduced costs. Notice that, in Bool(P), $LP_\emptyset$ and $LP_{F\emptyset}$ are equivalent because the unary costs of the remaining variables are all zeros. If the optimal cost OPT verifies $OPT > \theta$ (line 3), then we reached a conflict, an explanation can be obtained by analyzing the reduced costs of the removed values. Indeed, if $rc(i, a) < 0$ then the minimal cost solution with $x_{ia} = 1$ has a cost lower than OPT, therefore the removal of (i, a) is necessary to preserve an optimal cost OPT. The set of values verifying $rc(i, a) < 0$ explains the conflict. However, this explanation might not be minimal. The solver returns, OPT and the explanation and moves to phase 2.

If the constraint is not conflicting, then $OPT < \theta$ (line 4). We want to verify if some values are not GAC with the constraint. In our case, it corresponds to values having a non-zero minimal cost tuple. We once again use the reduced costs to detect a subset of those values. A value (i, a) verifying $rc(i, a) > \theta - OPT$ is not GAC. Indeed, its minimal cost tuple cost at least $OPT + rc(i, a) = \theta$, therefore (i, a) can be removed from Bool(P). We can't detect without extra computational work a dedicated explanation for each removal. Therefore, the solver computes a very straightforward explanation containing all the previously removed values. If possible this set is refined in Phase 2. A cost of 0, the set of all the values removed in `LinPass1`, and the explanation is returned.

In the case a domain wipe-out occurs, then VAC-lin returns the values of the variable and goes to Phase 2 (line 8). To limit the computation time of VAC-lin, we prioritize applying GAC on non-linear constraints and then incomplete GAC on linear constraints.

Algorithm 2: VAC-lin iteration - Phase 1: Instrumented AC

(* Revise variable i w.r.t. constraint $c_{\boldsymbol{S}}$ *);
Function `SimplePass`$(i, c_{\boldsymbol{S}})$
  $Killed \leftarrow \emptyset$;
  foreach $a \in d_i$ **do**
    if $\nexists t \in \tau(\boldsymbol{S})$ *s.t.* $t_i = a$ *and* $c_{\boldsymbol{S}}(t) \neq \top$ **then**
      $Killed \leftarrow Killed \bigcup (i, a)$;
  return *(0,*$\emptyset$*,*$Killed$*)*;

Function `LinPass1`$(c_{\boldsymbol{S}})$
  `/* Domain-Consistency(`$c_{\boldsymbol{S}}$`) returns the values removed by domain consistency */`
1 $BP \leftarrow$ `Domain-Consistency(`$c_{\boldsymbol{S}}$`)` ;
  if $c_{\boldsymbol{S}}$ *is not satisfiable* **then**
2    $KillerSet \leftarrow$ `Minimal-Explanation()`;
    return $(0, KillerSet, \emptyset)$
  $OPT \leftarrow$ `Optimal-Relaxed-Solution(`$c_{\boldsymbol{S}}$`)`;
3 **if** $OPT > \theta$ **then**
    $KillerSet \leftarrow \{(i, a) \mid rc(i, a) < 0\}$;
    return $(OPT, KillerSet, \emptyset)$
4 **else**
    $KillerSet \leftarrow \{(i, a) \mid (i, a)$ has been removed in Bool(P) $\}$;
    $Killed \leftarrow \{(i, a) \mid rc(i, a) > \theta - OPT\} \bigcup BP$;
    return *(0,*$KillerSet$*,*$Killed$*)*

Function `VAC-lin-Phase1()`
5 $P \leftarrow \{(i, c_{\boldsymbol{S}}) \mid c_{\boldsymbol{S}} \in C, i \in \boldsymbol{S}\}$;
  while $P \neq \varnothing$ **do**
    $(i, c_{\boldsymbol{S}}) \leftarrow P.Pop()$;
    if $c_{\boldsymbol{S}}$ *is a linear constraint* **then**
      $(OPT, KillerSet, Killed) \leftarrow$ `LinPass1(`$c_{\boldsymbol{S}}$`)`;
      if $OPT \neq 0$ **then**
        return $(c_S, KillerSet)$
    else
      $(OPT, KillerSet, Killed) \leftarrow$ `SimplePass(`$c_{\boldsymbol{S}}$`)`;
    foreach $(i, a) \in Killed$ **do**
      delete a from d_i;
6      $Q.Push(i, a)$;
7   killer$(i, a) \leftarrow (c_{\boldsymbol{S}}, KillerSet)$;
8   **if** $d_i = \varnothing$ **then return** $(i, \bigcup_{a \in d_i}(i, a))$;
9   **else** $P \leftarrow P \cup \{(j, c_{S'}) \mid c_{S'} \in C, S' \neq S, \{i, j\} \subset S', j \neq i\}$;
  return $(\emptyset, \emptyset)$;

Phase 2

In the second phase, we want to trace back the operations leading to a conflict in Bool(P) and collect a minimal subset of value deletions that is sufficient to explain it. We use a Boolean function M to mark the values with a zero cost in P necessary to explain the conflict. Initially, only the values in the explanation returned by phase 1 are marked. The marked values are also added to the queue R (line 20), this queue will be used in Phase 3. Our objective is to identify a set of values/tuples with non-zero costs that can be used as a source to move costs to the marked values. Ultimately if we move a cost to the values in the explanation returned by phase 1 then we can increase $c_\varnothing$. To do this, the solver revisits the queue Q starting from the last inserted value. For each marked value (i, a), it studies the explanation stored in killer (i,a). The idea is to use the values/tuples captured by killer (i,a) as a source to move cost to (i, a). If it meets a non-zero cost then we found a source, otherwise, if it meets a zero cost then the value is marked. In this phase, we also want to compute the maximal cost λ that can be sent to $c_\varnothing$. We use the same data structure as the one described in VAC: $k(i, a)$ corresponds to the number of quantum requested by value (i, a) and $k_{\boldsymbol{S}}(i, a)$ the number of quantum that (i, a) must extend to $c_{\boldsymbol{S}}$. We have $k(i, a) = \sum_{\boldsymbol{S}\in\boldsymbol{C}, i\in\boldsymbol{S}} k_{\boldsymbol{S}}(i, a)$. Similarly, $k(\boldsymbol{S}, \tau)$ defines the quantity of costs requested by tuple τ in cost function $\boldsymbol{S}$. Those requests are linked to the number of times a value/tuple appears in an explanation. We choose λ to be the maximal cost satisfying all the requests. For example, if a cost of 4 is available on value (i, a) and $k(i, a) = 2$, then $\lambda \leq \frac{4}{2} = 2$. Once again the procedure is different if the explanation is a non-linear or a linear constraint.

If the removal was propagated by a non-linear constraint then the procedure remains the same as in VAC (line 21). Let $c_{\boldsymbol{S}}$ be a non-linear constraint responsible for the removal of (i, a). We know that in Bool(P) every tuple $\tau \in \ell(\boldsymbol{S})$ verifying $\tau_i = a$ is forbidden. Either τ already have a cost $c_{\boldsymbol{S}}(\tau) > \theta$ in P (line 22) and was directly removed from Bool(P). In this case, we increase $k(\boldsymbol{S}, \tau)$ and verify if λ needs to be updated by computing $\frac{c_{\boldsymbol{S}}(\tau)}{k(\boldsymbol{S},\tau}$.

If $c_{\boldsymbol{S}}(\tau) = 0$, then there exists a value (j, b) removed in Bool(P) such that $\tau_j = b$. Value (j, b) explains the removal of τ and we trace it back (line 23). We update the different structure in function `Update-Structures`$((i, a), c_{\boldsymbol{S}}, (j, \tau_j))$.Structures, $k(j, b)$ and $k_S(j, b)$ are increased by $k(i, a)$, if $c_j(b) < \theta$ then the value is marked, otherwise λ is compared to $\frac{c_j(b)}{k(j,b)}$.

If a value (i, a) has been removed due to a linear constraint, we want to compute the minimal cost tuple with $x_i = a$ in the linear constraint. The procedure is described by the function `LinPass2` (line 10), it returns a cost corresponding to an approximation of the minimal cost tuple. There exist several possibilities. First, the minimal cost tuple with $x_i = a$ can also verify $x_b = j$ for some $(j, b) \in \mathsf{killer}(i, a)$. This can be discovered by considering a modified $LP_\emptyset$ where we add the constraints $x_{ia} = 1$ (line 10) and $x_{jb} = 0\ \forall (j, b) \in \mathsf{killer}(i, a)$ (line 11). If this new problem is not feasible then there exists no tuple verifying $x_i = a$ and $x_j \neq b\ \forall (j, b) \in \mathsf{killer}(i, a)$. Therefore, the minimal cost tuple necessarily verifies $x_b = j$ for some $(j, b) \in \mathsf{killer}(i, a)$. In this

case, the minimal cost tuple costs at least $\min_{(j,b)\in \mathsf{killer}(i,a)} c_j(b)$. Moreover, there exists no tuple with $x_{jb} = 0\ \forall (j,b) \in \mathsf{killer}(i,a)$ forbidden in Bool(P) but allowed in P. Therefore, it means that there is no tuple in $c_{\boldsymbol{S}}$ that may limit the value of λ. Hence, `LinPass2` returns a cost of $\lambda \times k(i,a)$. We use conflict explanation [Hebrard & Siala 2017] to refine the explanation (line 12).
Otherwise, if a solution exists with $x_{jb} = 0, \forall (j,b) \in \mathsf{killer}(i,a)$ and $x_{ia} = 1$, then it computes an optimal solution of the modified $LP_{\emptyset}$. Note that the values in $\mathsf{killer}(i,a)$ are forbidden, but we still analyze their reduced costs. The values verifying $rc(x_{jb}) \leq -OPT$ need to be traced back (line 13) and define the explanation of the removal. Indeed, the minimal cost tuple with $x_{jb} = 1$ and $x_{ia} = 1$ has a cost of at most $OPT - OPT = 0$, hence it is necessary to move some costs from x_{jb} to the linear constraint to obtain a non-zero cost on all the tuples verifying $x_i = a$. Otherwise, if $-OPT < rc(x_{jb}) < 0$, then the minimal cost tuple τ with $\tau_j = b$ verifies $c_{\boldsymbol{S}}(\tau) > 0$, to be more precise $c_{\boldsymbol{S}}(\tau) = OPT + rc(x_{jb})$. VAC-lin stops tracing whenever it finds a positive cost, therefore it is not necessary to trace back value (j,b). However, the cost τ (or the approximation of cost we have) needs to be returned as it may limit the value of λ line 14). It is interesting to point out that another strategy would have been to not consider τ as a limiting tuple and trace back (j,b). In Phase 3 the cost of τ would have been increased by the cost moved from (j,b) to the linear constraint and ensure us that no negative cost would be created. However, we have no guarantee that this would increase λ, indeed, by tracing (j,b) we might find a cost limiting the value of λ (or not). Moreover, VAC-lin works in iteration, hence if a better increase of $c_\varnothing$ is possible by tracing (j,b), this will be discovered in the next iteration of VAC-lin.
Finally, in both cases, the k, k_S structures are updated, the quantity of quantum requested by the values in the minimal explanation is increased by the quantity of quantum requested by (i,a) (lines 16,17) and those values are marked (structure M) if their unary cost is null (line 18). We also update λ (line 19) if necessary.

Phase 3

Finally, in phase 3, all EPTs are performed according to $k, k_{\boldsymbol{S}}$ structure and λ. After this sequence, we know a cost of λ can be moved to $c_\varnothing$ (line 24). In the case the conflict appeared in a linear constraint, then it is possible to apply algorithm Propagate (line 25), this is useful to detect if we can increase $c_\varnothing$ by a cost higher than the λ we approximated. Example 3.7 illustrates how to enforce VAC-lin on the

WCSP 3.1.
The space complexity of VAC-lin is dominated by the killer structure, for each constraint we associate to each value an explanation with maximal size $arity - 1$. Hence the space complexity is $O(er^2d)$ (vs $O(erd)$ for VAC), where r is the largest arity, e the number of constraints and d the maximal domain size. As for time complexity, enforcing GAC on non-linear constraints can be done in $O(nd^re)$ while algorithm

Algorithm 3: VAC-lin iteration - Phase 2: Computing λ

Function `LinPass2`$(c_{\boldsymbol{S}}, KillerSet, (i,a))$

Fix $x_{ia} = 1$;

foreach $(j,b) \in KillerSet$ **do** Fix $x_{jb} = 0$;

if $c_{\boldsymbol{S}}$ *is not satisfiable* **then**

killer (i,a).second $\leftarrow$ `Minimal-Explanation()`;

return $k(i,a) \times \lambda$

$OPT \leftarrow$ `Optimal-Relaxed-Solution`$(c_{\boldsymbol{S}})$;

killer (i,a).second $\leftarrow \{(j,b) \in KillerSet \mid rc(j,b) \leq -OPT\}$;

$Mintuple \leftarrow \min_{-OPT<rc(j,b)\leq 0} rc(j,b)$;

return $OPT + Mintuple$

Function `Update-Structures`$((i,a), c_{\boldsymbol{S}}, KillerSet)$

foreach $(j,b) \in KillerSet$ **do**

$k(j,b) \leftarrow k(j,b) + k(i,a)$;

$k_{\boldsymbol{S}}(j,b) \leftarrow k_{\boldsymbol{S}}(j,b) + k(i,a)$;

if $(c_j(b) = 0)$ **then** $M(j,b) \leftarrow$ true ;

else $\lambda \leftarrow \min(\lambda, \frac{c_j(b)}{k(j,b)})$;

Function `VAC-lin-Phase2()`

while $(Q \neq \varnothing)$ **do**

$(i,a) \leftarrow Q.Pop()$;

if $(M(i,a))$ **then**

$R.Push(i,a)$;

$c_{\boldsymbol{S}} \leftarrow$ killer$(i,a).first$;

if $c_{\boldsymbol{S}}$ *is not a linear constraint* **then**

foreach $\tau \in \ell(\boldsymbol{S})$ *s.t.* $\tau_i = a$ **do**

if $(c_{\boldsymbol{S}}(\tau) \neq 0)$ **then**

$k(\boldsymbol{S}, \tau) \leftarrow k(\boldsymbol{S}, \tau) + k(i,a)$;

$\lambda \leftarrow \min(\lambda, \frac{c_{\boldsymbol{S}}(\tau)}{k(\boldsymbol{S},\tau)})$;

else

Let $j \in \boldsymbol{S}, j \neq i$ be a variable that invalidates τ in Bool(P);

`Update-Structures`$((i,a), c_{\boldsymbol{S}}, (j, \tau_j))$

else

$OPT \leftarrow$ `LinPass2`$(c_{\boldsymbol{S}},$ killer$(i,a).second, (i,a))$;

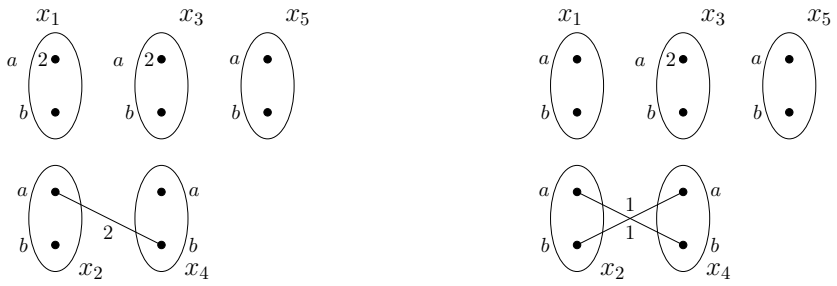
$OPT \leftarrow \frac{OPT}{k(i,a)}$;

if $OPT < \lambda$ **then** $\lambda \leftarrow OPT$;

`Update-Structures`$((i,a), c_{\boldsymbol{S}},$ killer$(i,a).second)$

Algorithm 4: VAC-lin iteration - Phase 3: Applying equivalence-preserving transformations

```
Function VAC-lin-Phase3(conflict)
    while (R ≠ ∅) do
        (j, b) ← R.Pop() ;
        c_S ← killer(j, b).first ;
        if c_S is not a linear constraint then
            foreach i ∈ S, i ≠ j, a ∈ D_i s.t. k_S(i, a) ≠ 0 do
                Extend(i, a, S, λ × k_S(i, a));
                k_S(i, a) ← 0 ;
            Project(S, j, b, λ × k(j, b)) ;
        else
            foreach (i, a) ∈ killer(j, b).second s.t. k_S(i, a) ≠ 0 do
                Extend(i, a, c_S, λ × k_S(i, a));
                k_S(i, a) ← 0 ;
            Project(S, j, b, λ × k(j, b)) ;
24  if conflict is a variable then UnaryProject(conflict, λ);
    else
        /* conflict is a linear constraint                    */
        OPT ← Optimal_Relaxed_Cost(conflict);
25      if OPT ≥ λ then Propagate(conflict);
        else Project(conflict, c_∅, λ);
```



$ctr1 : 7x_{1a} + 7x_{2a} + 3x_{3a} + 3x_{4a} + 3x_{5a} \geq 10$
$ctr2 : x_{1a} + x_{4b} \geq 1$
$c_\varnothing = 0$

$ctr1 : 7x_{1a} + 7x_{2a} + 3x_{3a} + 3x_{4a} + 3x_{5a} \geq 10$
$ctr2 : x_{1a} + x_{4b} \geq 1$
$c_\varnothing = 1$

Figure 3.1: (left) The original WCSP. (right) The WCSP after propagation of VAC-lin (the δ costs associated with the linear constraints are not written).

Propagate needs at most $O(rdlog(rd))$. Concerning phase 2, there are at most nd values in the queue P, each value is either associated with a non-linear constraint and it needs at most $O(d^{r-1})$, or with a linear constraint and it needs at most $O(rdlog(rd))$. If r is sufficiently large then the time complexity is $O(nd^r)$. Finally an $O(ed^r + rdlog(rd))$ applies to phase 3.

Example 3.7. *Let P be a WCSP with 5 Boolean variables $\{x_1, x_2, x_3, x_4, x_5\}$ with domain $\{a, b\}$, and constraints $ctr1 : 7x_{1a} + 7x_{2a} + 3x_{3a} + 3x_{4a} + 3x_{5a} \geq 10$, $ctr2 : x_{1a} + x_{4b} \geq 1$, $c_1(a) = 2$, $c_3(a) = 2$, and $c_{24}(a, b) = 2$. This is depicted by figure 3.1. Using algorithm Propagate does not increase the LB.*
If we apply VAC-lin. In Bool(P)*, x_{1a} and x_{3a} are directly removed, it follows by domain propagation on $ctr1$ that x_{2b} can be removed and we set* killer$(2, b) = \{x_{1a}, x_{3a}\}$*. It follows that x_{4b} is not AC with x_2 and can be removed. Finally $ctr2$ is infeasible with explanation $\{x_{1a}, x_{4b}\}$,* Bool(P) *is not AC.*
We set $\lambda = \top = 20$ and start tracing back the AC operations. $ctr2$ is infeasible because x_{1a} and x_{4b} has been removed. We directly have $c_1(a) = 2$ we can update the k structure $k(1, a) = 1$ and λ: $\lambda = \frac{c_1(a)}{k(1,a)} = 2$. Value x_{4b} has been removed because it has no support on c_{24}, we update the k of the two tuples satisfying $x4 = b$: $k(\{2, 4\}, (a, b)) = 1$, $k(\{2, 4\}, (b, b)) = 1$. The first tuple verifies $\frac{c_{24}(a,b)}{k(\{2,4\},(a,b))} = 2 \geq \lambda$. The second tuple verifies $c_{24}(b, b) = 0$: we need to trace back the removal of x_{2b}. We observe that if $x_{2b} = 1, x_{1a} = 0$ and $x_{3a} = 0$ then the constraint $ctr1$ is unsatisfied. We detect a minimal explanation:

- *The initial maximal achievable weight is 6.*
- *If x_{1a} is allowed then the maximal achievable weight is $6 + 7 = 13 \geq 10$. We add x_{1a} to the explanation and continue.*
- *If x_{3a} is allowed then the maximal achievable weight is $6 + 3 = 9 < 10$. Hence, x_{3a} is not needed to explain the conflict, we update the maximal achievable*

weight to 9.

We continue by tracing again x_{1a}, we update the k structure: $k(1,a) = k(1,a)+1 = 2$. We need to update λ: $\lambda = \frac{c_1(a)}{k(1,a)} = 1$.
We deduce the following EPTs:

- $extend(c_1, \{x_1 = a\}, ctr1, 1)$
- $project(c_2, ctr1, \{x_2 = b\}, 1)$
- $ectend(c_2, \{x_2 = b\}, c_4, 1)$
- $project(c_2, c_{24}, \{x_4 = b\}, 1)$
- $extend(c_4, \{x_4 = b\}, ctr2, 1)$
- $extend(c_1, \{x_1 = a\}, ctr2, 1)$

Constraint ctr2 now propagates a cost of 1 (see 3.1). ■

3.2.2 Discussion on VAC-lin

VAC-lin shares the same flaws as VAC but it exaggerates them. Indeed, both algorithms can produce small increases of lower bound leading to a very large (or even unbounded) number of iterations. In VAC-lin, this is enhanced by the fact we do several approximations when we apply VAC-lin. First, we over-estimate the number of requested quantum λ (k data structure) to ensure that no negative cost will be created. The approximation arises when one variable (i, a) is responsible for multiple removals through the same constraints. In this case the request counter $k(i, a)$ will be increased at each removal. However, depending on the cost distribution within the constraint it is possible to do better, as shown in example 3.8. For linear constraints, it requests to solve once again a modified $ILP_{\emptyset}(c_S)$. Note that this flaw has never been highlighted in the original VAC because it has only been implemented for binary constraints. Secondly, in function `LinPass2` from phase 2, we only get an approximation of the minimal cost tuple and the explanation. Obtaining better information would require solving an ILP or modifying `LinPass2` to take λ as a parameter. However, if λ is a parameter of `LinPass2`, then whenever λ is modified, we would need to re-do all the previous calls to `LinPass2` with the updated λ.

Example 3.8. *Suppose the following pattern of 3 Boolean variables and one ternary constraint appears in a bigger problem. The unary costs are all zeros except $c_3(a) = 2$, the ternary constraint is defined by table 3.4.*
If we apply VAC on this problem then when applying Phase 1, value $(3, a)$ is removed from Bool(P)*. With the removal of $(3, a)$ then $(1, a)$ and $(2, a)$ are not GAC with the ternary constraint. Suppose now a conflict occurred and we go to phase 2. Furthermore, suppose that both $(1, a)$, $(2, a)$ appear in the explanation and we found $k(1, a) = 1, k(2, a) = 1$. Then when tracing $(1, a)$, we get that $(3, a)$ is responsible*

x_1	x_2	x_3	Cost
a	a	a	5
a	a	b	5
a	b	a	0
a	b	b	5
b	a	a	0
b	a	b	5
b	b	a	5
b	b	b	0

Table 3.4: Ternary constraint

for the removal and set $k(3,a) = 1$ *with* $\lambda = 2$. *When tracing* $(2,a)$, $(3,a)$ *is again responsible and we set* $k(3,a) = 1+1 = 2$ *and* $\lambda = 1$. *However, in this situation, it is possible to keep* $\lambda = 2$. *Indeed, if* $(3,a)$ *transfers a cost of 2 to the ternary constraint then it is possible to transfer a cost of 2 to both* $(1,a)$ *and* $(2,a)$ *without introducing negative costs. This is possible only because* $c_{1,2,3}(a,a,a) > 2 \times \lambda = 4$, *otherwise,* $\lambda = 1$ *is the best we can do.* ■

3.3 Detecting Exactly One constraints

As described in section 3.1 each linear constraint is associated with a partition of its variables into EO sets. The intuitive partition we described is ideal to assert that multi-valued variables are assigned to exactly one value, but the algorithm will work the same way with any partition. This is interesting as we could capture conflicts involving the values of the linear constraint and encode them using an EO constraint. This extra information can help to derive a better lower bound. Unfortunately, to keep an MCKP we can't associate a linear constraint with more than one partition. However, we can observe that if for a given partition the values of multiple-valued variables all belongs to the same set then every solution of the associated MCKP implicitly verifies the unicity of assignment of the multi-valued variables. Indeed, each variable appears in only one EO set and for every EO set exactly one value will be chosen, therefore each variable will be assigned to at most one value. Our objective is now to associate a partition to each linear constraint, either it can be given as an input or we can define a procedure to detect and select them as in [Ansótegui *et al.* 2019] (for SAT) or [de Givry & Katsirelos 2017](where the constraints are directly added to the problem). We implemented a two-step procedure that first creates a conflict graph before finding a clique cover of each sub-graph induced by a linear constraint.

Conflict Graph

A conflict graph captures conflicts between two values. Each vertex corresponds to a value and an edge between two vertices expresses that no tuple using both

values can be part of an optimal solution. To obtain a conflict graph we assign one variable and propagate this information, if a value is removed then we create an edge between this value and the value assigned. We repeat this process for all possible values. Observe that a value can be removed either by a hard constraint e.g. domain consistency on a linear constraint, or by node consistency. The latter case requires a good initial lower and upper bound. Computing this conflict graph can be expensive as it needs to apply EDAC on non-linear constraints and the algorithm Propagate on linear constraints after each assignment. In practice on large instances, we only apply EDAC and domain propagation on linear constraints.

Clique Cover Detection

A clique in a graph is a subset of vertices such that every pair of vertices is adjacent. A clique in a conflict graph corresponds to an AMO constraint. A clique cover of the conflict graph defines a partition of the values into AMO constraints, which is exactly what we want to associate with our linear constraints (an AMO constraint can easily be transformed into an EO constraint by adding one variable corresponding to "not selecting any of the values in the AMO constraint"). The larger the clique the more likely it will be helpful during the search, hence we ideally want to obtain a minimum clique cover, but this is NP-hard to obtain. Instead, we opted for a simple greedy algorithm, it begins with an empty set of cliques, and then, for each variable, it selects the value having the largest degree in the conflict graph and tries to add it to an existing clique, if it is not possible then it creates a new clique with only this value. We can obtain a partition for a given linear constraint by finding a clique cover on the subgraph of the conflict graph induced by the values of the linear constraint (see Algorithm 5). It is also possible to run the clique detection on the whole conflict graph and add some of them directly to the problem.

3.4 Results

We implemented our approach in TOULBAR2, an exact WCSP solver in C++[5]. TOULBAR2's default variable ordering heuristic is the weighted degree heuristic [Boussemart *et al.* 2004b] with additional last conflict heuristic [Lecoutre *et al.* 2009]. By default TOULBAR2 uses the soft local consistency EDAC [de Givry *et al.* 2005] to compute bounds on binary table constraints, and our dedicated propagator for linear constraints. For warehouse, CPD, and some instances of the PB competition we also give the results when enforcing VAC or VAC-lin in preprocessing (VAC is implemented only for binary constraints). For all tests, we imposed a time limit of 30 minutes (except for CPD with 1 hour and 10 hours for Warehouse) on a single core of an Intel Xeon E5-2680 v3 at 2.50 GHz and 256 GB of RAM. We compared our PB propagator with other modeling approaches in protein design. We compared

[5] https://github.com/toulbar2/toulbar2 version 1.2.

Algorithm 5: AMO detection for linear constraints

```
Function Constraint_Graph(X, D)
    Create an empty graph G;
    foreach i ∈ X, a ∈ D_i do
        Add vertex (i, a) to G;
    foreach i ∈ X, a ∈ D_i do
        Assign x_i = a;
        /* Here, Propagate() returns the set of removed values.
           */
        Removed_Values ← Propagate();
        foreach (j, b) ∈ Removed_Value do
            Add edge (i, a) − (j, b) to G ;
    return G

Function CliqueCover(G, S)
    H ← G(S);
    Sort S in decreasing order of maximum degree in H;
    Create an empty list CC;
    foreach i ∈ S do
        Find a ∈ D_i such that deg((i, a)) in H is the largest;
        Add_To_Partition ← false;
        k ← 0;
        while k < |CC| and not Add_To_Partition do
            if (i, a) can be inserted in CC[k] then
                CC[k].append((i, a));
                Add_To_Partition ← true;
            k ← k + 1;
        if not Add_To_Partition then
            CC.append([(i, a)]);
    return CC
```

TOULBAR2 to the state-of-the-art ILP solver CPLEX 20.1 on knapsack problems with conflict graphs. We also compared TOULBAR2 on Pseudo Boolean Competition 2016 (PB16), the results are not competitive with recent PB solvers but those instances were previously out of reach by TOULBAR2.

3.4.1 Pseudo Boolean Competition 2016

We tested TOULBAR2 on the 1600 instances OPT-SMALLINT-LIN proposed by the PB16 competition.[6] All instances are Boolean and made up exclusively of linear constraints,[7] the number of variables, constraints, and the arity of the constraints vary greatly. We compared TOULBAR2 to the PB solvers NAPS [Sakai & Nabeshima 2015] v1.02b (winner of PB16), ROUNDINGSAT [Devriendt *et al.* 2021] v2 and to ILP solvers, CPLEX and SCIP v7.0.2 (SoPlex). We applied the settings to seek an exact solution with CPLEX ($epagap = epgap = epint = 0$ and $eprhs = 10^{-9}$), the default parameters were used for the other solvers. On all the instances, 1200 are solved by at least one solver. We report the number of solved instances in less than 1800 seconds by each solver for each family of the competition in Table 3.6. Table 3.5 reports for each solver the number of instances solved exclusively by this solver, and compares two-by-two the solving times. We consider a solver faster than another if it is at least 5% faster or if the instance is solved by one solver but not by the other one. Our approach does not seem ideal for this kind of problem, the effectiveness of TOULBAR2 is uneven between the families, it can be competitive (*caixa,primesdimacsnf...*) or not suited at all (*rand,radar...*). According to Table 3.5 TOULBAR2 is almost dominated by CPLEX and ROUNDINGSAT, the explanation may be that the PB16 competition is mainly composed by instances with a large number of constraints with a large arity, in this case, TOULBAR2 propagation might be slower than ROUNDINGSAT and weaker than CPLEX. The 5 instances only solved by TOULBAR2 are from the sub-family *auto-corr_ bern* (31 instances) and edgecross (20 instances) of the *minlplib2-pb-0.1.0* family, where constraints have a small arity (5 max). The same experiments were performed on the family *lion9-single-obj* from the OPT-BIGINT-LIN, these results show that our approach can also be applied to coefficients of large size without exceeding the state-of-the-art solvers except with rare exceptions.

We also experimented using VAC-lin in pre-processing on those instances. The full result is not reported as in many instances VAC-lin doesn't increase the LB or is too time-consuming to enforce. However, it is effective on some families of instances, such as the *area* family (*area_ delay*, *area_ Delay*, *area_ Opers*, *area_ partials*, *trarea_ ac*). Figure 3.2 shows the effect of enforcing VAC or VAC-lin in preprocessing on the solving time of the 69 instances of the *area* family. In practice, enforcing VAC is not worth it, it increases the lower bound for only 5 instances and is slower than the default TOULBAR2 on every instance. It was expected as VAC has been implemented only for binary constraints and those instances don't contain a large network of bi-

[6] http://www.cril.univ-artois.fr/PB16/

[7] We transform each equality constraint into two inequality constraints.

Table 3.5: Comparison of the solving time on the OPT-SMALLINT-LIN Pseudo Boolean Competition 2016, an entry reports how many times the solver in the column is faster than the solver in the row

	RoundingSat	NaPS	tb2	scip	cplex
RoundingSat	–	382	114	397	690
NaPS	677	–	242	601	761
tb2	924	694	–	792	871
scip	751	577	314	–	923
cplex	440	396	111	136	–
Exclusively solved	17	21	5	2	30

Table 3.6: Results for the Pseudo Boolean Competition 2016

Number of instance	RSat	NaPS	tb2	scip	cplex	VBS
area_delay,Delay,_opers,_partials	**59**	57	53	50	58	**59**	59
(bounded)_golomb_rulers	34	**12**	**12**	9	10	10	12
caixa	21	**21**	**21**	**21**	**21**	**21**	21
courseass	5	3	4	1	**5**	**5**	5
data	89	34	19	6	40	**48**	49
decomp	10	2	**8**	0	0	0	8
domset	15	0	0	0	0	0	0
dtproblems	60	**60**	40	56	**60**	**60**	**60**
EmployeeScheduling	17	0	12	0	12	**14**	14
factor	192	**192**	**192**	**192**	**192**	**192**	192
fctp	31	**31**	16	1	**31**	**31**	31
featureSubscription	20	19	**20**	0	0	0	20
flexray/fome	10	4	4	0	4	**5**	5
frb*opb	40	0	**16**	0	4	4	16
garden	7	5	**6**	5	**6**	**6**	6
graca	20	**20**	**20**	3	16	12	20
haplotype	8	**8**	8	0	0	0	8
heinz	45	22	18	6	22	**24**	30
j*opt	139	98	**100**	90	88	94	100
kullmann	7	1	0	0	**2**	**2**	2
logicsynthesis	74	62	34	32	70	**71**	71
marketsplit	40	9	8	8	11	**12**	12
milp, minlplib2,miplib/2003/3,mps	248	106	76	79	142	**151**	164
pbfvmcformulae	22	8	1	0	9	**16**	16
primesdimacscnf	156	128	121	118	**132**	129	133
radar	12	6	0	0	**12**	**12**	12
randbiglist,newlist	44	**44**	**44**	12	**44**	**44**	44
routing	15	**15**	**15**	14	**15**	**15**	15
sroussel	60	**23**	3	0	18	20	29
synthesisptlcmoscircuits	10	**10**	4	7	**10**	**10**	10
testset	6	**6**	**6**	**6**	**6**	**6**	6
trarea_ac	10	9	3	7	**10**	**10**	10
ttp	8	**2**	**2**	**2**	**2**	**2**	2
unibo	36	12	3	0	4	**14**	18
vtxcov,wnq	30	0	0	0	0	0	0
TOTAL	1600	1030	890	728	1057	**1100**	1201
BIGINT lion-single-obj	216	**196**	124	111	180	187	202

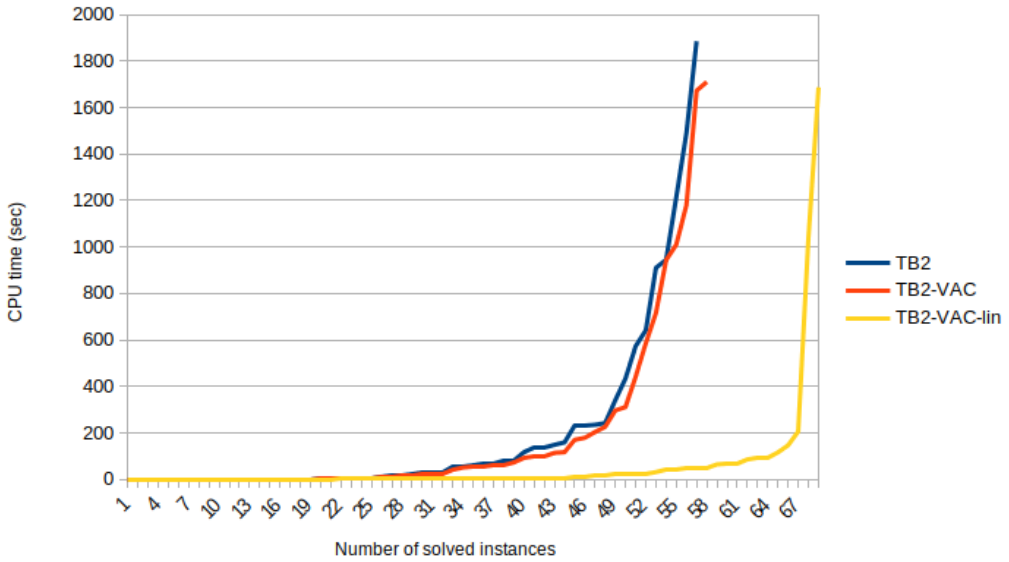


Figure 3.2: Cactus plot of CPU solving time for TOULBAR2 with VAC or VAC-lin in preprocessing for the OPT-SMALLINT-LIN area family.

nary constraints. However, VAC-lin shows good performances, on the 69 instances, 67 are solved within 1800 seconds when using VAC-lin in preprocessing and 56 without. The average time to solve those 56 instances is divided by 2 when enforcing VAC-lin in preprocessing, it goes from 164 seconds without VAC-lin to 80 seconds with VAC-lin. This approach is even competitive with SCIP and ROUNDINGSAT as shown in figure 3.3. However, CPLEX is still significantly more efficient on this benchmark.

3.4.2 XCSP3 competition

TOULBAR2 participated in the track Mini COP of the XCSP competition in 2022 and 2023[8]. All the instances are modeled in the XCSP format which includes all the major global constraints used in the CP community [Boussemart *et al.* 2016]. For example, you can find constraints to model comparison (AllDifferent, AllEqual...) counting (Sum, Cardinality...), or scheduling (Cumulative...). We encoded most of those constraints as a set of linear constraints. The score of the solvers reported in table 3.4.2,3.4.2 depends on the quality of the best upper bound and if they proved optimality. The result of the competition shows that TOULBAR2 was able to be competitive with other CP solvers. It is second in 2022[9] and we can observe it is the solver with the highest number of instances solved to optimality (see table 3.4.2). We participated in XCSP 2023 competition with 2 different solvers. Among the differences, LINTOULBAR2 includes VAC-lin but not TOULBAR2 . The instances

[8] https://www.xcsp.org/competitions/

[9] Full results: https://www.cril.univ-artois.fr/XCSP22/competitions/cop/mini-cop

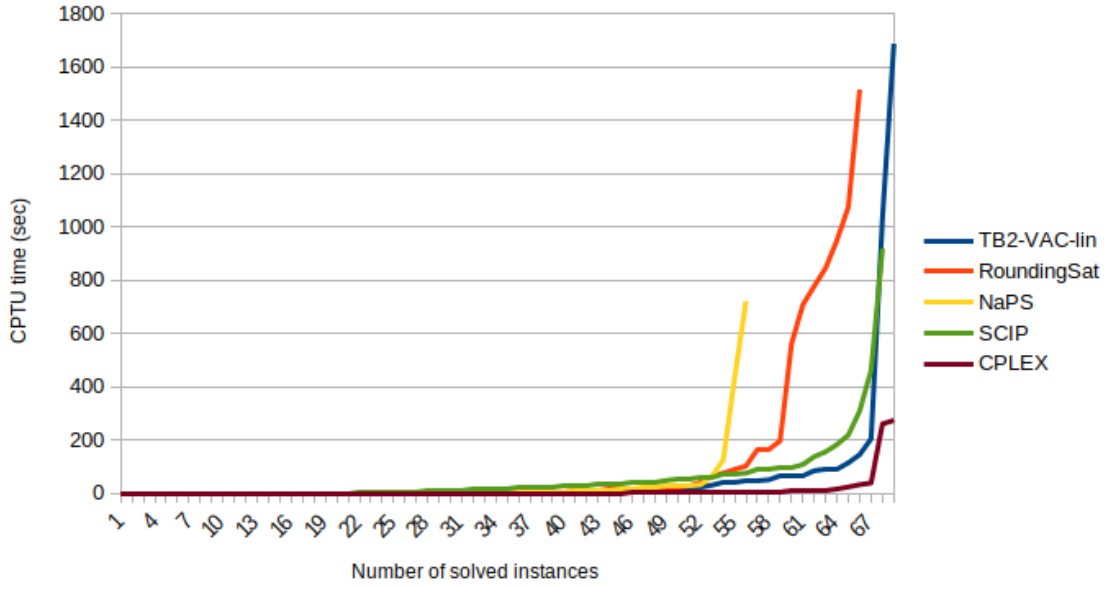


Figure 3.3: Cactus plot of CPU solving time for TOULBAR2 with VAC-lin in preprocessing and different solvers for the OPT-SMALLINT-LIN area family.

Solver	Score	Optimum	Best Bound
Virtual Best Solver	152.00	63	152
MISTRAL	93.00	34	99
TOULBAR2	86.00	51	87
MINIRBO	74.50	41	78
SAT4J-BOTH	58.50	39	60
SAT4J-RS	43.00	33	46
GLASGOW	31.50	21	34

Table 3.7: Result for the XCSP 2022 competition.

were harder for TOULBAR2 than the previous year, the number of instances solved to optimality dropped significantly. Still, TOULBAR2 was first on the mini COP track[10] (some solvers in table 3.4.2 are off competition). LINTOULBAR2 has shown good performance in some families of instances but was penalized because VAC-lin was not working correctly at the time of the competition.

3.4.3 Capacitated warehouse location problems

In the Warehouse Location Problem (WLP), a company considers opening warehouses at some candidate locations in order to supply its existing stores. The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply

[10]Full results: https://www.cril.univ-artois.fr/XCSP23/competitions/cop/mini-cop

Solver	Score	Optimum	Best Bound
Virtual Best Solver	148.00	43	105
MISTRAL	100.00	18	76
CHOCO	66.50	23	38
EXCHEQUER	42.50	24	18
TOULBAR2	39.50	14	25
LINTOULBAR2	32.00	12	19
SAT4J-BOTH	26.00	11	14
SAT4J-RESOLUTION	19.50	9	7
MINIRBO	13.00	13	0
SEAPEARL	7.00	3	0

Table 3.8: Result for the XCSP 2023 competition.

costs is minimized. Each store must be supplied by exactly one open warehouse. The capacitated WLP contains one multiple-choice knapsack constraint per warehouse such that the total demand of the assigned stores to the warehouse does not exceed its capacity. We add an extra linear constraint such that the sum of the capacity of the open warehouses is greater than the total demand of all the stores. This redundant constraint helps to reduce the search effort. We made experiments on 15 instances [Kratica *et al.* 2001][11] having from 100 (five *capmo* instances), 200 (*capmp*) to 300 (*capmq*) warehouses and stores (*i.e.,* up to 90,901 cost functions, including 301 linear constraints of arity 300), using the same CFN modeling approach as in [de Givry *et al.* 2005] (0/1 variables for warehouses and domain variables for stores representing which warehouse is its supplier) plus the additional PB constraints with EO partitions. A direct formulation with linear constraints and 0/1 variables is given to PB and ILP solvers.

Table 3.9 reports the number of solved instances within a 10-hour CPU time limit. We also report the mean solving time and number of backtracks (unsolved instances are discarded). We compared our approach with the same solvers as for the PB16 competition and also with the CP solver OR-TOOLS v9.0.9048 (using its flatzinc interface and free search option). CPLEX got the best results, being 19 times (resp. 115) faster than SCIP (resp. OR-TOOLS). TOULBAR2 was run with VAC or VAC-lin in prepossessing, in both cases 2/15 instances remain unsolved within the time limit. VAC-lin performs better than VAC on all instances except *capmq5, capmq3, capmq2*, where VAC is significantly faster (5900 sec vs 10700 sec on *capmq3*). Although it develops much more nodes ($\approx \times 1,000$) than its competitors, TOULBAR2 with or without VAC-lin was able to solve 5 instances (*capmo3, capmp2, capmp5, capmq2, capmq5*) faster than SCIP, the second-best competitor. It is also faster than OR-TOOLS on 9 instances. ROUNDINGSAT and NAPS could not solve any instances in less than 10 hours.

[11] https://forgemia.inra.fr/thomas.schiex/cost-function-library/-/tree/master/crafted/warehouses

Table 3.9: CPU time in seconds and number of backtracks (or search nodes for ILP solvers) in parentheses of 15 Capacitated Warehouse Location Problems.

	TB2-VAC	TB2-VAC-lin	OR-TOOLS	SCIP	CPLEX12.10
capmo1	6516 (794380)	1963 (543700)	1803 (82)	86 (243)	21 (544)
capmo2	-	-	30273 (100)	267 (1895)	89 (3265)
capmo3	539 (56939)	41 (12930)	444 (76)	133 (475)	9,68 (520)
capmo4	4889 (893422)	2458 (961056)	2337 (90)	79 (393)	21 (535)
capmo5	25724 (1157974)	3760 (495966)	5498 (81)	119 (540)	16 (780)
capmp1	-	-	4009 (120)	550 (835)	153 (5691)
capmp2	950 (42640)	737 (68270)	2431 (126)	3314 (15197)	57 (608)
capmp3	2963 (108036)	2002 (104123)	1878 (145)	111 (5)	7,2 (0)
capmp4	875 (72360)	654 (99297)	6618 (161)	617 (1044)	18 (639)
capmp5	1668 (81340)	421 (126054)	1553 (130)	1283 (1693)	192 (2911)
capmq1	2699 (233675)	1875 (44147)	6754 (118)	568 (19)	31 (0)
capmq2	1415 (14167)	2315 (237403)	8021 (197)	2209 (804)	76 (496)
capmq3	5925 (1316417)	10732 (568451)	5387 (177)	846 (24)	80 (642)
capmq4	1896 (133219)	1708 (9080)	17898 (229)	433 (5)	101 (1124)
capmq5	2019 (10957)	8692 (815237)	11219 (183)	7042 (5796)	44 (0)
Total solved	13	13	15	15	15
Average time	4,468 sec.	2,873 sec.	7,075 sec.	1,177 sec.	61,2 sec.

3.4.4 Knapsack problem with a conflict graph

We compare here TOULBAR2 and CPLEX on Knapsack with Conflict Graph (KPCG) [Bettinelli *et al.* 2017, Coniglio *et al.* 2021], a knapsack problem combined with binary constraints representing conflicts between pairs of variables. We use 6 different classes $C1$,$C3$,$C10$,$R1$,$R3$,$R10$. In three of them, the weight and the profit of each variable are correlated (class C) otherwise the profit is random between $[1, 100]$ (class R). The numbers $1, 3, 10$ correspond to a multiplying coefficient of the capacity, which has the effect of making the instances harder as the multiplier increases. In each class half of the instances have a capacity of 150, weights are uniformly distributed in $[20, 100]$, and the number of Boolean variables varies between $120, 250, 500$, and 1000. For the other half, the capacity is 1000, weights are uniformly distributed in $[250, 500]$, and the number of Boolean variables varies between $60, 120, 349$, and 501. Additionally, the density of the conflict graph varies from 0.1 to 0.9. In total, each class has 720 instances. We used a direct encoding for TOULBAR2 i.e we keep the Boolean variables, and if there is a conflict between x_i, x_j then we add a binary cost function between x_i and x_j with $c_{ij} = \top$. For CPLEX, we tried with both tuple and direct encodings (tuple encoding corresponds to the local polytope with integer variables) [Hurley *et al.* 2016]. Table 3.10 reports the number of instances solved by each solver. TOULBAR2 was more efficient than CPLEX with the tuple encoding and competitive with CPLEX using the direct encoding for four out of six classes. Moreover, TOULBAR2 finds the best solutions for the largest number of instances in every class. We also tested generating EO constraints from the conflict graph and associate them with the knapsack constraint (TOULBAR2-EO). This approach gives better results on the hardest benchmarks C10, R10 but not on

	TB2	TB2-EO	CPLEX$_t$	CPLEX$_d$	TB2	TB2-EO	CPLEX$_t$	CPLEX$_d$
C1	711	708	689	**720**	**720**	**720**	701	**720**
C3	592	593	487	**614**	703	**705**	513	640
C10	490	**492**	318	457	**592**	591	346	539
R1	**720**	**720**	705	**720**	**720**	**720**	705	**720**
R3	**720**	**720**	573	682	**720**	**720**	589	691
R10	554	**574**	365	519	630	**646**	384	572

Table 3.10: Number of solved instances (left) and number of times a solver found the best solution within the time limit (right) for six different classes of KPCG.

all the benchmarks. Even if TOULBAR2-EO derives better bounds, it has a different variable/value ordering than TOULBAR2 depending on the instances it reduces the number of nodes or not. This approach is also slowed down by the time taken to create the EO constraints (up to 175 seconds for problems with 1000 variables and high-density conflict graph) and a heavier propagation of the linear constraints.

3.4.5 Sequence of diverse solutions for CPD

A protein is a chain of simple molecules called amino acids. This sequence determines how the protein will *fold* into a specific 3D shape. The Computational Protein Design (CPD) [Allouche *et al.* 2014b] problem consists of identifying the sequence of amino acids that should fold into a given 3D shape. This problem can be modeled as a CFN[12] with unary and binary cost functions representing the energy of the protein but the criteria only approximate the reality, thus producing a sequence of diverse solutions increases the chance of finding the correct real sequence of amino acids. Each time a solution is found, a Hamming distance constraint is added to the model to enforce the next solution to be different from the previous ones. This is a greedy procedure, as it commits to each solution as soon as it is found, and therefore it cannot guarantee that the set of solutions it reports is optimal, even though each solution extends the set of solutions optimally.

The Hamming distance constraint can be directly encoded as a PB linear constraint. For each variable, a negative weight of -1 is associated with the value found in the last solution (other values having a zero weight) and the weighted sum must be greater than or equal to $-(|\boldsymbol{X}| - \zeta)$, where ζ corresponds to the required minimum Hamming distance.

This has been implemented in TOULBAR2 and compared to previous automata-based encoding approaches (ternary, hidden, and dual encodings from [Ruffini *et al.* 2021])

[12]Other paradigms such as ILP or Max-SAT have been tested but the experimental results using their corresponding state-of-the-art solvers were inferior to the CFN approach using TOULBAR2 [Allouche *et al.* 2014b, Allouche *et al.* 2021]. E.g., for CPD instance *1BK2.matrix.24p.17aa.usingEref_self_digit2* ($n = 24$, $d = 182$, $e = 300$), CPLEX 20.1 solves it in 42.84 seconds, TOULBAR2 in 0.37 seconds. RoundingSat [Devriendt *et al.* 2021] timed out after 10 hours.

on 30 instances [Traoré *et al.* 2013].[13] Selected instances have from 23 to 97 *residues* (variables) with maximum domain size from 48 to 194 *rotamers* (values). The number of unary and binary cost functions varies from 276 to 4,753. For each instance, we set the time limit of 1 hour and the solver halts after finding 10 diverse solutions. We set the parameter ζ for the minimum Hamming distance parameter for each new solution to $\zeta = 10$. We enforce VAC on unary and binary cost functions in preprocessing (options -A -d: -a=10 -div=10 -divm=(0 for dual, 1 for hidden, 2 for ternary, and 3 for the linear encoding)). Figure 3.4 reports the solving time of each encoding. The hidden and ternary encodings failed to give 10 diverse solutions for respectively one and 2 instances. The linear encoding is faster for 28 instances and it solves 24 of them in less than 30 seconds while dual, hidden, and ternary encodings solve respectively 12, 12, and 3 instances in less than 30 seconds.
Note that since we are computing a greedy sequence of solutions, the different encodings do not return the exact same sequence. Therefore, the set of diversity guarantee constraints is different, and each approach solves a slightly different sequence of problems. To be certain that the greedy approach does not deeply influence those results, we also tested the different encodings on instances with 10 initial diversity guarantee constraints. The task is to find one optimal solution. Figure 3.5 reports the solving time of each encoding. The obtained results are close to the previous one, with the linear encoding dominating the other approaches. We compared for each instance the number of nodes (Fig. 3.6) and time (Fig. 3.7) of the linear encoding and the dual encoding (previous TOULBAR2 default encoding). In all the instances the linear encoding needs fewer nodes than the dual encoding and except for two instances, the linear encoding is also faster. Automata-based encodings have the flaw of introducing extra variables that can disturb the variable ordering heuristic and local consistency algorithm (by default, EDAC during search, except partial $F\emptyset IC$ for PB constraints). While the linear encoding directly encodes the Hamming distance, it is heavier to propagate as we can see by comparing the number of nodes per second (403 for linear encoding and 1480 for dual encoding).

3.5 Conclusion and future work

It is now possible to model pseudo-Boolean linear constraints in deterministic and probabilistic graphical models. This provides greater modeling flexibility and allows a WCSP solver like TOULBAR2 to solve more problems, such as computational protein design problems with diversity guarantee or knapsack problems with conflict graphs. One of the weaknesses of our approach is that the algorithm fundamentally produces a suboptimal solution to the linear program. It propagates the pseudo-Boolean linear constraints one by one and is sensitive to the constraint ordering. To address this issue we extended VAC algorithm to handle linear constraints and

[13] `http://genoweb.toulouse.inra.fr/~tschiex/CPD-AIJ/Last35-instances`. We removed 5 instances (*1ENH.matrix.36p.17aa, 1STN.matrix.120p.18aa, HHR.matrix.115p.19aa, 1PGB.matrix.31p.17aa, 2CI2.matrix.51p.18aa*) on which TOULBAR2 timed out after 9,000 seconds even without diversity constraints [Allouche *et al.* 2021].

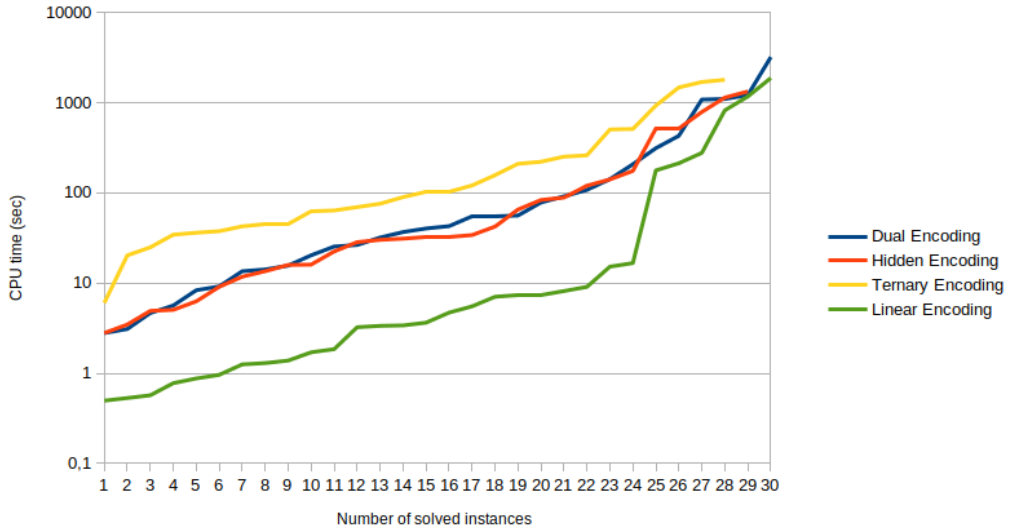


Figure 3.4: Cactus plot of CPU solving time (log scale) to find 10 diverse solutions for different encodings of Hamming distance constraints on CPD.

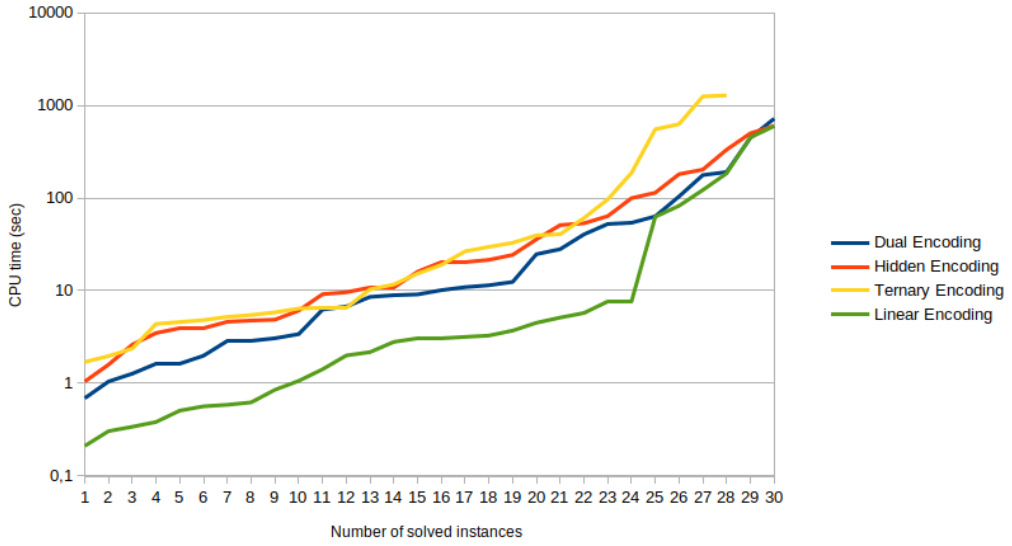


Figure 3.5: Cactus plot of CPU solving time (log scale) on 30 CPD instances with 10 diversity constraints using different encodings.

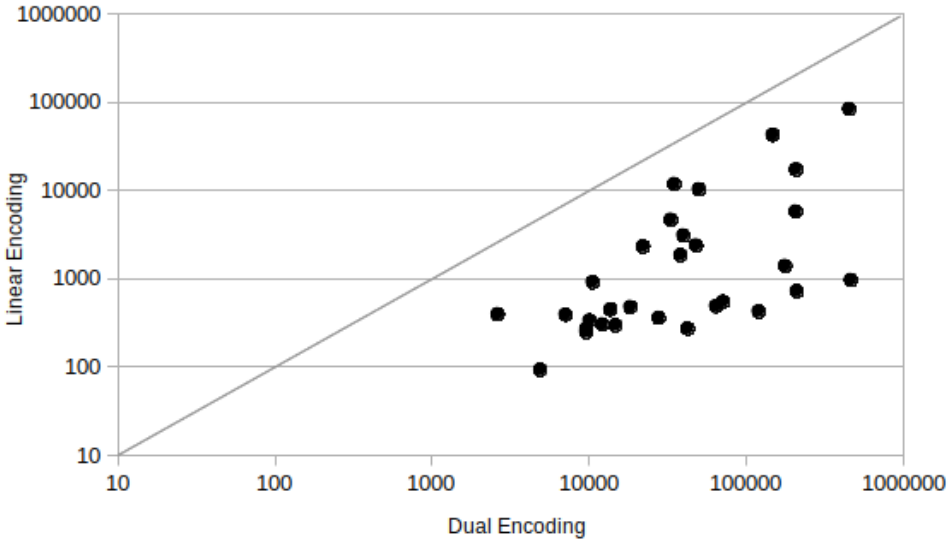


Figure 3.6: Comparison of the number of nodes on 30 CPD instances with 10 diversity guarantee constraints.

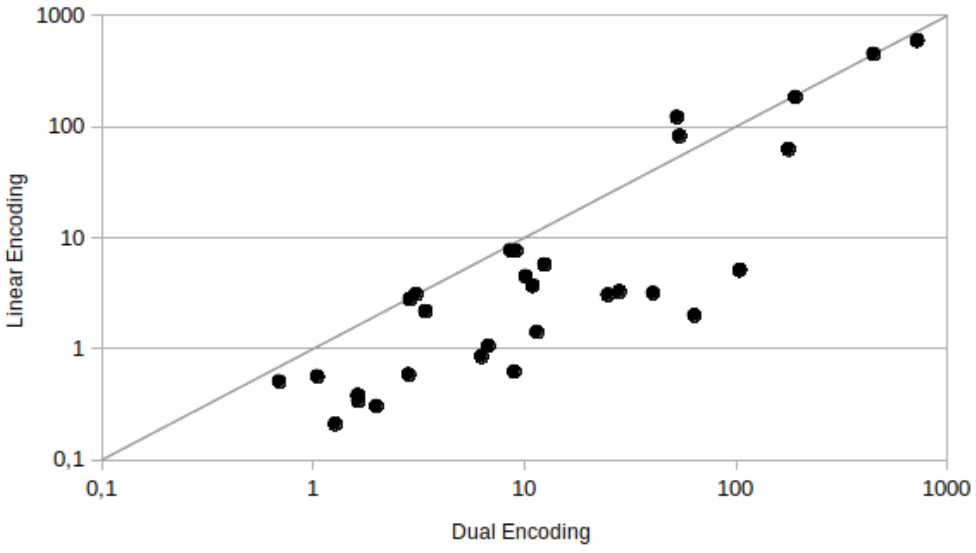


Figure 3.7: Comparison of CPU solving time in seconds on 30 CPD instances with 10 diversity guarantee constraints (log scale).

added the possibility of associating conflicting assignments with linear constraints. It showed its efficiency on several benchmarks.

Chapter 4

Virtual Pairwise Consistency

Contents

We introduced in chapter 2 several local consistency algorithms. In particular, we described Virtual Arc Consistency (VAC), which is achieved when the arc consistency closure of Bool(P) is non-empty. In this chapter, we define Virtual Pairwise Consistency (VPWC), which can be obtained by enforcing Pairwise Consistency on Bool(P). In constraint programming, Pairwise Consistency (PWC) is a consistency defined for non-binary CSP. In contrast to AC-based consistency, it has the possibility to exploit interactions between pairs of constraints. Therefore, it provides a stronger pruning but is more expensive to enforce. It was compared to generalized AC for solving non-binary CSPs given in extension [Samaras & Stergiou 2005a, Schneider & Choueiry 2018a, Wang & Yap 2019, Wang & Yap 2021]. It shows good performance in some benchmarks but is not preferred by default. Recent algorithms enforcing PWC on a CSP rely on a binary encoding, we also explore a similar idea in the CFN framework.

4.1 Pairwise Consistency

We recall that a Constraint Network (CN) is a triplet $< X, \boldsymbol{D}, \boldsymbol{C} >$, where X are the variables, $\boldsymbol{D}$ the domains, and $\boldsymbol{C}$ the constraints. A constraint $c_{\boldsymbol{S}} \in \boldsymbol{C}$ is defined by a pair $< \boldsymbol{S}, r_{\boldsymbol{S}} >$ where $\boldsymbol{S} \subseteq X$ is the *scope* $\boldsymbol{S} \subseteq X$ and $r_{\boldsymbol{S}}$ is a function $r_{\boldsymbol{S}} : \boldsymbol{D}_{\boldsymbol{S}} \to \{0; \top\}$. For $u, v \in \{0, \top\}$, we will denote the logical conjunction by $u \wedge v = u + v$ and the disjunction by $u \vee v = \min\{u, v\}$. We assume a CN contains all unary constraints, i.e., $\{i\} \in \boldsymbol{C}\ \forall i \in X$.
For any $B \subseteq A \subseteq X$, we define the *projection* of a constraint r_A: $\boldsymbol{D}_A \to \{0, \top\}$ onto variables B to be the constraint $r_A|_B$: $\boldsymbol{D}_B \to \{0, \top\}$ given by

$$r_A|_B(x) = \bigvee_{x' \in \boldsymbol{D}_A :\, x'|_B = x} r_A(x') \qquad \forall x \in \boldsymbol{D}_B. \tag{4.1}$$

We say that a pair of constraints $\{r_A, r_B\}$ is *Pairwise Consistent* (PWC) if they admit the same set of assignments to their shared variables, i.e.,

$$r_A|_{A\cap B} = r_B|_{A\cap B} \tag{4.2}$$

where '=' denotes here equality of functions. A CN is PWC if all possible pairs of its constraints are PWC. If we restrict PWC to pairs of constraints where one constraint is unary, we come back to (generalized) arc consistency (GAC) (see definition 2.1). PWC or GAC can be enforced on a CN P by iteratively forbidding assignments that violate (4.2). The minimal set of changes required to enforce PWC (or GAC) is unique, and the resulting CN is called the PWC (or GAC) *closure* of P.

We say that a local consistency ψ' is *not weaker* than a local consistency ψ if for every CN instance for which the ψ-consistency closure is empty, the ψ'-consistency closure is also empty. We say that ψ and ψ' are *equally strong* if ψ' is not weaker than ψ and *vice versa*. We say that ψ' is *strictly* stronger than ψ if ψ' is not weaker than ψ but they are not equally strong.

- For binary CNs, AC is equally strong as PWC.
- For non-binary CNs, PWC is strictly stronger than AC.
- For CSPs of arity more than 2 with all unary constraints, AC is equally strong as the local consistency obtained by enforcing PWC on constraint pairs $\{r_A, r_i\}$ (i.e., enforcing the equality $r_i = r_A|_{\{i\}}$) for all $i \in A \in \boldsymbol{S}$.

The PWC relation of constraints is clearly reflexive and symmetric. It is in general not transitive, but it satisfies the following weaker condition:

Theorem 4.1. *[Janssen et al. 1989] Let $C_1, \ldots, C_n \in \boldsymbol{S}$ be such that for every $i = 1, \ldots, n$, we have $C_1 \cap C_n \subseteq C_i$. Let, for every $i = 1, \ldots, n-1$, constraint r_{C_i} be PWC with $r_{C_{i+1}}$. Then r_{C_1} is PWC with r_{C_n}.*

Thus, enforcing PWC for some constraint pairs implies that the PWC condition holds also for some other pairs, which can simplify algorithms [Janssen *et al.* 1989, Schneider & Choueiry 2018b].

4.2 Dual Encoding of a Cost Function Network

Depending on the studied CN, the choice of the encoding can significantly impact the efficiency of the solver. A particular encoding may offer advantages due to its ability to provide a more concise representation and enable more efficient propagation and search algorithms. A well-known encoding is the *Dual encoding*, it allows encoding any CN into a binary CN [Samaras & Stergiou 2005b]. This encoding highlights the relation between pairs of constraints and allows a more powerful filtering by using arc consistency-based algorithms. However, it might introduce large domain variables. The dual encoding of a CN $P = (X, \boldsymbol{D}, \boldsymbol{C})$ is a binary CN Dual$(P) = (C, \bar{\boldsymbol{D}}, \bar{\boldsymbol{C}})$ where:

- The variables of the dual problem are the scope $\boldsymbol{C}$ of P.
- The domain of variable $A \in \boldsymbol{C}$ of the dual problem is $\bar{\boldsymbol{D}}_A = \{\tau \in \boldsymbol{D}_A \quad s.t \quad r_A(\tau) = 0\}$.
- The scopes are $\bar{\boldsymbol{C}} = \{\,\{A\} \mid A \in \boldsymbol{C}\,\} \cup \{\,\{A, B\} \mid A, B \in \boldsymbol{C},\ A \cap B \neq \emptyset\,\}$.
- The dual binary constraints with scope $\{A, B\} \in \bar{\boldsymbol{C}}$ are the *channeling constraint* $\bar{r}_{AB}$: $\boldsymbol{D}_A \times \boldsymbol{D}_B \to \{0, \top\}$ with values:

$$\bar{r}_{AB}(y, y') = \begin{cases} 0 & \text{if } y_i = y'_i\ \forall i \in A \cap B \\ \top & \text{otherwise} \end{cases} \qquad \forall y = (y_i)_{i \in A} \in \boldsymbol{D}_A,\ y' = (y'_i)_{i \in B} \in \boldsymbol{D}_B.$$

Note that we suppose that the unary constraints are in the set of constraints, this is not usually the case in the literature. In particular, the *hidden* and *double* encoding [Samaras & Stergiou 2005b] have been defined to keep the primal variables in the resulting binary encoding. With our assumption, the dual and double encoding are equivalent. The dual encoding can be used to enforce PWC on the primal problem.

Theorem 4.2. *[Janssen et al. 1989] Let P be a CN. P is PWC if and only if it's dual is AC.*

By taking advantage of theorems 4.2 and 4.1, PWC can be achieved by applying well-optimized AC algorithms [Schneider & Choueiry 2018b].

Similarly, the notion of dual encoding can be extended to CFN, it will help to produce better bounds with soft arc consistency based algorithms. The *dual encoding* of a CFN $P = (X, \boldsymbol{D}, \boldsymbol{S}, f)$ is a binary CFN $\text{Dual}(P) = (\boldsymbol{S} \setminus \{\emptyset\}, \bar{\boldsymbol{D}}, \bar{\boldsymbol{S}}, \bar{f})$ where:

- The variables of the dual problem are the scopes $\boldsymbol{S} \setminus \{\emptyset\}$ of P.
- The domain of variable $A \in \boldsymbol{S} \setminus \{\emptyset\}$ of the dual problem is $\bar{\boldsymbol{D}}_A = \boldsymbol{D}_A$.
- The scopes are $\bar{\boldsymbol{S}} = \{\emptyset\} \cup \{\,\{A\} \mid A \in \boldsymbol{S}\,\} \cup \{\,\{A, B\} \mid A, B \in \boldsymbol{S},\ A \cap B \neq \emptyset\,\}$.
- The dual nullary cost function is unchanged: $\bar{f}_\emptyset = f_\emptyset$.
- The dual unary cost function with scope $\{A\} \in \bar{\boldsymbol{S}}$ is the function $\bar{f}_A = f_A$.
- The dual binary cost function with scope $\{A, B\} \in \bar{\boldsymbol{S}}$ is the *channeling constraint* $\bar{f}_{AB}$: $\boldsymbol{D}_A \times \boldsymbol{D}_B \to \{0, \top\}$ with values:

$$\bar{f}_{AB}(y, y') = \begin{cases} 0 & \text{if } y_i = y'_i\ \forall i \in A \cap B \\ \top & \text{otherwise} \end{cases} \qquad \forall y = (y_i)_{i \in A} \in \boldsymbol{D}_A,\ y' = (y'_i)_{i \in B} \in \boldsymbol{D}_B.$$

Once again, as we make the assumption that all the unary cost functions are in the set of scopes, the dual and double encoding of a CFN are equivalent.

Example 4.1. *Let P be the CFN described in Example 2.9, represented by the hypergraph in Fig. 4.1(a). Then,* $\text{Dual}(P)$ *has 9 dual variables,* $y_1, y_{123}, y_{14}, \ldots, y_5$.

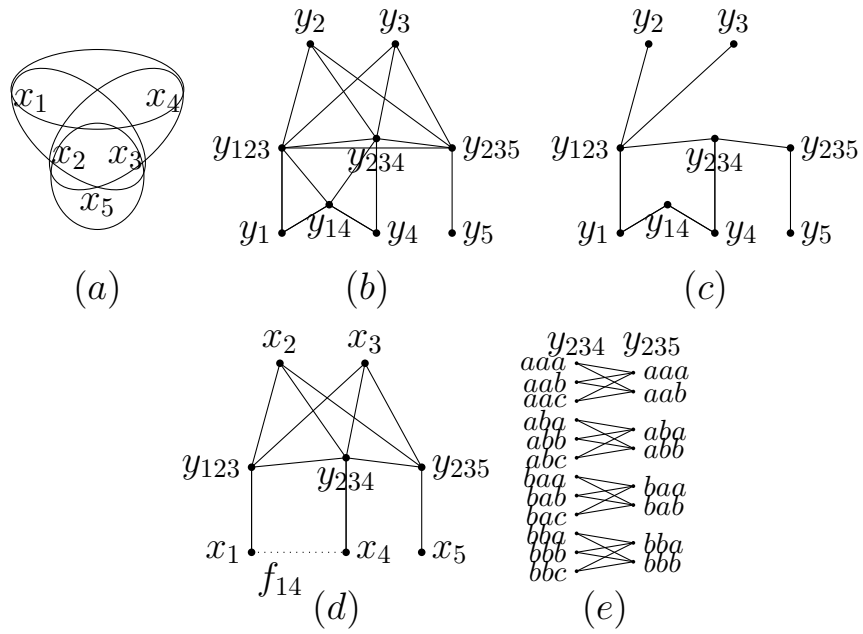


Figure 4.1: (a) Hypergraph of a CFN, (b) its dual graph, (c) a minimal dual graph, (d) the partial dual graph used in the experiments, (e) a binary channeling constraint created by the dual encoding (an edge depicts a 0-cost assignment).

The domain of the dual variables corresponds to the tuple of the corresponding primal constraint, hence the domain size of $y_{1,2,3}$ *and* $y_{2,3,4}$ *is 12, 8 for* $y_{2,3,5}$ *and 9 for* $y_{1,4}$. Dual(P) *has 16 binary channeling constraints, as shown by the constraint graph in Fig. 4.1(b). Using Theorem 4.1, a minimal dual graph can be produced with only 9 binary constraints (Fig. 4.1(c)).*

■

4.3 Virtual Pairwise Consistency

Following the idea of VAC, we introduce *Virtual Pairwise Consistency* (VPWC), a soft local consistency stronger than VAC.

Definition 4.1. *A CFN P is VPWC if the PWC closure of* Bool(P) *is non-empty.*

Combining Definition 4.1 and previous results [Janssen *et al.* 1989], we get that enforcing VPWC is possible using existing algorithms.

Theorem 4.3. *Let P be a CFN. P is VPWC if and only if* Dual(P) *is VAC.*

Proof. By theorem 4.2, we know that a CN has a non-empty PWC closure if and only if its dual has a non-empty AC closure. Clearly, for any CFN P we have Dual(Bool(P)) = Bool(Dual(P)). Therefore, P is VPWC iff Dual(Bool(P)) = Bool(Dual(P)) has a non-empty AC closure, which means Dual(P) is VAC. □

(a)

f_{123}	x_1	x_2	x_3	Cost
	a	a	a	0
	a	a	b	1
	b	a	a	1
	b	a	b	1
	c	a	a	1
	c	a	b	1

f_{234}	x_2	x_3	x_4	Cost
	a	a	a	1
	a	a	b	1
	a	a	c	1
f_1			x_1	
			b	2
			c	2

f_{14}	x_1	x_4	Cost
	a	a	2
	a	b	2
	a	c	2
f_2		x_2	
		a	0
		b	2

(b)

y_{123}	Cost
aaa	0
aab	1
baa	1
bab	1
caa	1
cab	1

y_{234}	Cost
aaa	1
aab	1
aac	1
y_1	
b	2
c	2

y_{14}	Cost
aa	2
ab	2
ac	2
y_2	
a	0
b	2

Table 4.1: (a) Original CFN. (b) dual unary cost functions (missing tuples have 0 cost).

Example 4.2. *Following Ex.4.1, we give the costs for each cost function in Table 4.1(a). VAC on this problem derives a lower bound of 2, since $x_1 = a$ is not consistent with r_{14}. VAC on the dual (Table 4.1(b)) derives a lower bound of 3, because (a) all values in y_{14} compatible with $y_1 = a$ (*i.e., *aa, ab, ac) have cost 2, and (b) all values compatible with $y_{123} = aaa$ in y_{234} (*i.e., *aaa, aab, aac) have a cost of 1, therefore they do not support $y_2 = a$, making it inconsistent in y_{123}. This leads to a lower bound of 3.*

■

The dual can help derive better lower bounds but introduces a possibly large number of variables with large domains, which may slow down the search. We propose to first dualize the problem and get a first strong lower bound, then return to the primal. The following shows that this is always possible without introducing higher order cost functions[1].

Theorem 4.4. *Let P be a CFN and let Q be a CFN equivalent to* Dual(P)*. Then there exists a CFN Q' equivalent to Q such that all binary constraints of Q' are hard and Q' has the same lower bound as Q.*

Proof. The main observation is that every dual binary cost function (the channeling constraint) is *piecewise-functional*. It means that c_{ij} has a block structure (see Fig. 4.1(e)): there exists a partition $H_i = \{\beta_1, ..., \beta_m\}$ of the domain $\boldsymbol{D}_i$ and a partition $B_j = \{\beta'_1, ..., \beta'_m\}$ of $\boldsymbol{D}_j$ such that for each $v \in \beta_k$ and $v' \in \beta'_l$ we have $c_{ij}(v, v') < \top$ whenever $k = l$ and $c_{ij}(v, v') = \top$ whenever $k \neq l$. In the context of the dual encoding, the channeling constraints start with a cost 0 whenever $k = l$. This implies that every value within the same block is initially supported by the

[1]This is unsurprising because the strongest bound that can be derived using EPTs is obtained using a linear program which includes pairwise consistency constraints [Werner 2010].

exact same set of values. A fact we can use to show that every EPT moving cost into c_{ij} can be matched with another EPT moving cost out of it without affecting the lower bound.
Indeed, if an EPT $project(c_i, c_{ij}, (i, v), \alpha)$ increases the cost of a value $v \in \beta_l \in H_i$, it implies to keep the non-negativity that for every value $v' \in \boldsymbol{D}_j$ supporting v we have $c_{ij}(v, v') \geq \alpha$. All the channeling constraints start with a cost 0 or $\top$. It follows that every value $v' \in \beta'_l$ moved a cost $\alpha' \geq \alpha$, corresponding to EPTs $extend(c_j, (j, v'), c_{ij}, \alpha')$ (α' might differ between values in β'_l). We also know that all the values in β_l are supported by the same values, we deduce that if it is possible to increase $c_i(v)$ by α then we can also increase the cost of every value in β_l by α. If all the values $v' \in \beta'_l$ extend exactly a cost $\alpha' = \alpha$ then no non-zero finite cost will remain in the channeling constraint after moving a cost α to the values $v \in \beta_l$. □

We can now summarize the base version of our approach. Given a CFN P, we apply EPTs to its dual encoding Dual(P) (using a VAC algorithm) to obtain a CFN Q with an increased lower bound. Theorem 4.4 lets us obtain from Q another CFN Q' in which all channeling cost functions are constraints. We can thus undo the dual encoding, i.e., obtain a CFN P', equivalent to P, such that $Q' = \text{Dual}(P')$. If Q was VAC then, by Theorem 4.3, P' is VPWC.

4.4 Experimental Results on UAI 2022 Competition

We won a recent competition on probabilistic graphical models. [2] We present results on a set of 120 tuning instances, where 63 have maximum arity of 3.

We evaluate three solvers: daoopt (version from UAI 2012 competition with 1-hour settings as given in [Otten *et al.* 2012]), cplex (version 20.1.0.0, forcing completeness with zero absolute and relative gaps, translating CFN to 0-1 LP by the tuple encoding [Hurley *et al.* 2016]), and toulbar2 (version 1.2.0) using two state-of-the-art methods, Variable Neighborhood Search (VNS) [Ouali *et al.* 2020] winner of UAI 2014 competition, [3] and Hybrid Best-First Search with VAC in preprocessing (VACpre-HBFS), including VAC integrality heuristics [Trösser *et al.* 2020]. [4] We implemented VPWC in the latest version of toulbar2. It is either enforced in preprocessing (and then converted back to the primal, see Theorem 4.4) (VPWCpre-HBFS) or maintained during search (HBFS-VPWC). EDAC is always enforced [Larrosa & Heras 2005, Sánchez *et al.* 2008], providing a default value ordering heuristic when no solution is found for solution-based heuristics [Demirovic *et al.* 2018]. The branching heuristic is *dom/wdeg* [Boussemart *et al.* 2004a] combined with *last conflict* [Lecoutre *et al.* 2009].

We use a slightly different binary encoding. We keep the original variables and the original binary cost functions unchanged, and only dualize the original

[2] https://uaicompetition.github.io/uci-2022, see MPE and MMAP entries.
[3] http://auai.org/uai2014/competition.shtml, http://miat.inrae.fr/toulbar2
[4] Options *-A -P=1000 -T=1000 -vacint -vacthr -rasps -raspsini* in toulbar2-vacint.

non-binary cost functions. It avoids creating extra dual variables and bijective channeling constraints with a partition of blocks but a single value per block. We add channeling constraints between dual and primal variables, this allows us to only add channeling constraints between those pairs of dual variables that are not redundant by Theorem 4.1. Observe the primal variables are exactly the same as the dual variables on unary cost functions. By adding all the channeling constraints between primal and dual variables (as in HVE) ensures that the minimal dual graph is connected for each separator even if we remove dual variables for the binary cost functions. Theorems 4.3 and 4.4 remain valid. Similarly, this encoding allows us to not add channeling constraints between dual variables with intersecting scopes equal to 1. Indeed, it has been shown that GAC is sufficient to enforce PWC if the size of the intersecting scope is 1 [Schneider & Choueiry 2018b]. The resulting non-minimal graph for Example 4.1 is shown in Fig. 4.1(d).

Furthermore, we apply this encoding only partially, indeed for high-arity constraints, a full dual encoding might mean a prohibitive amount of memory to store the dual domains.

Hence, only non-binary cost functions of arity less than 10 and fewer than 2^{15} non-forbidden tuples are dualized. Those remaining are lazily propagated by VAC/EDAC when they have less than three unassigned variables in their scope. The memory used by each channeling constraint between a pair of dual variables is restricted to at most 1MB (arbitrarily chosen). Larger channeling constraints are ignored. Additional preprocessing is performed beforehand for all the HBFS methods in order to find better bounds. An initial upper bound is found by local search [Neveu *et al.* 2004, Beuvin *et al.* 2021] and VAC-based heuristics [Trösser *et al.* 2020]. To reduce the problem size and improve lower bounds, we apply bounded variable elimination with a min-fill ordering [Dechter 1999, Larrosa 2000, Favier *et al.* 2011][5] and add ternary zero-cost functions on the *most-preferred triangles* (total memory space of extra ternary functions limited to 1MB).[6] It results in at most 6-ary (resp. zero-cost ternary) cost functions for 84 (resp. 81) instances, making our encoding applicable to 85 instances rather than 63. Finding a (quasi-)minimal dual graph (see Theorem 4.1) yielded 700.3 channeling constraints on average, a 4.5% savings compared to the complete dual graph.

The experiments were run on a single core of Intel Xeon E5-2683 2.1GHz processors with 1-hour CPU-time and 8GB memory limit. toulbar2 was able to solve optimally 86 instances using VACpre-HBFS or VNS. daoopt solved 92 instances and cplex 95 instances. Using our partial dual encoding with VPWC applied in preprocessing, VPWCpre-HBFS solved 95 instances, and when applied during search, HBFS-VPWC solved 99 instances, 15% above VACpre-HBFS, being the best exact

[5] It is done only if the median degree in the original problem is less than 8, eliminating variables with a current degree less than or equal to the original median degree.

[6] With additional options *-i -pils -p=-8 -O=-3 -t=1*. A triangle is defined by three variables involved in three binary cost functions. The score of a triangle is given by the average cost in the three functions. Triangles with the largest score are selected first. This approach allows to simulate soft path inverse consistency [Nguyen *et al.* 2017].

instance	(n,d,e,a)	(n',e',a')	(n'',d'',e'')	VACpre-HBFS time *(gap)*	VPWCpre-HBFS time *(gap)*	HBFS-VPWC time *(gap)*
Grids21	(1600,2,4800,2)	(799,2810,4)	(1628,16,4675)	- *(42.4%)*	- *(3%)*	1216.83
Promedas12	(1766,2,1766,3)	(826,1884,4)	(1373,16,2223)	5.17	6.34	7.7
ProteinFold11	(400,2,1160,2)	(190,604,4)	(381,16,1005)	- *(16.1%)*	8.48	12.43
wcsp12	(311,4,5732,3)	(305,5887,3)	(12708,64,70959)	- *(49.9%)*	- *(19.3%)*	- *(54.8%)*

Table 4.2: UAI 2022 detailed results on a selection of four instances for HBFS methods. '-' means the instance is unsolved in 1h. (in parentheses, remaining optimality gap).

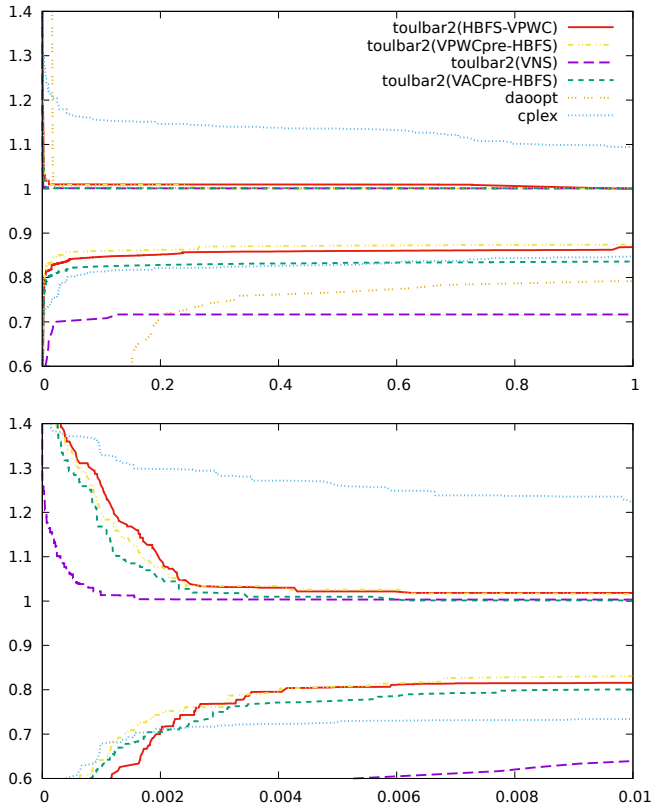


Figure 4.2: Normalized lower and upper bounds (y-axis) as time passes (x-axis in hour, zoomed on the bottom fig.) for cplex, daoopt, and toulbar2 on UAI 2022 tuning benchmark.

method for this benchmark.

Table 4.2 shows for a selection of UAI 2022 instances their size in terms of number of variables n, maximum domain size d, number of cost functions e, maximum arity a of the original problem, after preprocessing it with bounded variable elimination and adding triangles (n', d', e', a' with $d' = d$), and after applying our partial dual encoding (n'', d'', e'', a'' with $a'' = 2$). It gives also the CPU-time in seconds to solve an instance using HBFS methods or the remaining optimality gap if un-

solved after 1 hour. On Grids21, only HBFS-VPWC solves the instance. Notice the large improvement on the optimality gap by VPWCpre compared to VACpre. On Promedas12 and ProteinFolding11, VPWCpre-HBFS develops 13 and 32250 nodes, respectively, and takes about the same time as HBFS-VPWC which develops 4 and 992 nodes, respectively. For wcsp12, the size of the encoding slows down the search too much, suggesting harder limits for our partial dual encoding.

We report in Fig. 4.2 the average normalized lower and upper bounds as time passes (computed as in [Trösser *et al.* 2020]). Here VNS provides the best upper bounds in limited time whereas HBFS-VPWC is slightly slower than VPWCpre-HBFS, VACpre-HBFS, and VNS, but still faster than daoopt and cplex. Both VPWCpre-HBFS and HBFS-VPWC offer the best average lower bounds in less than 1 hour. HBFS-VPWC found 117 best solutions, VPWCpre-HBFS 112, VACpre-HBFS 106, VNS 105, daoopt 99, and cplex 95. VNS found 2 single-best solutions (wcsp11, wcsp12). For the competition, we combined VNS and HBFS-VPWC sequentially.[7]

4.5 Conclusion

We have defined virtual pairwise consistency and shown how it can efficiently be used in preprocessing or during search by applying the existing VAC algorithm to a dual encoding of the problem. In the future, we will explore the benefit of other binary encodings [Wang & Yap 2021] and adapt the VAC algorithm to the specific constraints of the encoding as it is done in CSPs [Schneider & Choueiry 2018a, Wang & Yap 2019]. Finding good heuristics to exploit a partial dual encoding in conjunction with bounded variable elimination and zero-cost function addition is also an interesting question.

[7] See toulbar2-ipr results on the UAI 2022 Tuning Leader Board. Multiple runs of VNS with increasing floating-point precision were done with a total amount of time of $\frac{1}{2}$h. The remaining time is allocated to HBFS-VPWC. Each search procedure gives its best solution found to the next search procedure. On UAI 2022 tuning instances, this approach found 119 best solutions, ranking first among our 7 tested methods.

Chapter 5

Conflict-free learning

Contents

5.1 Introduction

The idea behind learning mechanisms is to use the information provided by a solver during the search to learn a new constraint that will help for the end of the search, it has shown its efficiency in different paradigms CDCL for SAT [Marques-Silva & Sakallah 1999], NoGood recording for CSP [Dechter 1990, Katsirelos & Bacchus 2005], pseudo-Boolean resolution for PB solver [Chai & Kuehlmann 2003, Dixon & Ginsberg 2002, Elffers & Nordström 2018, Le Berre & Parrain 2010, Sheini & Sakallah 2006, Devriendt *et al.* 2021], MaxCDCL for MaxSAT [Li *et al.* 2021], MIP conflict analysis for MIP[Achterberg 2007]. Those learning methods are triggered when the solver encounters an infeasible problem, they are *conflict based*. Constraints learned that way are specifically designed to prune more inconsistent values. A more recent approach is conflict free learning [Witzig 2022], it relies on Farkas lemma [Farkas 1902] to produce *Farkas constraints* from a dual solution.

Definition 5.1 (Farkas constraint)**.** *Given an LP problem* $\min\{c^{\mathbf{T}}x | Ax = b, x \in \mathbb{Z}\}$ *and a dual solution* $\mathbf{y}$ *then the Farkas constraint is defined as:*

$$\mathbf{y}^{\mathbf{T}}Ax = \mathbf{y}^{\mathbf{T}}b \tag{5.1}$$

We denote in the following the Faraks constraint issued from dual solution $\mathbf{y}$ and problem P by $Farkas_P(\mathbf{y})$. Farkas constraints were typically used as a proof of infeasibility when the LP was decided infeasible. Witzig shows that learning a Farkas constraint on feasible nodes can be useful if the constraint is modified afterward, according to the information obtained by going deeper in the search tree [Witzig & Berthold 2020, Witzig 2022]. However, those constraints are derived from an aggregation of LP primal constraints, furthermore, they do not exploit any knowledge about the integrality of the variables. Therefore, those constraints will not help an LP solver to directly derive better bounds in the future, but can help to remove inconsistent values via domain propagation.
Even if those conflict-based/free learning are effective in their framework, they do not meet our needs in CFN. Indeed, WCSP solvers have the specificity to reparametrize the problem during the search. If the lower bound exceeds the upper bound it is not possible to directly point out a set of cost functions and trigger a procedure similar to conflict analysis. Furthermore, pruning strategies are limited in CFN because hard inconsistencies are not at the core of cost functions (except on some specific benchmarks). In WCSP solvers, the best way to cut the search space is to produce good bounds. Whereas previously defined learning mechanisms were exclusively designed to prune more values, it seems more interesting in those conditions to learn constraints helping derive better lower bounds.
Our approach is based on Farkas constraints. By itself, we know that a Farkas constraint will not help an LP solver to derive better bounds, but, we specify how to combine them to obtain constraints that are sound, logically redundant, but non-redundant with respect to the linear relaxation. In particular, similarly to Witzig's approach, we show that a Farkas constraint computed at an internal node can become useful if it is merged with Farkas constraints derived deeper in the search tree. Those new constraints will produce better bounds and may prune values.

5.2 Learning bounds using guarantee constraints

Let P_{MILP} be an ILP with Boolean decision variables expressed in standard form. P_{MILP} is defined by equation (5.2), where $x \in \{0,1\}^n$ is the decision variables vector, $s \in \mathbb{R}^m_+$ is the slack variables vector. The constraints are defined by $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. To model the Boolean decision variables, we suppose that $\forall 1 \leq i \leq n$ the bound constraint $x_i \leq 1$ is expressed in A. A slack variable x_{i0} is introduced to obtain the equality constraint $x_i + x_{i0} = 1$ or equivalently $-x_i - x_{i0} = -1$. The objective function is defined by $c_x \in \mathbb{R}^n$, $c_s \in \mathbb{R}^m$. In the literature, c_s is often omitted because it is equal to 0 in the conventional ILP standard form, however, they are noteworthy in this chapter. We compactly represent P_{LP} the LP relaxation of P_{MILP} and its associated dual problem P_D by equations (5.3), (5.4). Where $z = x \bigcup s$ with $z \in \mathbb{R}^{n+m}_+$. We define the objective function $c = c_x \bigcup c_s$. We also extend $A \in \mathbb{R}^{m \times (n+m)}$ to also represent the slack variables in the constraint. Observe that the local polytope of a WCSP (see equations Local Polytope in chapter

2) fits in the description of P_{MILP}.

$$\min\{c_x^T x + c_s^T s | Ax - s = b, x \in \{0,1\}, s \in \mathbb{R}^+\} \tag{5.2}$$

$$\min\{c^T z | Az = b, z \geq 0\} \tag{5.3}$$

$$\max\{b^T y | A^T y \leq c, y \in \mathbb{R}\} \tag{5.4}$$

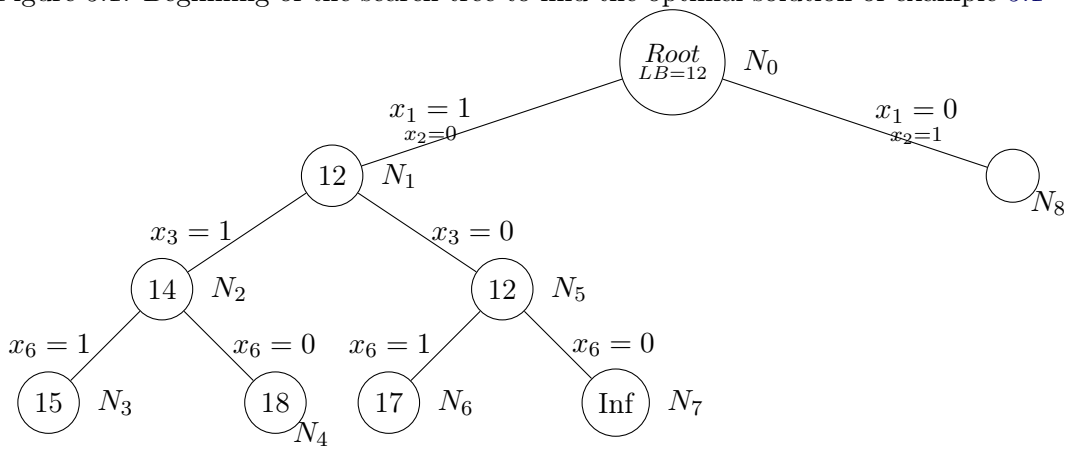
Suppose we solve P_{MILP} with a branch and bound procedure where each node corresponds to a problem similar to (5.2) but with a restricted domain. To restrict the domain of a variable we only modify the objective function. For example if $x_i = 1$ must be forbidden, then we set $c_i = \top$, where $\top$ corresponds to the current *ub*. This prevents the solver from choosing $x_i = 1$ because its cost is too high to be in the optimal solution. Hence, the difference between two nodes is exclusively the objective function. In particular, the set of primal constraints and dual variables are always described by LP's (5.3) and (5.4).
Our goal is to be able to learn one linear constraint per node (sub-problem) such that if we solve again the LP relaxation corresponding to this node, then, the learned linear constraint alone enforces a lower bound equal to the best-known lower bound for this sub-problem. If we learn such constraints then if another assignment leads to the same sub-problem (or a superset) we can directly derive the best-known lower bound without search. We can also hope that if we encounter a slightly different problem then the learned constraint still helps to derive a useful bound. Example 5.1 gives a concrete example of what we want to achieve.

Example 5.1. *Let P be an MILP problem:*

$$\begin{aligned}
&\min 3x_1 + 4x_2 + 6x_3 + 8x_4 + x_5 + 6x_6 \\
s.t& \\
&2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\
&x_1 + x_2 = 1 \\
&x_i + x_{i0} = 1 && x_i \in \{0,1\} \forall i \in [1,6] \\
&x_{i0} \geq 0 && \forall i \in [1,6] \\
&s \geq 0
\end{aligned}$$

Figure 5.1 shows the beginning of the search to find the optimal solution. At each node, an LP solver gives the optimal relaxed solution. We can see at node N_2 that the lower bound is 14, then at node N_3 and N_4 the optimal relaxed solution is 15 and 18. We can deduce that a better lower bound for the node N_2 is 15. In our learning procedure, we want to learn a constraint such that with this new constraint, the relaxed optimal solution at node N_2 is no better than 15. Similarly, at node N_5 and N_1, we want to learn a constraint justifying a lower bound of respectively 17

Figure 5.1: Beginning of the search tree to find the optimal solution of example 5.1



and 15.
Such constraints could enhance the future search, for example if from visiting the sub-tree issued from N_1 we derive the constraint $6x_{60} + x_{50} + 2s \geq 3$ and add it to the LP. Then when the solver visits node N_8, it finds an optimal relaxed solution of 13 without the extra constraint, while it finds 16 with the extra constraint and the search is over. In the following, we detail how we can derive such constraints. ■

Our learning strategy will learn constraints with particular properties, we call them *guarantee constraints*:

Definition 5.2 (Guarantee Constraint)**.** *Given an MILP $P : \min\{c_x^T x + c_s^T s | Ax - s = b, x \in \{0,1\}, s \in \mathbb{R}^+\}$, a constraint $w^T z = b'$ is a guarantee constraint of the bound γ if the LP $\min\{c^T z | w^T z = b', z \geq 0\}$, has optimal objective value γ.*

The largest bound a guarantee constraint can enforce is the optimal integer objective value of the problem. An obvious guarantee constraint with such bound for any MILP P with optimal objective value OPT is $c^T z = OPT$. This constraint enforces that the objective value must be equal to the optimal integer cost. An easy way to characterize a guarantee constraint is to use the following observation:

Observation 5.1. *Given an LP $\min\{c^T z | w^T z = b, z \geq 0\}$. If $\forall j, c_j \geq w_j$ then the optimal cost of this problem is at least b.*

Proof. The constraint $w^T z = b$ must be satisfied. We have $\forall j, c_j \geq w_j$, which implies constraint $c^T z \geq b$ will also be satisfied. Hence, the objective value of any solution will have a cost of at least b. □

We call the guarantee constraints verifying that property *memo constraints*:

Definition 5.3 (Memo Constraint)**.** *Given an MILP $P : \min\{c_x^T x + c_s^T s | Ax - s = b, x \in \{0,1\}, s \in \mathbb{R}^+\}$. A constraint $w^T z = \gamma$ is a memo constraint of P guaranteeing a bound γ if $\forall j, c_j \geq w_j$.*

The constraint $c^T z = OPT$ is also a memo constraint guaranteeing a lower bound OPT. A memo constraint can provide useful information on the problem we are trying to solve. The left-hand side of a memo constraint identifies a part of the objective function that is enough to justify a given lower bound represented by the right-hand side. We can find such information in the dual problem, therefore, it makes sense that Farkas constraints (definition 5.1) obtained from a dual feasible solution are memo constraints.

Lemma 5.1. *Let P_{MILP} be an MILP and P_{LP} its associated relaxed problem, if $\mathbf{y}$ is a dual feasible solution of P_{LP}, with objective value γ, then $Farkas_{P_{LP}}(\mathbf{y})$ will be a memo constraint of P_{MILP} guaranteeing a bound γ.*

Proof. First, we notice that for any proof constraint issued from a dual solution $\mathbf{y}$, the coefficient of a variable z_j is $\sum_{1 \leq i \leq m} \mathbf{y}_i a_{ij}$, where a_{ij} denotes the coefficient of variable z_j in the row i of matrix A. This corresponds exactly to the lhs of the dual constraint j under dual solution $\mathbf{y}$. Moreover, by definition of the dual, the rhs of dual constraint j is c_j. As $\mathbf{y}$ is a dual feasible solution, the coefficient of z_j in $Farkas_{P_{LP}}(\mathbf{y})$ is necessarily lower or equal to c_j. The rhs of $Farkas_{P_{LP}}(\mathbf{y})$ is exactly the objective value of the dual solution $\mathbf{y}$. Observation 5.1 concludes the proof. □

Notice that the Farkas constraint can have slack variables among its variables. Hence, in our learning mechanism, it is possible to learn constraints involving slack variables. This justifies the need to define the variables vector z.
From lemma 5.1, we can use a dual (not necessarily optimal) solution to derive a memo constraint justifying the LB associated with each node. In particular, we can compute a memo constraint at the leaf nodes, we distinguish 3 cases:

- P_{LP} is feasible, we can derive a memo constraint from the optimal dual solution.
- P_{LP} is infeasible, then the dual of P_{LP} is unbounded or infeasible, in both cases, we can obtain a memo constraint using a dual ray with an unbounded objective value. This is how conflict analysis is usually triggered in MILP, however, we keep the constraint as it is and do not pursue the conflict analysis as done in [Achterberg 2007] (see chapter 2 for a brief description).

In any case, we obtain one memo constraint at each leaf node justifying the LB associated with each of them. We can use those memo constraints as a starting point for our learning mechanism. Indeed, in a branch and bound procedure, the best-known lower bound of a sub-problem is equal to the best-known lower bound amongst its children. Hence, the idea is to learn a guarantee constraint using the guarantee constraints of the child nodes in a recursive manner, starting at the leaves. It remains to define how to combine the guarantee constraints of the children into one guarantee constraint.
It is tempting to resolve the variables one by one by using pseudo-Boolean resolution

on the current decision variable. However, the obtained constraint is not necessarily a guarantee constraint or it is not even possible to cancel the decision variable (see example 5.2. We need to look elsewhere.

Example 5.2. *Following example 5.1 and figure 5.1. Suppose that the UB is 100 and therefore removed values appear with a coefficient of 100 in the objective function. At node N_3 the LP is:*

$$\min 3x_1 + 100x_{10} + 100x_2 + 6x_3 + 100x_{30} + 8x_4 + x_5 + 6x_6 + 100x_{60}$$

s.t

$$\begin{aligned}
&c_1: \quad 2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\
&c_2: \quad x_1 + x_2 = 1 \\
&b_i: x_i + x_{i0} = 1 && \forall i \in [1,6] \\
&x_i \in \{0,1\}, \quad x_{i0} \geq 0 && \forall i \in [1,6] \\
&s \geq 0
\end{aligned}$$

Let λ_1, λ_2 be the dual variable associated to c_1, c_2 and $\mathbf{y}_i$ the dual variable associated to constraint b_i. A dual optimal solution is $\mathbf{y}_1 = 3, \mathbf{y}_3 = 6, \mathbf{y}_6 = 6$ with cost 15. The corresponding Farkas constraint is

$$3x_1 + 3x_{10} + 6x_3 + 6x_{30} + 6x_6 + 6x_{60} = 15 \tag{5.5}$$

It is a memo constraint guaranteeing a bound 15.

If we now look at node N_4, the objective function is:

$$\min 3x_1 + 100x_{10} + 100x_2 + 6x_3 + 100x_{30} + 8x_4 + x_5 + 100x_6$$

A dual optimal solution is $\lambda_1 = 2, \mathbf{y}_1 = -1, \mathbf{y}_5 = -1$ with cost 18. The corresponding Farkas constraint is

$$3x_1 - x_{10} + 4x_2 + 6x_3 + 8x_4 + x_5 - x_{50} + 12x_6 = 18 \tag{5.6}$$

It is a memo constraint guaranteeing a bound 18.
If we follow the reasoning used in PBO, and simplify constraints where a variable x_i and its opposite x_{i0} appear then constraint 5.5 using can be read 15=15. Therefore, we can't resolve x_6 and x_{60} from 5.5,5.6 with the cancellation rule.

■

5.2.1 The Fusion Resolution Rule

Nordstrom, Buss and Gocht, already defined a rule over pseudo-Boolean variables close to the one we want, this is the Fusion Resolution rule. The fusion resolution rule is described in [Buss & Nordström 2021] and is credited there to personal

communication with Stephan Gocht. For convenience, we use the notation l^σ with $\sigma \in \{0,1\}$, where $l^1 = l$ and $l^0 = \bar{l}$.

$$\frac{\alpha_j l_j + \sum_{i \neq j, \sigma \in \{0,1\}} \alpha_i^\sigma l_i^\sigma \geq b, \quad \alpha_j \bar{l}_j + \sum_{i \neq j, \sigma \in \{0,1\}} \alpha_i^\sigma l_i^\sigma \geq b'}{\sum_{i \neq j} \alpha_i^\sigma l_i^\sigma \geq \min(b, b')} \tag{5.7}$$

This rule is briefly mentioned as useful to get a better formulation if two constraints are differentiated only by one literal, but not used in an actual learning mechanism. Here we define a slightly more general version of the fusion resolution rule where the coefficients of the constraints can be different. Given two equality constraints where x_j and x_{j0} (the opposite value of x_j) each appear in one constraint. We recall that the vector z contains both the decision variables x_i and the slack variables s and x_{i0}.

$$w_j x_j + \sum_{\nu \neq \{j, j0\}} w_\nu z_\nu = b \tag{5.8a}$$

$$w'_{j0} x_{j0} + \sum_{\nu \neq \{j, j0\}} w'_\nu z_\nu = b' \tag{5.8b}$$

With $b, b', w_\nu, w'_\nu \in \mathrm{R}$. We consider the two sub-problems obtained by restricting the domain of x_j.

- If $x_j = 1$ and therefore $x_{j0} = 0$ then

$$\sum_{\nu \neq \{j, j0\}} w_\nu z_\nu = b - w_j \tag{5.9a}$$

$$\sum_{\nu \neq \{j, j0\}} w'_\nu z_\nu = b' \tag{5.9b}$$

- If $x_j = 0$ and therefore $x_{j0} = 1$ then

$$\sum_{\nu \neq \{j, j0\}} w_\nu z_\nu = b \tag{5.10a}$$

$$\sum_{\nu \neq \{j, j0\}} w'_\nu z_\nu = b' - w'_{j0} \tag{5.10b}$$

The optimal integer solution must satisfy at least one of the two systems. In particular, we can weaken each system by one constraint and only keep (5.9b) and (5.10a), the optimal integer solution must satisfy at least one of the two constraints. We specify the Fusion Resolution (FR) rule to capture this information in one constraint.

Definition 5.4 (Fusion Resolution Rule)**.**

$$\frac{w_j x_j + \sum_{\nu \neq \{j, j0\}} w_\nu z_\nu = b, \quad w'_{j0} x_{j0} + \sum_{\nu \neq \{j, j0\}} w'_\nu z_\nu = b'}{\sum_{\nu \neq \{j, j0\}} \max(w_\nu, w'_\nu) z_\nu \geq \min(b, b')} \tag{5.11}$$

Theorem 5.1. *For any pair of equality constraints, the fusion resolution rule is sound.*

Proof. We follow the same reasoning as the one described above. If an assignment satisfies (5.8a) and (5.8b), then it also satisfies either (5.9b) or (5.10a). Clearly, any solution satisfying one of the two constraints also satisfies (5.11). Therefore, any solution of (5.8a)-(5.8b) satisfies (5.11). □

Example 5.3. *Following example 5.2. We want to resolve x_6 from constraints (5.5) and (5.6) using the FR rule. To meet the requirement of the FR rule we transform (5.5) to remove x_6, we obtain the constraint*

$$3x_1 + 3x_{10} + 6x_3 + 6x_{30} = 9 \tag{5.12}$$

If we apply the FR rule then we obtain the constraint:

$$\frac{3x_1 - x_{10} + 4x_2 + 6x_3 + 8x_4 + x_5 - x_{50} + 12x_6 = 18 \quad 3x_1 + 3x_{10} + 6x_3 + 6x_{30} = 9}{3x_1 + 3x_{10} + 4x_2 + 6x_3 + 6x_{30} + 8x_4 + x_5 \geq 9} \tag{5.13}$$

This defines a memo constraint guaranteeing an LB of 9 for the node N_2 (see figure 5.1) whose objective function is:

$$\min 3x_1 + 100x_{10} + 100x_2 + 6x_3 + 100x_{30} + 8x_4 + x_5 + 6x_6$$

■

This rule verifies the properties we want, it removes one variable but does not increase the coefficients in the lhs. Hence, if we use the FR rule on 2 memo constraints then it produces a memo constraint. Furthermore, if initially x_j or x_{j0} were the only coefficients for which we had $c_j < w_j$ or $c_{j0} < w_{j0}$, then the FR rule produces a memo constraint from two non-memo constraints. However, the bound guaranteed by the obtained memo constraints depends on the two rhs. In example (5.3), we obtained a memo constraint of N_2 guaranteeing the bound 9 while we wanted to learn a constraint guaranteeing 15. The problem is that we needed to transform equation (5.5) to equation (5.12) in order to meet the requirement of the FR rule, but this operation decreased the rhs.
We adapt this rule to make it usable on any constraints and therefore without modifying the rhs. Given two equality constraints.

$$w_j x_j + x_{j0} w_{j0} + \sum_{\nu \neq \{j,j0\}} w_\nu z_\nu = b \tag{5.14a}$$

$$w'_j x_j + x_{j0} w'_{j0} + \sum_{\nu \neq \{j,j0\}} w'_\nu z_\nu = b' \tag{5.14b}$$

With $b, b', w_\nu, w'_\nu, w_j, w_{j0}, w'_j, w'_{j0} \in \mathrm{R}$. We consider the two sub-problems obtained by restricting the domain of x_j.

- If $x_j = 1$ and therefore $x_{j0} = 0$ then

$$w_j x_j + \sum_{\nu \neq \{j,j0\}} w_\nu z_\nu = b \tag{5.15a}$$

$$w'_j x_j + \sum_{\nu \neq \{j,j0\}} w'_\nu z_\nu = b' \tag{5.15b}$$

- If $x_j = 0$ and therefore $x_{j0} = 1$ then

$$w_{j0} x_{j0} + \sum_{\nu \neq \{j,j0\}} w_\nu z_\nu = b \tag{5.16a}$$

$$w'_{j0} x_{j0} + \sum_{\nu \neq \{j,j0\}} w'_\nu z_\nu = b' \tag{5.16b}$$

Once again, the optimal integer solution must satisfy at least one of the two systems and we weaken each system by one constraint to only keep (5.15b) and (5.16a). Observe that we did the reasoning with two Boolean variables x_j, x_{j0} verifying $x_j + x_{j0} = 1$. But the same reasoning could be done for a set of Boolean variables $x_{j0}, \ldots, x_{jn}$ verifying $\sum_{0 \leq k \leq n} x_{jk} = 1$. If we consider variable x_{j1}, we could define 2 systems, one with $x_{j1} = 1$ and one with $x_{j1} = 0$. A fact that can be used when working on ILPs similar to the local polytope of a WCSP.

We call this version Memo Resolution (MR).

Definition 5.5 (Memo Resolution Rule)**.**

$$\frac{w_j x_j + w_{j0} x_{j0} + \sum_{\nu \neq \{j,j0\}} w_\nu z_\nu = b, \quad w'_j x_j + w'_{j0} x_{j0} + \sum_{\nu \neq \{j,j0\}} w'_\nu z_\nu = b'}{w_j x_j + w'_{j0} x_{j0} + \sum_{\nu \neq \{j,j0\}} \max(w_\nu, w'_\nu) z_\nu \geq \min(b, b')} \tag{5.17}$$

Example 5.4. *Following example 5.2. Applying the MR rule directly on equations (5.5) and (5.6), to resolve x_6 gives:*

$$\frac{3x_1 - x_{10} + 4x_2 + 6x_3 + 8x_4 + x_5 - x_{50} + 12x_6 = 18 \quad 3x_1 + 3x_{10} + 6x_3 + 6x_{30} + 6x_6 + 6x_{60} = 15}{3x_1 + 3x_{10} + 4x_2 + 6x_3 + 6x_{30} + 8x_4 + x_5 + 6x_6 \geq 15} \tag{5.18}$$

This defines a memo constraint of N_2 guaranteeing a bound 15. ■

Currently, we are not able to state if the cutting planes method can derive the constraint derived by the FR or the MR rules.

The MR rule is not able to remove the decision variable x_j. However, if initially x_j or x_{j0} were the only coefficients for which we had $c_j < w_j$ or $c_{j0} < w_{j0}$, then the MR rule produces a memo constraint from two non-memo constraints. In this

sense, the MR rule is still resolving variable x_j. Furthermore, the MR rule is not symmetric, $MemoResolution(ctr1, ctr2, x_j) \neq MemoResolution(ctr2, ctr1, x_j)$, we can use this to our advantage and choose which coefficients we want to keep.
If we come back to our initial problem P_{MILP}, we can use the MR rule on the two memo constraints returned by the child nodes to resolve the decision variable and obtain a new constraint. In the following, we denote by $P_{x_j=0}$ (resp $P_{x_j=1}$) the problem P with additional restriction $x_j = 0$ (resp $x_j = 1$), each child node returns a memo constraint $memo_{x_j=0}$ (resp $memo_{x_j=1}$) guaranteeing a LB of $rhs_{x_j=0}$ (resp $rhs_{x_j=1}$).

Lemma 5.2. *Let P be a sub-problem. The constraint $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)$ is a memo constraint of P guaranteeing a lower bound of* $\min(rhs_{x_j=0}, rhs_{x_j=1})$.

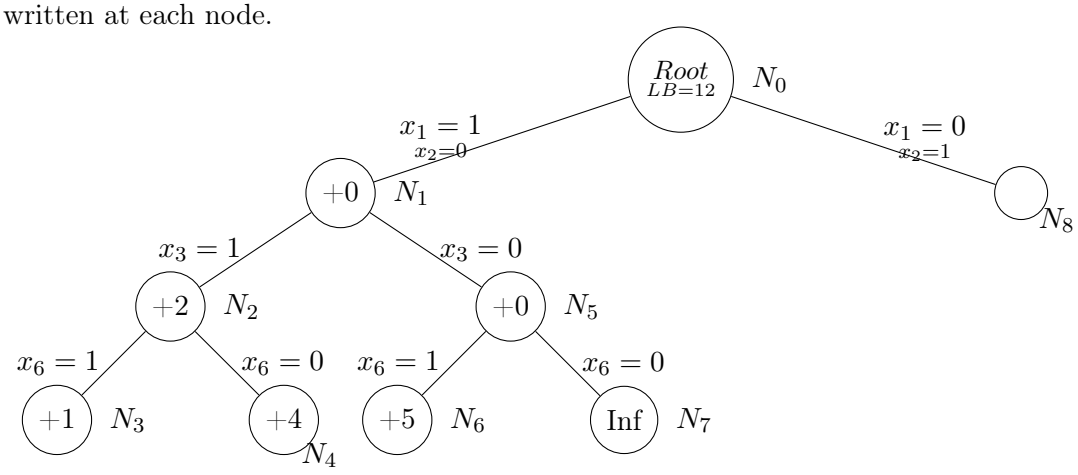
Proof. We define the two memo constraints by:

$$memo_{x_j=1} : \sum w_\nu z_\nu = rhs_{x_j=1}$$
$$memo_{x_j=0} : \sum w'_\nu z_\nu = rhs_{x_j=0}$$

The only difference between P and $P_{x_j=1}$ is the coefficient c_{j0}. Constraint $memo_{x_j=1}$ is a memo constraint of $P_{x_j=1}$ therefore $\forall \nu \neq j0$ we have $w_\nu \leq c_\nu$ and only w_{j0} can verify $w_{j0} > c_{j0}$. Similarly, coefficient w'_j in $memo_{x_j=0}$ is the only one that can verify $w'_j > c_j$. The two coefficients w'_j and w_{j0} do not appear in $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)$.
The rhs of $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)$ is $\min(rhs_{x_j=1}, rhs_{x_j=0})$. Observation 5.1 proves that this constraint is guaranteeing a lower bound of $\min(rhs_{x_j=1}, rhs_{x_j=0})$. □

The proof of lemma 5.2 justifies why in our learning process, we always choose to apply the MR rule in the direction: $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)$. The memo constraint returned by $MemoResolution(memo_{x_j=0}, memo_{x_j=1}, x_j)$ guarantees the minimum of the guaranteed bounds of the child nodes. Therefore, if $memo_{x_j=0}$,$memo_{x_j=1}$ justify the best-known lower bound of respectively $P_{x_j=0}$ and $P_{x_j=1}$, then the constraint $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)$ guarantees the best-known lower bound of P. We can learn this constraint (or an equivalent one). Notice that the MR rule returns a greater or equal constraint, therefore, we transform it into an equality constraint by adding a slack variable.
We can use the MR rule in a recursive procedure, we obtain one memo constraint at each node by applying the MR rule to resolve the next branching variable from the memo constraints of the two child nodes. At the leaves, we obtain a memo constraint from a dual solution. However, constraints learned that way will not help the solver to get better bounds. Indeed, they are all issued from the bounds and dual solutions obtained at leaf nodes, and a dual solution computed at a leaf node is too specific because it includes all the previously assigned variables. As a consequence, at each node, we learn a constraint depending on all the previously assigned variables, this makes the constraint useful only if we do the same set of

Figure 5.2: Beginning of the search tree of example 5.5. The increase of LB is written at each node.



assignments, which should never occur in a branch and bound. Put differently, the memo constraint we compute at one node shares redundant information with the memo constraint of the parent node. Indeed, a memo constraint identifies a subset of the objective function that is involved in justifying a bound, in our case, this corresponds to the LB of one node and it is equal to the LB of the parent node plus the local increase of LB. As the LB of the parent node depends on the previous decisions and removed values appear with a $\top$ objective value, those removed values will appear in the memo constraint of the child nodes.
We want to define a finer-grain learning mechanism where a memo constraint does not share redundant information with the memo constraint learned at the parent node. In other words, at each node, we are only interested in how the LB increases between this node and the leaves. We can notice that this quantity is obtained by summing the increase of LB at the current node with the minimal increase of LB between the child nodes and the leaves. Therefore, our new objective is to compute a guarantee constraint guaranteeing the minimal increase of LB between one node and the leaves by exploiting the guarantee constraints of the child nodes.

Example 5.5. *Following example 5.1, figure 5.2 shows the beginning of the search and highlights increases of LB. At each node, an LP solver gives the optimal relaxed solution. We can see at node root that the LB is 12, then it increases by 2 at node N_2, at node N_3 and N_4 it increases by 1 and 4. We can deduce that we could increase the LB by 3 at node N_2. In our learning procedure, we want to learn a constraint such that with this new constraint, the LB is increased by 3 at node N_2. Similarly, at node N_5 and N_1, we want to learn a constraint increasing the lower bound by respectively 5 and 3.* ■

It remains to somehow integrate the increase of LB at a current node in the recursive procedure. We know this information is contained in the dual solution but we are not able to capture it precisely. One possibility is to adapt the notion

of reparametrizations as defined in the CFN framework (see chapter 2) to the ILP framework. This would help us to modify the objective function along the obtained dual solutions and two dual solutions will not contain redundant information.

5.2.2 ILP and reparametrization

We use the reduced costs to produce an equivalent problem with a modified objective function. This result is well-known in the LP community studying the simplex algorithm, we give a simple proof.

Definition 5.6. *For any dual solution* $\mathbf{y}$*, the reduced costs* $rc^{\mathbf{y}}$ *with respect to* $\mathbf{y}$ *are* $rc^{\mathbf{y}} = c - A^T\mathbf{y}$.

Lemma 5.3. *Let* P *be an LP* $\{\min c^T z | Az = b, z \geq 0\}$ *and* $\mathbf{y}$ *a dual solution of* P *with cost* $\mathbf{y}^T b$ *and associated reduced costs* $rc^{\mathbf{y}}$*. Then the LP* $\{\min rc^{\mathbf{y}^T} z + \mathbf{y}^T b | Az = b, z \geq 0\}$ *is equivalent to* P*.*

Proof. The set of constraints is the same in both problems, therefore, we only need to show the equivalence of the objectives.

$$rc^{\mathbf{y}^T} z = (c - A^T\mathbf{y})^T z = c^T z - \mathbf{y}^T Az$$

Since z satisfies $Az = b$, we deduce that $rc^{\mathbf{y}^T} z = c^T z - \mathbf{y}^T b$. Hence, $rc^{\mathbf{y}^T} + \mathbf{y}^T b = c^T z$. □

Example 5.6 illustrates how a problem can be reparametrized.

Example 5.6. *Following example 5.1. Let* N_0 *be an MILP problem:*

$$\begin{aligned}
&\min 3x_1 + 4x_2 + 6x_3 + 8x_4 + x_5 + 6x_6 \\
s.t& \\
&c_1: \quad 2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\
&c_2: \quad x_1 + x_2 = 1 \\
&b_i: x_i + x_{i0} = 1 && \forall i \in [1,6] \\
&x_i \in \{0,1\}, \quad x_{i0} \geq 0 && \forall i \in [1,6] \\
&s \geq 0
\end{aligned}$$

Let λ_1, λ_2 *be the dual constraint associated to* c_1, c_2 *and* $\mathbf{y}_i$ *the dual variable associated to constraint* b_i*. A dual optimal solution* $\mathbf{y}$ *with cost 12 is:*

$$\begin{aligned}
\lambda_1 &= 2 \\
\lambda_2 &= -1 \\
\mathbf{y}_5 &= -1 \\
\mathbf{y}_6 &= -6
\end{aligned}$$

We deduce the following reduced cost:

$$\begin{aligned} rc^{\mathbf{y}}(x_1) &= 0 & rc^{\mathbf{y}}(x_{10}) &= 0 \\ rc^{\mathbf{y}}(x_2) &= 1 & rc^{\mathbf{y}}(x_{20}) &= 0 \\ rc^{\mathbf{y}}(x_3) &= 0 & rc^{\mathbf{y}}(x_{30}) &= 0 \\ rc^{\mathbf{y}}(x_4) &= 0 & rc^{\mathbf{y}}(x_{40}) &= 0 \\ rc^{\mathbf{y}}(x_5) &= 0 & rc^{\mathbf{y}}(x_{50}) &= 1 \\ rc^{\mathbf{y}}(x_6) &= 0 & rc^{\mathbf{y}}(x_{60}) &= 6 \\ & & rc^{\mathbf{y}}(s) &= 2 \end{aligned}$$

We define $\tilde{N_0}$ a problem equivalent to N_0:

$$\begin{aligned} & \min x_2 + x_{50} + 6x_{60} + 2s + 12 \\ s.t & \\ & c_1 : \quad 2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\ & c_2 : \quad x_1 + x_2 = 1 \\ & b_i : x_i + x_{i0} = 1 && \forall i \in [1,6] \\ & x_i \in \{0,1\}, \quad x_{i0} \geq 0 && \forall i \in [1,6] \\ & s \geq 0 \end{aligned}$$

For example, we can verify that the optimal integer solution $x_1 = 1, x_{20} = 1, x_3 = 1, x_{40} = 1, x_{50} = 1, x_6 = 1, s = 1$ has cost 15 in both problems. ■

If $\mathbf{y}$ defines a feasible dual solution, then all the reduced costs are positive. This lemma can also be applied to MILP, modifying the objective function according to the reduced costs gives an equivalent problem. Also, observe, that this lemma is true only for equality constraints, this explains why it is required to express the ILP in its standard form.
At each node of the search tree, we compute a dual solution $\mathbf{y}$ and use lemma 5.3 to reparametrize P into an equivalent $MILP$ $\tilde{P}$. In this reparametrization, the objective coefficient of the slack variables can be non-zero, this motivates why we introduced c_s. We suppose that all the constant terms appearing in the objective function are gathered in $c_\varnothing \in \mathbb{R}$. Both the dual solution $\tilde{\mathbf{y}} = 0$ of $\tilde{P}$ and $\mathbf{y}$ has objective $c_\varnothing$. Therefore, the dual solution computed by the children of $\tilde{P}$ does not contain redundant information with $\mathbf{y}$. They will capture only the specific information related to the child nodes. We can exploit this property to integrate reparametrization in our recursive procedure.

Lemma 5.4. *Let P be a sub-problem and $\tilde{P}$ a reparametrization of P obtained through lemma 5.3 and dual solution $\mathbf{y}$. Summing any dual feasible solution of $\tilde{P}_{x_j=0}$ with $\mathbf{y}$ leads to a dual solution satisfying all the dual constraints of P except the one corresponding to x_j.*

Proof. Let $\mathbf{y}$ be a dual solution of P and $rc^{\mathbf{y}}$ the associated reduced costs. Then the objective function $\tilde{c}$ of $\tilde{P}_{x_j=0}$ is $\forall \nu \neq j,\ \tilde{c}_\nu = rc^{\mathbf{y}}_\nu$ and $\tilde{c}_j = \top$. Therefore, the dual of $\tilde{P}$ is defined by:

$$\max\{b^T y + c_\varnothing | A^T y \leq \tilde{c}, y \in \mathrm{R}\} \tag{5.19}$$

Let $\mathbf{y}^{P_{x_j=0}}$ ($\mathbf{y}^{j0}$ for short) be a feasible solution of (5.19). Summing $\mathbf{y}$ and $\mathbf{y}^{j0}$ gives:

$$A^T\mathbf{y} + A^T\mathbf{y}^{j0} \leq c + \tilde{c}$$

Let A' be the matrix A without the column corresponding to variable x_j, we obtain:

$$\begin{aligned} A'^T\mathbf{y} + A'^T\mathbf{y}^{j0} &\leq A'^T\mathbf{y} + rc^{\mathbf{y}} \\ \Leftrightarrow A'^T\mathbf{y} + A'^T\mathbf{y}^{j0} &\leq A'^T\mathbf{y} + c - A'^T\mathbf{y} \\ \Leftrightarrow A'^T\mathbf{y} + A'^T\mathbf{y}^{j0} &\leq c \end{aligned}$$

Therefore, the dual solution $\mathbf{y} + \mathbf{y}^{j0}$ satisfies all the dual constraints of P except the one corresponding to x_j. □

Corollary 5.1. *Summing any memo constraint of $\tilde{P}_{x_j=0}$ with $Farkas_P(\mathbf{y})$ gives a constraint where all the coefficients in the lhs will be lower than their corresponding coefficient in the objective function of P except for x_j.*

Proof. For all the variables outside x_j, their coefficients in $Farkas_P(\mathbf{y})$ are lower than c. And their coefficients in $memo_{x_j=0}$ are lower than $\tilde{c} = rc^{\mathbf{y}}$. The end of the proof is similar to the proof of lemma 5.4. □

This lemma can easily be extended to the case where we study the child $\tilde{P}_{x_j=1}$. In this case, summing any dual solution of $\tilde{P}_{x_j=1}$ with $\mathbf{y}$ gives a dual solution satisfying all the dual constraints of P except the one corresponding to x_{j0}.

Theorem 5.2. *Let P be a sub-problem and $\tilde{P}$ a reparametrization of P obtained through lemma 5.3 and dual solution $\mathbf{y}$. Let $memo_{x_j=0}$,$memo_{x_j=1}$ be the memo constraints of respectively $\tilde{P}_{x_j=0}$ and $\tilde{P}_{x_j=1}$. The constraint $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)+ Farkas_P(\mathbf{y})$ is a memo constraint of P.*

Proof. We first observe that the two following constraints are equivalent:

$$MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j) + Farkas_P(\mathbf{y}) \tag{5.20}$$

$$MemoResolution(memo_{x_j=1} + Farkas_P(\mathbf{y}), memo_{x_j=0} + Farkas_P(\mathbf{y}), x_j) \tag{5.21}$$

From Corollary 5.1 we know that only the coefficient of x_j from constraint $memo_{x_j=0}+ Farkas_P(\mathbf{y})$ may be higher than its objective coefficient. By construction of the MR rule, this coefficient does not appear in $MemoResolution(memo_{x_j=1}+Farkas_P(\mathbf{y}), memo_{x_j=0}+ Farkas_P(\mathbf{y}), x_j)$. We reason similarly for x_{j0} and $memo_{x_j=1} + Farkas_P(\mathbf{y})$. □

Theorem 5.2 ensures that at each node of the search tree, summing the Farkas constraint and the constraint returned by the MR rule gives a memo constraint guaranteeing the minimal guaranteed bound of the child nodes. If, $memo_{x_j=0}$,$memo_{x_j=1}$ justify the best known lower bound for respectively $\tilde{P}_{x_j=0}$ and $\tilde{P}_{x_j=1}$ then constraint $MemoResolution(memo_{x_j=1}, memo_{x_j=0}, x_j)+Farkas_P(\mathbf{y})$ justifies the best known lower bound of P. With the reparametrization, we know that the learned constraints will depend on the current cost distribution but not directly on previous assignments. Moreover, a memo constraint of problem P is a guarantee constraint of all the reparametrizations of P. Therefore, during the search, if a sequence of reparametrizations leads to a problem equivalent to P then the memo constraint of P will still guarantee the correct bound.
Algorithm 6 explains how the reparametrization is integrated into the learning mechanism. It performs a branch and bound algorithm, where each node returns a memo constraint (lines 8,9). The function ChooseNextVar(DecVars) is a heuristic selecting the next branching decision (line 3). At each node, function GetDualSol(P) returns the optimal dual solution if it exists, otherwise, it returns a dual unbounded solution (line 1). This dual solution is used to compute a Farkas constraint (line 2). Once the two children have been visited it applies the MR rule on the returned memo constraints (line 4). Then, it computes the sum of this last constraint and the Farkas constraint (line 5). This $\geq$ constraint is transformed to an equality constraint by adding a slack variable (6). Finally, the constraint is learned (7) and returned (line 8).
This defines the basic approach. This can be adapted to fit the particularity of the solved instances. For example, if we consider an ILP corresponding to the local polytope of a WCSP, then, we know that the Boolean decision variables correspond to the direct encoding of multi-valued variables, where value (i, a) is represented by a Boolean variable x_{ia} taking value 1 if $x_i = a$, and a constraint $\sum_{a\in \boldsymbol{D}_i} x_{ia} = 1$ enforces that each variable must be assigned to only one value. We can integrate this information into our procedure. The lemmas and theorems are the same but instead of having variables x_j and their opposite x_{j0}, we have variables x_{ja} and their opposite $x_{jb} \forall b \in \boldsymbol{D}_j, b \neq a$. For example, if we branch on $x_{ja} = 1$ then all the variables $x_{jb} \forall b \in \boldsymbol{D}_j, b \neq a$ are set to 0.

Example 5.7. *Following example 5.6, we give a full run of our learning procedure and show how it can enhance the search. We had dual optimal solution* $\mathbf{y} : \lambda_1 =$

Algorithm 6: Learning sub-problems bounds with reparametrization

```
/* Performs a branch and bound and learn constraints using the
   dual solution at the leaf nodes;
P is the current LP ;
DecVar is the current decision variable and we branch it on
Value;
DecVars is a list of decision variables, and
ChooseNextVar(DecVars) an heuristic selecting a variable from
this list.  ;
UB is the global upper bound;
                                                              */
Function LearnDual(P,DecVar,Value,DecVars)
    assign(DecVar,Value,P) ;
    NewLB ← solve(P);
1   DualSol ← GetDualSol(P) ;
2   DPC ← Farkas_P(DualSol) ;
    if NewLB<UB then
        if DecVars is not empty then
            /* Visit the two child nodes                         */
3           NextDec← ChooseNextVar(DecVars) ;
            DualSol ← GetDualSol(P) ;
            P̃ ← Reparametrize(P,DualSol);
            memo1=LearnDual(P̃,NextDec,1,DecOrder);
            memo2=LearnDual(P̃,NextDec,0,DecOrder);
4           MemoCtr← MemoResolution(memo1,memo2,DecVar) ;
5           SumConstraint ← DPC+MemoCtr ;
6           SumEQConstraint ← AddSlackVariable(SumConstraint) ;
7           Learn SumEQConstraint ;
8           return SumConstraint ;
        else
            if NewLB < UB then
                UB ← NewLB;
9   return DPC
```

$2, \lambda_2 = -1, \mathbf{y}_5 = -1, \mathbf{y}_6 = -6$ *and obtained* $\tilde{N}_0$ *a reparametrization of* N_0*:*

$$
\begin{aligned}
& \min x_2 + x_{50} + 6x_{60} + 2s + 12 \\
s.t & \\
& c_1 : \quad 2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\
& c_2 : \quad x_1 + x_2 = 1 \\
& b_i : x_i + x_{i0} = 1 && \forall i \in [1, 6] \\
& x_i \in \{0, 1\}, \quad x_{i0} \geq 0 && \forall i \in [1, 6] \\
& s \geq 0
\end{aligned}
$$

We also compute a Farkas constraint from $\mathbf{y}$*:*

$$Farkas_{N_0}(\mathbf{y}) : \quad 3x_1 + 3x_2 + 6x_3 + 8x_4 + x_5 - x_{50} + 6x_6 - 6x_{60} - 2s = 12$$

Suppose the solver follows the branching decision made in the search tree depicted in figure 5.1. The solver branches on $x_1 = 1, x_2 = 0$*. The objective function of problem* N_1 *is:*

$$obj_{N_1} : \min 100x_2 + x_{50} + 6x_{60} + 2s + 12 \tag{5.22}$$

The optimal relaxed solution is 12, and the search continues.
The solver branches on $x_3 = 1$*. The objective function of problem* N_2 *is:*

$$obj_{N_2} : \min 100x_2 + 100x_{30} + x_{50} + 6x_{60} + 2s + 12 \tag{5.23}$$

The new optimal solution is 14. We denote its dual optimal solution by $\mathbf{y}^{N_2}$*:*

$$
\begin{aligned}
\lambda_1^{N_2} &= -1 \\
\lambda_2^{N_2} &= 2 \\
\mathbf{y}_3^{N_2} &= 3 \\
\mathbf{y}_5^{N_2} &= 1 \\
\mathbf{y}_6^{N_2} &= 6
\end{aligned}
$$

We obtain Farkas constraint:

$$Farkas_{N_2}(\mathbf{y}^{N_2}) : 3x_{30} + x_{50} - 4x_4 + 6x_{60} + s = 2 \tag{5.24}$$

The objective function is reparametrized and the solver branches on $x_6 = 1$*, hence the objective function of node* N_3 *is:*

$$obj_{N_3} : \min 100x_{10} + 100x_2 + +100x_{30} + 4x_4 + 100x_{60} + s + 14 \tag{5.25}$$

The optimal cost is 1. One optimal dual solution is $\mathbf{y}^{N_3}$*:*

$$\begin{aligned}\lambda_1^{N_3} &= -1\\ \lambda_2^{N_3} &= 2\\ \mathbf{y}_3^{N_3} &= 3\\ \mathbf{y}_6^{N_3} &= 6\end{aligned}$$

We obtain Farkas constraint:

$$Farkas_{N_3}(\mathbf{y}^{N_3}) : 3x_{30} - 4x_4 - x_5 + 6x_{60} + s = 1 \tag{5.26}$$

The solver backtracks and branches on $x_6 = 0$*. The objective function of node* N_4 *is:*

$$obj_{N_4} : \min 100x_{10} + 100x_2 + 100x_{30} + 4x_4 + 100x_6 + s + 14 \tag{5.27}$$

The optimal cost is 4. One optimal dual solution is $\mathbf{y}^{N_4}$*:*

$$\begin{aligned}\lambda_1^{N_4} &= 1\\ \lambda_2^{N_4} &= -2\\ \mathbf{y}_3^{N_4} &= -3\\ \mathbf{y}_5^{N_4} &= -1\end{aligned}$$

We obtain Farkas constraint:

$$Farkas_{N_4}(\mathbf{y}^{N_4}) : -3x_{30} + 4x_4 - x_{50} + 6x_6 - s = 4 \tag{5.28}$$

The solver backtracks. We can apply the MR rule on $Farkas_{N_3}$ *and* $Farkas_{N_4}$ *to resolve* x_6*, we obtain:*

$$\frac{3x_{30} - 4x_4 - x_5 + 6x_{60} + s = 1 \quad -3x_{30} + 4x_4 - x_{50} + 6x_6 - s = 4}{3x_{30} + 4x_4 + s \geq 1} \tag{5.29}$$

We transform this constraint into an equality constraint by introducing a slack variable s_1*. Then this constraint is summed with* $Farkas_{N_2}$ *:*

$$\begin{aligned} 3x_{30} + x_{50} - 4x_4 + 6x_{60} + s &= 2 && (5.30)\\ + && (5.31)\\ 3x_{30} + 4x_4 + s - s_1 &= 1 && (5.32)\\ \hline && (5.33)\\ Learn_{N_2} : 6x_{30} + x_{50} + 6x_{60} + 2s - s_1 &= 3 && (5.34)\end{aligned}$$

Constraint $Learn_{N_2}$ *is a memo constraint of* N_2 *guaranteeing an increase of LB of 3.*
By following the same procedure we obtain a memo constraint for node N_5*:*

$$Learn_{N_5} : 5x_3 + x_{50} + 2s - s_2 = 5 \tag{5.35}$$

We can now apply the MR rule on $Learn_{N_2}$ and $Learn_{N_5}$ to resolve x_3:

$$\frac{6x_{30} + x_{50} + 6x_{60} + 2s - s_1 = 3 \quad 5x_3 + x_{50} + 2s - s_2 = 5}{Learn_{N_1} : x_{50} + 6x_{60} + 2s \geq 3} \tag{5.36}$$

We add a slack variable s_3 and include the constraint in the LP.
The solver backtrack to N_0 and considers the right branch where $x_1 = 0, x_2 = 1$ the problem is:

$$\begin{aligned}
&\min 100x_1 + x_2 + 100x_{20} + x_{50} + 6x_{60} + 2s + 12 \\
&s.t \\
&c_1 : \quad 2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\
&c_2 : \quad x_1 + x_2 = 1 \\
&Learn_{N_1} : x_{50} + 6x_{60} + 2s - s_3 = 3 \\
&b_i : x_i + x_{i0} = 1 && \forall i \in [1, 6] \\
&x_i \in \{0, 1\}, \quad x_{i0} \geq 0 && \forall i \in [1, 6] \\
&s \geq 0
\end{aligned}$$

The optimal relaxed solution costs 16, and the search is over. Without the constraint $Learn_{N_1}$, the optimal relaxed solution is 13, and the search continues. ■

5.2.3 Value Removal and Learning

Theorem 5.2 is sound only if the objective function is modified only by reparametrization or branching decision. In particular values outside the decision variables should not be removed (see example 5.8). We discuss here how to integrate pruning operations into the learning mechanism.

Example 5.8. *Let P be an ILP problem:*

$$\begin{aligned}
\min 2x_1 + 3x_2 + 4x_3 \\
s.t \\
c_1 : x_1 + x_{10} = 1 \\
c_2 : x_2 + x_{20} = 1 \\
c_3 : x_3 + x_{30} = 1 \\
c_4 : x_4 + x_{40} = 1 \\
c_5 : 2x_1 + 2x_2 + 2x_3 - s_1 = 3 \\
c_6 : -x_1 + x_3 + x_4 - s_2 = 1 \\
\forall i \in [1, \ldots, 4] \quad x_i \in \{0, 1\} \\
\forall i \in [1, \ldots, 4] \quad x_{i0} \geq 0 \\
s_1, s_2 \geq 0
\end{aligned}$$

If we do the assignment $x_1 = 0$ then constraint c_5 assigns x_2 and x_3 by domain propagation. With this information, the optimal cost is 9. We derive the Farkas constraint

$$2x_1 + 2x_{10} + 3x_2 + 3x_{20} + 4x_3 + 4x_{30} = 9 \tag{5.37}$$

As you can see the variables x_{20}, x_{30} appear with positive coefficients, therefore this is not a memo constraint of $P_{x_1=0}$. ■

Suppose we have a MILP P with only equality constraints and Boolean decision variables. Let R define a set of value removals. Suppose a process outside branching decision removes a value x_j. As seen in the example, x_j can appear with a coefficient $w_j > c_j$ in the Farkas constraint, and therefore we do not obtain a memo constraint of P. However, this constraint is a memo constraint of $P_{x_j=0}$. One possibility to resolve x_j is to compute the memo constraint of $P_{x_j=1}$ and apply the MR rule. The memo constraint returned by $P_{x_j=1}$ depends on the process that removed x_j. We specify here the obtained constraint for a value removal induced by bound propagation and node consistency.

Bound propagation

Let's first have a look at equality constraints with no slack variables. Let $ctr_1 : \sum w_i x_i = b$ be a constraint of P and applying bound propagation on ctr_1 removes x_j. If we consider the problem $P_{x_j=1}$ then we obtain a direct failure as the constraint ctr_1 is not satisfied. We study the sub-problem defined by ctr_1 and the bound constraint of each variable:

$$\min \sum c_i x_i \tag{5.38a}$$

$$s.t \tag{5.38b}$$

$$ctr_1 : \sum w_i x_i = b$$

$$\forall i, x_i + x_{i0} = 1 \tag{5.38c}$$

We denote by λ the dual variable corresponding to ctr_1 and $\mathbf{y}_i$ the dual variables corresponding to constraints (5.38c) associated with i. The dual problem is :

$$\min \lambda b + \sum \mathbf{y}_i$$

$$s.t$$

$$\forall i, \lambda w_i + \mathbf{y}_i \leq c_i$$

There exist two cases in which x_j can be removed from ctr_1.

1. The first case is:

$$w_j + \sum_{i \notin R} \max(0, w_i) < b \tag{5.39}$$

We define a dual solution $\mathbf{y}$ by setting λ to any positive value, then:

$$\forall i \notin R, \mathbf{y}_i = \min(-\lambda w_i, 0, c_i)$$
$$\mathbf{y}_k = -\lambda w_k$$

The cost of this dual solution is

$$\lambda b - \lambda w_j + \sum_{i \notin R} \min(-\lambda w_i, 0, c_i) \tag{5.40}$$

This is equivalent to:

$$\lambda b - \lambda w_j - \lambda \sum_{i \notin R} \max(w_i, 0, -\frac{c_i}{\lambda}) \tag{5.41}$$

From assumptions (5.39) and $c_i \geq 0$, we deduce that this cost is >0, and increasing λ will produce a new solution with an increased objective value. Therefore, the dual problem is unbounded, and we can derive a Farkas constraint with an arbitrarily large rhs:

$$\sum_{i \in R} \lambda w_i + \sum_{i \notin R} (\lambda w_i + \min(-\lambda w_i, 0, c_i)) = \lambda b - \lambda w_k + \sum_{i \notin R} \min(-\lambda w_i, 0, c_i) \tag{5.42}$$

2. The second case is:

$$\sum_{i \in \boldsymbol{X} \setminus \{R\}} \max(0, w_i) > b \tag{5.43}$$

This is simply a symmetry of the first case and we can define an unbounded dual solution with the same reasoning.

Finally, we apply the MR rule to resolve variable x_j from the memo constraint (5.42) and the constraint obtained at node P. In practice, the negative coefficients of (5.42) are arbitrarily large, hence we directly weaken them because they will always be discarded when applying the MR rule. Therefore, the only variables appearing in the constraint are $\{i \in R, w_i > 0\}$. We say that those values provide an explanation for the removal of x_j. Their coefficients are arbitrarily large but it is possible to saturate them after applying the MR.

If the constraint admits a slack variable with a negative coefficient: $ctr_2 : \sum w_i x_i - s_1 = b$, then case (2) can't be triggered as the slack variable can always be used to satisfy the constraint tightly. We derive the same constraint if case (1) occurs.
If the slack variable appears with a positive coefficient, $ctr_3 : \sum w_i x_i + s_2 = b$ then case (1) can't be triggered and we proceed as usual for case (2).

Example 5.9. *Following example 5.8. We derived a Farkas constraint* $2x_1 + 2x_{10} + 3x_2 + 3x_{20} + 4x_3 + 4x_{30} = 9$.
The constraint $2x_1 + 2x_2 + 2x_3 - s_1 = 3$ *is responsible for the removal of* $x_{20} = 1, x_{30} = 1$. *If we try to assign* $x_{20} = 1$ *then we obtain the following dual problem:*

$$\begin{aligned} \max 3\lambda + \sum_{i=1}^{3} \mathbf{y}_i \\ 2\lambda + \Pi_1 &\leq \top \\ \Pi_1 &\leq 0 \\ 2\lambda + \Pi_2 &\leq \top \\ \Pi_2 &\leq 0 \\ 2\lambda + \Pi_3 &\leq 4 \\ \Pi_3 &\leq 0 \end{aligned}$$

The problem is unbounded, for example, $\lambda = 10, \pi_3 = -20$ *is a dual solution with cost 10,* $\lambda = 100, \pi_3 = -200$ *is a dual solution with cost 100. A Farkas constraint is:*

$$100(2x_1 + 2x_2 - x_{30} - s_1) = 100 \tag{5.44}$$

If we use the MR rule to resolve x_2 *we obtain:*

$$\frac{2x_1 + 2x_{10} + 3x_2 + 3x_{20} + 4x_3 + 4x_{30} = 9, \quad 100(2x_1 + 2x_2 - x_{30} - s_1) = 100}{200x_1 + 2x_{10} + 200x_2 + 4x_3 + 4x_{30} \geq 9} \tag{5.45}$$

With the same reasoning, we can resolve x_{30} *and obtain constraint:*

$$200x_1 + 2x_{10} + 3x_2 + 4x_3 \geq 9 \tag{5.46}$$

This constraint is a memo constraint of $P_{x_1=0}$ *guaranteeing a bound* 9. ■

5.2.3.1 Node Consistency

We say that a variable x_j is not node consistent if $c_j + c_\varnothing > ub$, where $c_\varnothing$ denotes a constant term appearing in the objective function. In our procedure, $c_\varnothing$ is increased along the reparametrization and always corresponds to the current lb. When a value is not node consistent it can be removed. We again study problem $P_{x_j=1}$, we obtain a direct bound violation because $c_j + lb > ub$. A dual solution justifying this is simply $\pi_j = c_j$. The obtained constraint is $c_j x_j + c_j x_{j0} = c_j$.
Let $w_j x_j + \sum w_i z_i \geq b$ be the constraint obtained by visiting the branch at node P. Applying the MR rule leads to:

$$\frac{c_j x_j + c_j x_{j0} = c_j \quad w_j x_j + \sum_{i \neq j} w_i z_i = b}{c_j x_j + \sum_{i \neq j} \max(w_i, 0) z_i} \geq \min(c_j, b) \tag{5.47}$$

This constraint is a memo constraint of P guaranteeing a lower bound of $\min(c_i, b)$. If $b \geq c_i$, then propagating the learned constraint will give a direct bound violation because $c_\varnothing + c_i \geq ub$.

Algorithm 7 shows how removal can be included in our learning procedure. At each node, bound propagation and node consistency are enforced (line 1). For each removal, a reason constraint is captured, it can be either (5.42) or (5.47) (line 2). When the procedure derives a new constraint, we iteratively use the reason constraint with the MR rule to make sure that removed values do not break the memo property (line 2).

5.3 Learning bounds in a CFN solver

The learning mechanism we defined can be used on any ILP with Boolean decision variables expressed in standard form. This last requirement is necessary only because we need lemma 5.3 to reparametrize the problem. However, if we can reparametrize the problem without relying on lemma 5.3 then a similar strategy could be defined for any ILP with Boolean decision variables. In particular, we know that CFN solvers based on soft consistency algorithms (see chapter 2), natively perform reparametrization corresponding to a dual solution of a local polytope. We can use those characteristics to adjust the learning mechanism and learn linear constraints in the CFN framework. This won't be helpful if the CFN solvers are not able to propagate the learned constraints, hence, we suppose they are extended with the dedicated propagator defined in chapter 3. We consider a problem $P < \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top >$ associated with its local polytope PrimalLin and $\mathcal{W} \subseteq \boldsymbol{C}$ as the set of linear $\geq$ constraints.

PrimalLin

$$\min Obj \stackrel{\text{def}}{=} c_\varnothing + \sum_{i \in \boldsymbol{X}, a \in \boldsymbol{D}_i} c_i(a) x_{ia} + \sum_{c_{\boldsymbol{S}} \in C^+ \setminus \mathcal{W}, \tau \in \ell(\boldsymbol{S})} c_{\boldsymbol{S}}(\tau) x_{\boldsymbol{S}:\tau} \tag{5.48a}$$

$$\text{s.t.} \quad \forall i \in \boldsymbol{X}, \quad \sum_{a \in \boldsymbol{D}_i} x_{ia} = 1 \tag{5.48b}$$

$$\forall c_{\boldsymbol{S}} \in C^+ \setminus \mathcal{W}, i \in \boldsymbol{S}, a \in \boldsymbol{D}_i \Big(\sum_{\tau \in \ell(\boldsymbol{S}), \tau_i = a} x_{\boldsymbol{S}:\tau} \Big) - x_{ia} = 0 \tag{5.48c}$$

$$Wx \geq b \tag{5.48d}$$

We denote by $\mathbf{y} = \{\Pi, \phi, \Lambda\}$ a dual solution of PrimalLin. Where π_i corresponds to constraint (5.48b), $\varphi_{ia:\boldsymbol{S}}$ to constraint (5.48c) and λ_k to the row k of (5.48d).
In an LP solver, we had access to a dual solution and we defined how to reparametrize the problem. For CFN solvers, we are in the opposite situation. The solver derives a reparametrization through a sequence of EPTs and we need to define the dual

Algorithm 7: Learning sub-problems bounds with reparametrization

```
/* Performs a branch and bound and learn constraints using the
   dual solution at the leaf nodes;
P is the current LP ;
DecVar is the current decision variable and we branch it on
Value;
DecVars is a list of decision variables, and
ChooseNextVar(DecVars) an heuristic selecting a variable from
this list.  ;
UB is the global upper bound;
                                                              */
Function LearnDualWithRemoval(P,DecVar,Value,DecVars)
    assign(DecVar,Value,P) ;
1   ValueRemoved ← ApplyNCandBP(P) ;
    foreach value ∈ ValueRemoved do
2       Reason(value) ← FindReason(value) ;
    NewLB ← solve(P);
    DualSol ← GetDualSol(P) ;
    DPC ← Farkas_P(DualSol) ;
    if NewLB< UB then
        if DecVars is not empty then
            /* Visit the two child nodes                         */
            NextDec← ChooseNextVar(DecVars) ;
            DualSol ← GetDualSol(P) ;
            P̃ ← Reparametrize(P,DualSol);
            memo1=LearnDualWithRemoval(P̃,NextDec,1,DecOrder);
            memo2=LearnDualWithRemoval(P̃,NextDec,0,DecOrder);
            MemoCtr← MemoResolution(memo1,memo2,DecVar) ;
            SumConstraint ← DPC+MemoCtr ;
            foreach value ∈ ValueRemoved do
                SumConstraint ←
                  MemoResolution(Reason(value),memo2,value) ;
3           SumEQConstraint ← AddSlackVariable(SumConstraint) ;
            Learn SumEQConstraint ; return SumConstraint ;
        else
            if NewLB < UB then
                UB ← NewLB;
    return DPC
```

solution. We already presented in section 3.1.3 how EPTs are reflected in the dual solution. But we also showed that when linear constraints are involved, then the dual solution of one node and its children contain redundant information (see example 3.6 in chapter 3). In particular, the lemma 5.4 and corollary 5.1 do not hold in this context. Therefore, within our learning mechanism, the recursive procedure will diffuse redundant information and we will not learn memo constraints.
However, we can still try to derive a dual solution sharing as little information as possible with the dual solution computed at a parent node. Given a node P, its child node $\tilde{P}$, associated to dual solutions $\mathbf{y}^P, \mathbf{y}^{\tilde{P}}$ with cost equal to $c^P_\varnothing$, $c^{\tilde{P}}_\varnothing$. The dual solution $\hat{\mathbf{y}} = \mathbf{y}^{\tilde{P}} - \mathbf{y}^P$ has a cost $c^{\tilde{P}}_\varnothing - c^P_\varnothing$, this corresponds to the increase of cost at node $\tilde{P}$. Ideally, we want to compute a memo constraint guaranteeing this bound. Unfortunately, $\hat{\mathbf{y}}$ may not be a dual feasible solution. Since the linear constraints are $\geq$ constraints, their dual variables must satisfy $\Lambda \geq 0$. Therefore if some λ_k verifies $\lambda^P_k > \lambda^{\tilde{P}}_k$ then $\hat{\lambda_k} < 0$ and the dual solution is not feasible (see example 5.10). Furthermore, it is not possible to derive a Farkas constraint from this particular dual unfeasible solution. Indeed, multiplying a $\geq$ constraint by a negative coefficient gives a $\leq$ constraint. We can't sum it with other $\geq$ constraints. We have no choice but to define a different dual solution $\hat{y}$.

$$\hat{\Pi} = \Pi^{\tilde{P}} - \Pi^P \tag{5.49a}$$

$$\forall \lambda_k \in \Lambda, \quad \hat{\lambda_k} = \max(\lambda^{\tilde{P}}_k - \lambda^P_k, 0) \tag{5.49b}$$

Unfortunately, $\hat{\mathbf{y}}$ may still not be a dual feasible solution (see example 5.10). We can still compute a Farkas constraint and use it in the learning approach, but it is not possible anymore to claim that we learn a memo constraint at each node (see example 5.10).

Example 5.10. *Following example 3.6 (which is the WCSP formulation of example 5.2). We had the initial WCSP:*

$$\begin{aligned} \min\, & 2x_{1a} + 3x_{2a} + 4x_{3a} \\ & s.t \\ c_{123} : & 2x_{1a} + 2x_{2a} + 2x_{3a} \geq 3 \\ c_{134} : & x_{1b} + x_{3a} + x_{4a} \geq 2 \\ \forall i \in [1, \ldots, 4] \quad & x_{ia}, x_{ib} \in \{0, 1\} \end{aligned}$$

We found a dual solution $\mathbf{y}^P = \{\lambda^P_{123} = 1.5, \lambda^P_{134} = 1, \pi^P_1 = -1, \pi^P_3 = 0, \pi^P_3 = 0 \pi^P_4 = -1\}$ *with cost 4.5 (rounded to 5).*
After the reparametrization and the assignment of $x_1 = b$ *we obtained problem* $\tilde{P}_{x_1=b}$*:*

$$\begin{aligned} & 100x_{1a} + x_{4b} + 5 + [\delta_{134} : x_{1b} + x_{3a} - x_{4b} - 1] + [\delta_{123} : 2x_{1a} - x_{1b} + 3x_{2a} + 3x_{3a} - 4] \\ & s.t \\ c_{123} : & 2x_{1a} + 2x_{2a} + 2x_{3a} \geq 3 \\ c_{134} : & x_{1b} + x_{3a} + x_{4a} \geq 2 \\ \forall i \in [1, \ldots, 4] \quad & x_{ia}, x_{ib} \in \{0, 1\} \end{aligned}$$

We obtained a dual solution $\mathbf{y}^{\tilde{P}} = \{\lambda^{\tilde{P}}_{123} = 0, \lambda^{\tilde{P}}_{134} = 0, \pi^{\tilde{P}}_1 = 0, \pi^{\tilde{P}}_2 = 3, \pi^{\tilde{P}}_3 = 4, \pi^{\tilde{P}}_4 = 0\}$ *with cost 7.*
The dual solution $\hat{\mathbf{y}} = \mathbf{y}^{\tilde{P}} - \mathbf{y}^{P}$ *is not feasible because* $\hat{\lambda}_{123} = -1.5$ $\hat{\lambda}_{134} = -1$.
If we define $\hat{\mathbf{y}}$ *as in equation (5.49a)-(5.49b) then we obtain* $\hat{\mathbf{y}} = \{\hat{\lambda}_{123} = 0, \hat{\lambda}_{134} = 0, \hat{\pi}_1 = 1, \hat{\pi}_2 = 3, \hat{\pi}_3 = 4, \hat{\pi}_4 = 1\}$ *with cost 9. This does not define a valid dual feasible solution of* $\tilde{P}_{x_1=b}$ *(the optimum with integer 0/1 variables is 7 and the optimal relaxed solution is 5). The obtained proof constraint is* $x_{1a} + x_{1b} + 3x_{2a} + 3x_{2b} + 4x_{3a} + 4x_{3b} + x_{4a} + x_{4b} = 9$. *This is not a memo constraint of* $\tilde{P}_{x_1=b}$. ■

This approach can be implemented in a CFN solver. However, we need to make sure that the learned constraints are expressible. For example, the tuple variable $x_{\boldsymbol{S}:\tau}$ appears in the local polytope PrimalLin and thus can be involved in the learned constraints. The CFN solver must be able to connect this tuple variable with the actual tuple τ of the cost function $\boldsymbol{S}$. In particular, if this tuple is removed then $x_{\boldsymbol{S}:\tau}$ should be removed from all the learned linear constraints and vice versa. This is a software engineering issue. It is unusual in CP solvers in general (and in TOULBAR2 in particular) to have the facility to be notified when a tuple becomes impossible, which is why we need to map tuples to values. One way to be sure those interactions are correctly modeled is to use the dual encoding of the CFN as defined in chapter 4.
The propagator Propagate has been defined only for $\geq$ constraints with positive integer coefficients. The learned constraint does not directly verify this requirement. To obtain integer coefficients, we need to round up all the coefficients of the rhs and the lhs. The cost functions contain only integer cost, hence the obtained constraint is still a memo constraint. One possibility to remove a negative coefficient is to use the cancellation rule 2.15. But we already stated earlier that the cancellation rule might break the memo property of the constraint. A second possibility is to weaken the negative coefficients. If the learned constraint is a memo constraint, then it remains a memo constraint after the weakening. Moreover, this weakening can lead to a strengthened saturation rule and produce a stronger constraint than the one obtained with the cancellation rule. Example 5.11 illustrates this.

Example 5.11. *Consider the following LP:*

$$\begin{aligned} \min x_1 + x_2 + x_3 + x_4 \\ 3x_1 + 2x_2 + x_3 - 2x_4 \geq 1 \\ x_1 + x_{10} = 1 \\ x_2 + x_{20} = 1 \\ x_3 + x_{30} = 1 \\ x_4 + x_{40} = 1 \end{aligned}$$

The optimal relaxed solution of this problem is $\{x_1 = \frac{1}{3}, x_2 = 0, x_3 = 0, x_4 = 0\}$ *with cost* $\frac{1}{3}$. *The constraint is already saturated. If we cancel* $-2x_4$ *using the constraint*

$x_4 + x_{40} = 1$, *we obtain* $3x_1 + 2x_2 + x_3 + 2x_{40} \geq 3$, *the optimal solution is still* $\{x_1 = \frac{1}{3}, x_2 = 0, x_3 = 0, x_4 = 0\}$. *But if we weaken the coefficient* $-2x_4$ *and saturate we obtain:* $x_1 + x_2 + x_3 \geq 1$ *with optimal cost 1.* ■

Example 5.12 illustrates a complete process to learn one constraint in a CFN solver.

Example 5.12. *Following example 5.10. We had the constraint* $x_{1a} + x_{1b} + 3x_{2a} + 3x_{2b} + 4x_{3a} + 4x_{3b} + x_{4a} + x_{4b} = 9$. *Values* x_{2b} *and* x_{3b} *have been removed by domain propagation. We can proceed as explained in section 5.2.3 and derive constraint:* $memo_{x_1=b} : 7x_{1a} + x_{1b} + 3x_{2a} + 4x_{3a} + x_{4a} + x_{4b} \geq 9$.
If we branch on $x_1 = a$ *then we derive constraint* $memo_{x_1=a} : 3x_{1a} + 7x_{1b} + x_{4a} + x_{4b} + 4x_{3a} \geq 8$.
Applying the MR rule leads to:

$$\frac{7x_{1a} + x_{1b} + 3x_{2a} + 4x_{3a} + x_{4a} + x_{4b} \geq 9, \quad 3x_{1a} + 7x_{1b} + x_{4a} + x_{4b} + 4x_{3a} \geq 8}{3x_{1a} + x_{1b} + 3x_{2a} + 4x_{3a} + x_{4a} + x_{4b} \geq 8} \tag{5.50}$$

The Farkas constraint issued from $\mathbf{y}^P = \{\lambda^P_{123} = 1.5, \lambda^P_{134} = 1, \pi^P_1 = -1, \pi^P_3 = 0, \pi^P_3 = 0, \pi^P_4 = -1\}$ *is:*

$$2x_{1a} + 3x_{2a} + 4x_{3a} - x_{4b} \geq 4.5 \tag{5.51}$$

Summing constraints (5.50), (5.51) *gives:*

$$5x_{1a} + x_{1b} + 6x_{2a} + 8x_{3a} + x_{4a} \geq 12.5 \tag{5.52}$$

This is not a memo constraint of P *but it guarantees a bound of* 5.5 *(rounded to 6).* ■

5.4 Experimental results

We tried to implement this approach in TOULBAR2 the same solver used in previous chapters. For the moment, we haven't converged to satisfying results. First, we have not defined yet how to integrate all the features of TOULBAR2 into the learning mechanism. We also know that TOULBAR2 is very sensitive to the constraint ordering. It is possible that this learning mechanism won't work unless we define a clever constraint ordering heuristic. Therefore, it is difficult to state if the learning mechanism doesn't help the solver or if we haven't found the correct configuration (or if the implementation is buggy).
As an alternative and to begin with an easier environment, we decided to implement it in a very basic ILP Python solver. It follows the algorithm 7. We rely on CPLEX python API to solve the LP relaxation and return an optimal dual solution with associated reduced costs. In contrast with TOULBAR2, in this ILP framework, we can learn constraints with slack variables, floating points, and negative coefficients.

We apply the saturation rule before learning the constraints, this is possible because the MR rule derives $\geq$ constraints. The variable ordering is fixed, and the solver branches on the first decision variable with a fractional value. This defines the default version of MemoBound.

MemoBound is not cleverly optimized and can't solve a large range of problems. It supports only ILPs with Boolean decision variables and linear or pairwise constraints encoded in a dual encoding. All experiments are conducted on a single core of an Intel Xeon E5-2680 v3 at 2.50 GHz and 256 GB of RAM. We compare our approach to TOULBAR2, RoundingSat [Devriendt *et al.* 2021] v2 with a linear search (option –opt-mode=linear) and CPLEX 22.7. The non-optimized bound propagation and explanation phases are costly, especially in Python. Hence, we are not interested in the solving time and only measure the number of nodes visited by the different approaches. In particular, the most relevant data is the difference of nodes between our Python script with or without learning. The other solvers are only there for reference.

5.4.1 Knapsack problem

We tested the different approaches on 20 randomly generated knapsack problems with 100, 150, 200, 250, and 300 Boolean variables taken from [Chu & Stuckey 2013][1]. The capacity is randomly generated and can go up to 40000 (for instances with 300 variables. The weights are randomly generated between 1 and 300.
The Knapsack problem has a bounded number of subproblems and the same problem may be encountered several times in the search. This is a good opportunity to see if the learning scheme is able to learn this.
We compare how the number of search nodes increases along the size of the knapsack. If we disable the learning in the Python script, then it needs on average 1315 nodes to solve the knapsack with 100 variables and 7076 to solve the one with 300 variables. Hence, the number of nodes has been multiplied by approximately 5.4. If we use MemoBound, the number of nodes is multiplied by 3.14, therefore the increase is almost linear. We deduce that For RoundingSat the number of decisions is multiplied by 4.65. Finally, CPLEX is very efficient and solves 55 problems at root. The number of nodes between 100 and 300 variables is multiplied by 5.11 (5.55 to 28), but the number of developed nodes is too small to be relevant. Figure 5.3 shows how the number of nodes increases along the size of the knapsack.
Remark that dynamic programming is designed to memoize the optimal solution of the different sub-problems. It would be interesting to formally define how our learning approach is related to dynamic programming on the knapsack problem.

[1] https://people.eng.unimelb.edu.au/pstuckey/dom-jump for mzn format or https://forgemia.inra.fr/thomas.schiex/cost-function-library/-/tree/master/crafted/knapsack for the opb/wcsp/lp format

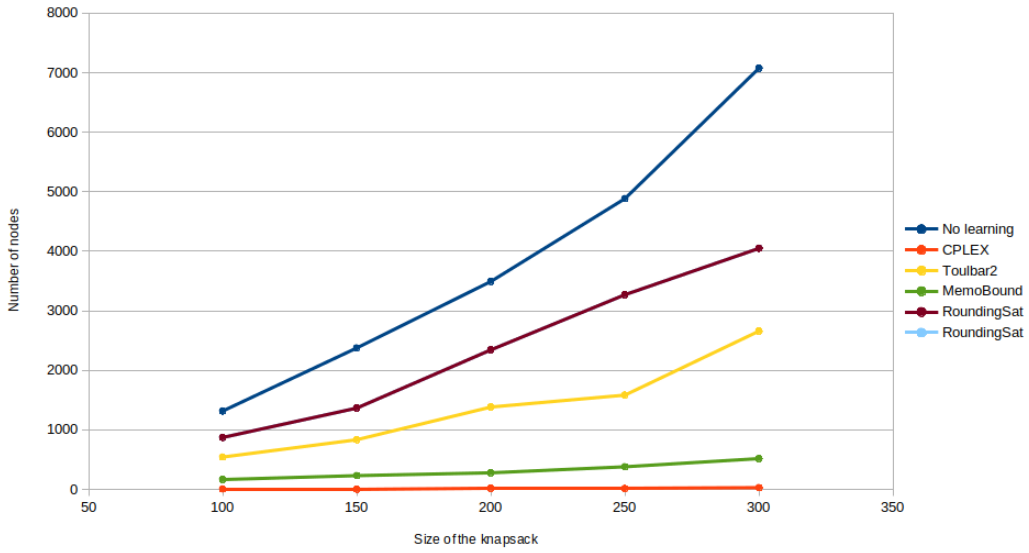


Figure 5.3: Comparison of the number of nodes to solve knapsack instances.

5.4.2 KPCG

We also tested our approach on the easiest instances of the KPCG benchmarks (see chapter 3). We present the results for different configurations in table 5.1. The letter 'R' corresponds to a random distribution of the weights and the letter 'C' to a distribution correlated to the profit. In superscript is noted a coefficient multiplying the capacity. The letter is followed by a number corresponding to the number of variables (120 or 250), and a decimal number giving the density of the conflict graph. For example, $R^3120 - 0.1$ are instances with a capacity multiplied by 3, random distribution, 120 variables, and a graph density of 0.1. Each class contains 10 instances. For all the instances we used a tuple encoding (this corresponds to the local polytope). On average, in those instances, MemoBound is able to divide by 4.16 (min 2.1, max 7.7) the number of nodes developed by the Python script. KPCG is a version of knapsack more challenging for the dynamic programming algorithm, but the learning scheme handles it without modification. This is seen because the relative benefit of MemoBound increases as we decrease the density, thus getting closer to pure knapsack. The cuts produced by CPLEX are very efficient and solve all the instances very quickly.

5.4.3 Kbtree problem

We tested our approach on randomly generated binary clique trees [De Givry *et al.* 2006]. Those problems contain only binary cost functions and can be decomposed into overlapping cliques. The variables appearing in two (or more) cliques are the *separator* variables. In those instances, the cliques follow a binary tree-like structure, with a bounded treewidth. Therefore, when a separator is fully assigned, then the prob-

	No learning	MemoBound	TOULBAR2	ROUNDINGSAT	CPLEX
$R^1120-0.1$	1030	226	213	2660	0.3
$R^1120-0.2$	1054	270	260	2558	0
$R^1120-0.3$	1004	298	270	2546	1.7
$R^1250-0.1$	2123	418	522	8985	0
$R^3120-0.1$	2272	472	476	5330	0.8
$R^3120-0.2$	3064	906	1026	6061	39.2
$R^3120-0.3$	3115	1487	1886	6650	66.8
$R^3250-0.1$	9423	1223	1679	17532	10.3
$C^1120-0.1$	10989	2646	1580	6064	0
$C^1120-0.2$	8672	2292	1151	8779	5.8
$C^1120-0.3$	6437	2156	1537	8043	73

Table 5.1: Number of nodes developed to solve different configurations of the KPCG problem.

	Variables	Constraints	No learning	MemoBound	TOULBAR2-BTD	ROUNDINGSAT	CPLEX
kb-7-2-3-2-60	44	190	7.79	5.9	5.58	4649	0 (573)
kb-7-2-4-2-60	92	406	43	17	16.33	64549	0 (1235)
kb-7-2-5-2-60	188	838	1240	262	37	-	0 (2556)
kb-8-2-3-2-60	51	246	13.89	8	12	7568	0 (758)
kb-8-2-4-2-60	107	526	128	40	40	153374	0 (1613)
kb-9-2-3-2-60	58	309	26	14	27	19153	0(979)
kb-9-2-4-2-60	122	661	457	156	95	-	0(2071)

Table 5.2: Average number of nodes developed to solve kbtree instances with different sizes. For CPLEX we give in parenthesis the number of iterations made by the simplex algorithm

lem is separated into two (or more) separate sub-problems. A solver can exploit that by storing for each instantiation of the separators the optimal solution of the created sub-problems. Thus, the solver will not solve two times the problem issued from the same assignment of the separators. This is the idea behind Back- track bounded by Tree Decomposition (BTD) [Jégou & Terrioux 2003]. It first computes a tree decomposition and uses it to make the search more efficient. TOULBAR2 has been augmented with BTD [De Givry *et al.* 2006] (option: -B=1 -O=-3 -Z -root=3). It also produces an ordering following the tree decomposition, which is given as input to MemoBound. ROUNDINGSAT was run with a linear search (option –opt-mode=linear). The parameters of the random generator are (w, s, h, d, t) where $w+1$ is the clique size, s the separator size, h the height of the clique tree, d the domain size, and t the constraint tightness (percentage of tuples with a non-zero cost (unit cost).). We generated 100 random instances with cliques of size 8,9,10 a separator size of 2, a height of 3,4,5, Boolean variables, and a tightness of 60. TOULBAR2-BTD solves directly those problems while other approaches solve the tuple encoding. Table 5.2 gives the size corresponding to the original instance, and the average number of nodes to solve the instances. The limit number of nodes was 1000000.

With the correct ordering, MemoBound seems to be able to learn the tree decomposition. For many instances it achieves similar speedups (in terms of nodes) than TOULBAR2-BTD without explicit knowledge of the tree decomposition (only the ordering). However, for some specific instances, the Python solver has abnormal behavior. For example, for kbree-7-2-5-2-60, the median number of nodes is 45 for MemoBound and 32 for TOULBAR2-BTD. Therefore, on the majority of instances TOULBAR2-BTD and MemoBound performs similarly. But the average number of nodes reported in table 5.2 is impacted by extreme values. In particular, there is one instance that is solved after 53000 nodes without learning and 6700 with learning while TOULBAR2-BTD needs only 45 nodes. We suspect that taking the first fractional value does not necessarily follow exactly the correct ordering and thus the search is negatively impacted. One way to improve this would be to dynamically detect the connected components and learn one constraint per component. Therefore, even if we do not follow the correct ordering, the learning mechanism will be able to derive one constraint per sub-problem.
ROUNDINGSAT has no information on the particular structure of the instances and therefore doesn't perform well. Finally, CPLEX is again very efficient and solves all the instances at the root node using the simplex algorithm and cuts.

5.5 Conclusion

We designed a novel conflict-free learning mechanism memorizing through linear constraints the lower bound of the encountered sub-problems. This approach can be embedded in different optimization paradigms such as ILP or CFN. The first results obtained on a basic Python script are encouraging. However, we haven't yet tested it in a fully functional solver. Maybe to obtain an effective learning mechanism it is necessary to explore different heuristics, such as constraint selection, constraint strengthening, or restarts. This conflict-free learning could also benefit from conflict-based learning. It would be also interesting to theoretically prove how the learning mechanism is related to dynamic programming and if it is able to simulate BTD.

Chapter 6

Conclusion

Contributions

In this thesis, we made several contributions to the state of the art in solving the WCSP problem. First, in chapter 3 we presented how to integrate linear constraints in a CFN. We defined how to represent them without introducing extra variables and designed a dedicated propagator. With this new feature, the WCSP solver toulbar2 is able to tackle new instances that were previously out of reach. In some families of instances, it dominates or is competitive with other state-of-the-art solvers. To improve the performance on some specific benchmarks we also extended the VAC algorithm to handle linear constraints and added the possibility of automatically detecting conflicts and embedding them in the linear constraints.
Secondly, we presented in chapter 4, Virtual Pair-Wise Consistency (VPWC) a newly defined soft local consistency. We show how to exploit VAC and dual encoding to efficiently enforce VPWC in preprocessing or during search. This has been tested on toulbar2 and shows good performance in the UAI 2022 competition [1].
Finally, we introduced a new conflict-free learning mechanism. This approach aims to learn the computed lower bounds based on dual proof constraints and reparametrization. We defined a recursive procedure learning a constraint at each node from the constraints computed at the child nodes. We show this method can be implemented in MILP or CFN frameworks. A basic Python script that implements the learning mechanism demonstrates encouraging preliminary results.

Perspectives

Most exciting future work concerns the learning mechanism. We only implemented the approach in a basic unoptimized Python script but it remains to see how it actually performs in an efficient fully functional solver (Toulbar2, SCIP or RoundingSAt). We also didn't explore the learning-related heuristics such as constraint selection, constraint strengthening, or restarts. This work could also be enhanced by an automatic detection of the connected components. It would make it possible to learn one constraint per component and obtain finer grains lower bounds. Finally, a good contribution would be to have theoretical proofs to relate the learning scheme to other approaches such as dynamic programming or BTD.

[1] https://uaicompetition.github.io/uci-2022/

Another direction could be to strengthen the lower bounds computed during the search. This could be done by finding a better ordering of the linear constraints. Maybe a machine learning-based approach could help to understand what are the key elements to find the best ordering. Otherwise, we could look at heuristics playing with the dual encoding. For example, it would be interesting to dualize as few cost functions as possible while keeping a good lower bound. This could be done dynamically during the search. Similarly, what we did in the UAI 2022 competition, we could try to define a heuristic cleverly selecting which empty cost function to add.

Chapter 7

Résumé en Français

Contents

7.1 Chapitre 1: Introduction

L'optimisation joue un rôle fondamental dans notre expérience quotidienne. Nous sommes sans cesse confrontés à de nouveaux défis, allant de la tournée de véhicules, à la planification, aux systèmes de recommandation, ou à la conception de protéines pour n'en citer que quelques-uns. Aborder ces problèmes est difficile car ils peuvent impliquer un grand nombre de variables avec des interactions complexes. La plupart d'entre eux sont hors de portée de la capacité cognitive humaine, nous nous appuyons donc sur l'amélioration incessante des algorithmes pour trouver la meilleure, ou du moins une bonne solution.
Il existe différentes approches de modélisation et de résolution, chacune étant conçue pour résoudre une gamme différente de problèmes. Parmi les deux paradigmes les plus courants, nous trouvons d'une part la *Programmation par Contraintes* (CP), un cadre basé sur la logique et l'inférence. D'autre part, la Programmation Linéaire en Nombres Entiers (ILP), spécialisée dans les interactions linéaires et faisant appel à des mathématiques avancées. ILP et CP offrent des approches complémentaires pour relever les défis complexes de l'optimisation. Cependant, il est fréquent qu'une fois qu'une technique montre son efficacité dans un paradigme, les chercheurs essaient de l'adapter à l'autre paradigme. Un exemple notable est l'Apprentissage de Clauses Basé sur les Conflits (CDCL). Il a été initialement défini pour le problème de satisfaction booléenne (SAT), pour lequel il est devenu un pilier des solveurs SAT modernes. Depuis lors, des approches dérivées ont été développées pour MaxSAT, l'optimisation pseudo-booléenne, le Problème de Satisfaction de Contraintes (CSP) ou ILP. Cette stratégie vise à apprendre des contraintes à partir des échecs rencontrés pendant la recherche. Les contraintes apprises amélioreront le reste de la

recherche en empêchant le solveur de refaire la même erreur.
Les *Modèles Graphiques* (GM) utilisent des graphes pour encoder les relations complexes entre les variables de décision, où les noeuds représentent des variables et les (hyper)arêtes représentent des dépendances ou des corrélations entre elles. GM offre un cadre flexible pour modéliser différents systèmes. Par exemple, les Réseaux de Fonction de Coût (CFN) sont des modèles graphiques non orientés impliquant des *fonctions de coût locales*. La tâche de trouver l'affectation minimisant la somme de toutes les fonctions de coût locales est connue sous le nom de Problème de Satisfaction de Contraintes Pondérées (WCSP). Ce problème se pose dans divers domaines tels que l'analyse d'images [Savchynskyy 2019] ou la bioinformatique [Allouche *et al.* 2014a]. La résolution de WCSP repose sur une *recherche arborescente* et la *propagation de contraintes*. Le solveur recherche une solution optimale en divisant l'espace de recherche en sous-problèmes plus petits. Les techniques de propagation de contraintes sont utilisées pour fournir une borne inférieure des sous-problèmes rencontrés. Obtenir de bonnes bornes inférieures est crucial pour éviter d'explorer des régions peu prometteuses.
Dans cette thèse, nous nous intéressons à diversifier la gamme d'instances modélisables et résolubles par un solveur WCSP. Nous montrons d'abord comment intégrer des contraintes linéaires dans un Réseau de Fonction de Coût (CFN). Ces contraintes sont expressives, compactes et sont au coeur de solveurs ILP très efficaces. Ainsi, traiter les contraintes linéaires dans les solveurs WCSP peut considérablement élargir leur utilisation pratique. Deuxièmement, nous définissons la Cohérence Virtual Pair-Wise, une nouvelle cohérence locale souple dérivant des bornes bonnes inférieurs. Enfin, guidés par le succès des méthodes d'apprentissage basées sur les conflits dans plusieurs domaines (comme SAT, l'optimisation pseudo-booléenne ou ILP), nous concevons un nouveau mécanisme d'apprentissage *sans conflit*. Il vise à mémoriser à travers une contrainte linéaire les bornes inférieures des sous-problèmes rencontrés. Si ce sous-problème apparaît une deuxième fois dans la recherche, propager la contrainte précédemment apprise aidera à obtenir une bonne borne inférieure. Nous montrons comment un tel mécanisme peut être intégré dans des solveurs MILP classiques, avant d'étendre cela aux solveurs WCSP.

Ce manuscrit est organisé comme suit:

- Le Chapitre 2 introduit le concept général derrière plusieurs paradigmes utilisés pour résoudre des problèmes combinatoires. Cela inclut la Programmation Linéaire en Nombres Entiers, la Programmation par Contraintes, les Réseaux de Fonction de Coût, SAT et l'Optimisation Pseudo-Booléenne.

- Le Chapitre 3 montre comment les contraintes linéaires peuvent être encodées et propagées dans un CFN. Nous proposons également un algorithme étendant la cohérence locale souple Virtual Arc [Cooper *et al.* 2010] (VAC) pour traiter les contraintes linéaires. Enfin, nous donnons les résultats expérimentaux obtenus sur des instances impliquant des contraintes linéaires. La plupart de la contribution de ce chapitre a été publiée à CPAIOR 2022 [Montalbano *et al.* 2022].

- Le Chapitre 4 exploite l'encodage dual d'un CFN et VAC pour imposer une cohérence locale douce nouvellement définie : cohérence Virtual Pair-Wise. Les résultats expérimentaux montrent les avantages d'une telle stratégie sur plusieurs benchmarks. Cette contribution est un travail collaboratif avec Tomas Werner de l'Université Technique de Prague et a été publiée à CPAIOR 2023 [Montalbano *et al.* 2023].

- Le Chapitre 5 introduit un mécanisme d'apprentissage sans conflit pour mémoriser les bornes inférieurs. Nous montrons comment intégrer cette approche dans des solveurs MILP classiques, avant d'étendre cela aux solveurs WCSP. Nous donnons quelques résultats préliminaires obtenus avec cette approche. Nous préparons actuellement une soumission présentant cette contribution.

7.2 Chapitre 2: Optimisation dans les modèles graphiques

Ce chapitre définit les notions importantes abordées dans le manuscrit. Il se concentre sur les modèles graphiques (GM) et les différents outils utilisés pour résoudre les problèmes d'optimisations liés au GM. D'un coté on trouve la Programmation Linéaire en Nombre Entier (ILP) qui se base sur des contraintes linéaires et les notions de problème primal/dual, et de l'autre la Programmation par Contrainte (CP) plus axé sur des algorithmes de cohérences locale et les contraintes globales. Les cadres ILP et CP offrent des approches complémentaires pour relever des défis d'optimisation complexes, l'ILP excellant dans les problèmes où les relations linéaires prédominent, tandis que la CP offre une flexibilité dans la modélisation de diverses contraintes et espaces de décision discrets. Il est important de noter que la situation est bien plus compliquée lorsqu'on discute des performances de résolution réelles. De nombreux autres paramètres peuvent impacter l'efficacité de la résolution, et l'ILP/CP peuvent se comporter mal ou bien sur des problèmes où nous nous attendions à l'opposé.
S'ensuit une présentation des problèmes de satisfaction de contrainte pondérées (WCSP), l'étude de ces derniers étant au coeur de la thèse. Un WCSP est un problème d'optimisation définit par des variables discrètes et des fonctions de coûts. Chaque fonction de coût est définit par un *scope* i.e un sous ensemble de variable, un coût i.e un entier positif, est associé à chaque tuple réalisable dans le scope. L'objectif est de trouver l'affectation minimisant la somme de toutes les fonctions de coût locales. Une des approches pour résoudre un WCSP s'appuie sur un algorithme de branch and bound et des algorithmes de cohérence locale souple (SLC). L'objectif des SLC est de produire un minorant en considérant des interactions locales entre les fonctions de coût. Différentes SLC sont présentées dans le manuscrit, et une hiérarchie se basant sur la qualité du minorant est donnée. Une des cohérences les plus fortes est Virtual Arc Consistency (VAC), celle ci s'obtient en appliquant la cohérence d'arc sur un CSP particulier appelé Bool(P) obtenu à partir du WCSP P. Enfin le chapitre se conclu par une introduction des paradigmes SAT et d'optimisation Pseudo Booléenne qui ont la particularité d'employer des

mécanismes d'apprentissage par conflit pendant la recherche (CDCL par exemple).

7.3 Chapitre 3: Modèles graphiques et contraintes linéaire

Les contraintes linéaires sont des contraintes expressives et compactes fournissant un outil puissant pour la modélisation et la résolution d'une large gamme de problèmes d'optimisation, notamment en informatique, en recherche opérationnelle et en intelligence artificielle [Boros & Hammer 2002]. Par défaut, dans les WCSP les contraintes sont représentées en extension, c'est à dire que tout les tuples et leur coût associé sont explicitement donnés. Malheureusement, représenter les contraintes linéaires de cette manière introduit un nombre de tuples exponentiel dans l'arité de la contrainte. Une possibilité serait de représenter les contraintes linéaires en intention et d'intégrer un solveur LP pour les gérer. Cependant, résoudre un LP à chaque noeud pourrait être très coûteux comme l'ont montré Hurley et al [Hurley *et al.* 2016] puisque les solveurs ILP peuvent être significativement plus lents que les solveurs WCSP dédiés. De plus, le solveur LP peut être sujet à une instabilité numérique, il n'est pas souhaitable d'introduire une instabilité numérique dans des solveurs exacts qui fonctionnent exclusivement avec des entiers pour garantir l'exactitude de leur solution.

Dans ce chapitre, nous présentons une méthode pour représenter dans un CFN toutes les contraintes linéaires sans introduire de variables supplémentaires. Nous étendons les algorithmes de cohérence locale souple pour propager ces contraintes. Plus précisément, nous nous concentrons sur les contraintes PB $\sum_{i\in\boldsymbol{S}} w_i x_i \geq C$ ainsi qu'une partition de ses variables en ensembles $A_1, \ldots, A_k \in \boldsymbol{S}$, $\bigcup_{j=1}^{k} A_j = \boldsymbol{S}$. Pour chaque partition A_j, une contrainte impose qu'une seule variable peut prendre la valeur un : $\sum_{x_j\in A_j} x_j = 1$. Une telle contrainte est connue sous le nom de contrainte *Exactly One* (EO), tandis que $\sum_{x_j\in A_j} x_j \leq 1$ est une contrainte *At Most One* (AMO). Cette formulation est plus générale qu'une seule contrainte PB. En particulier, elle nous permet d'étendre les contraintes PB aux variables à plusieurs valeurs. Soit $\boldsymbol{S}$ un ensemble de variables WCSP avec des domaines arbitraires et w_{iv} le poids associé à la valeur $v \in \boldsymbol{D}_i$. Pour chaque variable dans $\boldsymbol{S}$, nous utilisons des variables 0/1 x_{iv} qui prennent la valeur 1 si $x_i = v$ et 0 si $x_i \neq v$. La contrainte $\sum_{i\in\boldsymbol{S}, v\in\boldsymbol{D}_i} w_{iv} x_{iv} \geq C$ correspond au schéma décrit ci-dessus, avec des partitions $A_i = x_{iv} \mid v \in \boldsymbol{D}_i$. Sans perte de généralité, nous supposons que les poids et la capacité sont tous positifs.

Enfin, cette formulation admet le cas où il existe une contrainte AMO sur certaines partitions : nous ajoutons une autre variable 0/1 dans chaque telle partition et lui donnons un poids 0, de sorte que cette partition a maintenant une contrainte EO. Notez que toutes ces variables 0/1 sont utilisées pour représenter la contrainte mais n'apparaissent pas à l'extérieur de celle-ci, en particulier, nous ne branchons jamais sur ces variables.

Example 7.1. *Supposons que nous ayons un WCSP avec 2 variables* $X = x_1, x_2$ *avec des domaines* $\boldsymbol{D}_1 = 1, 2, 3$ *et* $\boldsymbol{D}_2 = 1, 2$*, nous pouvons exprimer une contrainte*

PB en utilisant des variables booléennes $x_{11}, x_{12}, x_{13}, x_{21}, x_{22}$.

$$\begin{aligned} 4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} &\geq 40 \\ x_{11} + x_{12} + x_{13} &= 1 \\ x_{21} + x_{22} &= 1 \\ x_{11}, x_{12}, x_{13}, x_{21}, x_{22} &\in 0, 1 \end{aligned}$$

Une solution à ce problème pourrait être $x_{13} = 1, x_{21} = 1$ *et correspond à* $x_1 = 3, x_2 = 1$ *dans le WCSP.* ■

La suite du chapitre montre qu'une contrainte linéaire peut être propagée à partir de la solution dual d'un problème de sac à dos à choix multiple. Cette propagation mène à une application d'une version faible de la cohérence locale souple Full zero-inverse. S'ensuit une discussion sur les différentes heuristiques liées à l'inclusion des contraintes linéaires (choix des variables de branchement, ordre de propagation...). Une des faiblesses de notre approche est que l'algorithme produit fondamentalement une solution sous-optimale au programme linéaire. Pour remédier à ce problème, nous avons étendu l'algorithme VAC pour gérer les contraintes linéaires et ajouté la possibilité d'associer des affectations conflictuelles avec des contraintes linéaires.

Enfin, l'approche a été implémenté dans le solveur Toulbar2 et testé sur plusieurs jeux d'instances. Les résultats montrent que ce travail offre une plus grande flexibilité de modélisation et permet un solveur WCSP tel que TOULBAR2 de résoudre davantage de problèmes, tels que des problèmes de conception de protéines computationnelles avec garantie de diversité ou des problèmes de sac à dos avec des graphes de conflit. Parfois TOULBAR2 est même compétitif avec les autres solveurs PBO ou ILP.

7.4 Chapitre 4: Cohérence Pair-Wise virutelle

Dans le chapitre 2 plusieurs algorithmes de cohérence locale ont été présentés. En particulier, nous avons décrit la Virtual Arc Consistency (VAC), qui est atteinte lorsque la fermeture de cohérence d'arc de Bool(P) n'est pas vide. Dans ce chapitre, nous définissons la Virtual Pair-Wise Consistency (VPWC), qui peut être obtenue en imposant la cohérence Pair-Wise sur Bool(P).

En programmation par contraintes, la cohérence Pair-Wise (PWC) est une cohérence définie pour les CSP non binaires. Contrairement à la cohérence basée sur les arcs, elle a la possibilité d'exploiter les interactions entre les paires de contraintes. Par conséquent, elle permet de filtrer plus efficacement les valeurs mais est plus coûteuse à imposer. Elle a été comparée à AC généralisée pour la résolution de CSP non binaires donnés en extension [Samaras & Stergiou 2005a, Schneider & Choueiry 2018a, Wang & Yap 2019, Wang & Yap 2021]. Elle montre de bonnes performances dans certains benchmarks mais n'est pas préférée par défaut. Les algorithmes récents imposant la PWC sur un CSP reposent sur un encodage binaire, nous explorons également une idée similaire dans le cadre des CFN. En effet,

ce chapitre présente *l'encodage dual* qui permet d'exprimer un CSP non-binaire en CSP binaire. Les algorithmes de cohérence par arc filtre plus efficacement les valeurs sur ce CSP binaire que sur le CSP original. Cependant, des variables avec de larges domaines et un grand nombres de contraintes peuvent être introduites dans le dual, ralentissant ainsi les algorithmes de filtrage. Cette encodage dual peut être étendu au WCSP et de la même manière les algorithmes de cohérence locale souple produisent des meilleurs minorants sur ce problème dual. En particulier, VAC n'a pas été implémentée pour les WCSP non-binaires, de ce fait, utiliser l'encodage dual permet d'appliquer VAC même sur un problème non-binaire. Il est montré qu'appliquer VAC sur l'encodage dual est équivalent à appliquer VPWC sur le problème original.

Un autre avantage de l'encodage dual réside dans sa flexibilité, l'utilisateur peut choisir quelle partie du problème originale il est souhaitable de dualiser (encodage dual partiel). De plus, pendant la recherche, après avoir appliquer VAC (ou tout autre algorithme) sur le problème dual, il est possible de le *dé-dualiser* et de poursuivre la recherche sur un problème non-binaire mais avec un premier bon minorant. Des résultats expérimentaux sur la compétition UAI 2022 montre que cette approche est pertinente et compétitive.

7.5 Chapitre 5: Apprentissage sans-conflit

L'idée derrière les mécanismes d'apprentissage est d'utiliser les informations fournies par un solveur pendant la recherche pour apprendre une nouvelle contrainte qui aidera pour le reste de la recherche. Cela a montré son efficacité dans différents paradigmes tels que CDCL pour SAT [Marques-Silva & Sakallah 1999], l'enregistrement de NoGood pour CSP [Dechter 1990, Katsirelos & Bacchus 2005], la résolution pseudo-Booléenne pour les solveurs PB [Chai & Kuehlmann 2003, Dixon & Ginsberg 2002, Elffers & Nordström 2018, Le Berre & Parrain 2010, Sheini & Sakallah 2006, Devriendt *et al.* 2021], MaxCDCL pour MaxSAT [Li *et al.* 2021], l'analyse de conflit MIP pour MIP [Achterberg 2007]. Ces méthodes d'apprentissage déclenchées lorsqu'un solveur rencontre un problème infaisable sont qualifiées *basées sur les conflits*. Les contraintes apprises de cette manière sont spécifiquement conçues pour filtrer davantage de valeurs incohérentes. Une approche plus récente est l'apprentissage sans conflit [Witzig 2022], qui repose sur le lemme de Farkas [Farkas 1902] pour produire des *contraintes de Farkas* à partir d'une solution duale.

Definition 7.1 (Contrainte de Farkas)**.** *Étant donné un problème LP* $\min c^{\mathbf{T}}x | Ax = b, x \in \mathbb{Z}$ *et une solution duale* $\mathbf{y}$*, la contrainte de Farkas est définie comme suit :*

$$\mathbf{y}^{\mathbf{T}}Ax = \mathbf{y}^{\mathbf{T}}b \tag{7.1}$$

Les contraintes de Farkas étaient généralement utilisées comme preuve d'infaisabilité lorsque le LP était déclaré infaisable. Witzig montre que l'apprentissage d'une contrainte de Farkas sur des noeuds réalisables peut être utile si la contrainte est modifiée selon les informations obtenues en approfondissant l'arbre de recherche [Witzig

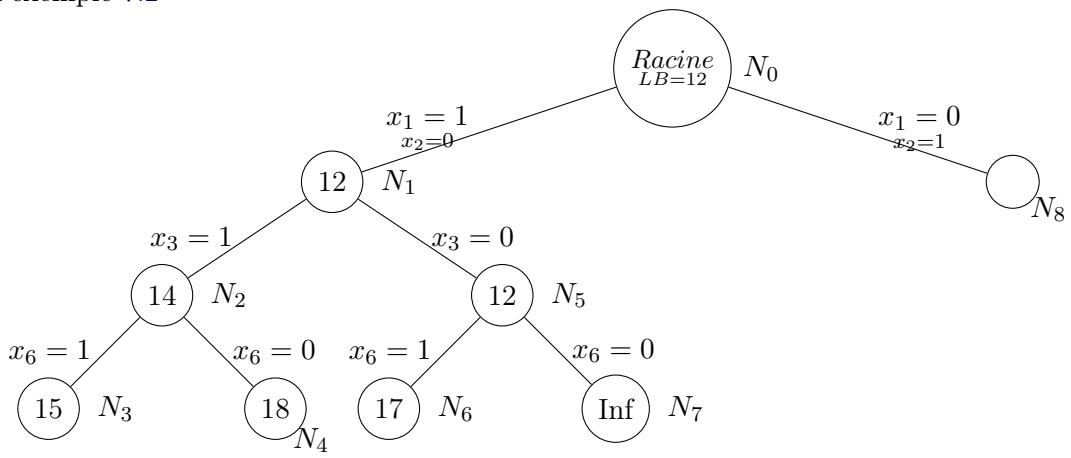
& Berthold 2020, Witzig 2022]. Cependant, ces contraintes sont dérivées d'une agrégation de contraintes primales LP, de plus, elles n'exploitent aucune connaissance sur l'intégralité des variables. Par conséquent, ces contraintes n'aideront pas un solveur LP à dériver directement de meilleures bornes à l'avenir, mais peuvent aider à éliminer les valeurs incohérentes via la propagation de domaine.

Même si ces apprentissages basés ou non sur les conflits sont efficaces dans leur cadre, ils ne répondent pas à nos besoins dans les CFN. En effet, les solveurs WCSP ont la particularité de reparamétrer le problème pendant la recherche. Si la borne inférieure dépasse la borne supérieure, il n'est pas possible de pointer directement un ensemble de fonctions de coût et de déclencher une procédure similaire à l'analyse de conflits. De plus, les stratégies de filtrage sont limitées dans les CFN car les incohérences dures ne sont pas au coeur des fonctions de coût (sauf sur certains benchmarks spécifiques). Dans les solveurs WCSP, la meilleure façon de réduire l'espace de recherche est de produire de bonnes bornes. Alors que les mécanismes d'apprentissages précédemment définis étaient exclusivement conçus pour filtrer davantage de valeurs, il semble plus intéressant dans ces conditions d'apprendre des contraintes aidant à dériver de meilleures bornes inférieures.

Notre approche est basée sur les contraintes de Farkas. Par elle-même, nous savons qu'une contrainte de Farkas n'aidera pas un solveur LP à dériver de meilleures bornes, mais nous spécifions comment les combiner pour obtenir des contraintes qui sont valides, logiquement redondantes, mais non redondantes par rapport à la relaxation linéaire. En particulier, de manière similaire à l'approche de Witzig, nous montrons qu'une contrainte de Farkas calculée à un noeud interne peut devenir utile si elle est fusionnée avec des contraintes de Farkas dérivées plus profondément dans l'arbre de recherche. Ces nouvelles contraintes produiront de meilleures bornes et pourront filtrer des valeurs.

Dans la suite du chapitre les auteurs définissent un mécanisme d'apprentissage dans le cadre MILP où la recherche de solution optimale se fait grâce à algorithme de branch and bound et la relaxation LP est résolue à chaque noeud. L'objectif est d'être en mesure d'apprendre une contrainte linéaire par noeud (sous-problème) de sorte que si nous résolvons à nouveau la relaxation LP correspondante à ce noeud, alors, la contrainte linéaire apprise à elle seule impose une borne inférieure égale à la meilleure borne inférieure connue pour ce sous-problème. Si nous apprenons de telles contraintes, alors si une autre affectation mène au même sous-problème (ou à un sur-ensemble), nous pouvons déduire directement la meilleure borne inférieure connue sans recherche. Nous pouvons également espérer que si nous rencontrons un problème légèrement différent, alors la contrainte apprise aide toujours à déduire une borne utile. L'exemple 7.2 donne un exemple concret de ce que nous voulons réaliser.

Figure 7.1: Début de l'arbre de recherche pour trouver la solution optimale de l'exemple 7.2



Example 7.2. *Soit P un problème MILP :*

$$\begin{aligned}
&\min 3x_1 + 4x_2 + 6x_3 + 8x_4 + x_5 + 6x_6 \\
&\textit{s.t.} \\
&2x_1 + 2x_2 + 3x_3 + 4x_4 + x_5 + 6x_6 - s = 10 \\
&x_1 + x_2 = 1 \\
&x_i + x_{i0} = 1 \\
&x_i \in \{0, 1\} \quad \forall i \in [1, 6] \\
&x_{i0} \geq 0 \quad \forall i \in [1, 6] \\
&s \geq 0
\end{aligned}$$

La figure 7.1 montre le début de la recherche pour trouver la solution optimale. À chaque noeud, un solveur LP donne la solution relaxée optimale. Nous pouvons voir au noeud N_2 que la borne inférieure est 14, puis au noeud N_3 et N_4 la solution relaxée optimale est 15 et 18. Nous pouvons en déduire qu'une meilleure borne inférieure pour le noeud N_2 est 15. Dans notre procédure d'apprentissage, nous voulons apprendre une contrainte telle qu'avec cette nouvelle contrainte, la solution relaxée optimale au noeud N_2 n'est pas inférieur à 15. De même, au noeud N_5 et N_1, nous voulons apprendre une contrainte justifiant une borne inférieure respectivement de 17 et 15.
De telles contraintes pourraient améliorer la recherche future, par exemple si en visitant le sous-arbre issu de N_1, nous déduisons la contrainte $6x_{60} + x_{50} + 2s \geq 3$ et l'ajoutons au LP. Ensuite, lorsque le solveur visite le noeud N_8, il trouve une solution relaxée optimale de 13 sans la contrainte supplémentaire, tandis qu'il trouve 16 avec la contrainte supplémentaire et la recherche est terminée. Dans la suite, nous détaillons comment nous pouvons dériver de telles contraintes. ■

La stratégie d'apprentissage apprend des contraintes avec des propriétés par-

ticulières, elles sont nommées *contraintes de garantie*, une contrainte de garantie est associée à une borne γ, si elle est ajoutée aux contraintes d'un MILP P, alors elle assure que le minorant de la relaxation linéaire est au moins γ. Il existe une classe de contrainte qui constitue des contraintes de garantie, ce sont les mémos contraintes. Les auteurs les caractérisent comme des contraintes dont les coefficients linéaires sont plus petits que les coefficients de la fonction objective. En particulier, les contraintes de Farkas sont des mémos contraintes. Ainsi, le mécanisme d'apprentissage est défini pour apprendre des mémos contraintes à chaque noeud garantissant la meilleure borne inférieure connue pour ce sous-problème. Pour ce faire, une approche récursive est définie où à chaque noeud une mémo contrainte est apprise en utilisant les mémos contraintes des deux noeuds enfants. Pour cela, les auteurs proposent une version alternative de la Fusion Resolution [Buss & Nordström 2021] permettant de combiner les mémos contraintes des noeuds enfants pour obtenir une mémo contrainte au noeud parent. Aussi, pour apprendre des contraintes plus précises, il est montré comment intégrer dans un solveur ILP la notion de reparamétrage utilisée dans les CFN. Pour compléter cette approche, les auteurs présentent une manière d'inclure des mécanismes de suppression de valeurs dans ce mécanisme d'apprentissage. Enfin, ils discutent des avantages et inconvenants à inclure ce processus dans un solveur WCSP.
Cette approche a été implémentée dans un petit solveur en python servant de preuve de concept. Les résultats sur les différents benchmarks sont prometteurs mais une implémentation du processus d'apprentissage dans un solveur complet et la définition d'heuristiques sont nécessaires pour conclure sur l'efficacité de cette approche.

7.6 Chapitre 5: Conclusion

Contributions

Dans cette thèse, nous avons apporté plusieurs contributions à l'état de l'art dans la résolution du problème WCSP. Tout d'abord, dans le chapitre 3, nous avons présenté comment intégrer des contraintes linéaires dans un CFN. Nous avons défini comment les représenter sans introduire de variables supplémentaires et avons conçu un propagateur dédié. Avec cette nouvelle fonctionnalité, le solveur WCSP TOULBAR2 est capable de traiter de nouvelles instances qui étaient auparavant hors de portée. Dans certaines familles d'instances, il domine ou est compétitif avec d'autres solveurs de pointe. Pour améliorer les performances sur certains benchmarks spécifiques, nous avons également étendu l'algorithme VAC pour gérer les contraintes linéaires et ajouté la possibilité de détecter automatiquement les conflits et de les intégrer dans les contraintes linéaires.

Deuxièmement, nous avons présenté dans le chapitre 4 la cohérence Virtual Pair Wise (VPWC) une cohérence locale souple nouvellement définie. Nous montrons comment exploiter VAC et l'encodage dual pour appliquer efficacement VPWC en prétraitement ou pendant la recherche. Cela a été testé sur TOULBAR2 et montre

de bonnes performances dans la compétition UAI 2022.

Enfin, nous avons introduit un nouveau mécanisme d'apprentissage sans conflit. Cette approche vise à apprendre les bornes inférieures calculées sur la base des contraintes de Farkas, et du reparamétrage. Nous avons défini une procédure récursive qui apprend une contrainte à chaque noeud à partir des contraintes calculées au niveau des noeuds enfants. Nous montrons que cette méthode peut être implémentée dans des cadres MILP ou CFN. Un script Python de base qui met en oeuvre le mécanisme d'apprentissage démontre des résultats préliminaires encourageants.

Perspectives

Le futur travail le plus excitant concerne le mécanisme d'apprentissage. Nous n'avons implémenté l'approche que dans un script Python de base non optimisé, mais il reste à voir comment elle se comporte réellement dans un solveur entièrement fonctionnel et efficace (Toulbar2, SCIP ou RoundingSAt). Nous n'avons pas non plus exploré les heuristiques liées à l'apprentissage telles que la sélection de contraintes, le renforcement des contraintes ou les redémarrages. Ce travail pourrait également être amélioré par une détection automatique des composantes connectées. Cela permettrait d'apprendre une contrainte par composant et d'obtenir des bornes inférieures plus fines. Enfin, une bonne contribution serait d'avoir des preuves théoriques pour relier le schéma d'apprentissage à d'autres approches telles que la programmation dynamique ou BTD.

Une autre direction pourrait être de renforcer les bornes inférieures calculées pendant la recherche. Cela pourrait être fait en trouvant un meilleur ordonnancement des contraintes linéaires. Peut-être qu'une approche basée sur l'apprentissage automatique pourrait aider à comprendre quels sont les éléments clés pour trouver le meilleur ordonnancement. Sinon, nous pourrions examiner les heuristiques jouant avec l'encodage dual. Par exemple, il serait intéressant de dualiser le moins de fonctions de coût possible tout en conservant une bonne borne inférieure. Cela pourrait être fait dynamiquement pendant la recherche. De même, ce que nous avons fait dans la compétition UAI 2022, nous pourrions essayer de définir une heuristique sélectionnant intelligemment quelle fonction de coût vide ajouter.

Bibliography

[Abío & Stuckey 2014] Ignasi Abío and Peter J Stuckey. *Encoding linear constraints into SAT*. In International Conference on Principles and Practice of Constraint Programming, pages 75–91. Springer, 2014. (Cited on page 29.)

[Achterberg 2007] Tobias Achterberg. *Constraint integer programming*. PhD thesis, 2007. (Cited on pages 36, 95, 99 and 134.)

[Allouche *et al.* 2012] David Allouche, Christian Bessière, Patrice Boizumault, Simon De Givry, Patricia Gutierrez, Samir Loudni, Jean-Philippe Metivier and Thomas Schiex. *Filtering decomposable global cost functions*. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 26, pages 407–413, 2012. (Cited on page 24.)

[Allouche *et al.* 2014a] David Allouche, Isabelle André, Sophie Barbe, Jessica Davies, Simon de Givry, George Katsirelos, Barry O'Sullivan, Steve Prestwich, Thomas Schiex and Seydou Traoré. *Computational protein design as an optimization problem*. Artificial Intelligence, vol. 212, pages 59–79, 2014. (Cited on pages 1 and 130.)

[Allouche *et al.* 2014b] David Allouche, Jessica Davies, Simon de Givry, George Katsirelos, Thomas Schiex, Seydou Traoré, Isabelle André, Sophie Barbe, Steve Prestwich and Barry O'Sullivan. *Computational Protein Design as an Optimization Problem*. Artificial Intelligence, vol. 212, pages 59–79, 2014. (Cited on page 79.)

[Allouche *et al.* 2016] David Allouche, Christian Bessiere, Patrice Boizumault, Simon De Givry, Patricia Gutierrez, Jimmy HM Lee, Ka Lun Leung, Samir Loudni, Jean-Philippe Métivier, Thomas Schiex*et al. Tractability-preserving transformations of global cost functions*. Artificial Intelligence, vol. 238, pages 166–189, 2016. (Cited on page 24.)

[Allouche *et al.* 2021] David Allouche, Sophie Barbe, Simon de Givry, George Katsirelos, Yahia Lebbah, Samir Loudni, Abdelkader Ouali, Thomas Schiex, David Simoncini and Matthias Zytnicki. Operations research and simulation in healthcare, chapter Cost Function Networks to Solve Large Computational Protein Design Problems. Springer, 2021. (Cited on pages 79 and 80.)

[Ansótegui *et al.* 2019] Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z Salamon, Josep Suy and Mateu Villaret. *Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints*. In International Conference on Principles and Practice of Constraint Programming, pages 20–36. Springer, 2019. (Cited on page 70.)

[Audemard *et al.* 2008] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Said Jabbour and Lakhdar Sais. *A generalized framework for conflict analysis.* In International conference on theory and applications of satisfiability testing, pages 21–27. Springer, 2008. (Cited on page 31.)

[Bacchus & Grove 2013] Fahiem Bacchus and Adam J Grove. *Graphical models for preference and utility.* arXiv preprint arXiv:1302.4928, 2013. (Cited on page 3.)

[Balas *et al.* 1996] Egon Balas, Sebastian Ceria, Gérard Cornuéjols and N Natraj. *Gomory cuts revisited.* Operations Research Letters, vol. 19, no. 1, pages 1–9, 1996. (Cited on page 7.)

[Bessiere *et al.* 2004] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich and Toby Walsh. *The tractability of global constraints.* In Principles and Practice of Constraint Programming–CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10, pages 716–720. Springer, 2004. (Cited on page 12.)

[Bettinelli *et al.* 2017] Andrea Bettinelli, Valentina Cacchiani and Enrico Malaguti. *A branch-and-bound algorithm for the knapsack problem with conflict graph.* INFORMS Journal on Computing, vol. 29, no. 3, pages 457–473, 2017. (Cited on pages 55 and 78.)

[Beuvin *et al.* 2021] François Beuvin, Simon de Givry, Thomas Schiex, Sébastien Verel and David Simoncini. *Iterated local search with partition crossover for computational protein design.* Proteins: Structure, Function, and Bioinformatics, 2021. (Cited on page 91.)

[Biere *et al.* 2021] Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh, editors. Handbook of satisfiability - second edition, volume 336 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2021. (Cited on pages 3 and 31.)

[Boros & Hammer 2002] Endre Boros and Peter L Hammer. *Pseudo-boolean optimization.* Discrete applied mathematics, vol. 123, no. 1-3, pages 155–225, 2002. (Cited on pages 39 and 132.)

[Boussemart *et al.* 2004a] F Boussemart, F Hemery, C Lecoutre and L Sais. *Boosting systematic search by weighting constraints.* In ECAI, volume 16, page 146, 2004. (Cited on page 90.)

[Boussemart *et al.* 2004b] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre and Lakhdar Sais. *Boosting systematic search by weighting constraints.* In ECAI, volume 16, page 146, 2004. (Cited on pages 58 and 71.)

[Boussemart *et al.* 2016] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard and Cédric Piette. *XCSP3: an integrated format for benchmarking*

combinatorial constrained problems. arXiv preprint arXiv:1611.03398, 2016. (Cited on page 75.)

[Buss & Nordström 2021] Sam Buss and Jakob Nordström. *Proof Complexity and SAT Solving.* Handbook of Satisfiability, vol. 336, pages 233–350, 2021. (Cited on pages 100 and 137.)

[Cappanera & Trubian 2005] Paola Cappanera and Marco Trubian. *A local-search-based heuristic for the demand-constrained multidimensional knapsack problem.* INFORMS Journal on Computing, vol. 17, no. 1, pages 82–98, 2005. (Cited on page 55.)

[Chai & Kuehlmann 2003] Donald Chai and Andreas Kuehlmann. *A fast pseudo-boolean constraint solver.* In Proceedings of the 40th annual Design Automation Conference, pages 830–835, 2003. (Cited on pages 32, 34, 95 and 134.)

[Chu & Stuckey 2013] Geoffrey Chu and Peter J Stuckey. *Dominance driven search.* In International Conference on Principles and Practice of Constraint Programming, pages 217–229. Springer, 2013. (Cited on page 122.)

[Coniglio *et al.* 2021] Stefano Coniglio, Fabio Furini and Pablo San Segundo. *A new combinatorial branch-and-bound algorithm for the Knapsack Problem with Conflicts.* European Journal of Operational Research, vol. 289, no. 2, pages 435–455, 2021. (Cited on page 78.)

[Cooper & Schiex 2004] Martin Cooper and Thomas Schiex. *Arc consistency for soft constraints.* Artificial Intelligence, vol. 154, no. 1-2, pages 199–227, 2004. (Cited on pages 16 and 19.)

[Cooper *et al.* 2010] Martin C Cooper, Simon de Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki and Tomas Werner. *Soft arc consistency revisited.* Artificial Intelligence, vol. 174, no. 7-8, pages 449–478, 2010. (Cited on pages 2, 16, 17, 22, 24, 25, 59, 60 and 130.)

[Cooper *et al.* 2020] Martin Cooper, Simon de Givry and Thomas Schiex. *Graphical models: queries, complexity, algorithms.* Leibniz International Proceedings in Informatics, vol. 154, pages 4–1, 2020. (Cited on page 3.)

[Davis *et al.* 1962] Martin Davis, George Logemann and Donald Loveland. *A machine program for theorem-proving.* Communications of the ACM, vol. 5, no. 7, pages 394–397, 1962. (Cited on page 29.)

[de Givry & Katsirelos 2017] Simon de Givry and George Katsirelos. *Clique cuts in weighted constraint satisfaction.* In Proc. of CP-17, pages 97–113, Melbourne, Australia, 2017. (Cited on page 70.)

[de Givry *et al.* 2005] Simon de Givry, Federico Heras, Matthias Zytnicki and Javier Larrosa. *Existential arc consistency: Getting closer to full arc consistency*

in weighted CSPs. In Proc. of IJCAI-05, pages 84–89, Edinburgh, Scotland, 2005. (Cited on pages 17, 20, 21, 24, 71 and 77.)

[De Givry *et al.* 2006] Simon De Givry, Thomas Schiex and Gerard Verfaillie. *Exploiting tree decomposition and soft local consistency in weighted CSP*. In Twenty-first National Conference on Artificial Intelligence-AAAI 2006, volume 1, pages 1115–p, 2006. (Cited on pages 123 and 124.)

[Dechter 1990] Rina Dechter. *Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition*. Artificial Intelligence, vol. 41, no. 3, pages 273–312, 1990. (Cited on pages 35, 95 and 134.)

[Dechter 1999] Rina Dechter. *Bucket Elimination: A Unifying Framework for Reasoning*. Artificial Intelligence, vol. 113, no. 1–2, pages 41–85, 1999. (Cited on page 91.)

[Dehani 2014] Djamel-Eddine Dehani. *La substituabilité et la cohérence de tuples pour les réseaux de contraintes pondérées*. PhD thesis, Artois, 2014. (Cited on page 22.)

[Demirovic *et al.* 2018] E Demirovic, G Chu and P J. Stuckey. *Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers*. In Proc. of CP-18, pages 99–108, Lille, France, 2018. (Cited on page 90.)

[Devriendt *et al.* 2021] Jo Devriendt, Ambros Gleixner and Jakob Nordström. *Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search*. Constraints, vol. 26, no. 1-4, pages 26–55, 2021. (Cited on pages 73, 79, 95, 122 and 134.)

[Dixon & Ginsberg 2002] Heidi E Dixon and Matthew L Ginsberg. *Inference methods for a pseudo-boolean satisfiability solver*. In AAAI/IAAI, pages 635–640, 2002. (Cited on pages 32, 95 and 134.)

[Dixon *et al.* 2004] Heidi E Dixon, Matthew L Ginsberg and Andrew J Parkes. *Generalizing Boolean satisfiability I: Background and survey of existing work*. Journal of Artificial Intelligence Research, vol. 21, pages 193–243, 2004. (Cited on page 32.)

[Dubois *et al.* 1993] Didier Dubois, Hélene Fargier and Henri Prade. *The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction*. In [Proceedings 1993] Second IEEE International Conference on Fuzzy Systems, pages 1131–1136. IEEE, 1993. (Cited on page 3.)

[Dyer 1984] Martin E. Dyer. *An $O(n)$ algorithm for the multiple-choice knapsack linear program*. Mathematical programming, vol. 29, no. 1, pages 57–63, 1984. (Cited on page 43.)

[Eén & Sörensson 2006] Niklas Eén and Niklas Sörensson. *Translating pseudo-boolean constraints into SAT.* Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, no. 1-4, pages 1–26, 2006. (Cited on page 32.)

[Elffers & Nordström 2018] Jan Elffers and Jakob Nordström. *Divide and Conquer: Towards Faster Pseudo-Boolean Solving.* In IJCAI, volume 18, pages 1291–1299, 2018. (Cited on pages 32, 35, 95 and 134.)

[Farkas 1902] Julius Farkas. *Theorie der einfachen Ungleichungen.* Journal für die reine und angewandte Mathematik (Crelles Journal), vol. 1902, no. 124, pages 1–27, 1902. (Cited on pages 36, 95 and 134.)

[Favier *et al.* 2011] A Favier, S de Givry, A Legarra and T Schiex. *Pairwise decomposition for combinatorial optimization in graphical models.* In IJCAI11P, IJCAI11L, 2011. Video demonstration at http://www.inra.fr/mia/T/degivry/Favier11.mov. (Cited on page 91.)

[Gebser *et al.* 2012] Martin Gebser, Benjamin Kaufmann and Torsten Schaub. *Conflict-driven answer set solving: From theory to practice.* Artificial Intelligence, vol. 187, pages 52–89, 2012. (Cited on page 32.)

[Gomory 1960] Ralph Gomory. *An algorithm for the mixed integer problem.* Technical report, RAND CORP SANTA MONICA CA, 1960. (Cited on pages 7 and 32.)

[Hebrard & Siala 2017] Emmanuel Hebrard and Mohamed Siala. *Explanation-based weighted degree.* In International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pages 167–175. Springer, 2017. (Cited on pages 58, 60, 62 and 65.)

[Hurley *et al.* 2016] Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki and Simon de Givry. *Multi-language evaluation of exact solvers in graphical model discrete optimization.* Constraints, vol. 21, no. 3, pages 413–434, 2016. (Cited on pages 17, 23, 39, 78, 90 and 132.)

[Janssen *et al.* 1989] Philippe Janssen, Philippe Jégou, Bernard Nouguier and Marie-Catherine Vilarem. *A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation.* In IEEE International Workshop on Tools for Artificial Intelligence, pages 420–421. IEEE Computer Society, 1989. (Cited on pages 86, 87 and 88.)

[Jégou & Terrioux 2003] Philippe Jégou and Cyril Terrioux. *Hybrid backtracking bounded by tree-decomposition of constraint networks.* Artificial Intelligence, vol. 146, no. 1, pages 43–75, 2003. (Cited on page 124.)

[Johnson & Padberg 1981] Ellis L Johnson and Manfred W Padberg. *A note of the knapsack problem with special ordered sets.* Operations Research Letters, vol. 1, no. 1, pages 18–22, 1981. (Cited on page 43.)

[Katsirelos & Bacchus 2005] George Katsirelos and Fahiem Bacchus. *Generalized nogoods in CSPs.* In AAAI, volume 5, pages 390–396, 2005. (Cited on pages 35, 95 and 134.)

[Kindermann & Snell 1980] Ross Kindermann and J Laurie Snell. Markov random fields and their applications, volume 1. American Mathematical Society, 1980. (Cited on page 3.)

[Koller & Friedman 2009] Daphne Koller and Nir Friedman. Probabilistic graphical models: principles and techniques. MIT press, 2009. (Cited on page 3.)

[Kolmogorov 2006] V Kolmogorov. *Convergent tree-reweighted message passing for energy minimization.* IEEE transactions on pattern analysis and machine intelligence, vol. 28, no. 10, pages 1568–1583, 2006. (Cited on page 17.)

[Komodakis *et al.* 2010] Nikos Komodakis, Nikos Paragios and Georgios Tziritas. *MRF energy minimization and beyond via dual decomposition.* IEEE transactions on pattern analysis and machine intelligence, vol. 33, no. 3, pages 531–552, 2010. (Cited on page 17.)

[Kratica *et al.* 2001] J. Kratica, D. Tosic, V. Filipovic and I. Ljubic. *Solving the Simple Plant Location Problems by Genetic Algorithm.* RAIRO Operations Research, vol. 35, pages 127–142, 2001. (Cited on pages 55 and 77.)

[Larrosa & Heras 2005] J. Larrosa and F. Heras. *Resolution in Max-SAT and its relation to local consistency in weighted CSPs.* In ijcai05p, pages 193–198, ijcai05l, 2005. (Cited on page 90.)

[Larrosa & Schiex 2003] Javier Larrosa and Thomas Schiex. *In the quest of the best form of local consistency for weighted CSP.* In IJCAI, volume 3, pages 239–244, 2003. (Cited on pages 20, 21 and 24.)

[Larrosa 2000] J. Larrosa. *Boosting search with variable elimination.* In Principles and Practice of Constraint Programming - CP 2000, volume 1894 of *LNCS*, pages 291–305, Singapore, September 2000. (Cited on page 91.)

[Larrosa 2002a] J. Larrosa. *On Arc and Node Consistency in weighted CSP.* In Proc. AAAI'02, pages 48–53, Edmondton, (CA), 2002. (Cited on page 17.)

[Larrosa 2002b] Javier Larrosa. *Node and arc consistency in weighted CSP.* In AAAI/IAAI, pages 48–53, 2002. (Cited on pages 19 and 24.)

[Le Berre & Parrain 2010] Daniel Le Berre and Anne Parrain. *The Sat4j library, release 2.2.* Journal on Satisfiability, Boolean Modeling and Computation, vol. 7, no. 2-3, pages 59–64, 2010. (Cited on pages 32, 95 and 134.)

[Lecoutre *et al.* 2009] C. Lecoutre, L Saïs, S. Tabary and V. Vidal. *Reasoning from last conflict(s) in constraint programming.* ai, vol. 173, pages 1592,1614, 2009. (Cited on pages 71 and 90.)

[Lee & Leung 2009] Jimmy HM Lee and Ka Lun Leung. *Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction.* In Twenty-First International Joint Conference on Artificial Intelligence, 2009. (Cited on page 24.)

[Lee & Leung 2012] Jimmy Ho-Man Lee and Ka Lun Leung. *Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction.* Journal of Artificial Intelligence Research, vol. 43, pages 257–292, 2012. (Cited on page 24.)

[Li *et al.* 2021] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet and Kun He. *Combining clause learning and branch and bound for MaxSAT.* In 27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. (Cited on pages 36, 95 and 134.)

[Little *et al.* 1963] John DC Little, Katta G Murty, Dura W Sweeney and Caroline Karel. *An algorithm for the traveling salesman problem.* Operations research, vol. 11, no. 6, pages 972–989, 1963. (Cited on page 6.)

[Maathuis *et al.* 2018] Marloes Maathuis, Mathias Drton, Steffen Lauritzen and Martin Wainwright. Handbook of graphical models. CRC Press, 2018. (Cited on page 3.)

[Marchand *et al.* 2002] Hugues Marchand, Alexander Martin, Robert Weismantel and Laurence Wolsey. *Cutting planes in integer and mixed integer programming.* Discrete Applied Mathematics, vol. 123, no. 1-3, pages 397–446, 2002. (Cited on page 7.)

[Marques-Silva & Sakallah 1999] Joao P Marques-Silva and Karem A Sakallah. *GRASP: A search algorithm for propositional satisfiability.* IEEE Transactions on Computers, vol. 48, no. 5, pages 506–521, 1999. (Cited on pages 30, 31, 95 and 134.)

[Martello & Toth 1987] Silvano Martello and Paolo Toth. *Algorithms for knapsack problems.* North-Holland Mathematics Studies, vol. 132, pages 213–257, 1987. (Cited on page 55.)

[Martins *et al.* 2014] Ruben Martins, Vasco Manquinho and Inês Lynce. *Open-WBO: A modular MaxSAT solver.* In Theory and Applications of Satisfiability Testing–SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 17, pages 438–445. Springer, 2014. (Cited on page 32.)

[Montalbano *et al.* 2022] Pierre Montalbano, Simon de Givry and George Katsirelos. *Multiple-choice knapsack constraint in graphical models.* In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pages 282–299. Springer, 2022. (Cited on pages 2, 24 and 130.)

[Montalbano *et al.* 2023] Pierre Montalbano, David Allouche, Simon De Givry, George Katsirelos and Tomáš Werner. *Virtual Pairwise Consistency in Cost Function Networks.* In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pages 417–426. Springer, 2023. (Cited on pages 2 and 131.)

[Moskewicz *et al.* 2001] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. *Chaff: Engineering an efficient SAT solver.* In Proceedings of the 38th annual Design Automation Conference, pages 530–535, 2001. (Cited on page 31.)

[Murty 1983] Katta G Murty. Linear programming. Springer, 1983. (Cited on page 6.)

[Neveu *et al.* 2004] Bertrand Neveu, Gilles Trombettoni and Fred Glover. *ID Walk: A Candidate List Strategy with a Simple Diversification Device.* In Proc. of CP, pages 423–437, 2004. (Cited on page 91.)

[Nguyen *et al.* 2017] Hiep Nguyen, Christian Bessiere, Simon de Givry and Thomas Schiex. *Triangle-based Consistencies for Cost Function Networks.* Constraints, vol. 22, no. 2, pages 230–264, 2017. (Cited on page 91.)

[Otten *et al.* 2012] Lars Otten, Alexander Ihler, Kalev Kask and Rina Dechter. *Winning the PASCAL 2011 MAP Challenge with Enhanced AND/OR Branch-and-Bound.* In DISCML'12 Workshop, at NIPS'12, Lake Tahoe, NV, 2012. (Cited on page 90.)

[Ouali *et al.* 2020] Abdelkader Ouali, David Allouche, Simon de Givry, Samir Loudni, Yahia Lebbah, Lakhdar Loukil and Patrice Boizumault. *Variable Neighborhood Search for Graphical Model Energy Minimization.* Artificial Intelligence, vol. 278, no. 103194, page 22p., 2020. (Cited on page 90.)

[Pisinger & Toth 1998] David Pisinger and Paolo Toth. *Knapsack problems.* In Handbook of combinatorial optimization, pages 299–428. Springer, 1998. (Cited on pages 5 and 43.)

[Pisinger 2005] David Pisinger. *Where are the hard knapsack problems?* Computers & Operations Research, vol. 32, no. 9, pages 2271–2284, 2005. (Cited on page 57.)

[Prusa & Werner 2013] Daniel Prusa and Tomas Werner. *Universality of the local marginal polytope.* In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1738–1743, 2013. (Cited on page 17.)

[Rossi *et al.* 2006] Francesca Rossi, Peter Van Beek and Toby Walsh. Handbook of constraint programming. Elsevier, 2006. (Cited on pages 3, 9, 10, 11, 12 and 18.)

[Ruffini *et al.* 2021] Manon Ruffini, Jelena Vucinic, Simon de Givry, George Katsirelos, Sophie Barbe and Thomas Schiex. *Guaranteed Diversity and Optimality in Cost Function Network Based Computational Protein Design Methods.* Algorithms, vol. 4, no. 6:168, 2021. (Cited on page 79.)

[Sakai & Nabeshima 2015] Masahiko Sakai and Hidetomo Nabeshima. *Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers.* IEICE TRANSACTIONS on Information and Systems, vol. 98, no. 6, pages 1121–1127, 2015. (Cited on pages 32 and 73.)

[Samaras & Stergiou 2005a] Nikolaos Samaras and Kostas Stergiou. *Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results.* Journal of Artificial Intelligence Research, vol. 24, pages 641–684, 2005. (Cited on pages 85 and 133.)

[Samaras & Stergiou 2005b] Nikolaos Samaras and Kostas Stergiou. *Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results.* Journal of Artificial Intelligence Research, vol. 24, pages 641–684, 2005. (Cited on pages 86 and 87.)

[Sánchez *et al.* 2008] M. Sánchez, S. de Givry and T. Schiex. *Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques.* Constraints, vol. 13, no. 1, pages 130–154, 2008. (Cited on page 90.)

[Savchynskyy 2019] Bogdan Savchynskyy. *Discrete graphical models—an optimization perspective.* Foundations and Trends® in Computer Graphics and Vision, vol. 11, no. 3-4, pages 160–429, 2019. (Cited on pages 1, 3, 17 and 130.)

[Schiex 2000] T. Schiex. *Arc consistency for soft constraints.* In Proc. of CP-00, pages 411–424, Singapore, 2000. (Cited on page 17.)

[Schneider & Choueiry 2018a] Anthony Schneider and Berthe Y Choueiry. *PW-CT: Extending compact-table to enforce pairwise consistency on table constraints.* In International Conference on Principles and Practice of Constraint Programming, pages 345–361. Springer, 2018. (Cited on pages 85, 93 and 133.)

[Schneider & Choueiry 2018b] Anthony Schneider and Berthe Y Choueiry. *PW-CT: Extending compact-table to enforce pairwise consistency on table constraints.* In International Conference on Principles and Practice of Constraint Programming, pages 345–361. Springer, 2018. (Cited on pages 86, 87 and 91.)

[Schrijver 1998] Alexander Schrijver. Theory of linear and integer programming. John Wiley & Sons, 1998. (Cited on pages 5 and 6.)

[Sheini & Sakallah 2006] Hossein M Sheini and Karem A Sakallah. *Pueblo: A hybrid pseudo-boolean SAT solver.* Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, no. 1-4, pages 165–189, 2006. (Cited on pages 32, 95 and 134.)

[Sontag *et al.* 2008] D Sontag, T Meltzer, A Globerson, Y Weiss and T Jaakkola. *Tightening LP Relaxations for MAP using Message-Passing.* In Proc. of UAI, pages 503–510, Helsinki, Finland, 2008. (Cited on page 17.)

[Sontag *et al.* 2012] D Sontag, D Choe and Y Li. *Efficiently Searching for Frustrated Cycles in MAP Inference.* In Proc. of UAI, pages 795–804, Catalina Island, CA, USA, 2012. (Cited on page 17.)

[Spielman & Teng 2004] Daniel A Spielman and Shang-Hua Teng. *Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time.* Journal of the ACM (JACM), vol. 51, no. 3, pages 385–463, 2004. (Cited on page 6.)

[Tourani *et al.* 2020] Siddharth Tourani, Alexander Shekhovtsov, Carsten Rother and Bogdan Savchynskyy. *Taxonomy of dual block-coordinate ascent methods for discrete energy minimization.* In Proc. of AISTATS-20, pages 2775–2785, Palermo, Sicily, Italy, 2020. (Cited on page 17.)

[Traoré *et al.* 2013] Seydou Traoré, David Allouche, Isabelle André, Simon de Givry, George Katsirelos, Thomas Schiex and Sophie Barbe. *A new framework for computational protein design through cost function network optimization.* Bioinformatics, vol. 29, no. 17, pages 2129–2136, 2013. (Cited on page 80.)

[Trösser *et al.* 2020] Fulya Trösser, Simon de Givry and George Katsirelos. *Relaxation-Aware Heuristics for Exact Optimization in Graphical Models.* In CPAIOR20P, pages 475–491, CPAIOR20L, 2020. (Cited on pages 18, 90, 91 and 93.)

[Vaidya 1989] P.M. Vaidya. *Speeding-up linear programming using fast matrix multiplication.* In 30th Annual Symposium on Foundations of Computer Science, pages 332–337, 1989. (Cited on page 17.)

[Van Hentenryck *et al.* 1992] Pascal Van Hentenryck, Helmut Simonis and Mehmet Dincbas. *Constraint satisfaction using constraint logic programming.* Artificial intelligence, vol. 58, no. 1-3, pages 113–159, 1992. (Cited on page 12.)

[Wang & Yap 2019] Ruiwei Wang and Roland HC Yap. *Arc consistency revisited.* In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pages 599–615. Springer, 2019. (Cited on pages 85, 93 and 133.)

[Wang & Yap 2021] Ruiwei Wang and Roland HC Yap. *Bipartite encoding: a new binary encoding for solving non-binary CSPs.* In Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, pages 1184–1191, 2021. (Cited on pages 85, 93 and 133.)

[Werner 2007] Tomas Werner. *A Linear Programming Approach to Max-sum Problem: A Review.* IEEE Trans. on Pattern Recognition and Machine Intelligence, vol. 29, no. 7, pages 1165–1179, July 2007. (Cited on page 17.)

[Werner 2010] Tomáš Werner. *Revisiting the Linear Programming Relaxation Approach to Gibbs Energy Minimization and Weighted Constraint Satisfaction.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 32, no. 8, pages 1474–1488, August 2010. (Cited on page 89.)

[Witzig & Berthold 2020] Jakob Witzig and Timo Berthold. *Conflict-Free Learning for Mixed Integer Programming.* In Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17, pages 521–530. Springer, 2020. (Cited on pages 96 and 135.)

[Witzig 2022] Jakob Witzig. Infeasibility analysis for mip. Technische Universitaet Berlin (Germany), 2022. (Cited on pages 36, 95, 96, 134 and 135.)

[Yamada *et al.* 2002] Takeo Yamada, Seija Kataoka and Kohtaro Watanabe. *Heuristic and exact algorithms for the disjunctively constrained knapsack problem.* Information Processing Society of Japan Journal, vol. 43, no. 9, 2002. (Cited on page 5.)

[Yap *et al.* 2020] Roland HC Yap, Wei Xia and Ruiwei Wang. *Generalized arc consistency algorithms for table constraints: A summary of algorithmic ideas.* In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 13590–13597, 2020. (Cited on page 12.)

[Zemel 1984] Eitan Zemel. *An $O(n)$ algorithm for the linear multiple choice knapsack problem and related problems.* Information Processing Letters, vol. 18, no. 3, pages 123–128, 1984. (Cited on page 43.)

[Zytnicki *et al.* 2009] M. Zytnicki, C. Gaspin, S. de Givry and T. Schiex. *Bounds Arc Consistency for Weighted CSPs.* Journal of Artificial Intelligence Research, vol. 35, pages 593–621, 2009. (Cited on pages 17, 19 and 24.)