Méthodes exactes pour l'apprentissage de la structure d'un réseau bayésien et les réseaux de fonctions de coûts

Fulya Trösser

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *31/03/2022* **par :**

Fulya TRÖSSER

Exact Methods for Bayesian Network Structure Learning and Cost Function Networks

---

JURY

MARTIN COOPER	Professeur titulaire	Président du Jury
JAMES CUSSENS	Professeur titulaire	Rapporteur
PETER VAN BEEK	Professeur titulaire	Rapporteur
CHRISTINE SOLNON	Professeur titulaire	Examinatrice
SIMON DE GIVRY	Chargé de recherche	Directeur
GEORGE KATSIRELOS	Chargé de recherche	Co-directeur

École doctorale et spécialité :
MITT : Domaine STIC : Intelligence Artificielle

Unité de Recherche :
Mathématiques et Informatique Appliqués de Toulouse (INRAE)

Directeurs de Thèse :
George KATSIRELOS et *Simon de GIVRY*

Rapporteurs :
James CUSSENS et *Peter van BEEK*

To Elsa and Philipp

"Don't judge each day by the harvest you reap but by the seeds that you plant."

Robert Louis Stevenson (1850-1894)

This thesis is the product of the seeds I have planted over three and a half years. It is finally time to harvest.

Acknowledgments

This thesis has been a great experience that allowed me to indulge my passion for applied mathematics and computer science, as well to see tangible results for a real-life problem whose solution is of great benefit to society.

I am deeply grateful to Simon and George who made this thesis possible for me. They have encouraged me to push my own limits and see that I could do things that I never thought I could. Their intelligence and diligence have been a great inspiration, and still they were incredibly humble, so much that they came up with the idea of calling our solver after my daughter's name, Elsa, for which I cannot thank them enough.

Next, I would like to express my thanks to the jury members, Martin Cooper, James Cussens, Christine Solnon and Peter van Beek, for granting me such a pleasant ending to my PhD, with their insightful feedback on my manuscript as well as their kind attitude throughout the defence.

Many thanks to Agence Nationale de la Recherche, to the DE-MO-GRAPH project, and to Philippe Jégou and Thomas Schiex for funding this PhD, and their benevolence within the context of Covid-19 and my maternity leave.

It is also thanks to all of my awesome colleagues at INRAE and their smiling faces that everything went so smoothly throughout my PhD. A big shout out to Thomas, Matthias, Sylvain and Fabienne for being available whenever I needed help.

Now I would like to briefly go a little bit back in time to mention all my professors who shone a light on the path before me. My admiration for Ali Tamer Ünal and Zeki Caner Taşkın, my BSc. professors from Boğaziçi University, was what made me decide to pursue a doctorate. Everything I learnt from them has been a sound basis for all my studies later on. During my MSc., I was extremely lucky to have Hadrien Cambazard, Gülgün Alpan, and Pierre Lemaire as my professors. Hadrien Cambazard was the one who made me realise the power of applied mathematics: one can not only solve the problems of the industry, but even schedule astronomical observations on a telescope in the optimal way :).

While we are still back in time, big hugs to Burak and İpek for always being there to listen to me and their willingness to help.

Special thanks to Sébastien Fréguin, as well as his fabulous colleagues from MAM La Chouette Violette, Magalie and Charlène, for the great care and attention they gave my daughter Elsa ever since she joined them. I cannot find words to express my gratitude. It was thanks to them that I had my peace of mind every day and was able to focus on my work.

A major highlight during my PhD years certainly was becoming friends with Emilie and Benjamin, and their lovely daughters Sophie and Pauline. They were my natural antidepressants and a true family to us!

Big thanks to the Ural and Trösser families for their endless support and encouragement. Although there is a big physical distance between all of us, it is great that you still make me feel loved and cherished!

Last but not least, I would like to (try to) put into words my infinite love for my own little family: Philipp and Elsa. It is true that life is not always super easy for us, and that we have our ups and downs, but one thing for sure, I would never exchange being "down" with the two of you for being "up" with anyone else in this world.

Please allow me to finish with the words of Isaac Newton (1642–1727), which summarises well all I wanted to say:
“If I have seen further, it is by standing on the shoulders of Giants.”

Thanks to all Giants in my life.

Contents

CHAPTER 1

Introduction

All real-life problems that we encounter involve a combination of several difficulties: too many variables, too many constraints, an arbitrary level of uncertainty, a limited amount of time to find a solution and the list goes on. If we try to solve such problems with a paper, a pen and a good reasoning, it will not take us much time to see that it is far from doable. On the other hand, if we have a computer do the exact same procedure that we were initially planning to do by hand, we will very likely be disappointed to see that the resulting solution quality is rather low. When we think we finally found the right approach that is supposed to give us the optimal solution, the computer might have different plans and take ages to come up with a single feasible solution.

Discrete optimisation is a sub-field of applied mathematics which deals with problems where the decision variables take a finite set of (nearly always integer) values. Most decisions in real life, especially in industry, involve discrete decisions such as employee scheduling [Ağralı *et al.* 2017], sales and operations planning [Taşkın *et al.* 2015], and production scheduling [Güngör *et al.* 2018]. *Integer Programming* (IP) and *Constraint Programming* (CP) are two of the most common paradigms to tackle such problems.

Another challenge often encountered in real-life problems, apart from the integrality requirement, is that there is an abundance of variables. The issue at this point is that, whenever we need to define some function over this many variables, the description of that function grows exponentially with the number of variables. This makes the *reasoning* task necessary for solving discrete optimisation problems much harder.

Graphical models are a type of data structure and very useful tools for reasoning. Undirected graphical models like *Cost Function Networks*, aka *Weighted Constraint Satisfaction Problems* (WCSP), and *Markov Random Fields* (MRF) can be used to give a factorised representation of a function, in which vertices of a graph represent variables of the function and (hyper)edges represent factors. The factors can be, for example, *cost functions*, in which case the graphical model represents a factorisation of a cost function, or *local probability tables*, in which case the model represents a non-normalised joint probability distribution [Koller & Friedman 2009].

The two models, WCSP and MRF, are equivalent under a $-\log$ transformation, hence the NP-complete cost minimisation query in WCSP is equivalent to the maximum a posteriori (MAP) assignment query in MRF. This optimisation problem has applications in many areas, such as image analysis [Li 2009, Geman & Graf-

figne 1986, Li 1994], speech recognition [Gravier *et al.* 1999, Gravier *et al.* 2000], and bioinformatics [Allouche *et al.* 2014].

Sometimes, optimisation is a (vital) means rather than an end. This happens within the context of Bayesian networks.

Bayesian networks (BN) [Pearl 1988] are probabilistic graphical models that represent joint probability distributions over random variables. Often the dependence relationships between the random variables is not known, and determining this relationship analytically is far from trivial. In this case, the usual way to go is to take a set of joint observations of the variables as an input and find a BN structure, i.e. the underlying directed acyclic graph (DAG), where each node corresponds to a random variable and each directed edge encodes direct conditional dependence, and which best explains the given data. This is known as the problem of *Bayesian network structure learning (BNSL) from discrete data*. The difficulty here is that the size of the search space of DAGs is exponential in the number of random variables. As a result, an intelligent approach needs to be developed.

There have been numerous approximate [Behjati & Beigy 2020, Dai *et al.* 2020, Liu *et al.* 2017, Scutari *et al.* 2019] and exact methods developed for BNSL in the last decades. Exact approaches include dynamic programming [Silander & Myllymäki 2006, Malone *et al.* 2011a], heuristic search [Yuan & Malone 2013, Fan & Yuan 2015], constraint programming [Van Beek & Hoffmann 2015], maximum satisfiability [Berg *et al.* 2014, Cussens 2008], breadth-first branch-and-bound search [Campos & Ji 2011, Fan *et al.* 2014, Malone *et al.* 2011b] and integer linear programming [Bartlett & Cussens 2017, Cussens *et al.* 2017].

One way to learn BNs from data is the score-and-search method [Heckerman *et al.* 1995], which is the method we mainly use in this work. This method uses a precalculated set of scores to formulate the BNSL as an optimisation problem. For each variable, there is a predefined set of candidate parent sets with their associated scores. With this information, it is possible to define a decomposable cost function. We then try to find a solution that minimises this cost function, which is a sum of local scores, and that returns a network that is acyclic.

BNSL is in essence an assignment problem: to each random variable we assign a parent set. If there was no acyclicity constraint, BNSL would be as simple as assigning to each random variable its lowest-cost candidate parent set. However, acyclicity is mandatory, hence the difficulty, otherwise this dissertation would not exist.

The main focus of this dissertation is on developing a set of methods in order to contribute towards handling BNSL instances with no bound on the arity of the parent sets. Nearly all the algorithms we designed for this purpose and presented herein are inspired by the well-known Virtual Arc Consistency (VAC) algorithm that is widely used in the WCSP framework. We see that our solver ELSA (the second Elsa that came to life during the course of this doctoral degree) using all these algorithms improve on the state-of-the-art. As a bonus, we also present some heuristics that lead to a more efficient use of VAC within branch-and-bound solvers for WCSPs.

This manuscript is organised as follows:

– Chapter 2 gives the necessary background on the relevant notions such as constraint programming, integer linear programming, cost function networks, Bayesian networks and search strategies, as well as the related existing work.

– Chapter 3 provides a more efficient inference algorithm for the acyclicity constraint, specifically a generalised arc consistency algorithm with significantly improved asymptotic complexity and practical performance.

– Chapter 4 presents an algorithm that computes an approximation to a well known lower bound for BNSL, which has significantly better practical performance. More precisely, we give a polynomial-time algorithm for detecting a subset of all violated cluster cuts provably increasing the lower bound and a greedy algorithm to increase the lower bound using the clusters cuts found.

– Chapter 5 demonstrates how we can exploit the structure of the parent set domains by using binary decision trees to efficiently maintain and manipulate them.

– Chapter 6 introduces two heuristics that allow to benefit from the information about the linear relaxation of the problem that the VAC algorithm produces.

– Chapter 7 concludes and points out future research directions.

CHAPTER 2

Background

Contents

This chapter introduces the basic concepts of constraint programming, integer linear programming, weighted constraint satisfaction problems, and Bayesian networks that the reader will encounter throughout this thesis.

2.1 Constraint Programming

Constraint programming (CP) [Rossi *et al.* 2006] is a paradigm aimed at solving combinatorial problems. It has successful applications in numerous domains including scheduling [Baptiste *et al.* 2001, Beck *et al.* 2011, Trilling *et al.* 2006], planning [Van Beek & Chen 1999], vehicle routing [Shaw 1998, De Backer *et al.* 2000, Guimarans *et al.* 2011], and bioinformatics [Barahona & Krippahl 2008, Backofen *et al.* 1999]. The first attempts to represent constrained problems as networks

go back to 1970's [Montanari 1974], as well as solving strategies [Freuder 1978, Freuder 1982, Borning 1981] and constraint-based solvers [Lauriere 1978].

In the CP framework, we have decision variables, which are to be assigned a value from a finite domain, and a set of constraints that impose restrictions on what values these decision variables can take. The task is to assign each decision variable to a value in a way that all constraints are satisfied. The decision variables together with the values they can take and the constraints imposed on them correspond to a problem, to which one can look for a feasible solution, in which case it is a decision problem that we call a *Constraint Satisfaction Problem*, or the best solution, in which case it is an optimisation problem that we call a *Constraint Optimisation Problem*.

2.1.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a triplet $\langle V, D, C \rangle$, where

- $V = \{v_1, \ldots, v_n\}$ is a set of n decision variables.
- $D = \{D(v_1), \ldots, D(v_n)\}$ is a set of functions, mapping decision variables to their domains.

  The size of the domain $D(v)$ of a decision variable v is denoted as d_v. Each domain $D(v) = \{S_1, \ldots, S_{d_v}\}$ consists of values that v can possibly take. The maximum domain size in a CSP is denoted as $d = \max_{v \in V} d_v$.
- C is a set of constraints.

Let $J \subseteq V$ be a subset of the variables, and $\ell(J)$ denote the Cartesian product $\prod_{v_i \in J} D(v_i)$ of the domains of the variables in J. An *assignment* $T \in \ell(J)$ is a mapping from each $v \in J$ to a value $S \in D(v)$. A *complete assignment* is an assignment to V. If an assignment maps v to $S \in D(v)$, we denote it by (v, S). A *constraint* $c \in C$ is a pair $\langle V_s, F \rangle$, where $V_s \subseteq V$ is the *scope* of the constraint and F is a *predicate* over $\prod_{v \in V_s} D(v)$ which accepts assignments to V_s that *satisfy* the constraint. For an assignment T to $V'_s \supseteq V_s$, let $T(V_s)$ be the restriction of T to V_s. We say that T satisfies $c = \langle V_s, F \rangle$ if $T(V_s)$ satisfies c. A problem is satisfied by T if T satisfies all constraints. Finding a solution which satisfies all the constraints is an NP-complete task. We say that a CSP is $empty$ if at least one of its domains is the empty set.

CSPs are typically solved by backtracking search, using propagators to reduce domains at each node and avoid parts of the search tree that are proved to not contain any solutions. These techniques will shortly be presented in detail in Section 2.1.4.

Example 2.1. *The well-known number placement puzzle sudoku is in fact a CSP. In its most famous form, we have a* 9×9 *square-shaped grid divided in* 3×3 *subgrids, of which we have nine. This grid is initially partially filled and the aim is to fill all*

the cells with a digit between 1 and 9, while making sure that the digits appearing in each row, each column and each subgrid are all different.

Let $v_{ij} \in V$ denote the cell at row $i \in \{1, \ldots, 9\}$ and column $j \in \{1, \ldots, 9\}$. These are the decision variables of this particular CSP. Then, we have $D(v_{ij}) = \{1, \ldots, 9\}$ for all i and j. For each subgrid $k \in \{1, \ldots, 9\}$, let $V_k \subset V$ be the set of cells in it. Notice that we have $|V_k| = 9$, $\bigcap_k V_k = \varnothing$ and $\bigcup_k V_k = V$.

For each row i, column j and subgrid k, we have the following constraints:

$$\begin{aligned} C_i &: v_{ij} \neq v_{ij'} && \forall i, j, j' \neq j \in \{1, \ldots, 9\} \\ C_j &: v_{ij} \neq v_{i'j} && \forall i, i' \neq i, j \in \{1, \ldots, 9\} \\ C_k &: v_{ij} \neq v_{i'j'} && \forall v_{ij}, v_{i'j'} \in V_k, i' \neq i, j' \neq j, k \in \{1, \ldots, 9\} \end{aligned}$$

$C = C_i \cup C_j \cup C_k$ is the set of constraints to be satisfied.

2.1.2 Constraint Optimisation Problems

CSPs being decision problems, the CP technology can also be efficiently used to solve optimisation problems by adding an objective function to minimise or maximise. Then, we have a Constraint Optimisation Problem (COP) which is a quadruplet $\langle V, D, C, f \rangle$ where V is a set of variables, D is the set of domains of these variables, C is a set of constraints on these variables, just like in a CSP, and finally, $f : \ell(V) \to \mathbb{R}$ is the objective function that evaluates the quality of a given solution by mapping an assignment $T \in \ell(V)$ to a real value.

Example 2.2. *The well-known Knapsack Problem can be cast as a COP. Let $I = \{1, \ldots, n\}$ be a set of items, each item i of which has a particular weight w_i and a value p_i. The aim is to pick a set of items such that their total value is maximised while making sure that their total weight does not exceed a predefined capacity K.*

Let $\{v_1, \ldots, v_n\}$ be the decision variables with $D(v_i) = \{0, 1\}$. We have $v_i = 1$ if we decide to take item i and $v_i = 0$ otherwise. We formulate the capacity constraint as follows:

$$\sum_{i \in I} v_i w_i \leq K$$

Finally, the objective function becomes:

$$\max_{T \in \mathbb{1}^n} T(v_i) p_i$$

where T is an assignment, $T(v_i) \in \{0, 1\}$ is the value that v_i takes in this assignment, and $\mathbb{1}^n$ is the set of all n-dimensional incidence vectors.

v_{11}	v_{12}
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9

v_{11}	v_{21}
2	1
2	3
2	4
2	5
2	6
2	7
2	8
2	9

v_{11}	v_{22}
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9

Table 2.1: A small portion of all the table constraints for the CSP formulation of sudoku presented in Example 2.1. We see the constraints between v_{11} and v_{12} (same row) for $v_{11} = 1$, between v_{11} and v_{21} (same column) for $v_{11} = 2$, and between v_{11} and v_{22} (same subgrid) for $v_{11} = 1$.

2.1.3 Major Types of Constraints

In this section, we will show different ways to represent constraints, as well as an example to see how each of them can be applied to the CSP formulation of sudoku that we saw in Example 2.1.

2.1.3.1 Table Constraints

Table constraints enumerate all allowed value combinations for the variables in their scope. Value combinations which do not appear are simply forbidden.

Example 2.3 (sudoku)**.** *We have seen in Example 2.1 how to formulate sudoku as a CSP. Here, we will see how the constraints can be represented as table constraints.*

For all variable pairs $\{x_{ij}, x_{i'j'}\}$ *that are concerned by a constraint because they are in the same row, column, or subgrid, we enumerate all allowed value combinations between these two variables. We have 9 rows, 9 columns and 9 subgrids, each of which contains 9 variables. Whenever we look at a set of 9 variables, we can choose* $\binom{9}{2} = 36$ *different pairs. For each pair of variables, we have* $9 \times 9 - 9 = 72$ *allowed value combinations. This gives us a total of* $(9 + 9 + 9) \times 36 \times 72 = 69984$ *value combinations to consider.*

In Table 2.1, we have a small portion of these 69984 *value combinations. In the left one, we see the value combinations for the pair* $\{v_{11}, v_{12}\}$*, where we have* $v_{11} = 1$*. We see that the assignment where both of them are 1 does not appear in the table, which simply means that it is forbidden. Similarly in the middle and the right table, we have a pair of variables which are in the same column and the same subgrid, respectively.*

2.1.3.2 Arithmetic Constraints

Arithmetic equality and inequality constraints arise naturally for many real-life problems. Although constraints of any degree can be handled in the CP framework, we focus on the linear ones here. They are basically in the form:

$$\sum_{i=1}^{n} k_i \times v_i \leq K$$
$$\sum_{i=1}^{n} k_i \times v_i = K$$
$$\sum_{i=1}^{n} k_i \times v_i \geq K$$

where v_i's are decision variables, k_i and K are real numbers, and n is an arbitrary positive integer.

Example 2.4 (sudoku)**.** *Let us redefine the decision variables for sudoku. For each variable v_{ij} that we used to have and each value $S \in \{1, \ldots, 9\}$ that it can take, we will define a new decision variable x_{ijS}. This new variable will be one if the cell at row i and column j is assigned S, and zero otherwise. Note that we will have $9 \times 9 \times 9 = 729$ of these variables. Similarly, we will replace the set V_k by X_k, the set of x_{ijS}'s such that the cell at row i and column j belongs to the subgrid k. As a result, our constraints become:*

$$\begin{aligned}
&C_i : x_{ijS} + x_{ij'S} \leq 1 && \forall i, j, j' \neq j, S \in \{1, \ldots, 9\} \\
&C_j : x_{ijS} + x_{i'jS} \leq 1 && \forall i, i' \neq i, j, S \in \{1, \ldots, 9\} \\
&C_k : x_{ijS} + x_{i'j'S} \leq 1 && \forall x_{ijS}, x_{i'j'S} \in X_k, k, S \in \{1, \ldots, 9\} \\
&C_{ij} : \sum_{S=1}^{9} x_{ijS} = 1 && \forall i, j \in \{1, \ldots, 9\}
\end{aligned}$$

For each variable pair in each row i, and each value S, we need to write down $\binom{9}{2} \times 9 = 324$ constraints. We have the same number of constraints for each column j and each subgrid k, as well. As a result, we have a total of $3 \times 9 \times 324 = 8748$ constraints. On top of these, we also have the constraints in C_{ij} to make sure that each cell ij is assigned exactly one digit, and we have 81 of these constraints.

2.1.3.3 Logical Constraints

Logical constraints allow us to capture the relations between the variables using *and* and *or* statements. Each decision variable corresponds to a *literal* that can be *true* or *false*. These literals are put together in clauses, where they can be positive (x) or negative ($\overline{x}$) in order to formulate a constraint.

Example 2.5 (sudoku)**.** *Let us redefine the decision variables for sudoku yet another time. This time, x_{ijS}'s are propositional variables, i.e. they are true is the cell at row i and column j is assigned S and false otherwise. Then, we have the following constraints:*

$$\begin{aligned}
C_i &: \overline{x}_{ijS} \vee \overline{x}_{ij'S} && \forall i, j, j' \neq j, S \in \{1, \ldots, 9\} \\
C_j &: \overline{x}_{ijS} \vee \overline{x}_{i'jS} && \forall i, i' \neq i, j, S \in \{1, \ldots, 9\} \\
C_k &: \overline{x}_{ijS} \vee \overline{x}_{i'j'S} && \forall x_{ijS}, x_{i'j'S} \in X_k, k, S \in \{1, \ldots, 9\} \\
C_{ij} &: \bigvee_{S \in \{1,\ldots,9\}} x_{ijS} && \forall i, j \in \{1, \ldots, 9\}
\end{aligned}$$

Like in Example 2.4, for each variable pair, each value and each row i, we need to write a total of $\binom{9}{2} \times 9 = 324$ constraints. We have 9 rows, 9 columns and 9 subgrids, so having 324 constraints for each of them gives us a total of $(9 + 9 + 9) \times 324 = 8748$ constraints. Additionally, we have the constraints in C_{ij} with $|C_{ij}| = 81$ to make sure each cell ij is assigned exactly one digit.

2.1.3.4 Global Constraints

A *global constraint* is a constraint that is informally defined over a non-fixed number of decision variables. It captures the relation between these variables at once, saving us from explicitly expressing thousands of constraints. This not only facilitates the modelling task, but also allows the CP solvers to have a more compact view of the problem structure.

Example 2.6 (sudoku)**.** *Let us recall the initial CSP formulation presented in Example 2.1, with decision variables v_{ij}, and the binary not-equal constraints c_i, c_j and c_k. Let V_i, V_j and V_k be the set of variables in each row i, column j and subgrid k, respectively. We can replace the binary constraints for each row i, column j and subgrid k by the well-known global constraint AllDifferent as follows:*

$$\begin{aligned}
C_i &: AllDifferent(V_i) && \forall i \in \{1, \ldots, 9\} \\
C_j &: AllDifferent(V_j) && \forall j \in \{1, \ldots, 9\} \\
C_k &: AllDifferent(V_k) && \forall k \in \{1, \ldots, 9\}
\end{aligned}$$

Here, we have 27 constraints instead of 78732 like in the previous examples using other types of constraints.

2.1.4 Search Procedure in CP

In CP, search is done by making decisions and updating domains in accordance with these decisions. A decision basically means assigning an uninstantiated variable a value from its current domain. Once this is done, very likely some values of the

other variables become incompatible with respect to some constraints, which are to be detected and removed.

In the following sections, we assume that we have a COP, and not a CSP, and that the COP at hand is a minimisation problem.

2.1.4.1 Constraint Propagation

The type of constraint propagation that we consider in this thesis is called *pruning* of the variable domains. The pruning task consists in going through the domains to detect and remove the *inconsistent* values. A value $S \in D(v)$ is inconsistent if it cannot appear in any complete solution that is based on the current partial solution. Similarly, a value $S \in D(v)$ is *consistent* if there exists a complete solution where v is assigned S.

Determining whether or not a value is inconsistent is usually an NP-hard task [Rossi *et al.* 2006]. This is why, CP solvers try to simplify this task by pruning values with respect to each constraint separately. Clearly, if a value is inconsistent with respect to a constraint, then it is also inconsistent with respect to the CSP; so it is safe to remove it. However, it is not necessarily always true that a value that is inconsistent with respect to the CSP is also inconsistent with respect to each of the constraints. This brings us to a point where we have to make a trade-off between the speed (i.e. running time) and the strength (i.e. how many of the inconsistent values removed) of the pruning. Some constraints achieve a better trade-off with substantial pruning and high computational efficiency.

Global constraints do achieve a good trade-off from this standpoint. Their ability to look at the *big picture*, i.e. all the variables at once and not one small subset of them at a time, allow them to remove large parts of, and most of the time *all* of the inconsistent values. They are perhaps the most important key to the success of CP, as their tailored algorithms perform a very strong propagation and search space reduction.

The task of pruning all the inconsistent values is like the holy grail in CP. As a consequence, designing specialised algorithms for global constraints in order to achieve this task, i.e. achieve *arc consistency*, has been a very popular research topic over the last decades.

Definition 2.7 (Generalised Arc Consistency)**.** *Let $P = \langle V, D, C \rangle$ be a CSP.*

A variable $v \in V$ is said to be generalised arc consistent (GAC) with respect to a constraint $c \in C = \langle V_s, F \rangle$, if for each value $S \in D(v)$, there exists at least one assignment $T \in \ell(V_S)$ with $T(v) = S$ and $F(T) = true$.

A variable $v \in V$ is said to be GAC with respect to the whole CSP P if it is GAC with respect to all its constraints in C.

A CSP P is said to be GAC if all its variables in V are GAC.

Example 2.8. *Let us retake the sudoku example and compare the amount of inconsistent values that we manage to prune when use binary not-equal constraints and when we use the global constraint AllDifferent.*

Assume that in the first row, the last six variables $\{v_{14}, v_{15}, v_{16}, v_{17}, v_{18}, v_{19}\}$ *are already assigned a digit from the set* $\{4, 5, 6, 7, 8, 9\}$*, and that the current domains for the first three variables are as follows:*

$$D(v_{11}) = \{1, 2, 3\} \qquad D(v_{12}) = \{2, 3\} \qquad D(v_{13}) = \{2, 3\}$$

Now let us look at the CSP formulation where we have the binary not-equal constraints below:

$$v_{11} \neq v_{12} \qquad v_{11} \neq v_{13} \qquad v_{12} \neq v_{13}$$

With these constraints, the only situation in which we are allowed to prune values is a variable having a singleton domain. The current domains given above, however, all have at least two values. In this case, we are unfortunately unable to prune values, because for each constraint we can always find a partial assignment that satisfies it. For example, the partial assignments $\{(v_{11}, 2), (v_{12}, 3)\}$ *and* $\{(v_{12}, 3), (v_{13}, 2)\}$ *satisfy the constraints* $v_{11} \neq v_{12}$ *and* $v_{12} \neq v_{13}$*, respectively. However, when these partial assignments are put together,* v_{11} *and* v_{13} *have the same value, which is not allowed. With the binary constraints, we are unable to see that* $2 \in D(v_{11})$ *is in fact an inconsistent value (so is* $3 \in D(v_{11})$*).*

If we had $AllDifferent(v_{11}, v_{12}, v_{13})$ *instead of the binary constraints, we would have access to one of the* $AllDifferent$ *propagators allowing us to prune large parts of inconsistent values. Alternatives include, but are definitely not limited to, one famous low-complexity algorithm [Régin 1994] which is based on matching theory [Lovász & Plummer 2009] and enforces GAC on* $AllDifferent$*, and, a slightly "weaker" but faster algorithm [López-Ortiz et al. 2003] enforcing domain consistency. By enforcing GAC on* $AllDifferent$*, we would be able to remove 2 and 3 from the domain of* v_{11}*, since obviously, these two digits need to be shared between* v_{12} *and* v_{13}*.*

2.1.4.2 Backtracking Search and Bounding

Backtracking search is one of the three main methods for solving CSPs and COPs, the other two being local search and dynamic programming [Rossi *et al.* 2006]. It is a complete method, meaning that it guarantees to find a solution if one exists.

Backtracking search is an iterative procedure which is based on making *decisions*. Starting from the root node, we make a decision for exactly one variable in accordance with our *branching strategy*, which can be one of the following three:

1. *Enumeration*: We create a branch for each value S in the domain of the variable v and in all the solutions explored in the subtree of this branch, v is assigned S. With this strategy, we branch on one variable at each level, so by the time we are at depth $n = |V|$, all variables have been assigned a value.

2. *Binary Choice Points*: At each node, we create two branches for a variable v and a value $S \in D(v)$. We either assign S to v, or we remove S from $D(v)$, i.e. $v = S$ or $v \neq S$.

3. *Domain Splitting*: With this strategy, we create two branches as well, but this time for the decisions $v \leq S$ and $v > S$.

Before we proceed to explore the subtree of a new branch, we first determine if it is worth exploring it. This is already because whenever we make a decision, we potentially make some set of values inconsistent. As a result, we need to make sure that the current partial assignment can be extended to a complete assignment. Furthermore, we need to know if this new branch promises any solution that is better than what we have found so far. To this end, at each node, we perform propagation and *bounding*. A backtracking search without these two subroutines is nothing but a complete enumeration. We want the part of the search tree that we explore to be as small as possible due to the combinatorial explosion that most optimisation problems inherently have.

Bounding is finding an optimistic and a pessimistic evaluation of the objective function. A *lower bound* (resp. an *an upper bound*) is the smallest (resp. largest) possible value that the objective function can attain. The tighter the bounds are, the lighter the search effort is, as the search terminates when the two bounds meet.

One valid approach to generate a lower (resp. an upper) bound is the greedy one which adds up the smallest (resp. largest) cost of a value in the domain of each variable. This is indeed a very easy way to calculate bounds, but these bounds tend to be very weak, as might be expected. For the initial upper bound generation, it is very common to use a local search algorithm that finds an approximate solution to the problem being solved. Once the search starts, the upper bound is updated whenever a new solution is found. When it comes to the lower bound, a pattern database heuristic [Felner *et al.* 2004], among other heuristic algorithms, can be used.

Another two crucial tools to reduce the size of the search tree are *variable ordering heuristics* and *value ordering heuristics*. The order in which variables are assigned has a huge impact on the performance of backtracking search algorithms [Gent *et al.* 1996]. However, finding the ordering that results in a minimal search tree size is at least as hard as solving the problem itself [Liberatore 2000], which is the reason why it is very common to use variable ordering heuristics. These heuristics include MinDom [Haralick & Elliott 1980], dom/ddeg [Bessiere & Régin 1996], dom/wdeg [Boussemart *et al.* 2004], last-conflict-based heuristic [Lecoutre *et al.* 2009] and impact-based heuristic [Refalo 2004]. These are however only generic approaches, so it is always a good idea to find ways to exploit problem specific structures in order to achieve better results. Value ordering is also important, because, if a problem has a solution, and a correct value is chosen for each variable, we can find a solution following a straight path without having to backtrack. One possible approach is to prioritise values that maximise the number of alternatives for future assignments [Haralick & Elliott 1980].

2.2 Integer Linear Programming

Although we do not directly employ Integer Linear Programming (ILP) in this thesis, our work has a lot of interesting references to it from a CP standpoint. Representing an optimisation problem in a CP or an ILP framework is like speaking two different languages both of which have their own strengths and weaknesses. Just like speaking two languages in real life is something great, deeply understanding CP and ILP as well as grasping the correspondences between them can turn out to be of great use in practice, as we will also see in this thesis.

2.2.1 Linear Programming

A linear program (LP) is the problem of finding

$$\min_x \{c^T x \mid x \in \mathbb{R}^n \wedge Ax \geq b \wedge x \geq 0\}$$

where c is an n-dimensional column vector, b is an m-dimensional column vector, A is an m x n matrix, and x is a column vector of n variables. A row A_i for $i \in \{1, \ldots, m\}$ corresponds to an individual linear constraint i and a column A_j for $j \in \{1, \ldots, n\}$ to a variable x_j. A feasible solution of this problem is one that satisfies $x \in \mathbb{R}^n \wedge Ax \geq b \wedge x \geq 0$, where $\geq$ is an element-wise comparison. On the other hand, an optimal solution is a feasible one that minimises the objective function $c^T x$.

The optimal solution to an LP can be found with the well-known Simplex algorithm [Murty 1983]. Although the theoretical complexity of the Simplex algorithm is exponential, in practice, its average-case complexity under various probability distributions is polynomial [Schrijver 1998].

The dual of an LP P in the above form is another LP D:

$$\max_y \{b^T y \mid y \in \mathbb{R}^m \wedge A^T y \leq c \wedge y \geq 0\}$$

where A, b, c are as before and y is the vector of m dual variables. Each row (constraint) in the primal corresponds to a column (variable) in the dual. Equivalently, each constraint in the dual corresponds to a variable in the primal. Given that, without loss of generality, the primal is a minimisation and the dual is a maximisation, the objective value of any dual feasible solution is a lower bound on the optimum of the primal (P). When P is satisfiable, its dual is also satisfiable and the values of their optima meet. We also note that the dual of the dual is the primal itself.

The point where the solutions of P and D meet, not only their optima have the same value, but also what we call the *complementary slackness conditions* hold. For a given feasible primal solution $\hat{x}$, let $slack^P_{\hat{x}}(i) = A_i\hat{x} - b_i$ be the slack of constraint i. Similarly, for a given feasible dual solution $\hat{y}$, let $slack^D_{\hat{y}}(j) = c - A^T\hat{y}$ be the

slack of constraint j. Complementary slackness states that in optimal solutions x^*, y^*, either the value of a variable x_j in P, or $slack^D_{y^*}(j)$ in D has to be 0. Similarly, either the value of a dual variable y_i in D, or $slack^P_{x^*}(i)$ in P has to be 0.

Given a dual feasible solution $\hat{y}$, $slack^D_{\hat{y}}(j)$ is the *reduced cost* of primal variable $j, rc_{\hat{y}}(j)$. The reduced cost $rc_{\hat{y}}(j)$ is the minimum amount by which the dual objective would increase over $b^T\hat{y}$ if x_j was forced to be non-zero in the primal. In a way, the reduced cost of a variable can be interpreted as an underapproximation of the marginal cost : the cost that we would have to add to the current lower bound if we absolutely wanted that variable to be non-zero.

2.2.2 Solving Strategies

As most real-life problems involve discrete decisions, having only continuous decision variables is far from being practical. This is why, we need to model such problems as an integer linear program (ILP), which is an LP where we replace the constraint $x \in \mathbb{R}^n$ by $x \in \mathbb{Z}^n$. Finding the optimal solution to an ILP is however an NP-hard task [Schrijver 1998]. This particular challenge along with the fact that ILPs are a very convenient tool to model all sorts of problems gained significant attention over the last decades and today, we have several recognised techniques for solving them.

In the following sections, we assume that we have an ILP which is a minimisation problem.

2.2.2.1 Branch-and-Bound

Branch-and-bound [Little *et al.* 1963] is a systematic enumeration of candidate solutions. The candidate solutions are explored in a rooted search tree where nodes represent a (partial) solution and the branches correspond to decisions that assign a variable a value or that reduces a variable's domain interval. Before exploring the subtree of a new branch, an estimation on the lower bound based on this decision is calculated. If this lower bound estimation is greater than or equal to the current upper bound, the subtree is immediately discarded as it does not promise any better solution. On the other hand, if the lower bound estimation is smaller than the current lower bound, the subtree is further explored. Whenever a new feasible solution is found, the upper bound is updated as the value of that solution. The search terminates when the upper and the lower bound meet.

This tells us that the efficiency of the branch-and-bound procedure relies heavily on a good quality lower and upper bound estimation, just like in backtracking search in CP. If there is no bounding mechanism, the algorithm would simply become an exhaustive search among exponentially many solution alternatives. Large parts of the search tree can be avoided thanks to tight lower and upper bounds.

The most commonly employed technique to generate lower bounds for an ILP is to solve its *linear relaxation*. The linear relaxation of an ILP is an LP obtained

by relaxing all the integrality constraints and thus allowing the integral variables to take real values.

At each node of the search tree, the linear relaxation of the current ILP is solved. If the objective function value at the LP's optimum is greater than or equal to the current upper bound, the node is fathomed. Otherwise, a variable with a fractional value in the optimal solution is picked and a new branch from the current node is created. This branch corresponds to a new constraint to be added in the ILP: either the variable is greater than or equal to the smallest integer that is greater than its fractional value, or the variable is less than or equal to the biggest integer that is less than its fractional value. For example, if a variable x turns out to be 3.4 in the LP, then the candidate branches are $x \leq 3$ and $x \geq 4$.

The search terminates when the linear relaxation of the current ILP has an optimal solution where all variables are assigned an integral value.

2.2.2.2 Branch-and-Cut

Branch-and-cut [Padberg & Rinaldi 1987] is a well-known method for solving ILPs. It is usually used when the number of constraints is too large (exponential) to be efficiently handled by the solver. Therefore, the trick is to initially ignore parts, or all of the constraints and introduce a subset of them to the ILP during the search procedure whenever they turn out to be able to *cut off* parts of the *search space*.

Here is a good moment to touch upon the geometry of LP and ILP.

In an LP with n variables, each constraint in the form of an inequality corresponds to a *half-space* in an n-dimensional Euclidean space, which is divided by a *hyperplane* consisting of the points where the right-hand side and the left-hand side of this inequality are equal. The intersection of finitely many half-spaces defines a *convex polytope*, and this polytope is the set of all feasible solutions, i.e. our search space.

A *face* of a convex polytope is any intersection of the polytope with a half-space such that none of the interior points of the polytope lie on the boundary of the half-space. Given an n-dimensional convex polytope, its *facets* are its $(n-1)$-dimensional faces, and its *vertices* are its 0-dimensional faces. The vertices are also called *extreme points*. The optimal solution of an LP lies in one of the vertices of its convex polytope [Wolsey 2020].

When we look at the convex polytope of the linear relaxation of an ILP, likely the vertices will not correspond to integral solutions, and the optimal solution is even less likely to be one. If we had the *convex hull* of (i.e. the smallest convex set containing) the points inside the convex polytope which correspond to an integral solution, then we would simply run the Simplex algorithm on it and be done solving the ILP. However, in most cases, there is an exponential number of inequalities, i.e. *facet-defining inequalities*, needed to describe this convex hull, and characterising them is not a straightforward task.

The aim of the branch-and-cut is to progressively cut off parts of the search space of the linear relaxation of the ILP in order to reduce it more and more towards

$S_1 \in D(v_1)$	$c_{v_1}(S_1)$
a	0
b	2

Table 2.2: c_{v_1}

$S_2 \in D(v_2)$	$c_{v_2}(S_2)$
a	1
b	0

Table 2.3: c_{v_2}

$S_1 \in D(v_1)$	$S_2 \in D(v_2)$	$c_{v_1v_2}(S_1, S_2)$
a	a	0
a	b	1
b	a	0
b	b	3

Table 2.4: $c_{v_1v_2}$

Table 2.5: Table costs for Example 2.10.

the convex hull of the integral solutions. The cutting is done by introducing a subset of the original set of constraints of the ILP which were initially omitted. A key issue of this method is to identify those constraints that are violated, i.e. the *violated inequalities*, because only the violated ones are able to cut through the search space. The task of extracting violated inequalities from a set of *valid inequalities* is called the *separation problem* and usually, the separation problem of an NP-hard problem is also NP-hard [Wolsey 2020].

2.3 Weighted Constraint Satisfaction Problems

What we encounter quite often in real life problems is that there are better and worse solutions, namely our preferences, rather than just feasible and infeasible solutions. *Weighted Constraint Satisfaction Problem* framework allows us to efficiently express our preferences in the form of *cost functions*.

Definition 2.9. A *cost function* $c_{V_S} : \ell(V_S) \to \mathbb{N}_{\geq 0}$ is defined over the scope $V_S \subseteq V$ and it associates a non-negative *integer* cost to each assignment $T \in \ell(V_S)$.

Cost functions can be represented in the form of a table as well as a Cost Function Network (CFN), as in Example 2.10. The *arity* of a cost function c_{V_S} is the cardinality of its scope, $|V_S|$, and in this work, we call the cost functions of arity *zero*, *one*, *two* as *nullary* (or *constant*), *unary* and *binary* cost functions, respectively. The constant cost function $c_\varnothing$ corresponds to the cost that we pay no matter what complete assignment $T \in \ell(V)$ we have.

Example 2.10. *Let v_1, v_2 be two variables with domains $D(v_1) = \{a, b\}$ and $D(v_2) = \{a, b\}$. Let $c_{v_1}(a) = 0$, $c_{v_1}(b) = 2$, $c_{v_2}(a) = 1$, $c_{v_2}(b) = 0$ be the unary cost values, and $c_{v_1v_2}(a, a) = 0$, $c_{v_1v_2}(a, b) = 1$, $c_{v_1v_2}(b, a) = 0$, $c_{v_1v_2}(b, b) = 3$ be the binary cost values. The costs in the form of tables can be seen in Table 2.5.*

The CFN for this example can be seen in Figure 2.1. Each variable $v \in \{v_1, v_2\}$ corresponds to a cell. Each value $S \in D(v)$ corresponds to a dot in the cell. A unary cost $c_v(S)$ is written next to the dot only if it is non-zero. If there is a non-zero binary cost $c_{vv'}$ between (v, S) and (v', S'), then an edge is drawn between the dots corresponding to these assignments.

Definition 2.11. A *Weighted Constraint Satisfaction Problem (WCSP)* [Cooper *et al.* 2010] is a quadruple $\langle V, D, C, k\rangle$ where V is a set of n variables $V = \{v_1, \ldots, v_n\}$, each variable $v_i \in V$ has a domain of possible values $D(v_i) \in D$, as in CSP. C is a set of cost functions, and k is a positive integer or infinity serving as the upper bound.

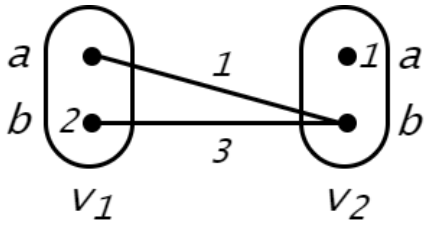


Figure 2.1: The CFN of Example 2.10.

WCSPs generalise CSPs as they can represent the same set of feasible solutions with infinite cost $c_{V_S}(T)) = k$ for forbidden assignments $T \in \ell(V_S)$, but additionally define a cost for feasible assignments. Without loss of generality, we assume all WCSPs contain a nullary cost function $c_\varnothing$, which represents a constant in the objective function, one unary cost function for each variable, and, at most one cost function for all other scopes. Since all costs are non-negative, $c_\varnothing$ is a valid lower bound on the cost of feasible solutions of the WCSP.

If the largest arity of any cost function in a WCSP is 2, then we say this is a *binary WCSP*. From now on, for the sake of simplicity, we will be talking about binary WCSPs only. However, all definitions and properties we present can easily be generalised to higher arities.

Given a WCSP P, the task is to find a complete assignment, i.e. a solution $T \in \ell(V)$ which minimises the sum of all cost functions, denoted as $c_P(T) = c_\varnothing + \sum_{v\in V} c_v(T(v)) + \sum_{c_{vv'}\in C} c_{vv'}(T(v), T(v'))$, and such that $c_P(T) < k$. The minimum value of this sum is denoted as $opt(P)$. This problem is NP-hard [Cooper & Schiex 2004].

2.3.1 Equivalence in WCSPs

A WCSP is usually solved with a branch-and-bound procedure, where maintaining a good lower bound is crucial. $c_\varnothing$ being the lower bound of a WCSP, one can transfer costs from unary and binary cost functions to $c_\varnothing$, thus increase the lower bound, by carrying out *equivalence preserving transformations* that we will see in detail in Section 2.3.2.

Given two WCSPs P_1, P_2 with the same set of variables and scopes, we say they have the same structure. If $c_{P_1}(T(V)) = c_{P_2}(T(V))$ for all $T \in \ell(V)$ then P_1 and P_2 are *equivalent* and they are *reparameterisations* of each other. We say that a reparameterisation is *better* if it has a higher $c_\varnothing$. Finding an optimal

reparameterisation with the highest possible $c_\varnothing$ using integer costs is an NP-hard task [Cooper & Schiex 2004] and therefore, it is common to approximate this task.

It has been shown [Cooper *et al.* 2010] that the optimal reparameterisation, which maximises $c_\varnothing$ using rational costs, is given by the dual of the following LP, called the *local polytope* of the WCSP:

$$\min c_\varnothing + \sum_{v \in V, S \in D(v)} c_v(S) x_{vS} + \sum_{c_{vv'} \in C, S \in D(v), S' \in D(v')} c_{vv'}(S, S') y_{vSv'S'} \tag{2.1a}$$

s.t.

$$\sum_{S \in D(v)} x_{vS} = 1 \qquad \forall v \in V \tag{2.1b}$$

$$x_{vS} = \sum_{S' \in D(v')} y_{vSv'S'} \qquad \forall v \in V, S \in D(v), c_{vv'} \in C \tag{2.1c}$$

$$0 \leq x_{vS} \leq 1 \qquad \forall v \in V, S \in D(v) \tag{2.1d}$$

$$0 \leq y_{vSv'S'} \leq 1 \qquad \forall c_{vv'} \in C, S \in D(v), S' \in D(v') \tag{2.1e}$$

This LP is the relaxation of the ILP formulation of a WCSP, where x_{vS} is a binary variable which is 1 if the WCSP variable v is assigned S, and 0 otherwise. Similarly, $y_{vSv'S'}$ is a binary variable which is 1 if both x_{vS} and $x_{v'S'}$ are both 1, indicating v is assigned S and v' is assigned S' at the same time. Constraints 2.1b ensure that each variable v is assigned exactly one value, whereas constraints 2.1c guarantee that an assignment where v is assigned S is chosen in case $x_{vS} = 1$. The objective function 2.1a is the decomposition of the total cost of an assignment.

From the optimal solution y^* of the dual of the above LP, the reparameterisation is extracted from the reduced costs $rc_{y^*}(x_{vS})$ and $rc_{y^*}(y_{vSv'S'})$ of each unary and binary cost function, respectively, by setting $c_v(S)$ to $rc_{y^*}(x_{vS})$, $c_{vv'}(S, S')$ to $rc_{y^*}(y_{vSv'S'})$, and $c_\varnothing$ to the objective function value of y^*. Because of the correspondence between reparameterisations and solutions of the dual of this LP, we use the two interchangeably.

Since finding the maximal $c_\varnothing$ is NP-hard, it is usually approximated by enforcing different types of *soft arc consistency* that we will see in Section 2.3.3. These soft arc consistencies are enforced by carrying out equivalence preserving transformations.

2.3.2 Equivalence Preserving Transformations

An *equivalence preserving transformation* (EPT) is an operation which transforms a WCSP into an equivalent WCSP with the same cost distribution for all assignments $T \in \ell(V)$.

Algorithm 1 gives three basic equivalence preserving transformations. `Project` projects weights from a binary cost function $c_{vv'}$, to a unary cost function c_v. `Extend`, on the contrary, transfers weights from a unary cost function to a binary

Algorithm 1: Main procedures used to carry out equivalence preserving transformations in binary WCSPs.

Require $\alpha \leq min\{c_{vv'}(T(v), T(v')) \colon T \in \ell(\{v, v'\})$ **and** $T(v) = S\}$
Procedure `Project`$(\{v, v'\}, v, S, \alpha)$**:**
  $c_v(S) \leftarrow c_v(S) \oplus \alpha$
  foreach $T \in \ell(\{v, v'\})$ *such that* $T(v) = S$ **do**
    $c_{vv'}(T(v), T(v')) \leftarrow c_{vv'}(T(v), T(v')) \ominus \alpha$
Require $\alpha \leq c_v(S)$
Procedure `Extend`$(v, S, \{v, v'\}, \alpha)$**:**
  foreach $T \in \ell(\{v, v'\})$ *such that* $T(v) = S$ **do**
    $\beta \leftarrow c_\varnothing \oplus c_v(T(v)) \oplus c_{v'}(T(v'))$
    $c_{vv'}(T(v), T(v')) \leftarrow ((c_{vv'}(T(v), T(v')) \oplus \beta) \ominus \beta) \oplus \alpha$
  $c_v(S) \leftarrow c_v(S) \ominus \alpha$
Require $\alpha \leq min\{c_v(S) \colon S \in D(v)\}$
Procedure `UnaryProject`(v, α)**:**
  foreach $S \in D(v)$ **do**
    $c_v(S) \leftarrow ((c_v(S) \oplus c_\varnothing) \ominus c_\varnothing) \ominus \alpha$
  $c_\varnothing \leftarrow c_\varnothing \oplus \alpha$

cost function. `UnaryProject` projects weights from a unary cost function to the constant cost function $c_\varnothing$. All cost moves are done with the operators $\oplus$ and $\ominus$. $\oplus$ is the aggregation operator which is either the usual addition operator $+$, or the addition-with-ceiling operator $+_k$ defined as

$$a +_k b = min\{a + b, k\} \qquad \forall a, b \in \{0, 1, \ldots, k\}$$

The partial inverse of the aggregation operator $\oplus$ is denoted by $\ominus$, and it is defined as

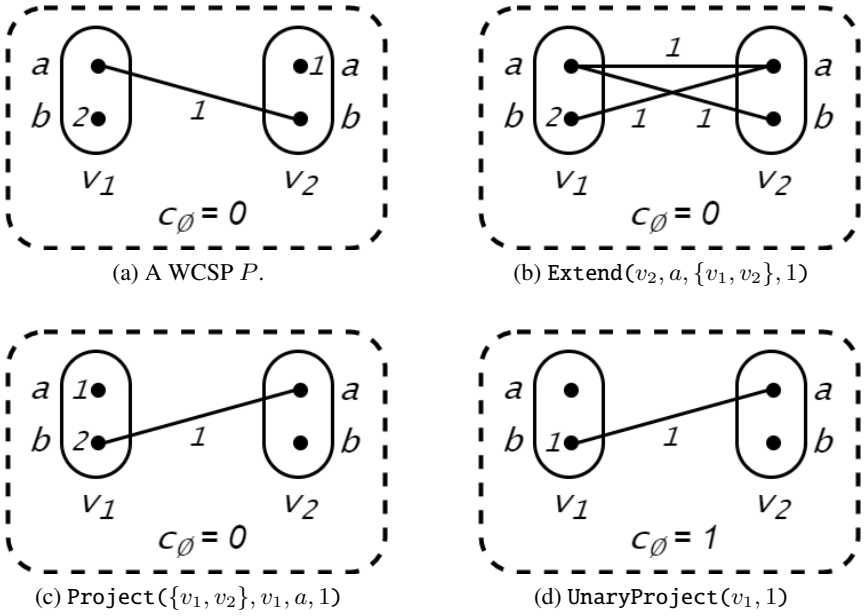
$$a \ominus b = a - b \qquad \forall a, b \in \{0, 1, \ldots, k-1\}, b \leq a$$
$$k \ominus b = k \qquad \forall b \in \{0, 1, \ldots, k\}$$

$\ominus$ is used in `Extend` and `UnaryProject` to make sure that forbidden values remain forbidden after any EPT.

Example 2.12. *Let us consider the WCSP P presented in Figure 2.2a. We have two variables v_1 and v_2, which both have the same domain $D(v_1) = D(v_2) = \{a, b\}$. Initially, the constant cost function $c_\varnothing$, i.e. the lower bound is 0. We carry out EPTs on P in the following order:*

- *From Figure 2.2a to Figure 2.2b:* `Extend`$(v_2, a, \{v_1, v_2\}, 1)$

  We transfer a cost of 1 from (v_2, a) to each assignment on $\{v_1, v_2\}$ where v_2 is assigned a, namely $\{(v_1, a), (v_2, a)\}$ and $\{(v_1, b), (v_2, a)\}$.



(a) A WCSP P.

(b) `Extend`$(v_2, a, \{v_1, v_2\}, 1)$

(c) `Project`$(\{v_1, v_2\}, v_1, a, 1)$

(d) `UnaryProject`$(v_1, 1)$

Figure 2.2: EPTs carried out on a WCSP P.

- *From Figure 2.2b to Figure 2.2c:* `Project`$(\{v_1, v_2\}, v_1, a, 1)$

  We transfer a cost of 1 to (v_1, a) *from each assignment on* $\{v_1, v_2\}$ *where* v_1 *is assigned a, namely* $\{(v_1, a), (v_2, a)\}$ *and* $\{(v_1, a), (v_2, b)\}$.

- *From Figure 2.2c to Figure 2.2d:* `UnaryProject`$(v_1, 1)$

  We transfer a cost of 1 to $c_\emptyset$ *from each value of* v_1.

As a result, we obtain another WCSP appearing in Figure 2.2d, which is equivalent to P *(just like the WCSPs in Figures 2.2b and 2.2c), but has a better lower bound with* $c_\emptyset = 1$.

2.3.3 Soft Arc Consistency

$c_\emptyset$, or equivalently the optimum of the dual of the local polytope (2.1) of a WCSP can be approximated by enforcing different types of *soft arc consistency*. We use the word "soft" because in WCSPs we do not have "hard" constraints like we do in CSPs. Some of these algorithms produce strong lower bounds, with unfortunately high computational cost. Conversely, some others have low complexity, but they provide weaker lower bounds. This is yet another point where we have a trade-off between inference strength and speed of inference.

Definition 2.13. A WCSP is *node consistent* [Cooper *et al.* 2010] if for all $v \in V$,

1. $\forall S \in D(v), c_v(S) \oplus c_\emptyset < k$
2. $\exists S \in D(v)$ such that $c_v(S) = 0$

Example 2.14. *The WCSP in Figure 2.2c is not node consistent since there is no value in $D(v_1)$ with a zero unary cost. However, the one in Figure 2.2d is node consistent, since there is at least one value in $D(v_1)$, a, with a zero unary cost.*

Node consistency (NC) is very intuitive in the sense that, if all values of a variable have a non-zero cost, obviously, no matter what value we assign to that variable, we will have to pay a certain non-zero cost. That cost is at least as large as the smallest cost of all the values, so we may as well transfer that to the lower bound. Enforcing NC is a simple task that can be done in time $O(nd)$ and this is in fact what `UnaryProject` does.

Our main focus is Virtual Arc Consistency (VAC) which computes high quality lower bounds [Cooper *et al.* 2010] but at a significant cost, so it is mostly used in preprocessing, rather than in every node of the search tree. In Chapter 6, we will discover ways to efficiently employ VAC in order to benefit from its good-quality lower bounds throughout the search.

Before moving on to VAC, let us now redefine *arc consistency* from a WCSP standpoint, as it will be relevant. This time we drop the word "generalised" since our focus here is only on binary WCSPs anyway.

Definition 2.15. A CSP P is *arc consistent* (AC) if it fulfils the following conditions:

1. For each binary constraint $c_{vv'} \in C$ and for each value $S \in D(v)$, there exists a value $S' \in D(v')$ such that the partial assignment $\{(v, S), (v', S')\}$ satisfies $c_{vv'}$. We say that the value $S' \in D(v')$ is a *support* for the value $S \in D(v)$.

2. For each binary constraint $c_{vv'} \in C$ and for each value $S' \in D(v')$, there exists a value $S \in D(v)$, i.e. a *support*, such that the partial assignment $\{(v, S), (v', S')\}$ satisfies $c_{vv'}$.

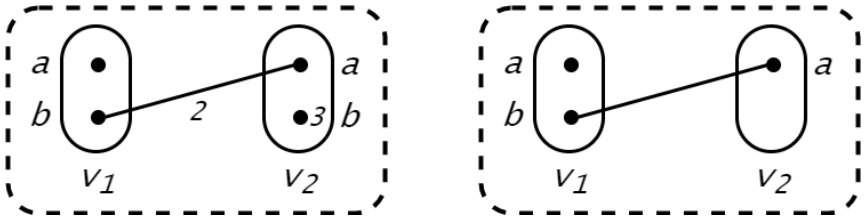
The arc consistent closure $AC(P)$ is the unique CSP which results from removing values from domains that violate the arc consistency property. $AC(P)$ is equivalent to P, i.e., it has exactly the same set of solutions. In particular, if $AC(P)$ is empty (has empty domains), P is unsatisfiable.

Similarly, a WCSP P is AC if it fulfils the following conditions:

1. For each binary cost function $c_{vv'} \in C$ and for each value $S \in D(v)$, there exists a value $S' \in D(v')$ such that $c_{vv'}(S, S') = 0$.

2. For each binary cost function $c_{vv'} \in C$ and for each value $S' \in D(v')$, there exists a value $S \in D(v)$ such that $c_{vv'}(S, S') = 0$.

Definition 2.16. Let $P = \langle V, D, C, k\rangle$ be a WCSP. Then $Bool(P) = \langle V, \overline{D}, \overline{C}\rangle$ is the CSP where, for all $v \in V$, S appears in $\overline{D}(v)$ if and only if $c_v(S) = 0$, and, for all $v, v' \in V$, $S \in D(v)$, $S' \in D(v')$, $\{(v, S), (v', S')\} \in \overline{C}$ if and only if $c_{vv'}(S, S') = 0$.



(a) P (b) $Bool(P)$

Figure 2.3: WCSP P and $Bool(P)$ mentioned in Example 2.17.

In other words, given a WCSP P, $Bool(P)$ is the classical CSP where values which have a unary cost greater than 0 are removed, and binary assignments having a cost greater than 0 become forbidden.

Example 2.17. *Let us consider the WCSP P presented in Figure 2.3a. We have two variables v_1 and v_2, which both have the same domain $D(v_1) = D(v_2) = \{a, b\}$. Non-zero costs are $c_{v_1v_2}(b, a) = 2$ and $c_{v_2}(b) = 3$.*

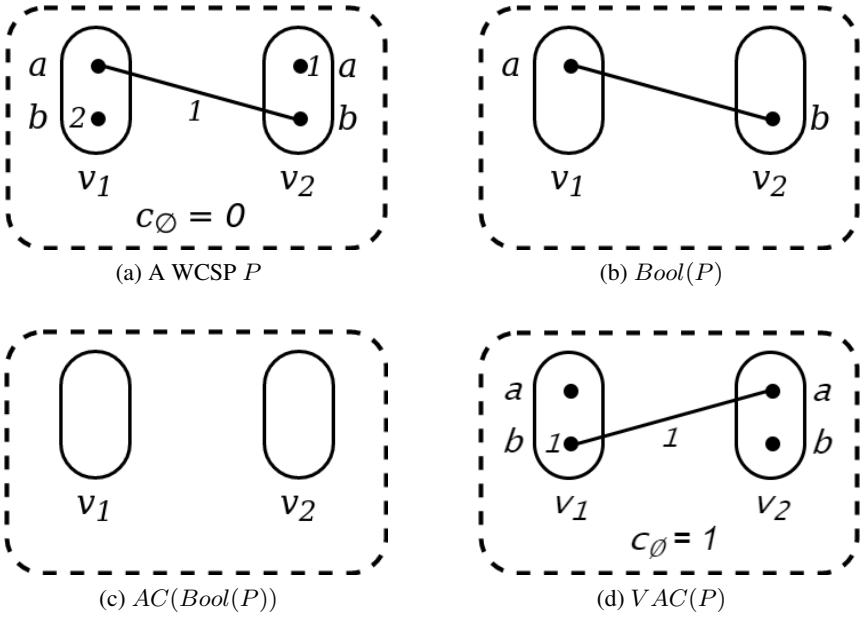
In $Bool(P)$ presented in Figure 2.3b, we see that $b \notin \overline{D}(v_2)$, i.e. $\overline{D}(v_2) = \{a\}$. Also, the assignment $\{(v_1, b), (v_2, a)\}$ which has a cost of 2 in P is simply forbidden in $Bool(P)$. Note that we draw an edge between two values when the corresponding assignment is forbidden.

By construction, $Bool(P)$ admits exactly the solutions of P with cost $c_\varnothing$, since all assignments that have non-zero cost in any cost function of P are mapped to forbidden tuples in $Bool(P)$. Thus, if $Bool(P)$ is unsatisfiable, this indicates $c_\varnothing < opt(P)$. In other words, the query "*Does the CSP $Bool(P)$ have a solution?*" is equivalent to the query "*Does the WCSP P have a solution with a total cost of $c_\varnothing$?*".

Definition 2.18. A WCSP P is *virtual arc consistent* (VAC) if the arc consistency closure of the CSP $Bool(P)$ is non-empty [Cooper *et al.* 2010]. The VAC closure $VAC(P)$ of a WCSP is a fixpoint that is not necessarily unique.

If $AC(Bool(P))$ is empty, or non-empty but still unsatisfiable, then $Bool(P)$ is unsatisfiable and hence $c_\varnothing < opt(P)$.

Example 2.19. *The WCSP P in Figure 2.4a is not VAC, because its $Bool(P)$, presented in Figure 2.4b has an empty AC closure, i.e. is not AC. It is because $a \in D(v_1)$ has no support in $D(v_2)$, and vice versa. When we remove these inconsistent values, we end up with empty domains, i.e. a domain wipe-out. We know that when $AC(Bool(P))$ is empty, there exist a sequence of EPTs which increase $c_\varnothing$ when applied to P [Cooper et al. 2010]. We see the resulting WCSP in Figure 2.4d, with an improved $c_\varnothing$ which is 1.*



(a) A WCSP P (b) $Bool(P)$

(c) $AC(Bool(P))$ (d) $VAC(P)$

Figure 2.4: Enforcing VAC on a WCSP P.

2.3.4 Exact Solution Methods for WCSPs

Exact solution methods for WCSPs are mostly based on branch-and-bound, although there are a few exceptions to this norm: core-guided MaxSAT solvers [Morgado *et al.* 2014], logic-based Benders decomposition [Davies & Bacchus 2011], cut generation [Sontag *et al.* 2008], to name a few. Coming back to branch-and-bound, expressing WCSP optimisation as an ILP (i.e. the local polytope (2.1) of WCSP) and using a solver for it is one approach that immediately comes to mind. However, ILP solvers need to solve the linear relaxation of the problem exactly to obtain a bound at each node of the branch-and-bound tree, an operation that is too expensive for the scale of problems encountered in many real-life applications. A faster alternative to this is to find a soft arc consistency transformation of a WCSP maximising $c_\varnothing$, using rational costs instead of integer costs, which is done by the *OSAC (Optimal Soft Arc Consistency)* algorithm [Cooper *et al.* 2010].

At this point, the advancements in Maximum a Posteriori (MAP) inference for Markov Random Fields (MRF) are also relevant, due to the fact that each WCSP corresponds to a CFN and the CFNs happen to have a tight relationship with probabilistic GMs. By exploiting the logarithmic property $\log(x \times y) = \log(x) + \log(y)$, it is possible to transform any probabilistic GM into a CFN using a $-\log$ transformation. This CFN defines a joint cost function which is in fact a controlled approximation of the $-\log()$ of the joint probability function of the probabilistic GM. Given that the logarithm function is monotonic, the cost minimisation query in WCSP is equivalent to the MAP assignment query in MRF [Cooper *et al.* 2020]. As a result, dual solvers developed for MAP assignment

in MRFs can also be used. For example, *TRW-S (Sequential Tree-Reweighted Message Passing)* [Schoenemann *et al.* 2014] is an algorithm designed to solve the LP-relaxation of the energy minimisation problem in MRFs, which has fixpoints satisfying the VAC conditions, and has proven to be very efficient in many practical applications [Werner 2007].

The most successful dedicated solvers use algorithms that in effect solve the linear relaxation of the WCSP *approximately* and therefore *potentially suboptimally*. Algorithms like VAC, OSAC, and TRW-S produce feasible solutions to the dual of the linear relaxation of the WCSP, which can be used as lower bounds. For the loss of precision that they give up, these algorithms gain significantly in computational efficiency. However, VAC, OSAC and TRW-S being complex algorithms, they are usually used only in the preprocessing, rather than the entire search procedure. During the search, other "lighter" alternatives like *EDAC (Existential Directional Arc Consistent)* [de Givry *et al.* 2005] tend to be the default option. EDAC is robust and was in fact the strongest polynomial-time algorithm to achieve soft arc consistency before OSAC and VAC came into play [Cooper *et al.* 2010].

For the implementations that we will present in Chapter 6, we are using ToulBar2[1] [Cooper *et al.* 2010], an open-source exact solver which has shown to be extremely successful as a black box solver [Hurley *et al.* 2016]. ToulBar2 performs hybrid best-first search (HBFS), combining the qualities of depth-first search and best-first search [Allouche *et al.* 2015]. It utilises quite a few sophisticated algorithms, like boosting search with variable elimination [Larrosa & Dechter 2003], which are either used by default (but can still be disabled) or made optional to the user. Default features include the EDAC algorithm to produce good-quality lower bounds during the search, as well as dom/wdeg and last-conflict heuristics for variable ordering.

2.4 Bayesian Networks

When we have a function over a large set of variables, the description of that function grows exponentially with the number of variables. Probabilistic GMs are powerful tools allowing us to express joint probability distributions over a large set of random variables in a compact manner. *Bayesian network* (BN) is a probabilistic GM that captures conditional dependencies between a set of random variables via a directed acyclic graph (DAG). Each vertex in the DAG corresponds to a random variable and the directed edges encode conditional dependencies.

BNs provide a compact representation of a given joint probability distribution by decomposing the description of the probability function. Let us consider an example to understand how they do this. We took an existing example [Koller & Friedman 2009] as a base and modified it so it is more relevant to the personal experience of the author: having gone through the Covid-19 outbreak and suffering

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

from allergic rhinitis every spring. In this example, we have six random variables with a predefined set of values they can take:

- *Season* $\in \{Spring, Summer, Autumn, Winter\}$
- *Number of daily cases (national Covid-19 reports)* $\in \{High, Medium, Low\}$
- *Hay fever* $\in \{True, False\}$
- *Covid-19* $\in \{True, False\}$
- *Loss of taste* $\in \{True, False\}$
- *High fever* $\in \{True, False\}$

We have $4 \times 3 \times 2 \times 2 \times 2 \times 2 = 192$ possible combinations of values of these six variables. The task of defining a joint probability distribution over 192 possible events is tedious. This is even a tiny example with only 6 variables having very small domain sizes. If we want to be able to deal with settings having a more realistic size, we have to find a way to simplify this task. Thanks to BNs, we can decompose complex probability distributions and define them as a product of factors. Let us consider the dependency relations between these random variables as shown in Figure 2.5.

In Figure 2.5, we have a directed acyclic graph (DAG) where vertices correspond to the random variables {*Season, Number of daily cases, Hay fever, Covid-19, Loss of taste, High fever*}. We associate with each vertex a conditional probability distribution (CPD), so that the DAG together with the local CPDs gives a decomposed representation of the original probability distribution. Thanks to this representation, we can calculate, for example, the probability of the event {*spring, medium rate, no hay fever, no Covid-19, loss of taste, no high fever*} as a product of these six factors:

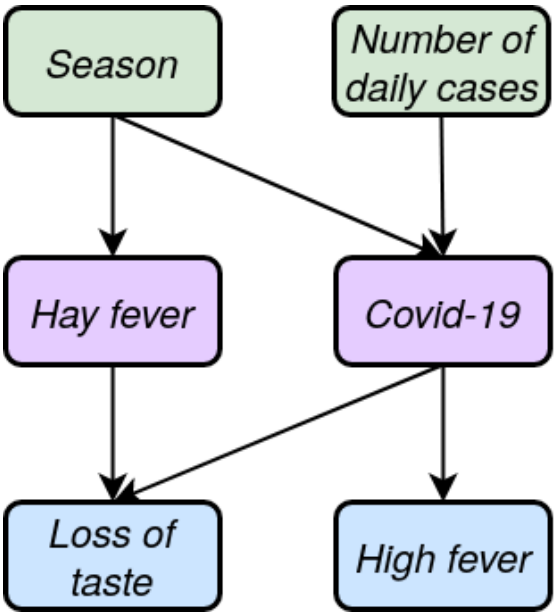


Figure 2.5: Dependency relations between variables represented as a directed graph. We observe that this graph happens to be acyclic.

- *P(Season = Spring)*
- *P(Daily cases = Medium)*
- *P(Hay fever = False | Season = Spring)*
- *P(Covid-19 = False | Season = Spring, Daily cases = Medium)*
- *P(Loss of taste = True | Hay fever = False, Covid-19 = False)*
- *P(High fever = False | Covid-19 = False)*

This calculation requires $3 + 2 + 4 + (4 \times 3) + (2 \times 2) + 2 = 27$ independent parameters: much smaller compared to $192 - 1 = 191$.

2.4.1 Bayesian Network Structure Learning

In this work, we are interested in learning a BN using discrete data. This is known as Bayesian Network Structure Learning (BNSL) in the scientific literature. Imagine since the first Covid-19 lockdown in France we have been collecting data in the following format:

Person ID	Date	Season	Number of daily cases	Hay fever	Covid-19	Loss of taste	High fever
1	17/03/2020	Spring	High	No	No	No	No
2	17/03/2020	Spring	High	No	Yes	No	Yes
...	...	...	...	...	...	...	...
1	18/03/2020	Spring	High	No	Yes	Yes	Yes
2	18/03/2020	Spring	High	No	Yes	No	Yes
...	...	...	...	...	...	...	...
1	01/06/2020	Summer	Low	Yes	No	Yes	No
2	01/06/2020	Summer	Low	No	No	No	No
...	...	...	...	...	...	...	...

Table 2.6: A set of daily observations on a group of people taking into account six criteria: season, number of daily Covid-19 cases, having hay fever, having Covid-19, presence of loss of taste, presence of high fever.

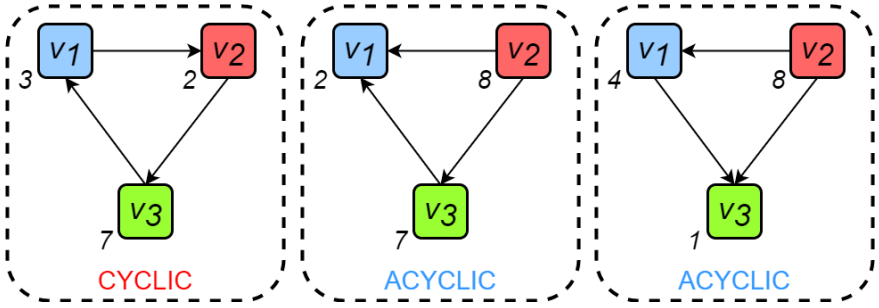
We take the data, which is a set of observations as input, and we try to construct the simplest BN that is most likely to reproduce that data. Learning the BN is a two-step-process: First, we learn the structure of the BN, in other words, the underlying DAG, and second, we learn the conditional probabilities by using the dependency relations embedded in the structure. For the BN in Figure 2.5, we learn the probability of each value of *Loss of taste* given *Hay fever* and *Covid-19*.

Learning the CPDs can be done in polynomial time and it is known that the BN learned from applying these two steps in sequence is globally optimal [Heckerman *et al.* 1995]. Once the BN is learnt from data, it can be used as a powerful tool for *inference* [Koller & Friedman 2009], e.g. finding the most likely assignment to a random variable given a set of observations on others. This is known as finding the Most Probable Explanation (MPE), and MPE lays at the heart of many real-life problems: finding the most probable diagnosis given symptoms and medical test results [Wasyluk *et al.* 2001], weather forecast given meteorological observations [Kennett *et al.* 2001, Cofiño *et al.* 2002], and many other problems in the fields of biotechnology [Allouche *et al.* 2013, Liu *et al.* 2016, Xing *et al.* 2017], economy [Demirer *et al.* 2006, Gemela 2001], sociology [Nedevschi *et al.* 2006, Sticha *et al.* 2006], computer vision [Yuille & Kersten 2006, Luo *et al.* 2005, Stassopoulou *et al.* 1998, Suk *et al.* 2008, Zhang & Ji 2011], risk analysis [Baksh *et al.* 2018, Johansson & Falkman 2008, Lee & Lee 2006, Sỳkora *et al.* 2018, Trucco *et al.* 2008], and so on [Sierra *et al.* 2018].

However, learning the structure, the DAG, is NP-hard, even if the number of parents for each random variable is limited to 2 [Chickering *et al.* 2004]. This difficult task involves finding the optimal graph in a superexponential search space, which expands drastically with each additional random variable.

2.4.2 Score-and-Search Method

Given a set V of random variables with $|V| = n$, a variable v has $2^{(n-1)}$ candidate parent sets. Let alone finding the optimal BN, even attributing a score to each



(a) Sum of scores = 12. (b) Sum of scores = 17. (c) Sum of scores = 13.

Figure 2.6: Three network alternatives for the variables and their candidate parent sets in Example 2.20. The first one has a cycle and is therefore infeasible; the middle one is feasible with a sub-optimal total score, and the last one is the optimal network.

of these parent sets would take a significant amount of time. At this point, it is only natural to ask how one handles these exponentially many candidate parent sets. Luckily, in the majority of cases one can show that many parent sets cannot appear in an optimal BN [de Campos & Ji 2010]. Therefore, those parent sets can be omitted from the very beginning. However, the number of candidate parent sets tends to be large even after the removal of those parent sets. This issue can be addressed by limiting the maximum size of parent sets for which scores are calculated [de Campos & Ji 2010, de Campos *et al.* 2018]. The limitation on the parent set size unfortunately prevents the optimal BN found from being *globally optimal*, since obviously we omit parts of the search space.

The approach which we use here for learning a BN from data is the *score-and-search* method. Given a set of multivariate discrete data $I = \{I_1, \ldots, I_N\}$, a *scoring function* $\sigma(G \mid I)$ measures the quality of the BN with underlying structure G. The BNSL problem asks to find a structure G that minimises $\sigma(G \mid I)$ for some scoring function σ, which happens to be an NP-hard problem [Chickering 1995].

Several scoring functions have been proposed for this purpose, including BDeu [Buntine 1991, Heckerman *et al.* 1995] and BIC [Schwarz 1978, Lam & Bacchus 1994]. These functions are *decomposable* and can be expressed as the sum of local scores which only depend on the set of parents (from now on, *parent set*) of each vertex: $\sigma_F(G \mid I) = \sum_{v \in V} \sigma_F^v(parents(v) \mid I)$ for $F \in \{BDeu, BIC\}$.

In this setting, we first compute local scores and then compute the structure of minimal score. We denote $PS(v)$ the set of candidate parent sets of $v \in V$. In the following, we assume that local scores are precomputed and given as input, as is common in works focusing on BNSL. We also omit explicitly mentioning I or F, as they are constant for solving any given dataset.

Example 2.20. *In Table 2.7, we have a set $V = \{v_1, v_2, v_3\}$ of three variables with their candidate parent sets and scores. In Figures 2.6a, 2.6b and 2.6c, three network alternatives are presented for these variables with their resulting sum of scores. The*

v	$S \in PS(v)$	$\sigma^v(S)$
v_1	$\{v_2, v_3\}$	2
	$\{v_2\}$	4
	$\{v_3\}$	3
	$\{\}$	10
v_2	$\{v_1, v_3\}$	1
	$\{v_1\}$	2
	$\{v_3\}$	6
	$\{\}$	8
v_3	$\{v_1, v_2\}$	1
	$\{v_1\}$	4
	$\{v_2\}$	7
	$\{\}$	9

Table 2.7: Candidate parent sets and their scores for each variable in Example 2.20.

network in Figure 2.6a has a very good total score, but unfortunately is not acyclic. The networks in Figures 2.6b and 2.6c are both feasible, i.e. acyclic networks, the latter being the optimal one.

2.4.3 GOBNILP

One of the state-of-the-art algorithms for BNSL is GOBNILP (Globally Optimal Bayesian Network learning using Integer Linear Programming) [Cussens 2011, Barlett & Cussens 2013]. It uses a specialised branch-and-cut algorithm in an ILP solver. At each node of the branch-and-bound tree, it generates cuts that improve the linear relaxation.

GOBNILP formulates the BNSL problem as the 0/1 ILP below:

$$\min \sum_{v \in V, S \in PS(v)} \sigma^v(S) x_{vS} \tag{2.2a}$$

$$s.t. \sum_{S \in PS(v)} x_{vS} = 1 \qquad \forall v \in V \tag{2.2b}$$

$$\sum_{v \in W, S \in PS^{-W}(v)} x_{vS} \geq 1 \qquad \forall W \subseteq V \tag{2.2c}$$

$$x_{vS} \in \{0, 1\} \qquad \forall v \in V, S \in PS(v) \tag{2.2d}$$

This ILP has a 0/1 variable x_{vS} for each candidate parent set S of each vertex v, where $x_{vS} = 1$ means that S is the parent set of v. The objective (2.2a) directly

encodes the decomposition of the scoring function. The constraint (2.2b) asserts that exactly one parent set is selected for each random variable. Finally, constraints (2.2c), called *cluster inequalities*, ensure that for each subset, i.e. cluster $W \subseteq V$, there is at least one variable $v \in W$ which is assigned a parent set S from the set $PS^{-W}(v) \subseteq PS(v)$ of its candidate parent sets that *do not* intersect W. They are based on the following property [Jaakkola *et al.* 2010] of directed acyclic graphs:

Theorem 2.21. *Let G be a directed graph over vertices V and let $parents(v)$ be the parents of vertex v in the graph. G is acyclic if and only if for every non-empty subset, i.e. cluster $W \subseteq V$ there is at least one vertex $v \in W$ with $parents(v) \cap W = \{\}$.*

A cluster W is a *violated cluster* when it does not satisfy the above condition, i.e. there is no variable $v \in W$ whose parents are *all* outside W. The cluster inequality (2.2c) for cluster W is violated when W is a violated cluster.

As there is an exponential number of cluster inequalities, GOBNILP generates only those that are violated and therefore improve the current linear relaxation. They are referred to as *cluster cuts*. This itself is a separation problem that happens to be NP-hard [Cussens *et al.* 2017], which GOBNILP also encodes and solves as an ILP. Interestingly, these inequalities are facet-defining-inequalities of the BNSL polytope [Cussens *et al.* 2017], so stand to improve the relaxation significantly.

2.4.4 CPBayes

CPBayes [Van Beek & Hoffmann 2015] is a CP-based method for BNSL. It uses the following COP model:

$$\begin{aligned} \min_{v \in V, S \in PS(v)} \quad & \sum_{v \in V} \sigma^v(S) \\ s.t. \quad & acyclic(V) \end{aligned}$$

In this model, each $v \in V$ is called a *parent set variable* with a set of candidate parent sets $PS(v)$, i.e. its domain. An immediate connection between the GOBNILP and CPBayes models is that the ILP variables $x_{vS}, \forall S \in PS(v)$ are the direct encoding [Walsh 2000] of the parent set variables of the CP model. Therefore, we use them interchangeably, e.g., we can refer to the value S in $D(v)$ as x_{vS}. Any complete assignment $T \in \ell(V)$ encodes a DAG: from the value $S = T(v)$ of each variable $v \in V$ we extract the directed edges which are present in the digraph. Finally, over the entire set V of variables, we have a global acyclicity constraint. As a result, any feasible solution to this model corresponds to a DAG.

CPBayes exploits symmetry and dominance relations present in the problem, subproblem caching, and a pattern database to compute lower bounds, adapted from heuristic search [Fan & Yuan 2015]. It performs a backtracking search algorithm where, at each level of the search tree, it instantiates one parent set variable and propagates domain values to remove the ones that are inconsistent with the current

partial assignment. For the global acyclicity constraint, it does not use a GAC propagator, but instead, it uses a polynomial-time satisfiability checker that detects whenever a current partial assignment is cyclic. We will see this satisfiability checker in more detail in Chapters 3 and 4.

It has been shown that CPBayes is competitive with GOBNILP in many datasets [Van Beek & Hoffmann 2015]. In contrast to GOBNILP, the inference mechanisms of CPBayes are very lightweight, which allows it to explore many orders of magnitude more nodes per time unit, even accounting for the fact that computing the pattern databases before search can sometimes consume considerable time. On the other hand, the lightweight pattern-based bounding mechanism can take into consideration only limited information about the current state of the search. Specifically, it can take into account the current total ordering implied by the DAG under construction, but no information that has been derived about the potential parent sets of each vertex, i.e., the current domains of parent set variables.

Remark. CPBayes and GOBNILP lie at the two extremes of the trade-off between inference strength and speed of inference: GOBNILP produces high-quality lower bounds whereas CPBayes has an outstanding rate of search nodes discovered per second. In this thesis, our main purpose is to achieve a better trade-off between these two extremes. We take the CPBayes solver as a base and incorporate several algorithms in it, in order to better deal with the global acyclicity constraint and the large domains. Throughout the chapters in this manuscript, we will add each of these algorithms one by one, and at the very end, we will call the resulting solver ELSA (**E**xact **L**earning of bayesian network **S**tructure using **A**cyclicity reasoning).

CHAPTER 3

Generalised Arc Consistency for the Global Acyclicity Constraint

Contents

3.1 Introduction

The global acyclicity constraint for BNSL is difficult to implement efficiently. van Beek and Hoffmann have given an algorithm which detects unsatisfiability of the acyclicity constraint in time $O(n^2d)$ [Hoffmann & van Beek 2013], where n is the number of random variables and d is the largest domain size. They then used this satisfiability checker to build a propagator that enforces generalised arc consistency (GAC) by probing, i.e., detecting unsatisfiability after assigning each individual value and pruning those values that lead to unsatisfiability. This gives a GAC propagator with complexity $O(n^3d^2)$. However, they later showed that this GAC propagator was unfortunately too costly to be practical [Van Beek & Hoffmann 2015].

Using this acyclicity checker, we show that we can enforce GAC in time $O(n^4d)$. Given that d is usually much larger than n, this presents a significant improvement over $O(n^3d^2)$. We later show that we can further reduce the complexity to $O(n^3d)$. Additionally, we propose a set of techniques that do not improve asymptotic complexity but improve performance.

3.2 Acyclicity Checker of van Beek and Hoffman

`acycChecker` presented in Algorithm 2 takes the following as input:

- $prefix$: a subset of parent set variables already placed in the ordering.

  Note that we treat $prefix$ as both a set and a sequence, as appropriate.

- V: the set of all parent set variables.

- D: domains of the variables in V.

  Note that the domains are the current domains at the moment `acycChecker` is called. Therefore, for example, the domain of an instantiated variable is a singleton domain consisting of its assigned value.

Algorithm 2: Acyclicity Checker

```
1  Procedure acycChecker(prefix, V, D):
2      O ← prefix
3      support ← [ ]
4      changes ← true
5      i ← size(prefix)
6      while changes do
7          changes ← false
8          foreach v ∈ V \ O do
9              if ∃S ∈ D(v) s.t. S ⊆ O then
10                 O_i ← v
11                 support_v ← S
12                 changes ← true
13                 i ← (i + 1)
14     return O
```

Its correctness is based on the following property [Jaakkola *et al.* 2010] of directed acyclic graphs.

Theorem 3.1. *Let G be a directed graph over vertices V and let $parents(v)$ be the parents of vertex v in the graph. G is acyclic if and only if for every non-empty subset $W \subseteq V$ there is at least one vertex $v \in W$ with $parents(v) \cap W = \{\}$.*

`acycChecker` greedily constructs an ordering O of the variables[1], such that if variable v is later in the ordering than v', then $v \notin parents(v')$. It does so by trying to pick a parent set S for an as yet unordered vertex such that S is entirely contained in the set of previously ordered vertices. The order in which the as yet unordered vertices are considered is unimportant, i.e. the algorithm is correct regardless of the order used. However, without loss of generality, we choose to consider variables in a lexicographic order in our implementation.

[1] Just like $prefix$, we treat O as both a sequence and a set, as appropriate.

If in fact all assignments yield cyclic graphs, `acycChecker` will eventually reach a point where all remaining vertices form a cycle among themselves, and it will return a partially constructed ordering. If there exists at least one complete assignment that gives an acyclic graph, it will be possible by Theorem 3.1 to select from a variable in $V \setminus O$ a parent set which does not intersect $V \setminus O$, hence is a subset of O. The value S chosen for each variable in line 9 also gives a witness of such an acyclic graph.

Theorem 3.2. `acycChecker` *returns a complete order if and only if there exists an instantiation of the variables that yields an acyclic DAG. It does this in* $O(n^2 d)$ *time.*

Proof. $\Rightarrow$ Let $i \in \{0, \ldots, (n-1)\}$ denote the while-loop iteration that is to be executed. `acycChecker` has placed i many variables in O and looks at the variables in $V \setminus O$. Suppose there exists an acyclic solution with the current set of domains. By Theorem 3.1, we know that there exists a variable $v \in (V \setminus O)$ with a value $S \in D(v)$ such that $S \cap (V \setminus O) = \{\}$. Therefore, at each iteration i, we can find a variable to place in O.

$\Leftarrow$ Now suppose the current set of domains cannot yield any acyclic solution. This means that there is at least one cycle $\{v_1 \rightarrow \ldots \rightarrow v_m \rightarrow v_1\}$. Let $W = \{v_1, \ldots, v_m\}$ be the set of variables involved in this cycle. Without loss of generality, assume `acycChecker` has placed all the variables in $V \setminus W$ in the ordering, hence we have $prefix = V \setminus W$. However, by Theorem 3.1, we know that there is no variable v in W with a value S in its domain such that $S \cap W = \{\}$. So the failure arises.

In the worst case scenario, both the while loop (line 6) and the for loop (line 8) will run n times, and the domain check (line 9) will run d times. This results in a complexity $O(n^2 d)$. □

Example 3.3 (Acyclicity checker on a small problem)**.** *Consider a BNSL problem with domains as shown in Table 3.1. Let O be a vector that is initially empty. Let* O_i *denote the variable* $v \in V$ *that is at position* i *in O.*

At the first while-loop iteration starting at line 6, `acycChecker` *looks at* v_0 *to see if it can place it in the order. Given that there is no value in* $D(v_0)$ *that allows it to be the first in the ordering, it moves on to the next variable,* v_1*, then to* v_2*.* v_2 *has the empty set in its domain and as a result,* O_0 *becomes* v_2*. At the next iteration, it looks at* v_0 *again. Seeing that it can now add* v_0 *to the ordering since it can have* v_2 *as its parent who has already been added to O, it sets* $O_1 = v_0$*. Moving on, it looks at* v_1 *and sees that it has a candidate parent set,* $\{v_2\}$*, which is a subset of the current O, so it adds* v_1 *to O and* O_2 *becomes* v_1*. It continues the same manner until it obtains the ordering* $\{v_2, v_0, v_1, v_3, v_4\}$.

Variable	Domain Value
v_0	$\{v_1\}$
	$\{v_2\}$
	$\{v_2, v_4\}$
v_1	$\{v_2\}$
	$\{v_3\}$
	$\{v_4\}$
	$\{v_2, v_3\}$
	$\{v_3, v_4\}$
v_2	$\{\}$
v_3	$\{v_0, v_4\}$
	$\{v_1, v_2\}$
v_4	$\{v_0\}$

Table 3.1: BNSL problem used as running example in this chapter.

3.3 Intuition Behind the GAC Propagator

Assume `acycChecker` terminates without detecting unsatisfiability. Then, it produces and returns a *valid ordering* $O = \{O_0, \ldots, O_i, \ldots, O_{n-1}\}$ of variables that it obtains by picking at each step i a variable with a parent set value in its domain that is a subset of the variables chosen at steps $0 \ldots (i-1)$.

Recall that from a valid ordering O, we can construct a support for the constraint: at each position i pick a parent set value S in the domain of $v = O_i$ such that $S \subseteq \{O_0, \ldots, O_{i-1}\}$. We say such a value is consistent with the ordering O. However, the choice of S has no impact on whether the assignment constructed is a support. Therefore, any choice will do and so a valid ordering provides support for all values which are consistent with it.

Consider now $S' \in D(v)$ which is inconsistent with O, i.e. $S' \not\subseteq \{O_0, \ldots, O_{i-1}\}$, therefore we have to probe to see if it is supported. By probe, we mean a sequence of operations: assume $v = S'$, propagate, prune $S' \in D(v)$ if propagation fails. We know that during the probe, nothing forces `acycChecker` to deviate from $\{O_0, \ldots, O_{i-1}\}$. So in a successful probe, it is *not incorrect* for `acycChecker` to construct a new valid ordering by placing the first $i-1$ elements $\{O_0, \ldots, O_{i-1}\}$ in the same manner, and trying to place another variable first, and not v, at position i. If there exists another variable v' with $S' \in D(v')$ such that $S' \subseteq \{O_0, \ldots, O_{i-1}\}$, then `acycChecker` can construct another valid ordering O' which is identical to O in the first $i-1$ positions $\{O_0, \ldots, O_{i-1}\}$ and in which v is *pushed down*, i.e. v is placed at a position later than i. Then all values $S \in D(v)$ consistent with O' are supported.

This observation provides a potential performance improvement over the prob-

Algorithm 3: Acyclicity Checker - The version taking v' and i' as input and making sure v' is placed later than position i'.

```
1  Procedure acycChecker-Constrained(prefix, V, D, v', i'):
2      O ← prefix
3      support ← {}
4      changes ← true
5      i ← size(prefix)
6      while changes do
7          changes ← false
8          foreach v ∈ V \ O do
9              if v = v' and i ≤ i' then
10                 continue
11             if ∃S ∈ D(v) s.t. S ⊆ O then
12                 O_i ← v
13                 support_v ← S
14                 changes ← true
15                 i ← (i + 1)
16     return O
```

Algorithm 4: GAC propagator probing with orders.

```
1  Procedure Acyclicity-GAC-n4d(V, D):
2      O ← acycChecker(∅, V, D)
3      if O ⊊ V then
4          return false
5      foreach v ∈ V do
6          i ← O^{-1}(v)
7          prefix ← {O_0, ..., O_{i-1}}
8          Mark each S ∈ D(v) s.t. S ⊆ prefix
9          while i < n and O is not empty do
10             v' ← v
11             i' ← i
12             O ← acycChecker-Constrained(prefix, V, D, v', i')
13             i ← O^{-1}(v)
14             prefix ← {O_0, ..., O_{i-1}}
15             Mark each S ∈ D(v) s.t. S ⊆ prefix
16     foreach v ∈ V, S ∈ D(v) do
17         if v = S is unmarked then
18             Prune v = S
19     return true
```

ing propagator: we only need to probe the values which are not supported by the ordering produced in the first run of `acycChecker`. Moreover, as we run it more in different probes, it produces new orders which support a set of values, which is a superset of the previously supported values. Therefore, we can decrease the number of probes that are necessary to find a support for all values. To say more precisely, we reduce the number of probes from $O(nd)$ (run `acycChecker` for each variable and value) to $O(n^2)$ (run `acycChecker` for each variable and position). `Acyclicity-GAC-n4d`, shown in Algorithm 4, exploits this insight. It ensures first that `acycChecker` can produce a valid ordering O. Then, for each variable v, it constructs a new ordering O' from O so that v is pushed down as much as possible. Finally, it prunes all parent set values $S \in D(v)$ that are inconsistent with O'.

3.4 The GAC Propagator

In `Acyclicity-GAC-n4d` presented in Algorithm 4, we first call `acycChecker` at line 2 to construct an initial valid ordering. If it cannot produce a complete valid ordering, we immediately return a failure. Then, in the for loop starting at line 5, we take each variable v one by one and in the while loop starting at line 9, we try to push v in the initial ordering as much as possible. We do this by calling a slightly different[2] version of `acycChecker`, called `acycChecker-Constrained` as we do not use it anymore[3], at line 12 with two additional input parameters. These are v' and i' and we use them to make sure that v is no earlier than position $i+1$ in the ordering. We also use the notation $O^{-1}(v)$ to get the index of v in the ordering O. Also note that here, $prefix$ given to `acycChecker-Constrained` is not necessarily the empty set, but the set of variables preceding v in O.

Whenever we manage to push a variable down, the ordering, the prefix and the position i of the variable are updated. Given the new $prefix$, we mark each value $S \in D(v)$ that is a subset of it as consistent. At the end of all iterations, we prune the values that were never marked.

Theorem 3.4. `Acyclicity-GAC-n4d` *enforces GAC on the Acyclicity constraint in* $O(n^4d)$.

Proof. We know that if a value $S \in D(v)$ of a variable $v \in V$ is consistent, then there exists a valid ordering Q that supports it. Here, we show that from any valid ordering O that does not support S, one can construct another valid ordering O' that supports it. We also show that for each variable v, there exists a valid ordering which supports all its consistent values and that `Acyclicity-GAC-n4d` always discovers one such valid ordering.

Let $v \in V$ and $S \in D(v)$. Also let $O = \{O_0, \ldots, O_{(n-1)}\}$ and $Q = \{Q_0, \ldots, Q_{(n-1)}\}$ be two valid orderings such that O does not support S whereas Q does. Assume $O_i = Q_j = v$ and let $O_p = \{O_0, \ldots, O_{(i-1)}\}$, $Q_p = \{Q_0, \ldots, Q_{(j-1)}\}$ be

[2]The difference comes from the two additional parameters v' and i' and the lines 9 and 10.

[3]The reason why will be clear in Section 3.4.1.

the *prefix* of v, i.e. the sets of variables that precede v in orders O and Q, respectively. Similarly, let $O_s = \{O_{i+1}, \ldots, O_{(n-1)}\}$, $Q_s = \{Q_{j+1}, \ldots, Q_{(n-1)}\}$ be the *suffix* of v, i.e. the sets of variables that succeed v in orders O and Q, respectively.

Let O' be the ordering O_p followed by Q_p, followed by v, followed by O_s, keeping only the first occurrence of each variable when there are duplicates. O' is a valid order: O_p is witnessed by the assignment that witnesses O, Q_p by the assignment that witnesses Q, v by S (as in Q) and O_s by the assignment that witnesses O. It also supports S, as required.

Going from O to O' can be done in steps: We go through the variables in Q_p one by one, namely $Q_0, \ldots, Q_{(j-1)}$, and the first variable we hit which comes after v in O we can insert it before v. Let this first variable be Q_p^*. We can then transform O to O'^1, in which we have O_p followed by $\{Q_p^*\}$, followed by $\{v\}$, followed by $O_s \setminus \{Q_p^*\}$. We can continue to push v down in this manner until we obtain O'^2 and so on and finally O'.

Once we can no longer push v down, it means that all consistent values are supported, otherwise we would be able to transform the current ordering to yet another valid ordering to support that value. As a result, the final ordering obtained will allow us to remove all inconsistent values and keep all those that are consistent.

We repeat the while loop at line 9 at most n times for each of the n variables, where we call `acycChecker` with complexity $O(n^2d)$, which results in $O(n^4d)$.

□

Example 3.5 (Running GAC on a small problem)**.** *Consider a BNSL problem with domains as shown in Table 3.1. We will now run* `Acyclicity-GAC-n4d` *presented in Algorithm 4 on this problem. From Example 3.3, we know that* `acycChecker` *at line 2 returns $O = \{v_2, v_0, v_1, v_3, v_4\}$. Let O_i denote the variable $v \in V$ that is at position i in O.*

Pushing variables down in the ordering. *During the for loop starting at line 5, we consider each variable $v \in V$ one by one and try to see if we can push it down in the order. At the first iteration, we look at variable v_0. It is at position $i = 1$, and its prefix is $\{v_2\}$. At line 15, we mark $S = \{v_2\} \in D(v_0)$ as consistent and call* `acycChecker` *to construct an ordering using $\{v_2\}$ as prefix such that v_0 is placed no earlier than position 2. O becomes $\{v_2, v_1, v_0, v_3, v_4\}$. At the next iteration at line 15, we mark $S = \{v_1\} \in D(v_0)$ as consistent and ask* `acycChecker-Constrained` *to construct an ordering using $\{v_2, v_1\}$ as prefix such that v_0 is placed no earlier than position 3. O becomes $\{v_2, v_1, v_3, v_0, v_4\}$. There is no additional value $S \in D(v_0)$ that we can mark as consistent. We can still attempt to push v_1 further down as we currently have $i = 3 < 4$. We ask* `acycChecker-Constrained` *to construct an ordering using $\{v_2, v_1, v_3\}$ as prefix such that v_0 is placed no earlier than position 4. It fails to construct one such ordering as v_0 is a necessary parent for v_4. As a result, the while loop for v_0 terminates and we prune the value $S = \{v_2, v_4\} \in D(v_0)$ since it was never marked as consistent during the iterations.*

3.4.1 Reducing Complexity to $O(n^3d)$

Following the proof of Theorem 3.4, given an initial valid ordering O, it is enough to find one variable in the suffix of v that we can insert before v. As a result, we do not need to ask `acycChecker-Constrained` to construct a new valid ordering every time. This improvement leads us to `Acyclicity-GAC-n3d` presented in Algorithm 5. As it can be seen, the call to `acycChecker-Constrained` is no longer called inside the while loop starting at line 9. Instead, we just find a variable in the suffix, namely the set $O \setminus (prefix \cup \{v\})$, who can have a parent set that is a subset of the current prefix. As a result, the complexity of the while loop decreases from $O(n^2d)$ to $O(nd)$.

Example 3.6 (Running the improved GAC propagator `Acyclicity-GAC-n3d` on a small problem)**.** *Consider a BNSL problem with domains as shown in Table 3.1. From previous examples, we know that* `acycChecker` *at line 2 in Algorithm 5 returns* $O = \{v_2, v_0, v_1, v_3, v_4\}$.

Pushing variables down in the ordering. *At the first iteration of the for loop starting at line 5, we look at variable* v_0. *It is at position* $i = 1$*, and its prefix is* $\{v_2\}$. *At line 5, we look for a variable in* $O \setminus (prefix \cup \{v\})$*, which is* $\{v_1, v_3, v_4\}$*, that we can insert just before* v_0. v_1 *is one such variable, as it can have* v_2 *as its parent. As a result, we add* v_1 *to prefix (line 13) and the current prefix becomes* $\{v_2, v_1\}$. *At the next iteration of the for loop, we look at the set* $\{v_3, v_4\}$ *to see if there is any variable we can insert just before* v_0. *We pick* v_3 *as it can have* $\{v_1, v_2\}$ *as its parent set. We then add* v_3 *to prefix (line 13) and the current prefix becomes* $\{v_2, v_1, v_3\}$. *Finally, at the next iteration, we look at the last variable in the current suffix,* v_4*, to see if we can insert it just before* v_0. *This is not doable, as the only parent* v_4 *can have is* v_0*, so it has to follow* v_0 *in the ordering.*

At the end of these for loop iterations, we obtained a new valid ordering where v_0 *comes as late as possible:* v_2, v_1, v_3, v_0, v_4. *Observe that we never maintained this order explicitly in the pseudo code, as we only need to know the set of variables appearing in the final prefix. Now, we can prune the values* $S \in D(v_0)$ *which are not a subset of* $\{v_2, v_1, v_3\}$. *This value happens to be* $\{v_2, v_4\}$.

3.5 Code Optimisation

We made two main modifications in the implementation of `Acyclicity-GAC-n3d` in Algorithm 5 in order to improve the practical performance.

The first one is that when we create a valid ordering by calling `acycChecker` at line 2, we also save the support of each variable for this ordering in a separate vector. Then, at line 12, before checking if there exists a parent set value $S \in D(w)$ that is a subset of the current prefix, we first see if the current support of w is consistent. This way, at times, we avoid traversing the entire domain of a variable w.

Algorithm 5: GAC propagator for acyclicity

```
1  Procedure Acyclicity-GAC-n3d(V, D):
2      O ← acycChecker(∅, V, D)
3      if O ⊊ V then
4          return false
5      foreach v ∈ V do
6          changes ← true
7          i ← O^{-1}(v)
8          prefix ← {O_0, ..., O_{i-1}}
9          while changes do
10             changes ← false
11             foreach w ∈ O \ (prefix ∪ {v}) do
12                 if ∃S ∈ D(w) s.t. S ⊆ prefix then
13                     prefix ← prefix ∪ {w}
14                     changes ← true
15         Prune {S | S ∈ D(v) ∧ S ⊈ prefix}
16     return true
```

Example 3.7 (Skipping domain traversal)**.** *Consider the BNSL problem with domains as shown in Table 3.1. Let $support$ be a vector that is initially empty. Let $support_v$ denote the value $S \in D(v)$ that is supported by O.*

Using supports saved in `acycChecker`**.** *During the run of* `acycChecker` *called at line 2, every time we add a new variable v to O, we also save the supporting value S that allows us to carry out this addition (line 11 in Algorithm 2). From Example 3.3, we know that* `acycChecker` *returns the ordering $O = \{v_2, v_0, v_1, v_3, v_4\}$ for this problem. The corresponding support vector is $\{\{v_2\}, \{v_2\}, \{\}, \{v_1, v_2\}, \{v_0\}\}$.*

Now assume we run the GAC propagator in Algorithm 5 and iterate over variable v_0. It is at position $i = 1$ in O and currently we have $prefix = \{v_2\}$. We try to see if there exists a variable w in $O \setminus (prefix \cup \{v_0\})$ that we can insert before v_0. We look at variable v_1 and instead of traversing all the values in its domain, we first check if $support_{v_1} = \{v_2\}$ is a subset of the current prefix. We are lucky and this is the case, so we can immediately add it to the prefix and move on to the next iteration.

The second improvement allows us to skip checking the domains of variables that can evidently be pushed to the end of the ordering. It is based on the following observation:

Observation 3.8. *If a variable v is placed at the last position of a valid ordering O, i.e. $O_{n-1} = v$, then all the values S in $D(v)$ are supported.*

Based on this observation, we do not need to probe the values of the variable at position $n - 1$. Moreover, we can try to see if there are any other variables that

Algorithm 6: Detecting tail variables.

```
1 tailVars ← ∅
2 tailVarsSupport ← ∅
3 foreach i ∈ {(n − 1), . . . , 0} do
4     v ← O_i
5     if v ⊆ tailVarsSupport then
6         break
7     tailVars ← tailVars ∪ {v}
8     tailVarsSupport ← tailVarsSupport ∪ support_v
```

can be swapped with it. We do this by going back from O_{n-1} one by one. We add O_i to the set of *tail variables* if it does not appear in the set $support_{n-1} \cup support_{n-2} \ldots \cup support_{i+1}$. As soon as we encounter the first variable appearing in the union of the supported values, we terminate. Then, during the propagation, we skip the variables added to the set of tail variables, since we know that in their domains there is no value to prune. The detection of tail variables is presented in Algorithm 6. Note that the detection of this set is not perfect. There are perhaps other supports that would give a larger set of tail variables, however, it is fine as this is supposed to be a heuristic only.

Example 3.9 (Tail variables)**.** *From Example 3.3, we know that* `acycChecker` *at line 2 of Algorithm 5 returns* $O = \{v_2, v_0, v_1, v_3, v_4\}$ *and saves a support vector* $\{\{v_2\}, \{v_2\}, \{\}, \{v_1, v_2\}, \{v_0\}\}$.

We initialise the sets $tailVars$ *and* $tailVarsSupport$ *as the empty set. We add* $O_4 = v_4$ *to the set* $tailVars$ *and its support* $\{v_0\}$ *to the set* $tailVarsSupport$*. Then we look at* $O_3 = v_3$*. It does not appear in* $tailVarsSupport$*, so we add it to* $tailVars$ *and its support* $\{v_1, v_2\}$ *to* $tailVarsSupport$*. At the next step, we look at* $O_2 = v_1$*.* v_1 *appears in* $tailVarsSupport = \{v_0, v_1, v_2\}$*, so we terminate and return the set* $\{v_3, v_4\}$ *as last variables.*

During the for loop starting at line 5 in Algorithm 5, we skip variables appearing in $tailVars$*.*

3.6 Experimental Results

3.6.1 Benchmarks and Settings

The datasets used in this chapter, as well as in Chapter 4 and Chapter 5, come from the UCI Machine Learning Repository[4], the Bayesian Network Repository[5], and the Bayesian Network Learning and Inference Package[6]. We have 51 medium datasets

[4] `http://archive.ics.uci.edu/ml`

[5] `http://www.bnlearn.com/bnrepository`

[6] `https://ipg.idsia.ch/software.php?id=132`

with $|V| < 64$, 18 large datasets with $64 \leq |V| < 128$, and 23 very large datasets with $|V| \geq 128$. For now, we omit the 23 very large data sets until the next chapter.

Local scores were computed from the datasets using B. Malone's code[7]. BDeu and BIC scores were used for medium datasets (less than 64 variables) and only BIC score for large datasets (above 64 variables). The maximum number of parents was limited to 5 for large datasets (except for `accidents.test` with maximum of 8), a high value that allows even learning complex structures [Scanagatta *et al.* 2015]. For example, `jester.test` has 100 random variables, a sample size of $4,116$ and $770,950$ parent set values. For medium datasets, no restriction was applied except for some BDeu scores (limit sets to 6 or 8 to complete the computation of the local scores within 24 hours of CPU-time [Lee & van Beek 2017a]).

For the experiments, we modified the C++ source of CPBayes v1.1 just to allow us to run it with datasets having more than 64 variables. For our own methods, we take this version of CPBayes as a base, integrate our GAC propagator in it, and call the resulting solver ELSA ^{gac}. In Chapter 4 and Chapter 5, we will sequentially integrate our lower bound mechanism and binary-decision-tree-based domains to ELSA ^{gac}, until we obtain ELSA which we have made publicly available.

All computations were performed on a single core of Intel Xeon E5-2680 v3 at 2.50 GHz and 256 GB of RAM with a 1-hour CPU time limit for the 51 medium datasets, as well as 3 of the large datasets: `kdd.ts`, `kdd.test`, and `kdd.valid`. For the remaining 15 large datasets, we had a 10-hour CPU time limit. For the preprocessing phase, we used two different settings depending on problem size $n = |V|$: $l_{min} = 20, l_{max} = 26, r_{min} = 50, r_{max} = 500$ if $n \leq 64$, else $l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$, where l_{min}, l_{max} are partition lower bound sizes and r_{min}, r_{max} are the number of restarts for the local search. All these settings apply to the experiments done for Chapters 4 and 5 as well.

3.6.2 Evaluation

For the evaluation, we compare CPBayes and ELSA ^{gac} in terms of the total search time in seconds, and total number of search nodes. In Table 3.2, we present these measures of CPBayes and ELSA ^{gac} to solve each dataset to optimality. In terms of the search time, we do not see either a clear improvement nor a downgrade for the smaller datasets, however, for ELSA ^{gac}, it improves by 48% for `bnetflix.ts`, 43% for `bnetflix.test` and 35% for `bnetflix.valid`, compared to CPBayes. The more striking observation is the consistent reduction in the number of search nodes, even though the reduction *rate* is not as high as that for the search times. We explore 21% less nodes for `bnetflix.ts`, 16% for `bnetflix.test` and 21% for `bnetflix.valid`, compared to CPBayes. This indicates that the average time we spend at a search node increases with GAC, but the overhead of GAC clearly pays off as the datasets get larger.

[7] `http://urlearning.org`

Dataset	$\|V\|$	$\sum \|d_v\|$	CPBayes time	CPBayes nodes	ELSA ^{gac} time	ELSA ^{gac} nodes
mildew1000_BIC	35	126	0	0	0	0
lympho_BIC	19	143	0	0	0	0
water1000_BIC	32	159	0	1974	0	**1941**
mildew1000_BDe	35	166	0.2	16127	**0.1**	**9267**
hailfinder100_BIC	56	167	0	0	0	0
tumour_BIC	18	219	0	0	0	0
barley1000_BIC	48	244	**1.2**	99485	1.3	**99403**
shuttle_BIC	10	264	0	0	0	0
hepatitis_BIC	20	266	0	60	0	**30**
tumour_BDe	18	274	0	0	0	0
lung-cancer_BIC	57	292	**2.7**	97071	2.8	**95654**
lympho_BDe	19	345	0	163	0	**72**
hailfinder500_BIC	56	418	2.6	128134	**2.5**	**115682**
carpo100_BIC	60	423	27.8	1262025	**27.6**	**1234813**
horse_BDe	28	490	**0.7**	65969	0.8	**64879**
horse_BIC	28	490	0	758	0	**581**
hepatitis_BDe	20	501	0	87	0	**49**
insurance1000_BIC	27	506	0	168	0	**94**
adult_BIC	15	547	0	19	0	**18**
zoo_BIC	17	554	0	45	0	**16**
spectf_BIC	45	610	3.5	145928	**3.3**	**143400**
sponge_BIC	45	618	**3.2**	178474	3.7	**177891**
flag_BIC	29	741	0.6	40155	**0.5**	**37607**
vehicle_BIC	19	763	0	78	0	**28**
adult_BDe	15	768	0	43	0	**35**
insurance1000_BDe	27	792	0	211	0	**129**
shuttle_BDe	10	812	0	0	0	0
bands_BIC	39	892	0.4	16083	0.4	**15856**
alarm1000_BIC	37	1002	**145.7**	11497094	147.8	**11096565**
segment_BIC	20	1053	0	0	0	0
flag_BDe	29	1324	**14.0**	877874	14.7	**852999**
voting_BIC	17	1848	0	46	0	**39**
voting_BDe	17	1940	0	47	0	**35**
autos_BIC	26	2391	0	123	0	**86**
zoo_BDe	17	2855	0	84	0	**50**
vehicle_BDe	19	3121	0	321	0	**176**
letter_BIC	17	4443	0	83	0	**75**
soybean_BIC	36	5926	1.5	22394	1.5	**21775**
msnbc.ts	17	6298	0	73	0	**43**
segment_BDe	20	6491	0	90	0	**71**
mushroom_BIC	23	13025	0	59	0	**40**
wdbc_BIC	31	14613	**337.5**	5931690	337.8	**5913953**
msnbc.test	17	16594	0	94	0	**51**
letter_BDe	17	18841	0	95	0	**55**
msnbc.valid	17	20673	0	0	0	0
nltcs.ts	16	22156	0	0	0	0
autos_BDe	26	25238	0.1	563	0.1	**340**
kdd.ts	64	43584	†	-	†	-
nltcs.valid	16	47097	0	39	0	**28**
nltcs.test	16	48303	0	58	0	**32**
steel_BIC	28	93026	931.5	4805144	**874.6**	**4798161**
kdd.test	64	152873	†	-	†	-
kdd.valid	64	197546	†	-	†	-
mushroom_BDe	23	438185	5.7	183	**5.6**	**118**
plants.ts	69	164640	†	-	†	-
baudio.ts	69	371117	†	-	†	-
bnetflix.ts	100	446406	456.2	424503	**236.3**	**331636**
plants.test	111	520148	†	-	†	-
jester.ts	100	531961	†	-	†	-
accidents.ts	100	568160	†	-	†	-
plants.valid	111	684141	†	-	†	-
jester.test	100	770950	†	-	†	-
baudio.test	100	1016403	†	-	†	-
bnetflix.test	100	1103968	2475.1	1035226	**1402.0**	**862913**
baudio.valid	69	1235928	†	-	†	-
bnetflix.valid	111	1325818	146.8	48792	**94.4**	**38227**
accidents.test	100	1425966	†	-	†	-
jester.valid	100	1463335	†	-	†	-
accidents.valid	100	1617862	†	-	†	-

Table 3.2: Comparison of CPBayes and ELSA ^{gac} in terms of the total search time in seconds, and total number of search nodes. Time limit for datasets above the line is 1h, for the rest 10h. Datasets are sorted by increasing total domain size.

3.7 Conclusion

We have presented a new GAC propagator for the global acyclicity constraint for BNSL that runs in time $O(n^3 d)$, along with several ideas improving its practical performance. Although we explore a smaller average number of nodes per second, the strong inference provided by GAC undoubtedly leads to more effective decisions during the search, reducing the overall size of the search tree. This overall reduction cancels out the additional efforts entailed by GAC for the smaller datasets, and outweighs for the larger ones. A low-complexity algorithm providing tight lower bounds can help us explore an even smaller part of the search tree, and multiply the gain we obtain from GAC. In the next chapter, we will focus on this aspect.

CHAPTER 4

Polynomial Class of Violated Cluster Inequalities

Contents

4.1 Introduction

In this chapter, we derive a lower bound that is computationally cheaper than that computed by GOBNILP. One of the issues hampering the performance of CPBayes is that it computes relatively poor lower bounds at deeper levels of the search tree. Intuitively, as the parent set variable domains get reduced by removing values that are inconsistent with the current ordering, the lower bound computation discards more information about the current state of the problem, making it weaker. We address this and therefore achieve a better trade-off between the inference strength and speed of inference by adapting the branch-and-cut approach of GOBNILP. However, instead of finding all violated cluster inequalities that may improve the LP lower bound, we only identify a subset of them.

We give in Section 4.3 a polynomial-time algorithm that discovers a class of cluster cuts that provably improve the linear relaxation. In Section 4.4, we give a greedy algorithm for solving the linear relaxation, inspired by similar algorithms for MaxSAT and WCSPs, in particular the VAC algorithm. We incorporate all these algorithms in ELSA ^{gac} and call the resulting solver ELSA ^{cluster}. Finally, in Section 4.5, we demonstrate that ELSA ^{cluster} shows a significantly improved performance, both in the size of the search tree explored and in runtime.

4.2 Violated Clusters

As we have seen in Chapter 2, we are using the CP model of CPBayes to formulate the BNSL problem. We have a set V of *parent set variables* corresponding to the random variables, each v of which has a set of candidate parent sets, $PS(v)$. Each of these candidate parent sets $S \in PS(v)$ has an associated *score* $\sigma^v(S)$, i.e. cost. We try to find the optimal, i.e. acyclic and lowest-cost network where we assign each variable $v \in V$ a parent set $S \in PS(v)$. This objective is the same as that of the ILP model (4.1) used by GOBNILP:

$$\min \sum_{v \in V, S \in PS(v)} \sigma^v(S) x_{vS} \tag{4.1a}$$

$$s.t. \sum_{S \in PS(v)} x_{vS} = 1 \qquad \forall v \in V \tag{4.1b}$$

$$\sum_{v \in W, S \in PS^{-W}(v)} x_{vS} \geq 1 \qquad \forall W \subseteq V \tag{4.1c}$$

$$x_{vS} \in \{0, 1\} \qquad \forall v \in V, S \in PS(v) \tag{4.1d}$$

Let us now recall the following theorem [Jaakkola *et al.* 2010], on which the constraints (4.1c) for each $W \subseteq V$, denoted as $cons(W)$, and `acycChecker` (reappearing in Algorithm 7 for the sake of self-containment) are based:

Theorem 4.1. *Let G be a directed graph over vertices V and let $parents(v)$ be the parents of vertex v in the graph. G is acyclic if and only if for every non-empty subset, i.e. cluster $W \subseteq V$ there is at least one vertex $v \in W$ with $parents(v) \cap W = \{\}$.*

W is a *violated cluster* if the parent set of each vertex $v \in W$ intersects W. On the other hand, recall that `acycChecker` returns an order of all variables if the current set of domains of the parent set variables may produce an acyclic graph, or a partially completed order if the constraint is unsatisfiable, indicating there exist violated clusters.

4.3 Restricted Cluster Detection

Consider the linear relaxation of the ILP (4.1), restricted to a subset $\mathcal{W}$ of all valid cluster inequalities, i.e., with constraint (4.1d) replaced by $0 \leq x_{vS} \leq 1 \forall v \in V, S \in PS(v)$ and with constraint (4.1c) restricted only to clusters in $\mathcal{W}$. We denote this $LP_{\mathcal{W}}$. We exploit the following property of this LP.

Theorem 4.2. *Let $\hat{y}$ be a dual feasible solution of $LP_{\mathcal{W}}$ with dual objective o. Also, let $varsof(W)$ denote the 0/1 variables involved in $cons(W)$. Then, if W is a cluster such that $W \notin \mathcal{W}$ and the reduced cost rc of all variables $varsof(W)$*

Algorithm 7: Acyclicity Checker

```
1  Procedure acycChecker(prefix, V, D):
2      O ← prefix
3      support ← {}
4      changes ← true
5      i ← size(prefix)
6      while changes do
7          changes ← false
8          foreach v ∈ V \ O do
9              if ∃S ∈ D(v) s.t. S ⊆ O then
10                 O_i ← v
11                 support_v ← S
12                 changes ← true
13                 i ← (i + 1)
14     return O
```

is greater than 0, there exists a dual feasible solution $\hat{y}$ of $LP_{\mathcal{W}\cup\{W\}}$ with dual objective $o' \geq o + minrc(W)$ where

$$minrc(W) = \min_{x_{vS} \in varsof(W)} rc_{\hat{y}}(x_{vS}) \tag{4.2}$$

Proof. The only difference from $LP_{\mathcal{W}}$ to $LP_{\mathcal{W}\cup\{W\}}$ is the extra constraint $cons(W)$ in the primal and corresponding dual variable y_W. In the dual, y_W only appears in the dual constraints of the variables $varsof(W)$ and in the objective, always with coefficient 1. Under the feasible dual solution $\hat{y} \cup \{y_W = 0\}$, these constraints have slack at least $minrc(W)$, by the definition of reduced cost. Therefore, we can set $\hat{y} = \hat{y} \cup \{y_W = minrc(W)\}$, which remains feasible and has objective $o' = o + minrc(W)$, as required. □

Theorem 4.2 gives a class of cluster cuts, which we call RC-clusters, for reduced-cost clusters, guaranteed to improve the lower bound. Importantly, this requires only a feasible, perhaps sub-optimal, solution.

This theorem is strongly linked to the VAC algorithm for WCSPs [Cooper *et al.* 2010], since it also uses the idea of performing propagation on the subset of domains that have reduced cost 0. When we talked about VAC in Chapter 2, we said that asking

"*Does the CSP $Bool(P)$ have a solution?*"

is equivalent to asking

"*Does the WCSP P have a solution with a total cost of $c_\varnothing$?*".

If the answer to the first question is "no", then there exist a sequence of EPTs which increase $c_\varnothing$ when applied to P. Similarly, here, asking

"*Does there exist an acyclic network where each variable $v \in V$ is assigned a parent set $S \in PS(v)$ such that $rc_{\hat{y}}(x_{vS}) = 0$?*"

is equivalent to asking

"*Does there exist an acyclic network whose total cost is equal to the current lower bound?*"

Again, if the answer to the first question is "no", then the current lower bound can be improved, as we have seen in the proof of Theorem 4.2. As opposed to the VAC algorithm, our method is more lightweight, as it only performs propagation on the acyclicity constraint, but may give worse bounds. The bound update mechanism in the proof of Theorem 4.2 is also simpler than VAC and more akin to the "disjoint core phase" in core-guided MaxSAT solvers [Morgado *et al.* 2013].

Variable	Domain Value	Cost
v_0	$\{v_2\}$	0
v_1	$\{v_2, v_4\}$	0
	$\{\}$	6
v_2	$\{v_1, v_3\}$	0
	$\{\}$	10
v_3	$\{v_0\}$	0
	$\{\}$	5
v_4	$\{v_2, v_3\}$	0
	$\{v_3\}$	1
	$\{v_2\}$	2
	$\{\}$	3

Table 4.1: BNSL problem used as running example in this chapter.

Example 4.3 (Running example)**.** *Consider a BNSL problem with domains as shown in Table 4.1 and let $\mathcal{W} = \emptyset$. Then, $\hat{y} = 0$ leaves the reduced cost of every variable to exactly its primal objective coefficient. The corresponding $\hat{x}$ assigns 1 to variables with reduced cost 0 and 0 to everything else. These are both optimal solutions, with cost 0 and $\hat{x}$ is integral, so it is also a solution of the corresponding ILP. However, it is not a solution of the BNSL, as it contains several cycles, including $C = \{v_0, v_2, v_3\}$. The cluster inequality $cons(W)$ is violated in the primal and allows us to increase the dual bound.*

We consider the problem of discovering RC-clusters within the CP model of CPBayes. First, we introduce the notation $LP_{\mathcal{W}}(D)$ which is $LP_{\mathcal{W}}$ with the additional constraint $x_{vS} = 0$ for each $S \notin D(v)$. Conversely, $D^{rc}_{\mathcal{W},\hat{y}}$ is the set of domains minus values whose corresponding variable in $LP_{\mathcal{W}}(D)$ has non-zero reduced cost under $\hat{y}$, i.e., $D^{rc}_{\mathcal{W},\hat{y}} = D'$ where $D'(v) = \{S \mid S \in D(v) \wedge rc_{\hat{y}}(x_{vS}) =$

Algorithm 8: Lower bound computation with RC-clusters

```
1 Procedure lowerBoundRC(V, D, 𝒲):
2     ŷ ← dualSolve(LP_𝒲(D))
3     while true do
4         W ← V \ acycChecker(V, D^rc_{𝒲,ŷ})
5         if W = ∅ then
6             return ⟨cost(ŷ), 𝒲⟩
7         W ← minimise(W)
8         𝒲 ← 𝒲 ∪ {W}
9         ŷ ← dualImprove(ŷ, LP_𝒲(D), W)
```

$0\}$ with $rc(x_{vS})$ being the reduced cost of x_{vS} in $LP_{\mathcal{W}}(D)$ under $\hat{y}$. With this notation, for values $S \notin D(v)$, i.e., S is pruned in $D(v)$, $x_{vS} = 1$ is infeasible in $LP_{\mathcal{W}}(D)$, hence effectively $rc_{\hat{y}}(x_{vS}) = \infty$.

Theorem 4.4. *Given a collection of clusters $\mathcal{W}$, a set of domains D and $\hat{y}$, a feasible dual solution of $LP_{\mathcal{W}}(D)$, there exists an RC-cluster $W \notin \mathcal{W}$ if and only if $D^{rc}_{\mathcal{W},\hat{y}}$ does not admit an acyclic assignment.*

Proof. $\Rightarrow$ Let W be such a cluster. Since for all $x_{vS} \in varsof(W)$, none of these are in $D^{rc}_{\mathcal{W},\hat{y}}$, so $cons(W)$ is violated and hence there is no acyclic assignment.

$\Leftarrow$ Consider once again `acycChecker`, in Algorithm 7. When it fails to find a witness of acyclicity, it has reached a point where $O \subsetneq V$ and for the remaining variables $C = V \setminus O$, all allowed parent sets intersect W. So if `acycChecker` is called with $D^{rc}_{\mathcal{W},\hat{y}}$, all values in $varsof(W)$ have reduced cost greater than 0, so W is an RC-cluster. □

Theorem 4.4 shows that detecting unsatisfiability of $D^{rc}_{\mathcal{W},\hat{y}}$ is enough to find an RC-cluster. Its proof also gives a way to extract such a cluster from `acycChecker`.

Algorithm 8 shows how Theorem 4.2 and Theorem 4.4 can be used to compute a lower bound. It is given the current set of domains D and the current pool of clusters $\mathcal{W}$ as input. It first solves the dual of $LP_{\mathcal{W}}(D)$, potentially suboptimally. Then, it uses `acycChecker` iteratively to determine whether there exists an RC-cluster W under the current dual solution $\hat{y}$. If that cluster is empty, there are no more RC-clusters, and it terminates and returns a lower bound equal to the cost of $\hat{y}$ under $LP_{\mathcal{W}}(D)$ and an updated pool of clusters. Otherwise, it minimises W (see section 4.3.1), adds it to the pool of clusters, and solves the updated LP. It does this by calling $dualImprove$, which solves $LP_{\mathcal{W}}(D)$ exploiting the fact that only the cluster inequality $cons(W)$ has been added.

Example 4.5. *Continuing our example, consider the behaviour of* `acycChecker` *at line 4 of Algorithm 8 with domains $D^{rc}_{\emptyset,\hat{y}}$ after the initial dual solution $\hat{y} = 0$. Since the empty set has non-zero reduced cost for all variables,* `acycChecker` *fails to place any variable in the ordering and returns $order = \{\}$ as a cluster,*

hence $C = V$. We postpone the discussion of minimisation to Section 4.3.1, other than to observe that W can be minimised to $C_1 = \{v_1, v_2\}$ at line 7. To the primal LP, we add $cons(C_1)$ that involves the variables $x_{v_1\{\}}$ and $x_{v_2\{\}}$ whose reduced costs are 6 and 10, respectively. We set the dual variable of C_1 to 6 in the new dual solution $\hat{y}_1$. The reduced costs of $x_{1\{\}}$ and $x_{v_2\{\}}$ are decreased by 6 and, importantly, $rc_{\hat{y}_1}(x_{v_1\{\}}) = 0$. In the next iteration of `lowerBoundRC`*,* `acycChecker` *is invoked on $D^{rc}_{\{C_1\},\hat{y}_1}$ and returns the cluster $\{v_0, v_2, v_3, v_4\}$. This is minimised to $C_2 = \{v_0, v_2, v_3\}$. The parent sets in the domains of these variables that do not intersect C_2 are $x_{v_2\{\}}$ and $x_{v_3\{\}}$, whose reduced costs are $10 - 6 = 4$ and 5, respectively. As a result, $minrc(C_2) = 4$, so we add $cons(C_2)$ to the primal and we set the dual variable of C_2 to 4 in $\hat{y}_2$. This brings up the dual objective to 10. The reduced cost of $x_{v_2\{\}}$ is 0, so in the next iteration* `acycChecker` *runs on $D^{rc}_{\{C_1,C_2\},\hat{y}_2}$ and succeeds with the order $\{v_2, v_0, v_3, v_4, v_1\}$, so the lower bound cannot be improved further. This also happens to be the cost of the optimal structure.*

Theorem 4.6. `lowerBoundRC` *terminates but is not confluent.*

Proof. It terminates because there is a finite number of cluster inequalities and each iteration generates one. In the extreme, all cluster inequalities are in $\mathcal{W}$ and the test at line 5 succeeds, terminating the algorithm.

To see that it is not confluent, consider an example with 3 clusters $C_1 = \{v_1, v_2\}, C_2 = \{v_2, v_3\}$ and $C_3 = \{v_3, v_4\}$ and assume that the minimum reduced cost for each cluster is unit and comes from $x_{v_2\{4\}}$ and $x_{v_3\{1\}}$, i.e., the former value has minimum reduced cost for C_1 and C_2 and the latter for C_2 and C_3. Then, if minimisation generates first C_1, the reduced cost of $x_{v_3\{1\}}$ is unaffected by $dualImprove$, so it can then discover C_3, to get a lower bound of 2. On the other hand, if minimisation generates first C_2, the reduced costs of both $x_{v_2\{4\}}$ and $x_{v_3\{1\}}$ are decreased to 0 by $dualImprove$, so neither C_1 nor C_3 are RC-clusters under the new dual solution and the algorithm terminates with a lower bound of 1. □

Theorem 4.7. `lowerBoundRC` *runs in $O(n^2 dmin(UB - LB, \sum_{v \in V} d_v))$ time, where LB is the current lower bound, and UB is the current upper bound, and d_v is the domain size of variable v.*

Proof. The term $n^2 d$ comes from `acycChecker` which is called at each while-loop iteration. The number of iterations is bounded by how much we can possibly increase the lower bound. There are two extreme cases:

1. We make the reduced cost of exactly one value $S \in D(v)$ at a time, and we have at most $\sum_{v \in V} d_v$ of these values

2. We increase the lower bound by 1 until we reach the current upper bound

The number of iterations is bounded by the minimum of these two. □

4.3.1 Cluster Minimisation

It is crucial for the quality of the lower bound produced by `lowerBoundRC` that the RC-clusters discovered by `acycChecker` are minimised, as the following example shows. Empirically, omitting minimisation rendered the lower bound ineffective.

Example 4.8. *Suppose that we attempt to use* `lowerBoundRC` *without cluster minimisation. Then, we use the cluster given by* `acycChecker`*,* $C_1 = \{v_0, v_1, v_2, v_3, v_4\}$*. We have* $minrc(C_1) = 3$*, given from the empty parent set value of all variables. This brings the reduced cost of* $x_{v_4\{\}}$ *to 0. It then proceeds to find the cluster* $C_2 = \{v_0, v_1, v_2, v_3\}$ *with* $minrc(C_2) = 2$ *and decrease the reduced cost of* $x_{v_3\{\}}$ *to 0, then* $C_3 = \{v_0, v_1, v_2\}$ *with* $minrc(C_3) = 1$*, which brings the reduced cost of* $x_{v_1\{\}}$ *to 0. At this point,* `acycChecker` *succeeds with the order* $\{v_4, v_3, v_1, v_2, v_0\}$ *and* `lowerBoundRC` *returns a lower bound of 6, compared to 10 with minimisation. The order produced by* `acycChecker` *also disagrees with the optimum structure.*

Therefore, when we get an RC-cluster W at line 4 of algorithm 8, we want to extract a minimal RC-cluster (with respect to set inclusion) from W, i.e., a cluster $C' \subseteq C$, such that for all $\emptyset \subset C'' \subset C'$, C'' is not a cluster.

Minimisation problems like this are handled with an appropriate instantiation of QuickXPlain [Junker 2004]. These algorithms find a minimal subset of constraints, not variables. We can pose this as a constraint set minimisation problem by implicitly treating a variable as the constraint "this variable is assigned a value" and treating acyclicity as a hard constraint.

However, the property of being an RC-cluster is not monotone. For example, consider the variables $\{v_1, v_2, v_3, v_4\}$ and $\hat{y}$ such that the domains restricted to values with 0 reduced cost are $\{\{v_2\}\}, \{\{v_1\}\}, \{\{v_4\}\}, \{\{v_3\}\}$, respectively. Then $\{v_1, v_2, v_3, v_4\}$, $\{v_1, v_2\}$ and $\{v_3, v_4\}$ are RC-clusters. but $\{v_1, v_2, v_3\}$ is not because the sole value in the domain of v_3 does not intersect $\{v_1, v_2, v_3\}$. We instead minimise the set of variables that does not admit an acyclic solution and hence *contains* an RC-cluster. A minimal unsatisfiable set that contains a cluster is an RC-cluster, so this allows us to use a minimisation algorithm for monotone predicates, such as the variants of QuickXPlain. We focus on RobustXPlain, which is called the deletion-based algorithm in SAT literature for minimising unsatisfiable subsets [Marques-Silva & Mencía 2020]. The main idea of the algorithm is to iteratively pick a variable and categorise it as either appearing in all minimal subsets of W, in which case we mark it as necessary, or not, in which case we discard it. To detect if a variable appears in all minimal unsatisfiable subsets, we only have to test if omitting this variable yields a set with no unsatisfiable subsets, i.e., with no violated clusters. This is given in pseudocode in Algorithm 9. This exploits a subtle feature of `acycChecker` as described in Algorithm 7: if it is called with a subset of V, it does not try to place the missing variables in the order and allows parent sets to use these missing variables. Omitting variables from the set given to `acycChecker` acts as omitting the constraint that these variables be assigned a value. The complexity of `minimiseCluster` is $O(n^3 d)$, where $n = |V|$ and $d = \max_{v \in V} |D(v)|$.

Algorithm 9: Find a minimal RC-cluster subset of W

```
1  Procedure minimiseCluster(V, D, C):
2      N = ∅
3      while C ≠ ∅ do
4          Pick c ∈ C
5          C ← C \ {c}
6          C' ← V \ acycChecker(N ∪ W, D)
7          if C' = ∅ then
8              N ← N ∪ {c}
9          else
10             C ← C' \ N
11     return N
```

4.4 Solving the Cluster LP

Solving a linear program is in polynomial time, so in principle $dualSolve$ can be implemented using any of the commercial or free software libraries available for this. However, solving this LP using a general LP solver is too expensive in this setting. As a data point, solving the dataset `steel_BIC` with our modified solver took 25,016 search nodes and 45 seconds of search, and generated $5,869$ RC-clusters. Approximately 20% of search time was spent solving the LP using the greedy algorithm that we describe in this section. CPLEX took around 70 seconds to solve $LP_{\mathcal{W}}$ with these cluster inequalities once. We believe this is due to the fact that we have nearly 100000 $x_{v,S}$ variables (see column 3 of Table 4.2 for this dataset.) and that the number of variables involved in each cluster inequality is large. While this data point is not proof that solving the LP exactly is too expensive, it is a pretty strong indicator. We have also not explored nearly linear time algorithms for solving positive LPs [Allen-Zhu & Orecchia 2015].

Our greedy algorithm is derived from Theorem 4.2. Observe first that $LP_{\mathcal{W}}$ with $\mathcal{W} = \emptyset$, i.e., only with constraints (4.1b) has optimal dual solution $\hat{y}_0$ that assigns the dual variable y_v of $\sum_{S \in PS(v)} x_{vS} = 1$ to $\min_{S \in PS(v)} \sigma^v(S)$. That leaves at least one of $x_{vS}, S \in PS(v)$ with reduced cost 0 for each $v \in V$. $dualSolve$ starts with $\hat{y}_0$ and then iterates over $\mathcal{W}$. Given $\hat{y}_{i-1}$ and a cluster W, it sets $\hat{y}_i = \hat{y}_{i-1}$ if W is not an RC-cluster. Otherwise, it increases the lower bound by $c = minrc(W)$ and sets $\hat{y}_i = \hat{y}_{i-1} \cup \{y_W = c\}$. It remains to specify the order in which we traverse $\mathcal{W}$.

We sort clusters by increasing size $|C|$, breaking ties by decreasing minimum cost of all original parent set values in $varsof(W)$. This favours finding non-overlapping cluster cuts with high minimum cost. In Section 4.5, we give experimental evidence that this computes better lower bounds.

$dualImprove$ can be implemented by discarding previous information and calling $dualSolve(LP_{\mathcal{W}}(D))$. Instead, it uses the RC-cluster W to update the

solution without revisiting previous clusters.

In terms of implementation, we store $varsof(W)$ for each cluster, not $cons(W)$. During $dualSolve$, we maintain the reduced costs of variables rather than the dual solution, otherwise computing each reduced cost would require iterating over all cluster inequalities that contain a variable. Specifically, we maintain $\Delta_{v,S} = \sigma^v(S) - rc_{\hat{y}}(x_{vS})$. In order to test whether a cluster W is an RC-cluster, we need to compute $minrc(W)$. To speed this up, we associate with each stored cluster a *support pair* (v, S) corresponding to the last minimum cost found. If $rc_{\hat{y}}(v, S) = 0$, the cluster is not an RC-cluster and is skipped. Moreover, parent set domains are sorted by increasing score $\sigma^v(S)$, so $S \succ S' \iff \sigma^v(S) > \sigma^v(S')$. We also maintain the maximum amount of cost transferred to the lower bound, $\Delta_v^{max} = \max_{S \in D(v)} \Delta_{v,S}$ for every $v \in V$. We stop iterating over $D(v)$ as soon as $\sigma^v(S) - \Delta_v^{max}$ is greater than or equal to the current minimum because $\forall S' \succ S, \sigma^v(S') - \Delta_{v,b} \geq \sigma^v(S) - \Delta_v^{max}$. We provide a detailed empirical analysis on the productivity of clusters in Section 4.5.2.3.

4.5 Experimental Results

For the experiments of this chapter, we take ELSA ^{gac} as a base, integrate all the algorithms presented herein, and call the resulting solver ELSA ^{cluster}. We produce another variant called ELSA ^{chrono}, which is the same as ELSA ^{cluster} except for the fact that it leaves clusters in chronological order, rather than the heuristic ordering presented in Section 4.4. To sum up, we carry out a comparative analysis between the following solvers:

- GOBNILP v1.6.3 using SCIP v3.2.1 with CPLEX v12.8.0
- CPBayes v1.1 compatible with $n > 64$
- ELSA ^{gac}
- ELSA ^{cluster}
- ELSA ^{chrono}

We have the same set of 51 medium ($|V| < 64$) and 18 large ($64 \leq |V| < 128$) datasets from the previous chapter. For medium datasets, as well as `kdd.ts`, `kdd.test`, and `kdd.valid`, we had a 1-hour CPU time limit, and a 10-hour CPU time limit for the remaining 15 large datasets. In addition to that, in Section 4.5.2, we share and study in detail the results for all 18 large datasets, and the 23 very large ($|V| > 128$) datasets which we ran with a time limit of 90 hours.

4.5.1 Evaluation

Dataset	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
carpo100_BIC	60	423	**0.6**	79.8 (27.8)	79.9 (27.6)	52.6 (0.1)	56.9 (0.1)
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	181.1 (147.8)	34.4 (1.0)	39.4 (3.9)
flag_BDe	29	1324	4.36	14.9 (14)	15.7 (14.7)	**1.0 (0.2)**	1.6 (0.7)
wdbc_BIC	31	14613	99.77	390.6 (337.5)	396.1 (337.8)	**56 (2.4)**	63.7 (4.7)
kdd.ts	64	43584	**327.63**	†	†	1452.3 (274.6)	2257.9 (960.0)
steel_BIC	28	93026	†	981.2 (931.5)	924.8 (874.6)	**124.2 (71.8)**	196.6 (139.6)
kdd.test	64	152873	1521.73	†	†	**1594.3 (224.4)**	1863.6 (246.1)
mushroom_BDe	23	438185	†	131.2 (5.7)	**130.8 (5.6)**	182.6 (58.9)	225.3 (66.6)
bnetflix.ts	100	446406	†	673.4 (456.2)	**453.2 (236.3)**	2103.1 (1900.9)	2482.1 (2727.0)
plants.test	111	520148	†	†	†	**28049.6 (26312.9)**	†
jester.ts	100	531961	†	†	†	**21550.5 (21003.7)**	29868.0 (29251.5)
accidents.ts	100	568160	**1273.96**	†	†	2302.2 (930)	2887.7 (1280.2)
plants.valid	111	684141	†	†	†	**17801.6 (14080.2)**	†
jester.test	100	770950	†	†	†	**30186.8 (29455)**	†
bnetflix.test	100	1103968	†	2725.6 (2475.1)	**1651.7 (1402)**	10333.1 (10096.5)	10079.1 (9799.5)
bnetflix.valid	111	1325818	†	511.7 (146.8)	**457.7 (94.4)**	10871.7 (10527.7)	10749.1 (10363.3)
accidents.test	100	1425966	4975.64	†	†	**3641.7 (680.7)**	4290.7 (866.0)

Table 4.2: Comparison of ELSA ^{cluster} against GOBNILP, CPBayes, ELSA ^{gac}, and ELSA ^{chrono}, in terms of total running (and search) time in seconds. Time limit for the blue datasets at the end is 10 hours, and for the rest it is 1 hour. Datasets are sorted by increasing total domain size for each time limit category. For CPBayes as well as all variants of ELSA we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

We compare the runtime to solve each dataset to optimality with GOBNILP, CPBayes, ELSA ^{gac}, ELSA ^{cluster}, and ELSA ^{chrono} in Table 4.2. The preprocessing time for CPBayes as well as all variants of ELSA is roughly the same; this is why for them, we report in parentheses and compare the time spent during search only.

In Table 4.2, we exclude the datasets which were solved within the time limit by GOBNILP and have a search time of less than 10 seconds for CPBayes and all variants of ELSA. We also exclude 8 datasets that were not solved to optimality by any method. This leaves us 17 "more interesting" results to analyse here out of 69 total. The results in the same format for all 69 datasets can be seen in Table 4.5 at the end of this chapter.

Comparison to GOBNILP. CPBayes was already proven to be competitive to GOBNILP [Van Beek & Hoffmann 2015]. Our results in Table 4.2 confirm this while showing that neither is clearly better. When it comes to our solver ELSA ^{cluster}, all datasets solved within the time limit by GOBNILP are solved, unlike CPBayes. On top of that, ELSA ^{cluster} solves 9 more datasets optimally than GOBNILP.

Comparison to ELSA ^{gac}. ELSA ^{cluster} takes several orders of magnitude less search time than ELSA ^{gac} to optimally solve most datasets, the only exception being `bnetflix.valid`. For `bnetflix.valid`, the number of search nodes went down from $38,227$ to $6,834$ ($\downarrow 82\%$) for `bnetflix.valid`, the search time went up from 457.7 to 10,871.7 ($\uparrow 2275\%$). We should note that the average domain size for this dataset is nearly 12,000. As a result, the total number of domain value iterations carried out during `lowerBoundRC` is also particularly large: 244,154,531. Coming back to the overall picture, ELSA ^{cluster} proved optimality for 8 more datasets than ELSA ^{gac} within the time limit.

Comparison to ELSA ^{chrono}. We see that the cluster ordering heuristic improves the bounds computed by our greedy dual LP algorithm significantly. Compared to not ordering the clusters, we see an improved search time for all datasets (except for `bnetflix.test` and `bnetflix.valid`, and the decrease is negligible). On top of that, 3 more datasets solved to optimality.

4.5.2 Further Analysis on Large and Very Large Datasets

We made a longer run with a 90-hour CPU-time limit for 18 large and 23 very large datasets on a cluster of Intel Xeon E5-2683 v4 at 2.10 GHz and 128 GB of RAM. We used the following parameters: $l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$ for partition lower bound min/max sizes l_{min}, l_{max}, and local search min/max number of restarts r_{min}, r_{max}. *E.g.,* we ran `./elsa -B 20 -r 15 -R 30 -t 324000 kdd.test.jkl` for solving `kdd.test` within the time limit. We compared ELSA ^{cluster} with GOBNILP with default parameters, except optimality gap set to zero, and CPBayes.

4.5.2.1 Quality of Lower Bounds and Time per Node

We measured the quality of the initial lower bound lb found at the root node of the search tree as the relative distance $\frac{o-lb}{o}$ to the best solution o found by any method within the 90-hour CPU-time limit. Results are given in Table 4.3.

GOBNILP usually produced the best lower bounds. CPBayes got the worst ones and ELSA ^{cluster} made good progress towards GOBNILP but still remained below except a few exceptions (GOBNILP stopped at the root node due to exhausted memory for `accidents.valid`, `dna`, `kosarek.valid`, `msweb.test`, and `msweb.valid`).

In Figure 4.1, we show the mean CPU-time per search node when information was available for a subset of the 41 large and very large datasets. We sort the data by increasing CPU-time independently for each solver. Here, GOBNILP is several orders of magnitude slower than CPBayes and our approach. ELSA ^{cluster} was usually slower than CPBayes, up to 457.4 times for `bnetflix.test`, except on a few cases, *e.g.,* ELSA ^{cluster} was 9.7 faster than CPBayes on `pumsb_star.test`.

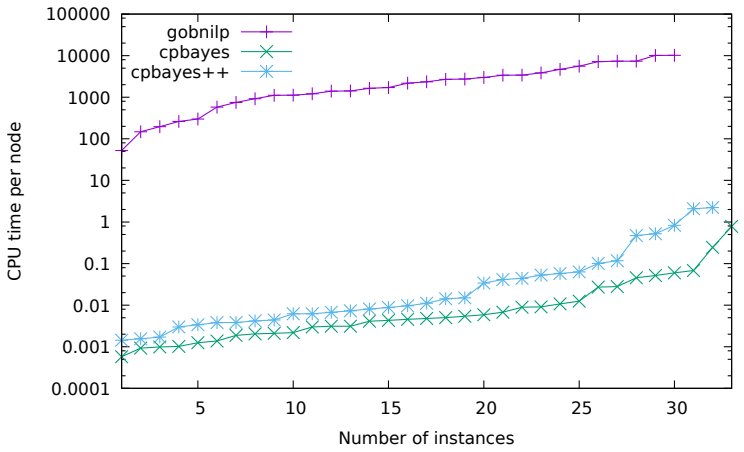


Figure 4.1: CPU-time (in seconds) per search node on a subset of the 41 large and very large datasets.

In Figure 4.2, we compare the number of nodes in the search tree for CPBayes and ELSA ^{cluster} on datasets solved by both of them. Because the number of commonly solved datasets is too small for large and very large datasets, we do this comparison on the medium datasets with less than 64 variables. The points below the line $y = x$ indicate a smaller search tree for ELSA ^{cluster}. The reduction in the search tree size is very clear.

4.5.2.2 Quality of Solutions and Optimality Proofs

We measured the quality of a BNSL solution s as the relative distance $\frac{s-o}{o}$ to the best solution o found by any method within the 90-hour CPU-time limit. The results are given in Table 4.4. Recall that both CPBayes and ELSA ^{cluster} have a local

Problem	$\|V\|$	$\sum_{v \in \mathbf{V}} \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}
kdd.ts	64	43584	⋆**0.03%**	2.51%	⋆0.50%
kdd.test	64	152873	⋆**0.02%**	⋆1.75%	⋆0.20%
plants.ts	69	164640	⋆**2.42%**	35.01%	7.28%
kdd.valid	64	197546	⋆**0.13%**	2.60%	⋆0.49%
baudio.ts	100	371117	**2.00%**	11.34%	⋆3.63%
pumsb_star.ts	163	394992	⋆**0.19%**	24.69%	0.46%
tretail.ts	135	435976	⋆0.17%	0.48%	**0.07%**
bnetflix.ts	100	446406	**3.23%**	⋆20.86%	⋆8.73%
plants.test	69	520148	**2.58%**	34.32%	⋆5.98%
jester.ts	100	531961	**4.20%**	13.03%	⋆6.44%
kosarek.ts	190	556189	⋆**0.00%**	2.13%	0.44%
accidents.ts	111	568160	⋆**0.00%**	8.36%	⋆0.38%
plants.valid	69	684141	**2.57%**	34.61%	⋆6.66%
msweb.ts	294	732213	**0.25%**	1.83%	0.62%
diabetes-5000	413	754563	**0.26%**	0.67%	0.67%
jester.test	100	770950	**4.76%**	13.29%	⋆6.37%
tretail.test	135	897478	**2.88%**	4.44%	3.16%
baudio.test	100	1016403	**2.22%**	11.92%	⋆3.10%
pumsb_star.test	163	1034955	23.79%	68.12%	**21.61%**
tretail.valid	135	1087404	**0.67%**	2.60%	2.60%
bnetflix.test	100	1103968	**5.49%**	⋆24.48%	⋆11.69%
kosarek.test	190	1192386	**1.06%**	4.38%	2.60%
baudio.valid	100	1235928	**2.37%**	12.09%	⋆3.32%
pumsb_star.valid	163	1271525	33.69%	75.89%	**21.63%**
kosarek.valid	190	1312600	**2.20%**	5.84%	5.84%
bnetflix.valid	100	1325818	**5.38%**	⋆25.26%	⋆11.24%
accidents.test	111	1425966	⋆**0.00%**	20.03%	⋆3.71%
jester.valid	100	1463335	**5.64%**	17.03%	⋆7.63%
msweb.test	294	1597487	0.53%	3.81%	**0.37%**
accidents.valid	111	1617862	**3.57%**	34.77%	7.83%
dna.ts	180	1849860	**1.07%**	5.53%	1.52%
msweb.valid	294	1869374	**0.78%**	3.59%	3.59%
tmovie.ts	500	1918883	**35.15%**	-	-
pigs-5000	441	1984359	**24.50%**	-	-
dna.test	180	2019003	**1.07%**	5.66%	⋆1.62%
book.ts	500	2071722	**7.56%**	-	-
dna.valid	180	2561134	3.90%	9.92%	**3.72%**
tmovie.valid	500	2610026	**49.06%**	-	-
tmovie.test	500	2778556	**53.78%**	-	-
book.test	500	2794588	**15.08%**	-	-
book.valid	500	3020475	**20.24%**	-	-

Table 4.3: Quality of initial lower bounds on 18 large and 23 very large datasets. The best method is shown in bold. '⋆': optimum proved, '-': no lower bound reported.

search procedure in preprocessing. It helps the search find good initial solutions.

Concerning optimality proofs, ELSA ^{cluster} solved 17 datasets, GOBNILP solved 9, and CPBayes 4 only. GOBNILP ran out-of-memory (128GB) on 11 datasets whereas ELSA ^{cluster} took less than 8GB on all the benchmark.

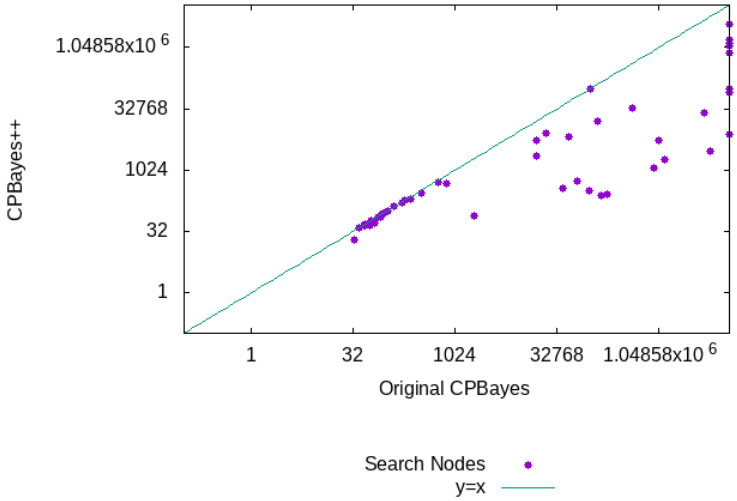


Figure 4.2: Number of search tree nodes for CPBayes and ELSA ^{cluster} on medium datasets with less than 64 variables.

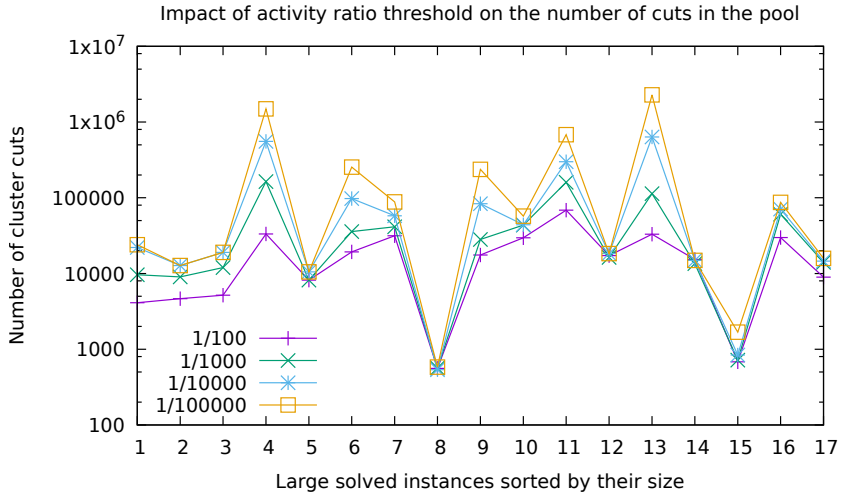


Figure 4.3: Number of cluster cuts in the pool at the end of the search when solving large datasets depending on the activity ration threshold.

Problem	$\|V\|$	$\sum_{v \in \mathbf{V}} \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}
kdd.ts	64	43584	⋆**0.00%**	0.17%	⋆**0.00%**
kdd.test	64	152873	⋆**0.00%**	⋆**0.00%**	⋆**0.00%**
plants.ts	69	164640	⋆**0.00%**	3.02%	1.48%
kdd.valid	64	197546	⋆**0.00%**	0.16%	⋆**0.00%**
baudio.ts	100	371117	1.62%	1.29%	⋆**0.00%**
pumsb_star.ts	163	394992	⋆**0.00%**	4.76%	4.76%
tretail.ts	135	435976	⋆**0.00%**	0.29%	0.29%
bnetflix.ts	100	446406	0.01%	⋆**0.00%**	⋆**0.00%**
plants.test	69	520148	0.01%	2.61%	⋆**0.00%**
jester.ts	100	531961	10.29%	0.50%	⋆**0.00%**
kosarek.ts	190	556189	⋆**0.00%**	1.62%	1.62%
accidents.ts	111	568160	⋆**0.00%**	0.37%	⋆**0.00%**
plants.valid	69	684141	1.74%	2.40%	⋆**0.00%**
msweb.ts	294	732213	9.87%	**0.00%**	**0.00%**
diabetes-5000	413	754563	2.86%	**0.00%**	**0.00%**
jester.test	100	770950	7.45%	0.18%	⋆**0.00%**
tretail.test	135	897478	6.93%	0.03%	**0.00%**
baudio.test	100	1016403	13.69%	0.92%	⋆**0.00%**
pumsb_star.test	163	1034955	102.62%	**0.00%**	**0.00%**
tretail.valid	135	1087404	17.60%	0.38%	**0.00%**
bnetflix.test	100	1103968	0.41%	⋆**0.00%**	⋆**0.00%**
kosarek.test	190	1192386	†15.70%	**0.00%**	**0.00%**
baudio.valid	100	1235928	12.46%	0.49%	⋆**0.00%**
pumsb_star.valid	163	1271525	155.68%	**0.00%**	**0.00%**
kosarek.valid	190	1312600	†25.15%	**0.00%**	**0.00%**
bnetflix.valid	100	1325818	7.07%	⋆**0.00%**	⋆**0.00%**
accidents.test	111	1425966	⋆**0.00%**	1.71%	⋆**0.00%**
jester.valid	100	1463335	7.86%	0.68%	⋆**0.00%**
msweb.test	294	1597487	†14.99%	**0.00%**	**0.00%**
accidents.valid	111	1617862	†433.17%	0.32%	**0.00%**
dna.ts	180	1849860	†16.82%	**0.00%**	**0.00%**
msweb.valid	294	1869374	†14.82%	**0.00%**	**0.00%**
tmovie.ts	500	1918883	**0.00%**	-	-
pigs-5000	441	1984359	†**0.00%**	-	-
dna.test	180	2019003	†22.33%	0.51%	⋆**0.00%**
book.ts	500	2071722	**0.00%**	-	-
dna.valid	180	2561134	†25.77%	**0.00%**	**0.00%**
tmovie.valid	500	2610026	**0.00%**	-	-
tmovie.test	500	2778556	**0.00%**	-	-
book.test	500	2794588	†**0.00%**	-	-
book.valid	500	3020475	†**0.00%**	-	-

Table 4.4: Solution quality on 18 large and 23 very large datasets. The best method is shown in bold. '⋆': optimum found, '†': out-of-memory, '-': no solution found.

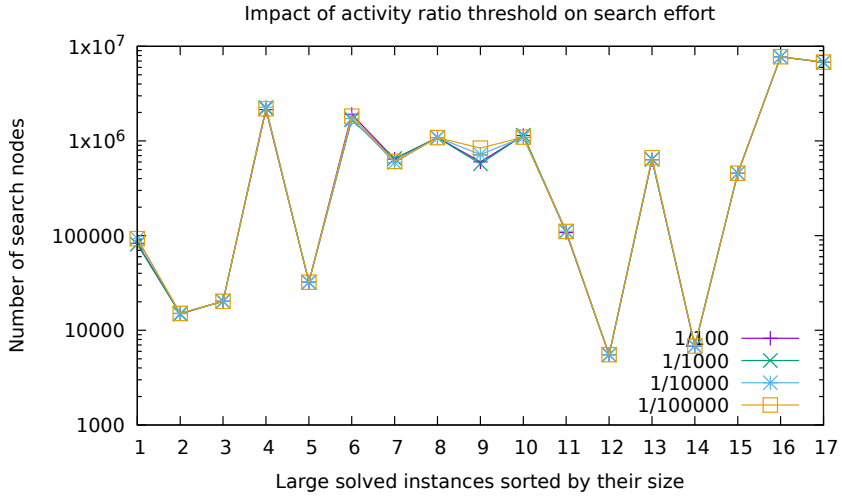


Figure 4.4: Number of search nodes for solving large datasets depending on the activity ration threshold.

4.5.2.3 Impact of the Cluster Pool Activity Threshold

In practice, on very large datasets 97.6% of unproductive clusters are detected by support pairs and 8.6% of the current domains are visited for the rest. To keep a bounded-memory cluster pool, we discard frequently unproductive clusters. We throw away large clusters with a productive ratio $\frac{\#productive}{\#productive+\#unproductive}$ smaller

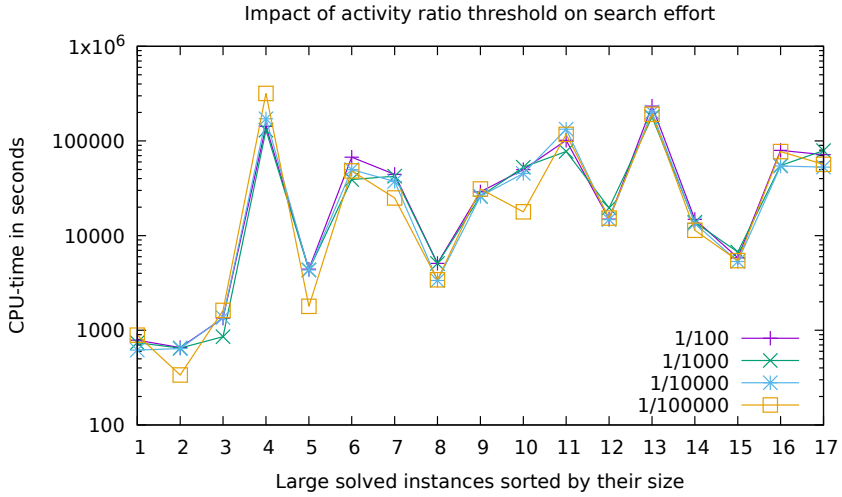


Figure 4.5: CPU-time for solving large datasets depending on the activity ratio threshold.

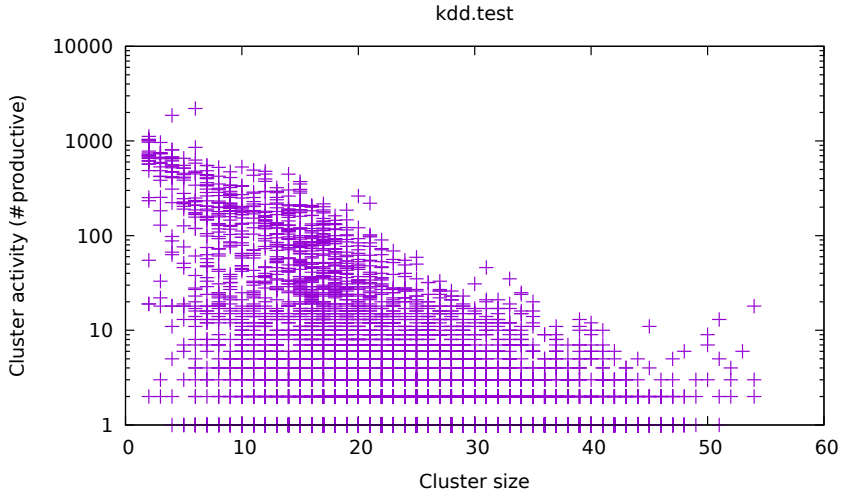


Figure 4.6: Cluster activity ($\#productive$) w.r.t. size for $9,018$ cuts kept in the cluster pool at the end of the search, after $14,840$ search nodes in 613 seconds on `kdd.test` with 64 random variables.

than $\frac{1}{1{,}000}$. Clusters of size 10 or less are always kept because they are often more productive and their number is bounded.

We carried out one last experiment to see if the activity threshold has an impact on the overall performance of our approach. Figure 4.3 gives the total number of cluster cuts in the pool at the end of the search for 17 solved datasets. Let a_W be the number of times cluster W has improved the lower bound. We found that keeping W in the cluster pool during $100 \times a_W$ or up to $100,000 \times a_W$ search nodes does not change the whole search effort in terms of total search nodes to solve a problem (Figure 4.4). However it may increase CPU-time, but this could also be due to performance fluctuations of the cluster (Figure 4.5).

By default, we choose an activity threshold of $\frac{1}{1{,}000}$, *i.e.* keeping each cluster W during $1,000 \times a_W$ search nodes. Clusters of size less than or equal to 10 were always kept. Such clusters are often more productive and their number is bounded, see for example cluster activities in `kdd.test` (Figure 4.6). On 41 large and very large datasets, the number of cuts found in preprocessing ranged from 328 (`accidents.test`) to $4,011$ (`kosarek.valid`) with a mean of $2,011.4$. At the end of the search, or the 90-hour CPU-time limit, there were from 85 (`pumsb_star.valid`) to $163,710$ (`baudio.ts`) clusters in the pool with a mean of $26,638.3$ clusters.

4.6 Conclusion

We have presented a new set of inference techniques for BNSL using constraint programming, centred around the expression of the acyclicity constraint. These

new techniques exploit and improve on previous work on linear relaxations of the acyclicity constraint and the associated propagator. The resulting solver ELSA ^{cluster} explores a different trade-off on the axis of strength of inference versus speed, with GOBNILP on one extreme and CPBayes on the other. We showed experimentally that the trade-off we achieve is a better fit than either extreme, as our solver ELSA ^{cluster} outperforms both GOBNILP and CPBayes.

The major obstacle towards better scalability to larger datasets is the fact that domain sizes grow exponentially with the number of variables, which is to some degree unavoidable (like the `bnetflix.valid` case). Therefore, in the next chapter, will focus on exploiting the structure of these domains to improve performance.

Table 4.5: Comparison of ELSA ^{cluster} against GOBNILP, CPBayes, ELSA ^{gac}, and ELSA ^{chrono}, in terms of total running (and search) time in seconds. Time limit for the blue datasets at the end is 10 hours, and for the rest it is 1 hour. Datasets are sorted by increasing total domain size for each time limit category. For CPBayes as well as all variants of ELSA we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
mildew1000_BIC	35	126	0.37	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
lympho_BIC	19	143	0.16	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
water1000_BIC	32	159	0.16	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
mildew1000_BDe	35	166	0.5	0.5 (0.2)	0.5 (0.1)	0.4 (0.1)	0.6 (0.1)
hailfinder100_BIC	56	167	0.43	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
tumour_BIC	18	219	0.18	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
barley1000_BIC	48	244	0.29	1.5 (1.2)	1.5 (1.3)	1.6 (1.4)	1.9 (1.6)
shuttle_BIC	10	264	1.81	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hepatitis_BIC	20	266	0.63	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.4 (0.0)
tumour_BDe	18	274	0.39	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
lung-cancer_BIC	57	292	0.57	3.8 (2.7)	3.9 (2.8)	1.1 (0.0)	1.4 (0.0)
lympho_BDe	19	345	0.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
hailfinder500_BIC	56	418	0.36	3.4 (2.6)	3.3 (2.5)	1.2 (0.5)	1.2 (0.4)
carpo100_BIC	60	423	**0.6**	79.8 (27.8)	79.9 (27.6)	52.6 (0.1)	56.9 (0.1)
horse_BDe	28	490	0.71	1.2 (0.7)	1.4 (0.8)	0.6 (0.0)	0.6 (0.0)
horse_BIC	28	490	0.99	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.5 (0.0)
hepatitis_BDe	20	501	0.93	0.4 (0.0)	0.4 (0.0)	0.5 (0.0)	0.4 (0.0)
insurance1000_BIC	27	506	**0.51**	32.4 (0.0)	32.6 (0.0)	32.8 (0.0)	33.9 (0.0)
adult_BIC	15	547	0.33	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
zoo_BIC	17	554	0.42	0.1 (0.0)	0.0 (0.0)	0.1 (0.0)	0.1 (0.0)

Table 4.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
spectf_BIC	45	610	1.52	4.3 (3.5)	4.1 (3.3)	**0.8 (0.0)**	1.2 (0.1)
sponge_BIC	45	618	1.45	5.0 (3.2)	5.5 (3.7)	1.8 (0.0)	2.0 (0.0)
flag_BIC	29	741	1.28	1.1 (0.6)	1.0 (0.5)	0.4 (0.0)	0.5 (0.0)
vehicle_BIC	19	763	1.48	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
adult_BDe	15	768	0.67	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
insurance1000_BDe	27	792	**0.7**	34.0 (0.0)	34.2 (0.0)	34.3 (0.0)	37.5 (0.0)
shuttle_BDe	10	812	4.03	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
bands_BIC	39	892	3.96	0.7 (0.4)	0.7 (0.4)	0.6 (0.2)	0.5 (0.2)
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	181.1 (147.8)	34.4 (1.0)	39.4 (3.9)
segment_BIC	20	1053	4.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
flag_BDe	29	1324	4.36	14.9 (14)	15.7 (14.7)	**1.0 (0.2)**	1.6 (0.7)
voting_BIC	17	1848	1.87	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
voting_BDe	17	1940	1.91	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.1 (0.0)
autos_BIC	26	2391	**11.83**	17.2 (0.0)	17.3 (0.0)	19.2 (0.0)	18.3 (0.0)
zoo_BDe	17	2855	19.02	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
vehicle_BDe	19	3121	5.68	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
letter_BIC	17	4443	3.95	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
soybean_BIC	36	5926	**40.19**	45.2 (1.5)	45.4 (1.5)	50.8 (3.0)	54.3 (5.7)
msnbc.ts	17	6298	8.61	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.3 (0.0)
segment_BDe	20	6491	29.15	0.4 (0.0)	0.4 (0.0)	0.5 (0.1)	0.6 (0.1)
mushroom_BIC	23	13025	1561.82	4.1 (0.0)	4.2 (0.0)	4.3 (0.2)	4.4 (0.2)
wdbc_BIC	31	14613	99.77	390.6 (337.5)	396.1 (337.8)	**56.0 (2.4)**	63.7 (4.7)
msnbc.test	17	16594	57.96	0.4 (0.0)	0.4 (0.0)	0.5 (0.1)	0.5 (0.1)
letter_BDe	17	18841	111.73	0.4 (0.0)	0.4 (0.0)	0.7 (0.3)	0.7 (0.3)

Table 4.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
msnbc.valid	17	20673	23.14	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
nltcs.ts	16	22156	15.3	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
autos_BDe	26	25238	715.89	146.6 (0.1)	146.9 (0.1)	**145.8 (0.8)**	156.3 (0.8)
kdd.ts	64	43584	**327.63**	†	†	1452.3 (274.6)	2257.9 (960.0)
nltcs.valid	16	47097	40.91	0.9 (0.0)	0.9 (0.0)	1.0 (0.1)	1.0 (0.1)
nltcs.test	16	48303	56.58	1.0 (0.0)	1.0 (0.0)	1.2 (0.3)	1.2 (0.3)
steel_BIC	28	93026	†	981.2 (931.5)	924.8 (874.6)	**124.2 (71.8)**	196.6 (139.6)
kdd.test	64	152873	**1521.73**	†	†	1594.3 (224.4)	1863.6 (246.1)
kdd.valid	64	197546	†	†	†	†	†
mushroom_BDe	23	438185	†	131.2 (5.7)	**130.8 (5.6)**	182.6 (58.9)	225.3 (66.6)
plants.ts	69	164640	†	†	†	†	†
baudio.ts	69	371117	†	†	†	†	†
bnetflix.ts	100	446406	†	673.4 (456.2)	**453.2** (236.3)	2103.1 (1900.9)	2482.1 (2727.0)
plants.test	111	520148	†	†	†	**28049.6 (26312.9)**	†
jester.ts	100	531961	†	†	†	**21550.5 (21003.7)**	29868.0 (29251.5)
accidents.ts	100	568160	**1273.96**	†	†	2302.2 (930)	2887.7 (1280.2)
plants.valid	111	684141	†	†	†	**17801.6 (14080.2)**	†
jester.test	100	770950	†	†	†	**30186.8 (29455)**	†
baudio.test	100	1016403	†	†	†	†	†
bnetflix.test	100	1103968	†	2725.6 (2475.1)	**1651.7** (1402.0)	10333.1 (10096.5)	10079.1 (9799.5)
baudio.valid	69	1235928	†	†	†	†	†
bnetflix.valid	111	1325818	†	511.7 (146.8)	**457.7** (94.4)	10871.7 (10527.7)	10363.3 (10749.1)
accidents.test	100	1425966	4975.64	†	†	**3641.7 (680.7)**	4290.7 (866.0)
jester.valid	100	1463335	†	†	†	†	†

Table 4.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
accidents.valid	100	1617862	†	†	†	†	†

CHAPTER 5

Binary Decision Trees For BNSL

Contents

5.1 Introduction

Let us recall the asymptotic complexity of the algorithms we have seen in Chapters 3 and 4:

- `Acyclicity-GAC-n3d`: $O(n^3d)$
- `lowerBoundRC`: $O(n^2dmin(UB - LB, \sum_{v \in V} d_v))$

These all have a factor of d, i.e. the largest domain size in a BNSL problem. We know that in theory, d is bounded by $2^{(n-1)}$, and in practice, it is bounded by some integer m, which is indeed smaller than $2^{(n-1)}$, but much larger than n in most cases.

This limitation on the practical performance of our algorithms led us to think of ways to exploit the BNSL specific features. Possibly the most striking feature is the fact that the domain values are not some arbitrary values, but are *sets*. Hence, we can use existing data structures for sets of sets to achieve better performance. We should, however, acknowledge that the options are *not* unrestricted, as the data structure in question needs to be flexible to allow value removals and retrievals, capable of taking into account the cost information, and answering queries like

- Does there exist a parent set value that is a subset of a given set S?
- Does there exist a parent set value that is a subset of S with an associated cost of 0?
- Does there exist a parent set value that is a superset of a given set S?

In order to answer one of these queries, if we iterate over each value in the domain of a variable, depending on our luck we can stop at the first or the last value we hit. Therefore, especially for large BNSL problems, answering these queries with a better complexity than $O(d)$ can potentially be huge time gain. In this chapter, we utilise Binary Decision Trees for this purpose. At the end, we see that even a rather naive implementation can result in a significant performance improvement for some datasets.

5.2 Binary Decision Trees

A Binary Decision Tree (BDT) is a decision support tool that models a set of decisions and their outcomes in the form of a rooted binary tree. They provide a way to run queries with conditional control statements. The branches in a BDT correspond to decision-making steps that lead to a desired result.

BDTs are a popular instrument in Machine Learning (ML) [Cardie 1993, Elaidi *et al.* 2018] with plenty of practical uses, a typical one being *classification* [Jensen & Arnspang 1999]. Classification is the task of sorting an input dataset based on some given features. Each node in a BDT represents a feature to be classified, and each branch represents a value that the node can assume. As might be expected, the branches in a BDT may represent either *true* or *false*.

In Figure 5.1, we have a BDT where we classify a set of shapes. These shapes may belong to a *class* such as *circle, ellipse, square, rectangle*. Each ellipse denotes a node and next to each node, we have a cloud showing the input set of that node. If a node classifies its input set with respect to a feature, then we write that feature inside the node. We create a dashed branch from a node if there is at least one

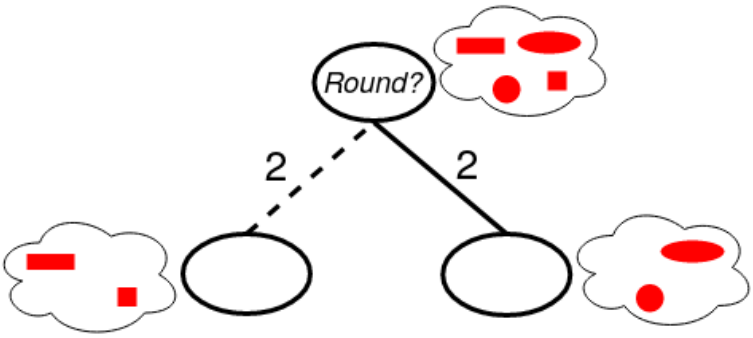


Figure 5.1: A BDT which is not fully grown.

element in its input set which *does not* have this feature, and similarly, we create a solid branch from a node if there is at least one element in its input set which *has* this feature. To each branch we also associate the number of elements flowing to the child node.

Classification, and therefore branching stops when the input set of a node is *pure enough* with respect to the amount of error allowed. For example, if we allow an error of 1%, it is enough for 99 elements out of 100 to be in the same class. The node where classification stops is called a *leaf node*. We denote the leaf nodes with an empty ellipse.

In ML, the building process of a BDT is called *training* and it is done using a *training dataset*. Once the training phase is over and the structure of the BDT is fully known, it is then used to classify the entire dataset. A common challenge in this particular context is to find a BDT structure that does not *overfit* the data. Overfitting is a situation where a BDT perfectly fits the training data but fails to generalise the unseen data. One possible reason why overfitting may occur is that the BDT is *fully grown*. We say that a BDT is fully grown when it classifies the training data into completely pure sets, i.e. sets where all elements belong to the same class, modulo the error we allow.

The BDT in Figure 5.1 is not fully grown, as the rightmost and leftmost leaf nodes do not have pure sets. They both contain shapes which belong to different classes. So here, a certain amount of error, i.e. impurity is allowed, and from an overfitting standpoint this is rather positive. The BDT in Figure 5.2 performs one more classification step to check if the symmetry axes of the shapes are of same length or not. We see that it is fully grown, has zero error, and possibly more prone to overfitting data.

Although a fully grown BDT is not necessarily preferred in the ML framework, in our case, this is what we want to have in order to represent the domain $D(v)$ of a parent set variable v. We want to classify each domain value, i.e. parent set $S \in D(v)$ into $d_v = |D(v)|$ many groups, i.e. one group per each parent set. The reason why will get clear in the next section.

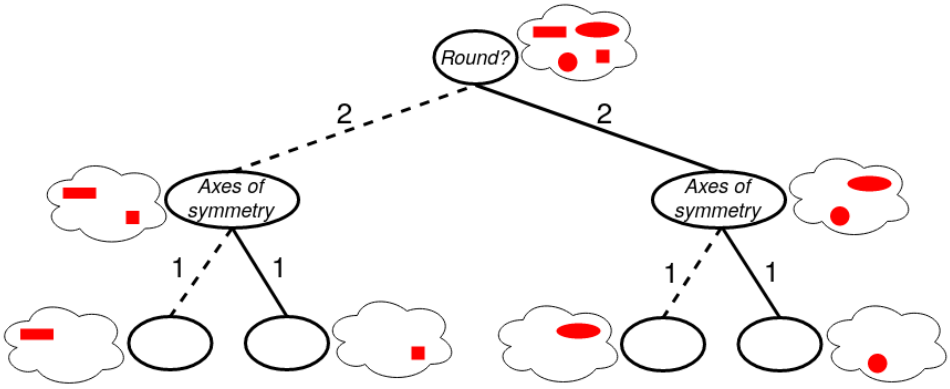


Figure 5.2: A fully-grown BDT.

5.3 Using BDTs to Represent Domain Values in BNSL

The BDT of each variable v consists of $n = |V|$ levels. The first $n - 1$ levels are for each variable in $V \setminus \{v\}$. The naive approach is to branch on variables at each level in a lexicographic order, however, a different order can also be used, and moreover, the variable ordering *does not have to be* identical for each branch. The last level consists of the leaf nodes, in each of which there is exacly one value $S \in D(v)$. For each value $S \in D(v)$, there is a unique path, i.e. a sequence of branches leading from the root node to the leaf node. This also implies that for every value $S \in D(v)$, there is *at least one edge* which is used only by that value. This is an important property which allows us do add and remove values when needed.

Each node η has the following attributes:

- ID
- Variable index
- False child node η_f
- True child node η_t
- False edge counter
- True edge counter

The root is created before all the others are explored in a depth-first manner. The variable index u of a node η is its classifier. If there exists at least one parent set value $S \in D_\eta(v) \subseteq D(v)$, where $D_\eta(v)$ is the set of values taken as an input by η, then a true child η_t is created for η. S, as well as all other parent set values which contain u, are transferred to η_t. We denote the set of these values as $D^u_\eta(v)$. The number of parent set values being transferred to η_t is kept as the "true edge counter" of η. Similarly, if there exists any parent set value $S \notin D_\eta(v)$, then a false child η_f

is created for η. We denote the set of these values as D_η^{-u}. All the parent set values in D_η^{-u} are transferred to η_f and the false edge counter of η is set to $|D_\eta^{-u}|$.

Variable	Domain Value
v_0	$\{v_1\}$
	$\{v_2\}$
	$\{v_3\}$
	$\{v_1, v_2\}$

Table 5.1: BNSL problem used throughout this chapter. We have $V = \{v_0, v_1, v_2, v_3\}$ and $n = |V| = 4$, and we focus on the domain of variable v_0.

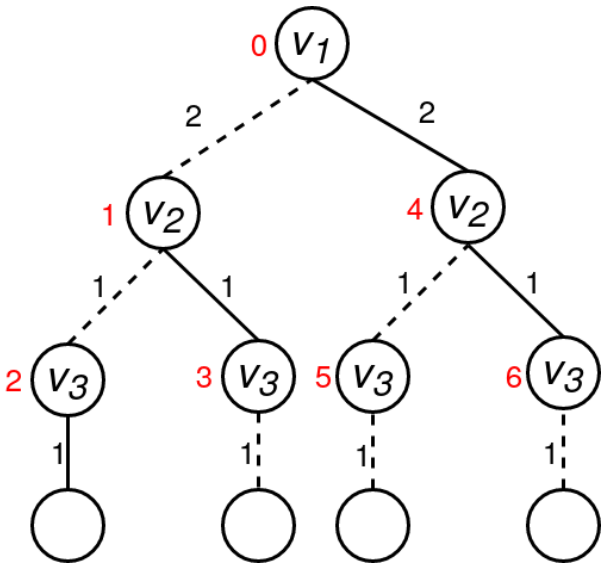


Figure 5.3: BDT of v_0 from the BNSL problem presented in Table 5.1

Example 5.1. *In Table 5.1, we present the domain of a variable v_0, which belongs to a BNSL problem with 4 variables $V = \{v_0, v_1, v_2, v_3\}$. $D(v_0)$ has 4 values in it and in Figure 5.3, we see the BDT of v_0. This BDT has 1 level for each variable in $V \setminus \{v_0\}$, and an additional level for the leaf nodes, which we also call sink. The root node takes all 4 values as input and classifies them into two groups: those containing v_1 and not containing v_1. As a result, a false child is created for the root, and, the values $\{v_2\}$ and $\{v_3\}$ are transferred to this new false child, increasing the false edge counter to 2. Then, from the false child, $\{v_3\}$ goes further down via the dashed edge, and $\{v_2\}$ via the solid edge. Finally, each of these values are transferred to a leaf node with an appropriate edge (dashed or solid). Subsequently, a true child for the root is also created, and, the values $\{v_1\}$ and $\{v_1, v_2\}$ are transferred to this new true child, increasing the true edge counter to 2. We continue*

in this manner until all values find their way to a leaf node. The red numbers next to each node denote its ID, as well as the order in which each node is created. We note that this BDT reaches its largest width, which is 4, at the second last level. By convention, the largest width of a BDT is simply its width.

5.4 Main Operations in BDTs

5.4.1 Creation of BDTs

At the beginning of the search, we create a BDT for each variable $v \in V$ which we maintain dynamically during the entire search. That is to say, whenever a domain value $S \in D(v)$ is pruned or restored, the BDT of v will be updated accordingly.

Let us first see how we construct a BDT. In Algorithm 10, we see the subroutine which constructs the BDT of $v \in V$. We have `constructBDT`, where we create a root node, whose variable index is yet to be set. Then, we trigger a recursive procedure called `exploreNode` which has 6 input parameters:

- A BDT node η node whose variable index will be set and children will be created
- The set of input parent set values $D_\eta(v) \subseteq D(v)$
- The level, i.e. depth of η
- The set of variables which we have so far branched on until we reached η, i.e. $seenVars$

For the initial call to `exploreNode`, the input parameters above are $root$, $D(v)$ i.e. the entire domain, 0, the empty set, respectively.

In $chooseVar$ which is called at line 6, although we can use whatever heuristic we want, for now we assume we pick the first variable $v_i \in V \setminus (seenVars \cup \{v\})$ as the next variable to branch on. We set the variable index of η to v_i, and we divide $D_\eta(v)$ into two sets: the set $D_\eta^{v_i}(v)$ of values S such that $v_i \in S$ and the set $D_\eta^{-v_i}(v)$ of values S^- such that $v_i \notin S^-$. We then set the false edge counter and the true edge counter of η to $d_v^{-v_i}$ and $d_v^{v_i}$, respectively.

Now that we know on which variable we will branch, we set the variable index of η to $nodeVar$. If at line 9, we see that η is in fact at the last level before the sink, we immediately set its children to sink, under the condition that the corresponding edge counters are positive. Otherwise, we continue to expand the BDT in a depth-first manner. If the false edge counter of η is positive, then we create a new false child for it and explore that. For that, we give as input to `exploreNode` the false child node η_f, $D_\eta^{-nodeVar}(v)$ which is indeed $D_{\eta_f}(v)$, the depth of the next level, and the set of seen variables in which we now added $nodeVar$. Once the false branch is fully grown, we create and explore the true child provided that the true edge counter of η is positive.

Algorithm 10: Constructing a BDT and exploring each node η starting from the root.

```
1  Procedure constructBDT(v):
2      η ← root
3      D_η(v) ← D(v)
4      exploreNode(η, D_η(v), 0, {})
5  Procedure exploreNode(η, D_η(v), level, seenVars):
6      η.varIndex = chooseVar(V \ (seenVars ∪ {v}))
7      η.falseCounter ← d_v^{-η.varIndex}
8      η.trueCounter ← d_v^{η.varIndex}
9      if level = nbLevels − 2 then
10         if η.falseCounter > 0 then
11             η_f ← sink
12         if η.trueCounter > 0 then
13             η_t ← sink
14         return
15     seenVars ← seenVars ∪ {η.varIndex}
16     if η.falseCounter > 0 then
17         η_f ← newNode()
18         exploreNode(η_f, D_η^{-η.varIndex}(v), level + 1, seenVars)
19     if η.trueCounter > 0 then
20         η_t ← newNode()
21         exploreNode(η_t, D_η^{η.varIndex}(v), level + 1, seenVars)
```

5.4.2 Maintaining BDTs

Our solver being based on CPBayes, there are several symmetry cost-based and symmetry-based pruning mechanisms being used. In addition to that, whenever we backtrack during the search, some of the pruned values are restored. Therefore, from the time BDTs are created, until the moment we actually need to run queries on them, very likely there have been quite a few such changes and we need to make sure that the BDTs correspond exactly to the current domain. We do this via the procedures `pruneParentSet` and `restoreValue` presented in Algorithms 11 and 12.

`pruneParentSet` follows the path indicated by a given parent set S from root to sink. Whenever the variable index of a node η that we hit appears in S, we decrease the true edge counter by 1 and go to the true child η_t, and, when it does not, we decrease the false edge counter by 1 and go to the false child η_f. Note that the update in the counters will result in zero-counters for some nodes, whose corresponding branch is used only by S. `restoreValue` does exactly the same thing as `pruneParentSet`, only that it increases the counters it meets by one. Another important observation to make at this point is that, we *never delete* any

node. We only render a node *unreachable* by making the counter of the branch leading to it zero. We can see the counters as *flow capacities*, and if the true (false) counter is zero, we simply cannot go to the true (false) child.

Algorithm 11: Prune a parent set value $S \in D(v)$.

```
1  Procedure pruneParentSet(S):
2      η ← root
3      while η ≠ sink do
4          if η.varIndex ∈ S then
5              η.trueCounter ← η.trueCounter − 1
6              η ← η_t
7              continue
8          if η.varIndex ∉ S then
9              η.falseCounter ← η.falseCounter − 1
10             η ← η_f
11             continue
```

Algorithm 12: Restore a parent set value $S \in D(v)$.

```
1  Procedure restoreValue(S):
2      η ← root
3      while η ≠ sink do
4          if η.varIndex ∈ S then
5              η.trueCounter ← η.trueCounter + 1
6              η ← η_t
7              continue
8          if η.varIndex ∉ S then
9              η.falseCounter ← η.falseCounter + 1
10             η ← η_f
11             continue
```

5.4.3 Navigating Through BDTs

We very often need to ask if a particular node that we encounter has a false child or a true child. `hasTrueChild` and `hasFalseChild`, presented in Algorithm 13, answer these questions. `hasTrueChild` returns $true$ if a true child η_t has been created for η, *and* the true counter is positive, *and* η_t is *unmasked*. Likewise, `hasFalseChild` returns $true$ if a false child η_f has been created for η, *and* the false counter is positive, *and* η_f is *unmasked*. By default, all nodes are unmasked. We will make use of the masking feature in Section 5.6.

Algorithm 13: Checking if a node η has a true or a false child and if it is reachable.

```
1  Procedure hasTrueChild(η):
2      if η_t exists and η.trueCounter > 0 and η_t unmasked then
3          return true
4      return false
5  Procedure hasFalseChild(η):
6      if η_f exists and η.falseCounter > 0 and η_f unmasked then
7          return true
8      return false
```

5.5 BDT in the GAC Propagator

Let us first recall how the GAC propagator works. Whenever we iterate on a variable given an initial valid ordering, we try to see how much we can push it down in the ordering towards the end. To see if we can push a variable further down, we check if there exists a value in its domain that is a subset of the current prefix (see line 12 in Algorithm 14). This is where the BDTs come into play with the procedure `hasValueThatIsSubsetOf` presented in Algorithm 15.

Algorithm 14: GAC propagator for acyclicity, using BDT features.

```
1  Procedure Acyclicity-GAC-n3d(V, D):
2      O ← acycChecker(∅, V, D)
3      if O ⊊ V then
4          return false
5      foreach v ∈ V do
6          changes ← true
7          i ← O^{-1}(v)
8          prefix ← {O_0, ..., O_{i-1}}
9          while changes do
10             changes ← false
11             foreach w ∈ O \ (prefix ∪ {v}) do
12                 if BDT[w].hasValueThatIsSubsetOf (prefix) then
13                     prefix ← prefix ∪ {w}
14                     changes ← true
15         foreach u ∉ (prefix ∪ {v}) do
16             BDT[v].pruneParentSetsWith (u)
17     return true
```

We know that in the BDT of a variable $w \in V$, we have a unique path for each parent set value $S \in D(w)$. In complement to this, we know that there is no path from root to sink for $S \notin D(w)$. Hence, it is enough to check if there exists a

path from root to sink that respects the given prefix. We start from the root, and we try all alternative paths to reach the sink. Whether we are allowed to follow the dashed or solid branch of a node η that we encounter depends on the variable index of η. If $\eta.varIndex \in prefix$, we may follow both branches, otherwise exclusively the dashed branch. If the node we are attempting to reach following a particular branch, i.e. η_t or η_f is reachable, we go to that node and continue the same manner. If, however, at a node η before we reach the sink, there is no reachable child node where we are allowed to go, `reachSink` called from η returns false. If all alternatives fail like this, `reachSink` called from the root returns false. Otherwise, as soon as one of these alternative paths that we follow actually reaches the sink, we immediately stop trying alternatives and return true.

Algorithm 15: Procedure to check if a variable has at least one value in its domain that is a subset of the given prefix. $nbNodes$ is the total number of nodes in the BDT of the variable in question.

```
1  Procedure hasValueThatIsSubsetOf(prefix):
2      return reachSink (root, prefix)
3  Procedure reachSink(η, prefix):
4      if η is sink then
5          return true;
6      if η.varIndex ∈ prefix then
7          if η.hasTrueChild() and reachSink (η_t, prefix) then
8              return true;
9          if η.hasFalseChild() and reachSink (η_f, prefix) then
10             return true;
11         return false;
12     else
13         if η.hasFalseChild() and reachSink (η_f, prefix) then
14             return true;
15         return false;
```

Here, we should emphasise the fact that a BDT merges common prefixes, thanks to which we avoid exploring the entire subtree of a node as soon as we realise that there is no value therein which is a subset of a given set. For example, assume we want to check if there exists a value $S \subseteq \{2, 3, 5\}$. Also assume we have $root.varIndex = 1$, $root_f.varIndex = root_t.varIndex = 2$. We do not bother to explore the subtree rooted at $root_t$, because all values in that subtree contain 1 in them.

Once a variable v is pushed down as much as possible in the valid ordering, we proceed to remove inconsistent values from its domain. These are the values $S \in D(v)$ such that for all $u \notin prefix$, i.e. all u which comes after v in the final ordering, $u \in S$. This is done by `pruneParentSetsWith` presented in Algorithm 16.

Algorithm 16: Prune parent sets containing a particular variable.

```
1  Procedure pruneParentSetsWith(u):
2      reachNodeWithVarIndex(u, root)
3      updateCounters()
4  Procedure reachNodeWithVarIndex(u, node):
5      if η.varIndex = u then
6          if η.hasTrueChild() then
7              η.trueCounter ← 0
8          return
9      if η.hasTrueChild() then
10         reachNodeWithVarIndex(u, η_t)
11     if η.hasFalseChild() then
12         reachNodeWithVarIndex(u, η_f)
```

In pruneParentSetsWith, we find *all* the nodes with $varIndex = u$ in the BDT of v. Starting from the root, we follow both solid and dashed branches of any node with a different variable index than u. Whenever we encounter a node η with $\eta.varIndex = u$, we set its true counter to 0, and stop going further down from thereon. By setting $\eta.trueCounter$ to 0, we make not only η_t unreachable, but also the sink for all the values using that branch. Once the true counter of all the nodes with $varIndex = u$ are set to 0, we update the counters at line 3. updateCounters detects all the nodes which henceforth became unreachable, and sets both the true and the false counter of these nodes to zero. Then, it traverses the BDT from the bottom to the top, and makes sure that for each node η, the sum $\eta.trueCounter + \eta.falseCounter$ is equal to the counter of the branch leading to η.

5.6 BDT in Cluster Reasoning Algorithms

As we have seen in Chapter 4, we use acycChecker both in lowerBoundRC, the algorithm detecting a set of cluster cuts which provably improve the linear relaxation, and minimiseCluster, the algorithm minimising the clusters found in lowerBoundRC.

An important detail to remember however is that in these algorithms, we focus on the variables with a reduced cost of zero. We use at this point the *masking* feature which we mentioned earlier. In lowerBoundRC presented in algorithm 17, just before we solve the dual of $LP_{\mathcal{W}}(D)$, we call $resetMasks$ to mask *all* the nodes (except the root) of the BDT of all variables in order to make them unreachable. Then, once we have $\hat{y}$, we unmask the nodes which are used in the paths for the values appearing in $D^{rc}_{\mathcal{W},\hat{y}}$: the set of values with a reduced cost of zero in $\hat{y}$. Similarly, at each iteration of the while loop starting at line 5, the new cluster W we discover creates a non-empty set of parent set values whose reduced cost

Algorithm 17: Lower bound computation with RC-clusters

```
1  Procedure lowerBoundRC(V, D, 𝒲):
2      resetMasks()
3      ŷ ← dualSolve(LP_𝒲(D))
4      unmaskValues(D^rc_{𝒲,ŷ})
5      while true do
6          W ← V \ acycChecker(V, D^rc_{𝒲,ŷ})
7          if W = ∅ then
8              return ⟨cost(ŷ), 𝒲⟩
9          W ← minimise(W)
10         𝒲 ← 𝒲 ∪ {W}
11         ŷ ← dualImprove(ŷ, LP_𝒲(D), W)
12         unmaskValues(D^rc_{𝒲,ŷ})
```

become zero. As a result, we unmask the nodes used in the paths of those values as well.

Another point where we make use of BDTs is when we look for the next variable to be added in the set O in `acycChecker`. At line 8 in Algorithm 18, we check if the variable v on which we iterate has a value in its domain that is a subset of the current members of the set O. We do this by making a call to `hasValueThatIsSubsetOf`, which we have seen in Section 5.5.

Algorithm 18: Acyclicity Checker

```
1  Procedure acycChecker(prefix, V, D^rc_{𝒲,ŷ}):
2      O ← prefix
3      changes ← true
4      i ← size(prefix)
5      while changes do
6          changes ← false
7          foreach v ∈ V \ O do
8              if v hasValueThatIsSubsetOf (O) then
9                  O_i ← v
10                 changes ← true
11                 i ← (i + 1)
12     return O
```

5.7 Improving the Runtime

5.7.1 How BDTs Affect Runtime

Runtime is proportional to various metrics of the size of a BDT. For example, constructing a BDT is proportional to the number of nodes it will eventually have given the variable ordering being used, since obviously, different orderings result in a different BDT structure. On the other hand, finding whether a subset of a specific set is contained in the set of sets represented by the tree, as in `hasValueThatIsSubsetOf`, is proportional in $width \times depth$ *in the worst case*.

Minimising one metric does not necessarily optimise all metrics and at any rate, finding the ordering which optimises any one metric is NP-hard [Laurent & Rivest 1976]. There are indeed exact methods for solving this NP-hard problem [Bessiere *et al.* 2009, Avellaneda 2019], nevertheless, whether it is worth the time investment or not in our context remains as a future direction for the time being. In the meantime, we employ a heuristic algorithm in order to hopefully reduce the BDT sizes.

5.7.2 Information Gain

One of the widely used methods for learning a BDT is based on the notion of *information gain*. Let us first formally define in general terms what it is.

Definition 5.2 (Information Gain)**.** ***Information gain (IG)*** *is the expected reduction in entropy, i.e. the amount of information resulting from a decision taken. This reduction from one state to another is calculated as*

$$IG(\mathcal{T}, x) = H(\mathcal{T}) - H(\mathcal{T}|x) \tag{5.1}$$

where $\mathcal{T}$ is a ***training set*** *which we want to classify, $H(\mathcal{T})$ is its* ***entropy****, and $H(\mathcal{T}|x)$ is its* ***conditional entropy*** *given an* ***attribute*** *x. They are calculated as*

$$H(\mathcal{T}) = -\sum_i P(t_i) \times \log P(t_i) \tag{5.2a}$$

$$H(\mathcal{T}|x) = \sum_{a \in vals(x)} \frac{|\mathcal{T}^{x=a}|}{|\mathcal{T}|} H(\mathcal{T}^{x=a}) \tag{5.2b}$$

$$= \sum_{a \in vals(x)} P(x = a) H(\mathcal{T}^{x=a}) \tag{5.2c}$$

$P(t_i)$ is the ***probability*** *of an element in the set $\mathcal{T}$ belonging to a class i, e.g. the probability of a shape belonging to the square class in the set given to the BDT in Figure 5.4, which is 0.25. Note that the logarithm is to the base 2.*

$\mathcal{T}^{x=a}$ is the set of elements in $\mathcal{T}$ for which the attribute x has value a. Therefore, the ratio $\frac{|\mathcal{T}^{x=a}|}{|\mathcal{T}|}$ is also the ***probability*** *of an element having $x = a$, $P(x = a)$. For*

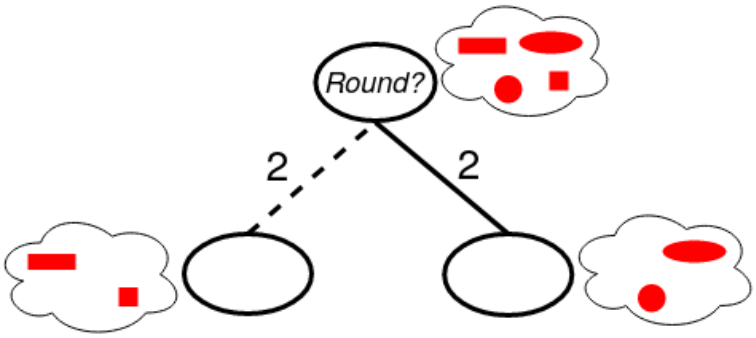


Figure 5.4: The BDT for shapes from Section 5.2.

Variable	Domain Value
v_0	$\{v_1\}$
	$\{v_2\}$
	$\{v_3\}$
	$\{v_1, v_2\}$

Table 5.2: $n = |V| = 4$, $V = \{v_0, v_1, v_2, v_3\}$.

example, we have $P(Round? = 1) = \frac{|\mathcal{T}^{Round?=1}|}{|\mathcal{T}|} = \frac{2}{4} = 0.5$ *for the set given to the BDT in Figure 5.4.* $H(\mathcal{T}^{x=a})$ *is the* ***entropy*** *of the training set* ***conditioned on*** $x = a$.

In our context, we calculate the IG for each variable $v \in V$ with respect to another variable u. $\mathcal{T}$ corresponds to the set of all *possible (not candidate)* parent sets of v. We have only two classes: parent sets which appear in the set of *candidate parent sets*, i.e. the domain $D(v)$, and those which do not. The attribute corresponds to containing, or not containing, another given variable u.

Example 5.3. *Let us recall the domain of* v_0 *in the small BNSL problem with* $V = \{v_0, v_1, v_2, v_3\}$ *and* $n = 4$ *which we have seen earlier in this chapter, now shown in Table 5.2. The set of all possible parent sets of* v_0*, denoted* $\mathcal{T}_{v_0}$*, is of size* $2^{n-1} = 8$ *and consists of the following:*

$\{\}$	$\{v_1, v_2\}$
$\{v_1\}$	$\{v_1, v_3\}$
$\{v_2\}$	$\{v_2, v_3\}$
$\{v_3\}$	$\{v_1, v_2, v_3\}$

Let us first calculate the entropy of the domain of $D(v_0)$*. We know from equation 5.2a that it is equal to*

$$H(\mathcal{T}_{v_0}) = -P(D(v_0)) \log P(D(v_0)) - P(D^-(v_0)) \log P(D^-(v_0))$$

where $P(D(v_0))$ is the probability of a parent set in $\mathcal{T}_{v_0}$ also appearing in $D(v_0)$, which is equal to

$$P(D(v_0)) = \frac{d_{v_0}}{|\mathcal{T}_{v_0}|}$$

Recall that we use d_{v_0} to denote the domain size of v_0. The probability of a parent set in $\mathcal{T}_{v_0}$ ***not*** *appearing in $D(v_0)$ is*

$$P(D^-(v_0)) = \frac{|\mathcal{T}_{v_0}| - d_{v_0}}{|\mathcal{T}_{v_0}|}$$

As a result, we have

$$\begin{aligned} H(\mathcal{T}_{v_0}) &= -\frac{d_{v_0}}{|\mathcal{T}_{v_0}|} \log \frac{d_{v_0}}{|\mathcal{T}_{v_0}|} - \frac{|\mathcal{T}_{v_0}| - d_{v_0}}{|\mathcal{T}_{v_0}|} \log \frac{|\mathcal{T}_{v_0}| - d_{v_0}}{|\mathcal{T}_{v_0}|} \\ &= -\frac{4}{8} \log \frac{4}{8} - \frac{8-4}{8} \log \frac{8-4}{8} \\ &= 1 \end{aligned}$$

Now let us focus on the attribute of containing v_3.

The set of all possible parent sets of v_0 containing v_3, denoted $\mathcal{T}_{v_0}^{v_3}$, is of size $2^{n-2} = 4$ and consists of the following:

$$\begin{array}{rr} \{v_3\} & \{v_1, v_3\} \\ \{v_2, v_3\} & \{v_1, v_2, v_3\} \end{array}$$

Similarly, the set of all possible parent sets of v_0 ***not*** *containing v_3, denoted $\mathcal{T}_{v_0}^{-v_3}$, is also of size $2^{n-2} = 4$ and consists of the following:*

$$\begin{array}{rr} \{\} & \{v_2\} \\ \{v_1\} & \{v_1, v_2\} \end{array}$$

We denote the set of candidate parent sets of v_0 containing v_3 as $D^{v_3}(v_0)$, and it has only one element, $\{\{v_3\}\}$, and is of size $d_{v_0}^{v_3} = 1$.

The set of candidate parent sets of v_0 ***not*** *containing v_3 is denoted as $D^{-v_3}(v_0)$, in which we have $\{\{v_1\}, \{v_2\}, \{v_1, v_2\}\}$, and which is of size $d_{v_0}^{-v_3} = 3$.*

Now that we have all this information, let us calculate the entropies $H(\mathcal{T}_{v_0}^{v_3})$ and $H(\mathcal{T}_{v_0}^{-v_3})$ using equation 5.2a:

$$
\begin{aligned}
H(\mathcal{T}_{v_0}^{v_3}) &= -\frac{d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|}\log\frac{d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|} - \frac{|\mathcal{T}_{v_0}^{v_3}| - d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|}\log\frac{|\mathcal{T}_{v_0}^{v_3}| - d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|} \\
&= -\frac{1}{4}\log\frac{1}{4} - \frac{3}{4}\log\frac{3}{4} \\
&= 0.811 \\
H(\mathcal{T}_{v_0}^{-v_3}) &= -\frac{d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|}\log\frac{d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|} - \frac{|\mathcal{T}_{v_0}^{-v_3}| - d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|}\log\frac{|\mathcal{T}_{v_0}^{-v_3}| - d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|} \\
&= -\frac{3}{4}\log\frac{3}{4} - \frac{1}{4}\log\frac{1}{4} \\
&= 0.811
\end{aligned}
$$

Now that we have all the components, we can calculate $IG(\mathcal{T}_{v_0}, v_3)$:

$$
\begin{aligned}
IG(\mathcal{T}_{v_0}, v_3) &= H(\mathcal{T}_{v_0}) - \frac{|\mathcal{T}_{v_0}^{v_3}|}{|\mathcal{T}_{v_0}|}H(\mathcal{T}_{v_0}^{v_3}) - \frac{|\mathcal{T}_{v_0}^{-v_3}|}{|\mathcal{T}|}H(\mathcal{T}_{v_0}^{-v_3}) \\
&= 1 - \frac{4}{8}0.811 - \frac{4}{8}0.811 \\
&= 0.189
\end{aligned}
$$

Below is the general IG formula for a variable $v \in V$ in BNSL terms:

$$
\begin{aligned}
IG(\mathcal{T}_v, u) &= H(\mathcal{T}_v) - \frac{|\mathcal{T}_v^u|}{|\mathcal{T}_v|}H(\mathcal{T}_v^u) - \frac{|\mathcal{T}_v^{-u}|}{|\mathcal{T}_v|}H(\mathcal{T}_v^{-u}) && \text{(5.3a)} \\
&= H(\mathcal{T}_v) && \text{(5.3b)} \\
&- \frac{|\mathcal{T}_v^u|}{|\mathcal{T}_v|}\left[-\frac{d_v^u}{|\mathcal{T}_v^u|}\log\frac{d_v^u}{|\mathcal{T}_v^u|} - \frac{|\mathcal{T}_v^u| - d_v^u}{|\mathcal{T}_v^u|}\log\frac{|\mathcal{T}_v^u| - d_v^u}{|\mathcal{T}_v^u|}\right] && \text{(5.3c)} \\
&- \frac{|\mathcal{T}_v^{-u}|}{|\mathcal{T}_v|}\left[-\frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\log\frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} - \frac{|\mathcal{T}_v^{-u}| - d_v^{-u}}{|\mathcal{T}_v^{-u}|}\log\frac{|\mathcal{T}_v^{-u}| - d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] && \text{(5.3d)} \\
&= H(\mathcal{T}_v) && \text{(5.3e)} \\
&+ \frac{1}{|\mathcal{T}_v|}\left[d_v^u\log\frac{d_v^u}{|\mathcal{T}_v^u|} + (|\mathcal{T}_v^u| - d_v^u)\log\frac{|\mathcal{T}_v^u| - d_v^u}{|\mathcal{T}_v^u|}\right] && \text{(5.3f)} \\
&+ \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u}\log\frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} + (|\mathcal{T}_v^{-u}| - d_v^{-u})\log\frac{|\mathcal{T}_v^{-u}| - d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] && \text{(5.3g)}
\end{aligned}
$$

5.7.3 Simplifying the IG Formula

The size of the training set $\mathcal{T}_v$ for a variable $v \in V$ is 2^{n-1} (and similarly 2^{n-2} for $\mathcal{T}_v^u$ and $\mathcal{T}_v^{-u}$). As n grows, not only it gets cumbersome to calculate these terms

with powers of 2 present in the IG formula 5.3a, but also we eventually exceed the precision of the floating point types. In order to overcome this problem, we exploit the fact that for large enough n, $\frac{2^{n-2}-d_v^u}{2^{n-2}} \simeq 1$ and $\log \frac{2^{n-2}-d_v^u}{2^{n-2}} \simeq 0$. Therefore the second term inside the big brackets in 5.3f and 5.3g is zeroed out.

Let us now think how we can factor out the term $\frac{1}{|\mathcal{T}_v|} = \frac{1}{2^{n-1}}$ outside the big brackets in 5.3f and 5.3g. If we consider $2^{n-1} \log \frac{2^{n-1}-d_v^u}{2^{n-1}}$, we cannot necessarily say it is approximately 0, because the log goes to 0 at the same rate that 2^{n-1} grows, so their growth rates cancel each other out. It may still be true, but it is not as obvious. In any case, we have already managed to simplify the formula a bit, so let us write it down before we move on:

$$IG(\mathcal{T}_v, u) = H(\mathcal{T}_v) \tag{5.4}$$
$$+ \frac{1}{|\mathcal{T}_v|}\left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|}\right] \tag{5.5}$$
$$+ \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] \tag{5.6}$$

The fact that we need the IG of a variable just to compare to that of another allows us to factor out $\frac{1}{2^{n-1}}$. Assume in addition to u, we have another variable (attribute) w. When we want to check if $IG(\mathcal{T}_v, u)$ is greater than $IG(\mathcal{T}_v, w)$, we have the following inequality:

$$IG(\mathcal{T}_v, u) - IG(\mathcal{T}_v, w) \overset{?}{\geq} 0 \iff \tag{5.7}$$
$$H(\mathcal{T}_v) + \frac{1}{|\mathcal{T}_v|}\left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|}\right] + \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] \tag{5.8}$$
$$-H(\mathcal{T}_v) - \frac{1}{|\mathcal{T}_v|}\left[d_v^w \log \frac{d_v^w}{|\mathcal{T}_v^w|}\right] - \frac{1}{|\mathcal{T}_v|}\left[d_v^{-w} \log \frac{d_v^{-w}}{|\mathcal{T}_v^{-w}|}\right] \overset{?}{\geq} 0 \iff \tag{5.9}$$
$$\frac{1}{|\mathcal{T}_v|}\left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|}\right] + \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] \tag{5.10}$$
$$-\frac{1}{|\mathcal{T}_v|}\left[d_v^w \log \frac{d_v^w}{|\mathcal{T}_v^w|}\right] - \frac{1}{|\mathcal{T}_v|}\left[d_v^{-w} \log \frac{d_v^{-w}}{|\mathcal{T}_v^{-w}|}\right] \overset{?}{\geq} 0 \iff \tag{5.11}$$
$$d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|} + d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} \tag{5.12}$$
$$-d_v^w \log \frac{d_v^w}{|\mathcal{T}_v^w|} - d_v^{-w} \log \frac{d_v^{-w}}{|\mathcal{T}_v^{-w}|} \overset{?}{\geq} 0 \iff \tag{5.13}$$
$$d_v^u \log d_v^u - d_v^u \log |\mathcal{T}_v^u| + d_v^{-u} \log d_v^{-u} - d_v^{-u} \log |\mathcal{T}_v^{-u}| \tag{5.14}$$
$$-d_v^w \log d_v^w + d_v^w \log |\mathcal{T}_v^w| - d_v^{-w} \log d_v^{-w} + d_v^{-w} \log |\mathcal{T}_v^{-w}| \overset{?}{\geq} 0 \iff \tag{5.15}$$

$$d_v^u \log d_v^u - d_v^u(n-2) + d_v^{-u} \log d_v^{-u} - d_v^{-u}(n-2) \tag{5.16}$$

$$-d_v^w \log d_v^w + d_v^w(n-2) - d_v^{-w} \log d_v^{-w} + d_v^{-w}(n-2) \overset{?}{\geq} 0 \iff \tag{5.17}$$

In a nutshell, we choose u over w when the following holds:

$$d_v^u \log d_v^u - d_v^u(n-2) + d_v^{-u} \log d_v^{-u} - d_v^{-u}(n-2) \geq$$
$$d_v^w \log d_v^w - d_v^w(n-2) + d_v^{-w} \log d_v^{-w} - d_v^{-w}(n-2)$$

As to which threshold we use to decide when we switch to this simplified comparison, we want it to be small enough to avoid overflow and at the same time large enough that the simplification does not skew the results too much. For example, if $n = 5$, we do not want to simplify. We considered $n = 32$ to be a safe choice, since it is far from the limits of a double precision IEEE 754 number.

5.8 Related Work

BDTs are definitely not the only way to represent variables whose domains are sets of sets. We can mention the set variables in CSP [Gervet 1997], as well as Decision Diagrams (DDs) such as Binary Decision Diagrams (BDDs) [Akers 1978], Algebraic Decision Diagrams (ADDs) [Bahar *et al.* 1997], Zero-Suppressed BDDs (ZDDs) [Minato 1993] as the most widely used tools.

The domain of a set variable is traditionally defined by two sets (I, O), denoting that the domain contains all the sets s which contain I, i.e. $I \subseteq s$, and whose intersection with O is the empty set, i.e. $s \cap O = \{\}$. Set variables are convenient for implementing inclusion and disjointness constraints. On the contrary, they are not very flexible when it comes to handling cardinality and lexicographic constraints, which is why there have been attempts to cope with this limitation. One of them is the Length-Lex Ordering, which provides a direct encoding of the cardinality and lexicographic information, by totally ordering a set domain first by length, then lexicographically [Gervet & Van Hentenryck 2006]. For us, the main limitation of set variables is that their domain is defined by just two sets and not as an enumeration of all possible values, which leads to a significant loss of precision. Such a description of the domains would probably be compatible with our GAC algorithm, however, it would be problematic for `lowerBoundRC`, where we need to know the reduced cost of each individual value.

Sets can be represented by DDs, as well. DDs were first introduced in the form of a binary decision program [Lee 1959], which is a type of computer program representing a switching circuit. It was already shown that switching circuits can be represented in Boolean algebra [Shannon 1938], so this was a more conducive representation alternative for the computation of the outputs of switching circuits. Binary decision programs and BDDs are equivalent in a way, but BDDs have the advantage of allowing graphical representation. BDDs are in fact a more compact form of BDTs: one can transform a BDT into a BDD by traversing the BDT from

bottom to top and reducing two nodes whose subtrees are isomorphic into one node. The size of the BDD is also affected by the ordering of the variables, which led to the design of efficient heuristics to find "good" orderings [Rice & Kulhari 2008].

When it comes to ADDs and ZDDs, they are a particular type of BDDs. ZDDs have a fixed variable ordering and thanks to their special reduction rule, they are better at compressing sparse sets. ADDs on the other hand differ from BDDs with their ability to represent arbitrary real-valued functions, unlike BDDs which represent Boolean functions.

Although DDs are powerful tools for various combinatorial problems [Bergman *et al.* 2016, Bergman *et al.* 2014b, Bergman *et al.* 2014a], they are unfortunately not suitable for our case. We need a data structure which allows us to dynamically add and restore values during the search procedure. This issue could technically be addressed by creating a DDs from scratch at every node of the search tree, with respect to the current state of the domains. This is what we initially tried, but the preliminary results were far from promising. The other option is to use the existing union/intersection algorithms for BDDs and ZDDs. They might work out well, especially if the DDs are much smaller than the domain of the variable, but they risk doubling the size of the DD with each value removal. They also make memory management more complicated, since we need to add or remove nodes as we descend a branch and on backtracking.

Even if modifying DDs was somewhat practical, there is still the need to associate a score to each value within this data structure. ADDs are technically capable of doing that, but modifying the ADDs every time we make the reduced cost of a set of values 0 would potentially blow up their sizes. Both these requirements oblige us to have a unique path for each value, which is not the case in none of the DDs mentioned above, but it is in BDTs.

5.9 Experimental Results

In this chapter, we integrate the BDT-based domains as well as the information gain heuristic in ELSA ^{cluster}. We carry out a comparative analysis between the following solvers:

- GOBNILP v1.6.3 using SCIP v3.2.1 with CPLEX v12.8.0
- CPBayes v1.1 compatible with $n > 64$
- ELSA ^{cluster}
- ELSA ^{BDT}: uses BDTs in all algorithms mentioned in this Chapter
- ELSA ^{IG} : differs from ELSA ^{BDT} by using information gain heuristic during the construction of BDTs

We have the same set of 51 medium ($|V| < 64$) and 18 large ($64 \leq |V| < 128$) datasets from the previous chapter. For medium datasets, as well as `kdd.ts`, `kdd.test`, and `kdd.valid`, we had a 1-hour CPU time limit, and a 10-hour CPU time limit for the remaining 15 large datasets. For the analysis, we focus on the same 17 datasets as in Chapter 4, and all the results in the same format for all 69 datasets can be seen at the end of this chapter in Tables 5.6, 5.7, and 5.8.

5.9.1 Overall Comparison Between All Solvers

In Table 5.3, we see that ELSA ^{cluster} remains to be the winner in terms of the number of datasets solved to optimality within the time limit. However, we attain the shortest runtime for several datasets such as `steel_BIC`, `kdd.test`, `jester.ts`, and `bnetflix.test`.

5.9.2 Comparison Against ELSA ^{cluster}

In Table 5.4, we see that with BDTs (with or without IG), the runtime is either improved or not particularly affected for the medium datasets. However, for large datasets, this is unfortunately not the case. Instead, for these datasets we have one of the two extreme cases: the runtime is either substantially improved, e.g. `jester.ts` and all `bnetflix`, or significantly deteriorated, e.g. `accidents.ts`, `plants.valid` and `accidents.test`. It seems that, some datasets benefit from the acceleration of the queries by BDTs so much, that the workload brought along by BDTs is cancelled out. For others, however, the gain is not sufficient to outweigh operational costs.

5.9.3 Comparison Between ELSA ^{BDT} and ELSA ^{IG}

In Table 5.5, we compare these two solvers in terms of the average number of nodes of the BDTs (denoted), the total time needed to create them (denoted ), as well as the total time and the search time in parentheses. If anything, these results show us that information gain is indeed a heuristic algorithm, i.e. we cannot expect a consistent improvement with it. To give an example, using information gain does not guarantee us a smaller BDT, and it may even double the number of nodes like it does for `flag.BDe`. It also clearly increases the time needed for the construction, but this increase is microscopic with respect to the scale of these datasets. Precisely, the biggest time difference, which is 39.6 seconds, occurs for `accidents.test`, and with any solver we need at least 1 hour to solve this dataset.

Table 5.3: Comparison of ELSA ^{BDT} and ELSA ^{IG} against GOBNILP, CPBayes, and ELSA ^{cluster}, in terms of total running time in seconds. Time limit for the blue instances at the end is 10 hours, and for the rest it is 1 hour. Instances are sorted by increasing total domain size for each time limit category. For CPBayes as well as all variants of ELSA we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
carpo100_BIC	60	423	0.6	79.8 (27.8)	52.6 (0.1)	47.2 (0.2)	**47.0 (0.1)**
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	34.4 (1.0)	34.3 (4.5)	34.0 (4.3)
flag_BDe	29	1324	4.36	14.9 (14)	1.0 (0.2)	1.1 (0.3)	**1.0 (0.2)**
wdbc_BIC	31	14613	99.77	390.6 (337.5)	56 (2.4)	**52.7 (2.7)**	59 (2.7)
kdd.ts	64	43584	**327.63**	†	1452.3 (274.6)	1395.1 (254.7)	1379.5 (239.7)
steel_BIC	28	93026	†	981.2 (931.5)	124.2 (71.8)	**114 (62.7)**	125.6 (76.2)
kdd.test	64	152873	1521.73	†	1594.3 (224.4)	1451.0 (108.5)	**1438.7 (90.7)**
mushroom_BDe	23	438185	†	**131.2 (5.7)**	182.6 (58.9)	147.0 (24.1)	149.8 (26.8)
bnetflix.ts	100	446406	†	**673.4 (456.2)**	2103.1 (1900.9)	839.2 (622.3)	897.0 (680.5)
plants.test	111	520148	†	†	**28049.6 (26312.9)**	†	†
jester.ts	100	531961	†	†	21550.5 (21003.7)	21473.0 (20742.7)	**16015.3 (15374.2)**
accidents.ts	100	568160	**1273.96**	†	2302.2 (930.0)	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	†	†	**17801.6 (14080.2)**	†	†
jester.test	100	770950	†	†	**30186.8 (29455)**	†	31207.7 (30339.1)
bnetflix.test	100	1103968	†	2725.6 (2475.1)	10333.1 (10096.5)	1900.8 (1650.4)	**1869.9 (1619.8)**
bnetflix.valid	111	1325818	†	**511.7 (146.8)**	10871.7 (10527.7)	2113.7 (1749.9)	2112.3 (1746.6)
accidents.test	100	1425966	4975.64	†	**3641.7 (680.7)**	17727.7 (14560.9)	20589.3 (17196)

Table 5.4: Comparison of ELSA ^{cluster}, ELSA ^{BDT} and ELSA ^{IG} in terms of total running time and search time in seconds. Time limit for the blue instances at the end is 10 hours, and for the rest it is 1 hour. Instances are sorted by increasing total domain size for each time limit category. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
carpo100_BIC	60	423	52.6 (0.1)	47.2 (0.2)	**47.0 (0.1)**
alarm1000_BIC	37	1002	34.4 (1.0)	34.3 (4.5)	**34.0 (4.3)**
flag_BDe	29	1324	1.0 (0.2)	1.1 (0.3)	**1.0 (0.2)**
wdbc_BIC	31	14613	56.0 (2.4)	**52.7 (2.7)**	59 (2.7)
kdd.ts	64	43584	1452.3 (274.6)	1395.1 (254.7)	**1379.5 (239.7)**
steel_BIC	28	93026	124.2 (71.8)	**114.0 (62.7)**	125.6 (76.2)
kdd.test	64	152873	1594.3 (224.4)	1451.0 (108.5)	**1438.7 (90.7)**
mushroom_BDe	23	438185	182.6 (58.9)	**147.0 (24.1)**	149.8 (26.8)
bnetflix.ts	100	446406	2103.1 (1900.9)	**839.2 (622.3)**	897.0 (680.5)
plants.test	111	520148	**28049.6 (26312.9)**	†	†
jester.ts	100	531961	21550.5 (21003.7)	21473.0 (20742.7)	**16015.3 (15374.2)**
accidents.ts	100	568160	**2302.2 (930.0)**	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	**17801.6 (14080.2)**	†	†
jester.test	100	770950	**30186.8 (29455)**	†	31207.7 (30339.1)
bnetflix.test	100	1103968	10333.1 (10096.5)	1900.8 (1650.4)	**1869.9 (1619.8)**
bnetflix.valid	111	1325818	10871.7 (10527.7)	2113.7 (1749.9)	**2112.3 (1746.6)**
accidents.test	100	1425966	**3641.7 (680.7)**	17727.7 (14560.9)	20589.3 (17196)

Table 5.5: Comparison of ELSA ^{BDT} and ELSA ^{IG} in terms of average BDT sizes (🌳) and total time spent to create all BDTs (⏱). Time limit for the blue instances at the end is 10 hours, and for the rest it is 1 hour. Instances are sorted by increasing total domain size for each time limit category. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
carpo100_BIC	60	423	**201.5** 🌳 0.0 ⏱	47.2 (0.2)	375.7 🌳 0.0 ⏱	47.0 (0.1)
alarm1000_BIC	37	1002	**403.8** 🌳 0.0 ⏱	34.3 (4.5)	758.8 🌳 0.0 ⏱	34.0 (4.3)
flag_BDe	29	1324	**450.4** 🌳 0.0 ⏱	1.1 (0.3)	986.2 🌳 0.0 ⏱	1.0 (0.2)
wdbc_BIC	31	14613	**3835.9** 🌳 0.0 ⏱	52.7 (2.7)	7890.6 🌳 0.1 ⏱	59 (2.7)
kdd.ts	64	43584	17665.8 🌳 0.1 ⏱	1395.1 (254.7)	**15955.9** 🌳 0.4 ⏱	**1379.5 (239.7)**
steel_BIC	28	93026	**16073.3** 🌳 0.1 ⏱	**114.0 (62.7)**	40174.0 🌳 0.3 ⏱	125.6 (76.2)
kdd.test	64	152873	62548.6 🌳 0.4 ⏱	1451.0 (108.5)	**51480.7** 🌳 1.4 ⏱	**1438.7 (90.7)**
mushroom_BDe	23	438185	**64656.3** 🌳 0.2 ⏱	147.0 (24.1)	123941.3 🌳 1.1 ⏱	149.8 (26.8)
bnetflix.ts	100	446406	160500.9 🌳 3.0 ⏱	**839.2 (622.3)**	**154816.8** 🌳 11.0 ⏱	897.0 (680.5)
plants.test	111	520148	**114790.5** 🌳 1.7 ⏱	†	138612.7 🌳 7.7 ⏱	†
jester.ts	100	531961	158942.0 🌳 3.7 ⏱	21473.0 (20742.7)	**114897.2** 🌳 13.9 ⏱	**16015.3 (15374.2)**
accidents.ts	100	568160	222061.2 🌳 5.5 ⏱	13921.2 (12328.3)	**208849.9** 🌳 19.8 ⏱	**13555.3 (12008.6)**
plants.valid	111	684141	**149565.8** 🌳 2.3 ⏱	†	178192.2 🌳 10.1 ⏱	†
jester.test	100	770950	206566.4 🌳 4.8 ⏱	†	**155281.7** 🌳 20.3 ⏱	**31207.7 (30339.1)**
bnetflix.test	100	1103968	359724.9 🌳 7.1 ⏱	1900.8 (1650.4)	**348861.9** 🌳 28.7 ⏱	**1869.9 (1619.8)**
bnetflix.valid	111	1325818	417987.3 🌳 8.4 ⏱	2113.7 (1749.9)	**413199.0** 🌳 34.7 ⏱	2112.3 (1746.6)

Table 5.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
accidents.test	100	1425966	534753.2	12.2	**17727.7 (14560.9)**	**519021.7**	51.8	20589.3 (17196)

5.10 Conclusion

In this chapter, we attempted to address an issue inherent to BNSL problems: large domain sizes. The results show us that BDTs can be powerful tools to solve maybe not all, but definitely some large BNSL problems. As future work, one might consider a more efficient implementation of the BDTs, to see if they can be beneficial for an extended collection of datasets. On the other hand, the information gain heuristic, especially for certain datasets as we saw, may result in larger BDTs. In this case, switching off the information gain heuristic at a certain depth, or not using it at all for some variables, is envisageable.

Table 5.6: Comparison of ELSA ^{BDT} and ELSA ^{IG} against GOBNILP, CPBayes, and ELSA ^{cluster}, in terms of total running time in seconds. Time limit for the blue instances at the end is 10 hours, and for the rest it is 1 hour. Instances are sorted by increasing total domain size for each time limit category. For CPBayes as well as all variants of ELSA we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
mildew1000_BIC	35	126	0.37	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
lympho_BIC	19	143	0.16	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
water1000_BIC	32	159	0.16	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
mildew1000_BDe	35	166	0.5	0.5 (0.2)	0.4 (0.1)	0.5 (0.1)	0.5 (0.1)
hailfinder100_BIC	56	167	0.43	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
tumour_BIC	18	219	0.18	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
barley1000_BIC	48	244	0.29	1.5 (1.2)	1.6 (1.4)	2.8 (2.6)	2.7 (2.4)
shuttle_BIC	10	264	1.81	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hepatitis_BIC	20	266	0.63	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
tumour_BDe	18	274	0.39	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
lung-cancer_BIC	57	292	0.57	3.8 (2.7)	1.1 (0.0)	1.1 (0.1)	1.1 (0.0)
lympho_BDe	19	345	0.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
hailfinder500_BIC	56	418	0.36	3.4 (2.6)	1.2 (0.5)	1.7 (0.9)	1.6 (0.8)
carpo100_BIC	60	423	0.6	79.8 (27.8)	52.6 (0.1)	47.2 (0.2)	**47.0 (0.1)**
horse_BDe	28	490	0.71	1.2 (0.7)	0.6 (0.0)	0.6 (0.0)	0.5 (0.0)
horse_BIC	28	490	0.99	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
hepatitis_BDe	20	501	0.93	0.4 (0.0)	0.5 (0.0)	0.4 (0.0)	0.4 (0.0)
insurance1000_BIC	27	506	0.51	32.4 (0.0)	32.8 (0.0)	29.4 (0.0)	29.6 (0.0)
adult_BIC	15	547	0.33	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
zoo_BIC	17	554	0.42	0.1 (0.0)	0.1 (0.0)	0.0 (0.0)	0.0 (0.0)

Table 5.6 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
spectf_BIC	45	610	1.52	4.3 (3.5)	0.8 (0.0)	0.9 (0.2)	0.8 (0.1)
sponge_BIC	45	618	1.45	5 (3.2)	1.8 (0.0)	1.8 (0.0)	1.8 (0.0)
flag_BIC	29	741	1.28	1.1 (0.6)	0.4 (0.0)	0.5 (0.1)	0.5 (0.0)
vehicle_BIC	19	763	1.48	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
adult_BDe	15	768	0.67	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
insurance1000_BDe	27	792	0.7	34.0 (0.0)	34.3 (0.0)	31.3 (0.0)	31.8 (0.0)
shuttle_BDe	10	812	4.03	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
bands_BIC	39	892	3.96	0.7 (0.4)	0.6 (0.2)	1.0 (0.6)	0.8 (0.5)
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	34.4 (1.0)	34.3 (4.5)	34.0 (4.3)
segment_BIC	20	1053	4.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
flag_BDe	29	1324	4.36	14.9 (14)	1.0 (0.2)	1.1 (0.3)	**1.0 (0.2)**
voting_BIC	17	1848	1.87	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
voting_BDe	17	1940	1.91	0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
autos_BIC	26	2391	**11.83**	17.2 (0.0)	19.2 (0.0)	17.1 (0.0)	17 (0.0)
zoo_BDe	17	2855	19.02	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
vehicle_BDe	19	3121	5.68	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
letter_BIC	17	4443	3.95	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
soybean_BIC	36	5926	**40.19**	45.2 (1.5)	50.8 (3)	46.2 (5.6)	45.4 (4.7)
msnbc.ts	17	6298	8.61	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
segment_BDe	20	6491	29.15	0.4 (0.0)	0.5 (0.1)	0.5 (0.1)	0.5 (0.1)
mushroom_BIC	23	13025	1561.82	4.1 (0.0)	4.3 (0.2)	4 (0.1)	4.1 (0.1)
wdbc_BIC	31	14613	99.77	390.6 (337.5)	56 (2.4)	**52.7 (2.7)**	59 (2.7)
msnbc.test	17	16594	57.96	0.4 (0.0)	0.5 (0.1)	0.4 (0.1)	0.5 (0.1)
letter_BDe	17	18841	111.73	0.4 (0.0)	0.7 (0.3)	0.5 (0.1)	0.6 (0.1)

Table 5.6 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
msnbc.valid	17	20673	23.14	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.6 (0.0)
nltcs.ts	16	22156	15.3	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
autos_BDe	26	25238	715.89	146.6 (0.1)	145.8 (0.8)	**144.0 (0.6)**	149.4 (0.7)
kdd.ts	64	43584	**327.63**	†	1452.3 (274.6)	1395.1 (254.7)	1379.5 (239.7)
nltcs.valid	16	47097	40.91	0.9 (0.0)	1.0 (0.1)	1.2 (0.3)	1.2 (0.4)
nltcs.test	16	48303	56.58	1.0 (0.0)	1.2 (0.3)	1.3 (0.3)	1.3 (0.4)
steel_BIC	28	93026	†	981.2 (931.5)	124.2 (71.8)	**114 (62.7)**	125.6 (76.2)
kdd.test	64	152873	1521.73	†	1594.3 (224.4)	1451.0 (108.5)	**1438.7 (90.7)**
kdd.valid	64	197546	†	†	†	†	†
mushroom_BDe	23	438185	†	**131.2 (5.7)**	182.6 (58.9)	147.0 (24.1)	149.8 (26.8)
plants.ts	69	164640	†	†	†	†	†
baudio.ts	69	371117	†	†	†	†	†
bnetflix.ts	100	446406	†	**673.4 (456.2)**	2103.1 (1900.9)	839.2 (622.3)	897.0 (680.5)
plants.test	111	520148	†	†	**28049.6 (26312.9)**	†	†
jester.ts	100	531961	†	†	21550.5 (21003.7)	21473.0 (20742.7)	**16015.3 (15374.2)**
accidents.ts	100	568160	**1273.96**	†	2302.2 (930.0)	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	†	†	**17801.6 (14080.2)**	†	†
jester.test	100	770950	†	†	**30186.8 (29455)**	†	31207.7 (30339.1)
baudio.test	100	1016403	†	†	†	†	†
bnetflix.test	100	1103968	†	2725.6 (2475.1)	10333.1 (10096.5)	1900.8 (1650.4)	**1869.9 (1619.8)**
baudio.valid	69	1235928	†	†	†	†	†
bnetflix.valid	111	1325818	†	**511.7 (146.8)**	10871.7 (10527.7)	2113.7 (1749.9)	2112.3 (1746.6)
accidents.test	100	1425966	4975.64	†	**3641.7 (680.7)**	17727.7 (14560.9)	20589.3 (17196)
jester.valid	100	1463335	†	†	†	†	†

Table 5.6 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
accidents.valid	100	1617862	†	†	†	†	†

Table 5.7: Comparison of ELSA ^{cluster}, ELSA ^{BDT} and ELSA ^{IG} in terms of total running time and search time in seconds. Time limit for the blue instances at the end is 10 hours, and for the rest it is 1 hour. Instances are sorted by increasing total domain size for each time limit category. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
mildew1000_BIC	35	126	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
lympho_BIC	19	143	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
water1000_BIC	32	159	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
mildew1000_BDe	35	166	0.4 (0.1)	0.5 (0.1)	0.5 (0.1)
hailfinder100_BIC	56	167	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
tumour_BIC	18	219	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
barley1000_BIC	48	244	1.6 (1.4)	2.8 (2.6)	2.7 (2.4)
shuttle_BIC	10	264	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hepatitis_BIC	20	266	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
tumour_BDe	18	274	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
lung-cancer_BIC	57	292	1.1 (0.0)	1.1 (0.1)	1.1 (0.0)
lympho_BDe	19	345	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
hailfinder500_BIC	56	418	1.2 (0.5)	1.7 (0.9)	1.6 (0.8)
carpo100_BIC	60	423	52.6 (0.1)	47.2 (0.2)	47.0 (0.1)
horse_BDe	28	490	0.6 (0.0)	0.6 (0.0)	0.5 (0.0)
horse_BIC	28	490	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
hepatitis_BDe	20	501	0.5 (0.0)	0.4 (0.0)	0.4 (0.0)
insurance1000_BIC	27	506	32.8 (0.0)	29.4 (0.0)	29.6 (0.0)
adult_BIC	15	547	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
zoo_BIC	17	554	0.1 (0.0)	0.0 (0.0)	0.0 (0.0)
spectf_BIC	45	610	0.8 (0.0)	0.9 (0.2)	0.8 (0.1)

Table 5.7 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
sponge_BIC	45	618	1.8 (0.0)	1.8 (0.0)	1.8 (0.0)
flag_BIC	29	741	0.4 (0.0)	0.5 (0.1)	0.5 (0.0)
vehicle_BIC	19	763	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
adult_BDe	15	768	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
insurance1000_BDe	27	792	34.3 (0.0)	31.3 (0.0)	31.8 (0.0)
shuttle_BDe	10	812	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
bands_BIC	39	892	0.6 (0.2)	1.0 (0.6)	0.8 (0.5)
alarm1000_BIC	37	1002	34.4 (1.0)	34.3 (4.5)	34.0 (4.3)
segment_BIC	20	1053	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
flag_BDe	29	1324	1.0 (0.2)	1.1 (0.3)	1.0 (0.2)
voting_BIC	17	1848	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
voting_BDe	17	1940	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
autos_BIC	26	2391	19.2 (0.0)	17.1 (0.0)	17 (0.0)
zoo_BDe	17	2855	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
vehicle_BDe	19	3121	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
letter_BIC	17	4443	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
soybean_BIC	36	5926	50.8 (3)	46.2 (5.6)	45.4 (4.7)
msnbc.ts	17	6298	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
segment_BDe	20	6491	0.5 (0.1)	0.5 (0.1)	0.5 (0.1)
mushroom_BIC	23	13025	4.3 (0.2)	4 (0.1)	4.1 (0.1)
wdbc_BIC	31	14613	56.0 (2.4)	52.7 (2.7)	59 (2.7)
msnbc.test	17	16594	0.5 (0.1)	0.4 (0.1)	0.5 (0.1)
letter_BDe	17	18841	0.7 (0.3)	0.5 (0.1)	0.6 (0.1)
msnbc.valid	17	20673	0.5 (0.0)	0.5 (0.0)	0.6 (0.0)

Table 5.7 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
nltcs.ts	16	22156	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
autos_BDe	26	25238	145.8 (0.8)	144 (0.6)	149.4 (0.7)
kdd.ts	64	43584	1452.3 (274.6)	1395.1 (254.7)	1379.5 (239.7)
nltcs.valid	16	47097	1.0 (0.1)	1.2 (0.3)	1.2 (0.4)
nltcs.test	16	48303	1.2 (0.3)	1.3 (0.3)	1.3 (0.4)
steel_BIC	28	93026	124.2 (71.8)	114.0 (62.7)	125.6 (76.2)
kdd.test	64	152873	1594.3 (224.4)	1451.0 (108.5)	1438.7 (90.7)
kdd.valid	64	197546	†	†	†
mushroom_BDe	23	438185	182.6 (58.9)	147.0 (24.1)	149.8 (26.8)
plants.ts	69	164640	†	†	†
baudio.ts	69	371117	†	†	†
bnetflix.ts	100	446406	2103.1 (1900.9)	839.2 (622.3)	897.0 (680.5)
plants.test	111	520148	28049.6 (26312.9)	†	†
jester.ts	100	531961	21550.5 (21003.7)	21473.0 (20742.7)	16015.3 (15374.2)
accidents.ts	100	568160	2302.2 (930.0)	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	17801.6 (14080.2)	†	†
jester.test	100	770950	30186.8 (29455)	†	31207.7 (30339.1)
baudio.test	100	1016403	†	†	†
bnetflix.test	100	1103968	10333.1 (10096.5)	1900.8 (1650.4)	1869.9 (1619.8)
baudio.valid	69	1235928	†	†	†
bnetflix.valid	111	1325818	10871.7 (10527.7)	2113.7 (1749.9)	2112.3 (1746.6)
accidents.test	100	1425966	3641.7 (680.7)	17727.7 (14560.9)	20589.3 (17196)
jester.valid	100	1463335	†	†	†
accidents.valid	100	1617862	†	†	†

Table 5.8: Comparison of ELSA ^{BDT} and ELSA ^{IG} in terms of average BDT sizes (🌳) and total time spent to create all BDTs (⏱). Time limit for the blue instances at the end is 10 hours, and for the rest it is 1 hour. Instances are sorted by increasing total domain size for each time limit category. † indicates a timeout.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}			ELSA ^{IG}		
mildew1000_BIC	35	126	🌳 82.9	⏱ 0.0	0.2 (0.0)	🌳 114.9	⏱ 0.0	0.2 (0.0)
lympho_BIC	19	143	🌳 77.3	⏱ 0.0	0.1 (0.0)	🌳 112.9	⏱ 0.0	0.1 (0.0)
water1000_BIC	32	159	🌳 92.6	⏱ 0.0	0.2 (0.0)	🌳 144.2	⏱ 0.0	0.2 (0.0)
mildew1000_BDe	35	166	🌳 92.4	⏱ 0.0	0.5 (0.1)	🌳 147.7	⏱ 0.0	0.5 (0.1)
hailfinder100_BIC	56	167	🌳 103.3	⏱ 0.0	0.5 (0.0)	🌳 154.0	⏱ 0.0	0.5 (0.0)
tumour_BIC	18	219	🌳 82.9	⏱ 0.0	0.1 (0.0)	🌳 161.6	⏱ 0.0	0.1 (0.0)
barley1000_BIC	48	244	🌳 151.8	⏱ 0.0	2.8 (2.6)	🌳 223.0	⏱ 0.0	2.7 (2.4)
shuttle_BIC	10	264	🌳 52.6	⏱ 0.0	0.0 (0.0)	🌳 122.4	⏱ 0.0	0.0 (0.0)
hepatitis_BIC	20	266	🌳 111.8	⏱ 0.0	0.3 (0.0)	🌳 189.1	⏱ 0.0	0.3 (0.0)
tumour_BDe	18	274	🌳 100.5	⏱ 0.0	0.1 (0.0)	🌳 198.3	⏱ 0.0	0.1 (0.0)
lung-cancer_BIC	57	292	🌳 160.8	⏱ 0.0	1.1 (0.1)	🌳 263.4	⏱ 0.0	1.1 (0.0)
lympho_BDe	19	345	🌳 160.7	⏱ 0.0	0.2 (0.0)	🌳 249.4	⏱ 0.0	0.2 (0.0)
hailfinder500_BIC	56	418	🌳 191.3	⏱ 0.0	1.7 (0.9)	🌳 367.3	⏱ 0.0	1.6 (0.8)
carpo100_BIC	60	423	🌳 201.5	⏱ 0.0	47.2 (0.2)	🌳 375.7	⏱ 0.0	47.0 (0.1)
horse_BDe	28	490	🌳 203.7	⏱ 0.0	0.6 (0.0)	🌳 395.9	⏱ 0.0	0.5 (0.0)
horse_BIC	28	490	🌳 219.5	⏱ 0.0	0.4 (0.0)	🌳 391.0	⏱ 0.0	0.4 (0.0)
hepatitis_BDe	20	501	🌳 197.3	⏱ 0.0	0.4 (0.0)	🌳 339.6	⏱ 0.0	0.4 (0.0)

Table 5.8 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
insurance1000_BIC	27	506	226.1 0.0	29.4 (0.0)	394.0 0.0	29.6 (0.0)
adult_BIC	15	547	151.5 0.0	0.0 (0.0)	293.5 0.0	0.0 (0.0)
zoo_BIC	17	554	234.9 0.0	0.0 (0.0)	344.0 0.0	0.0 (0.0)
spectf_BIC	45	610	305.4 0.0	0.9 (0.2)	504.4 0.0	0.8 (0.1)
sponge_BIC	45	618	344.5 0.0	1.8 (0.0)	474.9 0.0	1.8 (0.0)
flag_BIC	29	741	296.9 0.0	0.5 (0.1)	572.2 0.0	0.5 (0.0)
vehicle_BIC	19	763	246.3 0.0	0.2 (0.0)	442.8 0.0	0.2 (0.0)
adult_BDe	15	768	194.6 0.0	0.0 (0.0)	386.3 0.0	0.0 (0.0)
insurance1000_BDe	27	792	307.7 0.0	31.3 (0.0)	600.9 0.0	31.8 (0.0)
shuttle_BDe	10	812	130.6 0.0	0.0 (0.0)	247.4 0.0	0.0 (0.0)
bands_BIC	39	892	394.2 0.0	1.0 (0.6)	661.7 0.0	0.8 (0.5)
alarm1000_BIC	37	1002	403.8 0.0	34.3 (4.5)	758.8 0.0	34.0 (4.3)
segment_BIC	20	1053	186.1 0.0	0.2 (0.0)	648.3 0.0	0.2 (0.0)
flag_BDe	29	1324	450.4 0.0	1.1 (0.3)	986.2 0.0	1.0 (0.2)
voting_BIC	17	1848	457.2 0.0	0.1 (0.0)	847.2 0.0	0.1 (0.0)
voting_BDe	17	1940	450.1 0.0	0.0 (0.0)	893.1 0.0	0.0 (0.0)
autos_BIC	26	2391	686.6 0.0	17.1 (0.0)	1560.6 0.0	17 (0.0)
zoo_BDe	17	2855	923.8 0.0	0.1 (0.0)	1418.7 0.0	0.1 (0.0)
vehicle_BDe	19	3121	848.3 0.0	0.3 (0.0)	1529.5 0.0	0.3 (0.0)

Table 5.8 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
letter_BIC	17	4443	938.5	0.0	0.1 (0.0)	1736.7	0.0	0.1 (0.0)
soybean_BIC	36	5926	1687.3	0.0	46.2 (5.6)	3391.9	0.0	45.4 (4.7)
msnbc.ts	17	6298	1610.7	0.0	0.2 (0.0)	2441.8	0.0	0.2 (0.0)
segment_BDe	20	6491	809.6	0.0	0.5 (0.1)	3167.4	0.0	0.5 (0.1)
mushroom_BIC	23	13025	3871.2	0.0	4 (0.1)	5787.1	0.0	4.1 (0.1)
wdbc_BIC	31	14613	3835.9	0.0	52.7 (2.7)	7890.6	0.1	59 (2.7)
msnbc.test	17	16594	3674.2	0.0	0.4 (0.1)	4974.8	0.0	0.5 (0.1)
letter_BDe	17	18841	2771.6	0.0	0.5 (0.1)	6012.7	0.0	0.6 (0.1)
msnbc.valid	17	20673	4399.8	0.0	0.5 (0.0)	5897.5	0.0	0.6 (0.0)
nltcs.ts	16	22156	3934.4	0.0	0.4 (0.0)	5465.6	0.0	0.4 (0.0)
autos_BDe	26	25238	5943.5	0.0	144 (0.6)	14218.0	0.1	149.4 (0.7)
kdd.ts	64	43584	17665.8	0.1	1395.1 (254.7)	15955.9	0.4	1379.5 (239.7)
nltcs.valid	16	47097	7368.1	0.0	1.2 (0.3)	8458.7	0.1	1.2 (0.4)
nltcs.test	16	48303	7397.0	0.0	1.3 (0.3)	8504.7	0.1	1.3 (0.4)
steel_BIC	28	93026	16073.3	0.1	114.0 (62.7)	40174.0	0.3	125.6 (76.2)
kdd.test	64	152873	62548.6	0.4	1451.0 (108.5)	51480.7	1.4	1438.7 (90.7)
kdd.valid	64	197546	84590.1	0.5	†	61594.7	1.8	†
mushroom_BDe	23	438185	64656.3	0.2	147.0 (24.1)	123941.3	1.1	149.8 (26.8)
plants.ts	69	164640	39108.0	0.5	†	48971.957	2.95	†

Table 5.8 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
baudio.ts	69	371117	84552.2	1.9 †	89714.630	10.25 †
bnetflix.ts	100	446406	160500.9	3.0 839.2 (622.3)	154816.8	11.0 897.0 (680.5)
plants.test	111	520148	114790.5	1.7 †	138612.7	7.7 †
jester.ts	100	531961	158942.0	3.7 21473.0 (20742.7)	114897.2	13.9 16015.3 (15374.2)
accidents.ts	100	568160	222061.2	5.5 13921.2 (12328.3)	208849.9	19.8 13555.3 (12008.6)
plants.valid	111	684141	149565.8	2.3 †	178192.2	10.1 †
jester.test	100	770950	206566.4	4.8 †	155281.7	20.3 31207.7 (30339.1)
baudio.test	100	1016403	215677.6	5.0 †	267653.9	26.6 †
bnetflix.test	100	1103968	359724.9	7.1 1900.8 (1650.4)	348861.9	28.7 1869.9 (1619.8)
baudio.valid	69	1235928	242546.6	5.9 †	350898.0	36.1 †
bnetflix.valid	111	1325818	417987.3	8.4 2113.7 (1749.9)	413199.0	34.7 2112.3 (1746.6)
accidents.test	100	1425966	534753.2	12.2 17727.7 (14560.9)	519021.7	51.8 20589.3 (17196)
jester.valid	100	1463335	342123.1	8.0 †	335342.3	38.8 †
accidents.valid	100	1617862	622315.5	14 †	560968.0	50.5 †

CHAPTER 6

VAC Integrality

Contents

6.1 Introduction

In this chapter, we switch focus from *learning* Bayesian networks (BNs) to *reasoning* with Graphical Models (GMs), which also includes reasoning with BNs. As a result of the great interest in performing Most Probable Explanation (MPE) on BNs, there have been also efforts to develop efficient methods for it. This eventually brings us to cost minimisation in Weighted Constraint Satisfaction Problems (WCSPs), since we know that each WCSP corresponds to a Cost Function Network (CFN) and that MPE in BNs is equivalent to cost minimisation in CFNs under a $-\log$ transformation. Here, we make contributions to the state of the art in minimisation queries on WCSPs. The main purpose in this chapter is to find ways to *use the Virtual Arc Consistency (VAC) algorithm more efficiently within branch-and-bound solvers for WCSPs*. These improvements allow us to use VAC not only in preprocessing, but during the search as well.

The technical tool we use to improve VAC is based on a property first developed in the context of COMBILP method [Haller *et al.* 2018], which solves the linear relaxation and uses the solution of the relaxation to find a decomposition of the

problem: the "easy" part corresponding to the set of integral variables in the linear relaxation, and a combinatorial part containing the variables assigned fractional values. They then proceed to solve the combinatorial subset exactly and if that solution can be combined with the easy part without incurring extra cost, it reports optimality. Otherwise, it moves some variables from the easy part to the combinatorial part and iterates. Crucially, they identify integral variables by identifying a condition called Strict Arc Consistency (Strict AC) on the dual solution produced, and can therefore be used with approximate dual LP solvers, like VAC and TRW-S, which may produce a suboptimal dual LP solution for which no corresponding primal solution exists.

We make several contributions here. First, we relax Strict AC, the condition that COMBILP uses to detect integrality. We show in Section 6.2 that the relaxed condition admits larger sets of integral variables and that the sets are in a sense maximal. Second, we show that a class of fixpoints of an LP solver like VAC implies a specific set of integral variables regardless of the dual solution it finds, even when those variables do not satisfy the Strict AC condition in this solution. This avoids the need to bias the LP solver towards solutions that contain Strictly AC variables. On the practical side, we introduce two simple techniques that exploit this property within a branch-and-bound solver. The first, given in Section 6.4 modifies the branching heuristic to avoid branching on Strictly AC variables, as that is unlikely to be informative. The second, given in Section 6.5, is a variant of the well-known RINS heuristic in integer programming ([Danna *et al.* 2005]), which optimistically assumes that the set of Strictly AC variables assigned their integral values actually appear in the optimal solution and solves a restricted sub-problem to help quickly identify high quality solutions. In Section 6.6, we show that integrating these techniques in the ToulBar2 solver [Cooper *et al.* 2010] improves performance significantly over the state of the art in some families of instances.

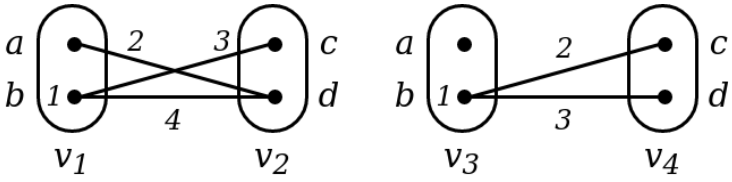
6.2 Strict Arc Consistency and VAC-integrality

Savchynskyy et al. introduced Strict Arc Consistency ([Savchynskyy *et al.* 2013]) as a way to partition a WCSP into an "easy" part, which can be solved exactly by an LP solver and a "hard" combinatorial part.

Throughout this chapter, we assume all WCSPs are Node Consistent (NC), i.e. $\min_{T \in \ell(V_S)} c_{V_S}(T) = 0$ for all scopes V_S. Otherwise, the WCSP can be trivially reparameterised to make it NC and increase the lower bound.

Definition 6.1 (Strict Arc Consistency [Savchynskyy *et al.* 2013])**.** *A variable $v_i \in V$ is Strictly Arc Consistent (Strictly AC) if it satisfies the following two conditions:*

1. *There exists a unique value $a \in D(v_i)$ such that $c_{v_i}(a) = 0$,*
2. *There exists a unique tuple $\{(v_i, a), (v_j, c)\}$ which satisfies $c_{v_i v_j}(a, c) = 0$ $\forall c_{v_i v_j} \in C$.*



(a) P_1 (b) P_2

Figure 6.1: Two WCSPs P_1 and P_2.

The value a is called the Strictly AC value of v_i.

Example 6.2. *Let $P_1 = \langle V, D, C, k\rangle$ be a WCSP where $V = \{v_1, v_2\}$, $D = \{\{a, b\}, \{c, d\}\}$, C is the set of cost functions and k is an arbitrary upper bound. This instance is presented as a cost function network in Figure 6.1a.*

Variable v_1 is Strictly AC since it has a unique value $a \in D(v_1)$ with $c_{v_1}(a) = 0$ and there exists a unique zero-cost tuple $\{(v_1, a), (v_2, c)\}$ with its neighbour v_2.

Given a WCSP P and a subset V_S of its variables such that all variables in V_S are Strictly AC, we can solve P restricted to V_S exactly by assigning the Strictly AC value to each variable. This property gives a natural partition of a WCSP into the set of Strictly AC variables and the rest. This partition was used by Savchynskyy et al. [Savchynskyy *et al.* 2013] and in a refined algorithm introduced later [Haller *et al.* 2018]. These algorithms exploit the solvability of the Strictly AC subset of variables and only need to solve the smaller non-Strictly-AC subset using a combinatorial solver.

Our first contribution here is to note that the Strict AC property is stronger than necessary. In particular, we can weaken the second condition as follows:

Definition 6.3 (VAC-integrality)**.** *A variable $v_i \in V$ is VAC-integral if it satisfies the following two conditions:*

1. *There exists a unique value $a \in D(v_i)$ such that $c_{v_i}(a) = 0$,*
2. *There exists at least one tuple $\{(v_i, a), (v_j, c)\}$ such that $c_{v_iv_j}(a, c) + c_{v_j}(c) = 0 \quad \forall c_{v_iv_j} \in C$.*

The value a is the VAC-integral value of v_i.

Example 6.4. *Let $P_2 = \langle V, D, C, k\rangle$ be a WCSP where $V = \{v_3, v_4\}$, $D = \{\{a, b\}, \{c, d\}\}$, C is the set of cost functions and k is an arbitrary upper bound. This instance is presented as a cost function network in Figure 6.1b.*

Variable v_3 is VAC-integral since it has a unique value $a \in D(v_3)$ with $c_{v_3}(a) = 0$ and with its neighbour v_4, there exists at least one zero-cost tuple satisfying the second condition in Definition 6.3, which are $\{(v_3, a), (v_4, c)\}$ and $\{(v_3, a), (v_4, d)\}$.

The difference between VAC-integrality[1] and Strict AC is that in VAC-integrality, the second condition requires that the witness value appears in at least one rather than exactly one 0-cost tuple in each incident constraint. The VAC-integral subset of a WCSP maintains the main property of Strict AC, namely that it is exactly solvable by inspection and its optimal solution has cost 0. The optimal solution, as in Strict AC, simply assigns to each VAC-integral variable its VAC-integral value. By definition, this has cost 0.

Since VAC-integrality is a relaxation of Strict AC, every Strictly AC variable is also VAC-integral, as is the case for v_1 in instance P_1 in Figure 6.1a. The inverse does not hold, i.e. v_3 in P_2 in Figure 6.1b is not Strictly AC. However, this only holds for instances that are at a VAC fixpoint. Both P_1 and P_2 are VAC since the AC closure of their $Bool$ is non-empty.

Proposition 6.5. *If a WCSP P is VAC and a variable v_i is Strictly AC then it is also VAC-integral.*

Proof. Let v_i be a Strictly AC variable in P, and let v_j be one of its neighbours. Let $\{(v_i, a), (v_j, c)\}$ be the unique tuple which satisfies $c_{v_i v_j}(a, c) = 0$. Suppose $c_{v_j}(c) > 0$. Then we can extend $c_{v_j}(c)$ to $c_{v_i v_j}(k, b)$ for all $k \in D(v_i)$. Given that $\{(v_i, a), (v_j, c)\}$ was the unique tuple with $c_{v_i v_j}(a, c) = 0$, now all $c_{v_i v_j}$ are non-zero after this extension. In that case, we can project some non-zero binary cost to $c_{v_i}(a)$, the unique value with zero unary cost. After this projection, all values in the domain of v_i have a non-zero unary cost; meaning that we can project some cost to $c_\varnothing$ and increase the lower bound. However, this is contradictory as P was already VAC. As a result, now we know that, for a Strictly AC variable v_i in a WCSP P which is VAC, we have not only $c_{v_i v_j}(a, c) = 0$, but also $c_{v_j}(c) = 0$. Therefore, given that for a VAC-integral variable v_i there can be one or more tuples with $c_{v_i v_j}(a, c) + c_{v_j}(c) = 0$, the set of VAC-integral variables is at least as large as the set of Strictly AC variables and it includes all Strictly AC variables. □

This shows us that VAC-integrality is a less restrictive property than the Strict Arc Consistency, meaning that Strict Arc Consistency implies VAC-integrality whereas the contrary does not occur.

6.2.1 Link to Integer Linear Programming

Let us recall LP called the local polytope of a WCSP P from Chapter 2:

[1] We change the name of the property from "consistency", which implies an algorithm that achieves said consistency, to "integrality". Adding the VAC term will become clear after Proposition 6.7.

$$\min c_\varnothing + \sum_{v_i \in V, a \in D(v_i)} c_{v_i}(a) x_{v_i a} + \sum_{c_{v_i v_j} \in C, a \in D(v_i), c \in D(v_j)} c_{v_i v_j}(a,c) y_{v_i a, v_j c}$$

s.t.

$$\begin{aligned}
&\sum_{a \in D(v_i)} x_{v_i a} = 1 && \forall v_i \in V \\
&x_{v_i a} = \sum_{c \in D(v_j)} y_{v_i a, v_j c} && \forall v_i \in V, a \in D(v_i), c_{v_i v_j} \in C \\
&0 \leq x_{v_i a} \leq 1 && \forall v_i \in V, a \in D(v_i) \\
&0 \leq y_{v_i a, v_j c} \leq 1 && \forall c_{v_i v_j} \in C, a \in D(v_i), c \in D(v_j)
\end{aligned} \tag{6.1}$$

We know that from the optimal solution of the above LP, the reparameterisation is extracted from the *reduced costs* $r(x_{v_i a})$ and $r(y_{v_i a, v_j b})$ of each variable and binary cost function, respectively, by setting $c_{v_i}(a)$ to $r(x_{v_i a})$ and $c_{v_i v_j}(a,c)$ to $r(y_{v_i a, v_j b})$ and setting $c_\varnothing$ to the optimum of the LP (6.1).

As with Strict AC, VAC-integrality implies integrality of the corresponding primal solution by complementary slackness.

Proposition 6.6. *The VAC-integral variables in an optimal dual solution of* (6.1) *correspond to the variables v for which there exists a unique a with $x_{va} = 1$ and $x_{vb} = 0$ for $b \neq a$ in the corresponding optimal primal solution.*

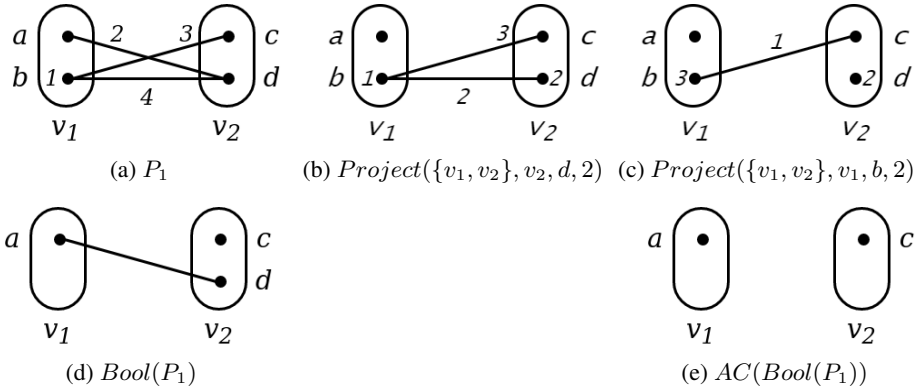
Proof. Given an optimal dual solution of the local polytope LP (6.1), for each VAC-integral variable v with VAC-integral value a, the primal solution must have $x_{vb} = 0$ for all $b \in D(v) \setminus \{a\}$ by complementary slackness and hence $x_{va} = 1$. □

Note that in the case of approximate dual LP solvers like VAC and TRW-S, unlike OSAC, this observation does not hold: if the dual solution is not optimal, there is no primal solution with the same cost. Rather, we use Strict AC and VAC-integrality as proxies for conditions which would lead to integrality in optimal solutions, while maintaining the property that they admit zero cost solutions.

One complication with both Strict AC and VAC-integrality is that any lower bound given by a dual solution can in fact be produced by several dual solutions, but they do not all give the same VAC-integrality subset. One way to deal with this is to bias the LP solver towards solutions that maximise the VAC-integral subset [Haller *et al.* 2018]. Here we propose another method, given by the following observation.

Proposition 6.7. *Given a WCSP P which is VAC and a variable v_i, if in $AC(Bool(P))$ it holds that $\overline{D}(v_i) = \{a\}$ then v_i is VAC-integral with value a.*

Proof. Since $AC(Bool(P))$ is arc consistent, if a value remains in the domain of v_i in $Bool(P)$, it has unary cost 0 in P and is supported by tuples and values of cost 0 in all incident constraints. Conversely, if a value is removed in $Bool(P)$, either it



(a) P_1 (b) $Project(\{v_1, v_2\}, v_2, d, 2)$ (c) $Project(\{v_1, v_2\}, v_1, b, 2)$

(d) $Bool(P_1)$ (e) $AC(Bool(P_1))$

Figure 6.2: WCSP P_1 in (a) with its different reparameterisations in (b) and (c) obtained with cost projections, its $Bool$ in (d) and the AC closure of $Bool(P_1)$ in (e).

has non-zero unary cost in P or some non-zero amount of cost can be moved onto it [Cooper *et al.* 2010]. □

Example 6.8. *Let $P_1 = \langle V, D, C, k\rangle$ be a WCSP where $V = \{v_1, v_2\}$, $D = \{\{a, b\}, \{c, d\}\}$, C is the set of cost functions and k is an arbitrary upper bound. This instance is presented as a cost function network in Figure 6.2a.*

WCSPs in Figure 6.2b and 6.2c are two different reparameterisations of P_1 and therefore equivalent to P_1. They have the same lower bound $c_\varnothing = 0$ and they yield the same total cost for any complete assignment $A \in \ell(V)$. They are obtained sequentially by projecting a cost of 2 from the binary cost functions $c_{v_1v_2}(a, d)$ and $c_{v_1v_2}(b, d)$ onto the unary cost function $c_{v_2}(d)$ (Figure 6.2b), and then, by projecting a cost of 2 from the binary cost functions $c_{v_1v_2}(b, c)$ and $c_{v_1v_2}(b, d)$ onto the unary cost function $c_{v_1}(b)$ (Figure 6.2c).

In P_1 we are able to identify only v_1 as VAC-integral while there are in fact other reparameterisations which would allow us to identify v_2 as VAC-integral as well (Figures 6.2b and 6.2c). However, we do not need to find those reparameterisations because by running the VAC algorithm on P_1, which runs AC on $Bool(P_1)$, we see that both v_1 and v_2 have singleton domains in $AC(Bool(P_1))$. As a result, they are both VAC-integral by Proposition 6.7.

The effect of Proposition 6.7 is that the class of dual feasible solutions which have the same $AC(Bool(P))$ produce the same set of VAC-integral variables, even though most of these solutions do not satisfy Definition 6.3. Therefore, this observation allows us to construct a larger VAC-integral subset than that given by collecting the variables that satisfy the VAC-integrality property given a dual solution. This has the advantage that we do not need to modify the LP solver to be biased towards specific dual solutions and is easy to use with VAC, which explicitly maintains $Bool(P)$. We can apply the same reasoning to find Strict AC sets of variables, using the following condition.

Proposition 6.9. *If a WCSP P is VAC, then a VAC-integral variable v_i is Strictly AC if and only if all its neighbours are VAC-integral.*

Proof. Let v_i be a VAC-integral variable.

$\Rightarrow$ Assume v_i is Strictly AC as well. Then, following Definition 6.1 and Proposition 6.5, we know that there exists a unique value $a \in D(v_i)$ with $c_{v_i}(a) = 0$ and a unique tuple $\{(v_i, a), (v_j, c)\}$ which satisfies $c_{v_i v_j}(a, c) = 0$ and $c_{v_j}(c) = 0$. Given that P is VAC, the $Bool(P)$ is Arc Consistent and there is no further pruning to do. Assume $\exists c \neq b \in D(v_j)$ such that $c_{v_j}(c) = 0$. Then, c should appear in the domain of v_j in $Bool(P)$ as well. In that case, however, c will not have any support in variable v_i in $Bool(P)$, as a is the unique value in the domain of v_i in $Bool(P)$ and the tuple $\{(v_i, a), (v_j, c)\}$ is forbidden since it has a non-zero binary cost in P. As a result, b has to be the unique value in $D(v_j)$ with a zero unary cost. Therefore, v_j has to be VAC-integral.

$\Leftarrow$ Let v_j be a neighbour of v_i. Given that v_i is VAC-integral, we know that there exists at least one tuple such that $c_{v_i v_j}(a, c) + c_{v_j}(c) = 0$. By assumption, v_j is VAC-integral, it means that it has exactly one value b such that $c_{v_j}(c) = 0$. Therefore, there cannot be more than one tuple with $c_{v_i v_j}(a, c) + c_{v_j}(c) = 0$. As a result, v_i is also Strictly AC when all its neighbours are VAC-integral. □

6.2.2 Complexity Analysis

It is natural to ask whether the presence of a large VAC-integral subset makes the problem easier to solve, in the sense of fixed parameter tractability [Downey & Fellows 2013]. Unfortunately, this turns out not to be the case. Let us define a class of WCSPs, denoted ALMOST-INTEGRAL-WCSP, which are VAC with $n - 1$ VAC-integral variables. We show that it is NP-complete, which implies that WCSP is para-NP-complete for the parameter of number of non-VAC-integral variables.

Theorem 6.10. ALMOST-INTEGRAL-WCSP *is NP-Complete.*

Proof. Membership in NP is obvious since this is a subclass of WCSP. For hardness, we reduce from binary WCSP. Let $P = \langle V, D, C, k\rangle$ be an arbitrary WCSP and assume that it is VAC. We construct $P' = \langle V \cup V' \cup \{q\}, D \cup D' \cup D(q), C', k + |C|\rangle$, where V' and D' are copies of the variables in V and their domains D, respectively, including the unary cost functions and q has domain $\{a, b\}$ with $c_q(a) = c_q(b) = 0$. For each cost function with scope $\{v_i, v_j\}$ in C, P' has two cost functions with scopes $\{v_i, v_j, q\}$ and $\{v_i', v_j', q\}$.

Each variable in P has at least one value with unary cost 0, since it is VAC. Let this value be a for all variables. We define the ternary cost functions to be $c_{v_i v_j q}(a, a, a) = 0$, $c_{v_i v_j q}(u, v, a) = k$ when $u \neq a$ or $v \neq a$, $c_{v_i v_j q}(u, v, b) = c_{v_i v_j}(u, v) + 1$ for all u, v. Similarly, $c_{v_i' v_j' q}(a, a, b) = 0$, $c_{v_i' v_j' q}(u, v, b) = k$ when $u \neq a$ or $v \neq a$, $c_{v_i' v_j' q}(u, v, a) = c_{v_i v_j}(u, v) + 1$ for all u, v.

P' is an instance of ALMOST-INTEGRAL-WCSP with q the non-VAC-integral variable. Indeed, $c_{v_i}(a) = 0$ for all variables $v_i \in V \cup V'$ and it is supported by the

zero cost tuple (a, a, a) in each ternary constraint. All other values appear only in ternary tuples with non-zero cost hence will be pruned in $AC(Bool(P'))$.

P has optimum solution of cost c if and only if P' has optimum of cost $c + |C|$. Indeed, when we assign q to a or b, the problem is decomposed into independent binary WCSPs on V and V'. One of these admits the all-a, 0-cost assignment and the other is identical to P with an extra cost of 1 per cost function.

□

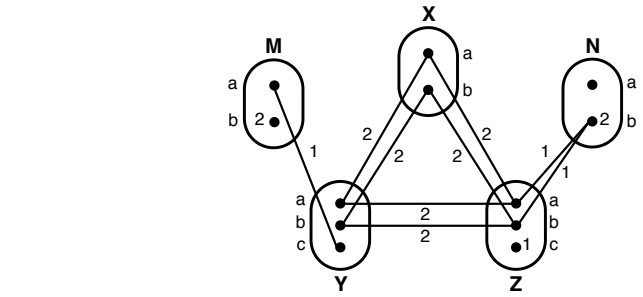
Although this construction uses ternary cost functions, we can convert them to binary using the hidden encoding [Bacchus *et al.* 2002]. This preserves arc consistency, hence it also preserves VAC, so the result holds also for binary WCSPs.

This theorem is not surprising. For example, Haller *et al.* mentioned that there is no theoretical reason to expect Strictly AC variables to maintain their assignment in any optimal solution. This just provides a specific formal explanation for this statement.

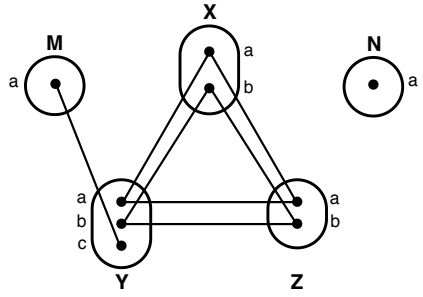
6.3 Stratified VAC

The foundation of all the heuristics we present in this chapter is the implementation of VAC in ToulBar2 [Cooper *et al.* 2010], which is restricted to binary WCSPs. In this implementation, the non-zero binary costs $c_{v_i v_j}(a, c)$ are stratified. Specifically, they are sorted in decreasing order and placed in a fixed number L of buckets. The minimum cost θ_t of each bucket $t \in \{1, \ldots, L\}$ defines a sequence of thresholds $(\theta_1, \ldots, \theta_L)$. At each θ_t from 1 to L, it constructs the $Bool_{\theta_t}(P)$, the $Bool(P)$ constructed removing values with a unary cost greater than or equal to θ_t as well as forbidding all binary relations with a binary cost greater than or equal to θ_t, and iterates on it until no domain wipe-out occurs. After θ_L, it follows a geometric schedule $\theta_{t+1} = \frac{\theta_t}{2}$ until $\theta_t = 1$. The reader is referred to [Cooper *et al.* 2010] for more details.

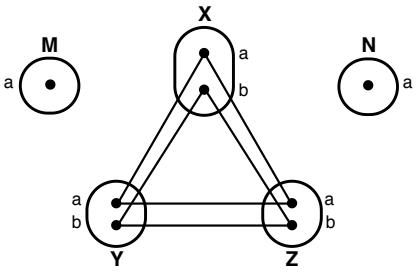
For a smaller θ_t, $Bool_{\theta_t}(P)$ is more restricted, i.e. the domain sizes are reduced. The overall, informal observation is that the set of VAC-integral variables expands as θ_t gets smaller, and saturates at some point, which is usually before $\theta_t = 1$. However, even after this saturation, the domain sizes of non-VAC-integral variables do not necessarily cease shrinking. For the heuristic we present in Section 6.5, we aim to choose a threshold θ where we have a good compromise between the number of VAC-integral variables and the domain sizes of non-VAC-integral variables. This way, we increase the size of the easy (VAC-integral) part and decrease the complexity of the difficult part, while hopefully keeping most (perhaps all) values appearing in the optimal solution. In an informal sense, we consider those VAC-integral variables that were present with a higher θ to be more informative, and more likely to appear in an optimal solution. For example, if we assign against the VAC-integral value for a high value θ_t, the cost of the best possible solution is at least $c_\varnothing + \theta_t$, whereas for $\theta_{t'} = 1$, the cost of the best possible solution that disagrees with the VAC-integral value can only be shown to be $c_\varnothing + 1$. Thus, the



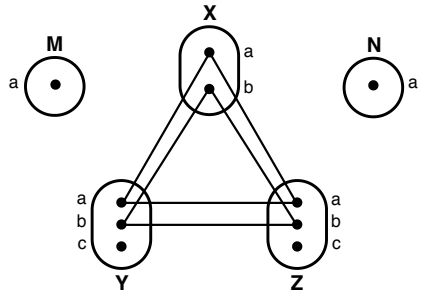
(a) A WCSP P.



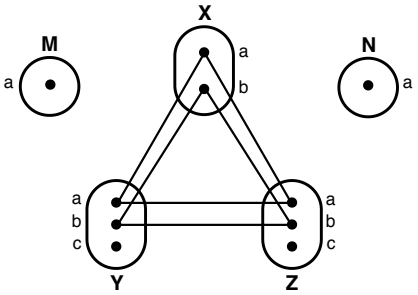
(b) $Bool_1(P)$



(c) $AC(Bool_1(P))$



(d) $Bool_2(P)$



(e) $AC(Bool_2(P)) = Bool_2(P)$

Figure 6.3: A WCSP P, $Bool(P)$ constructed with $\theta = 1$ and $\theta = 2$ together with their AC closures.

higher the θ for which a variable is VAC-integral, the less tight the relaxation needs to be for the corresponding VAC-integral value to appear in an optimal solution.

Example 6.11. *Let $P = \langle V, D, C, k\rangle$ be a WCSP where $V = \{M, X, Y, Z, N\}$, $D = \{\{a, b\}, \{a, b\}, \{a, b, c\}, \{a, b, c\}, \{a, b\}\}$, C is the set of cost functions and k is an arbitrary upper bound. This instance is presented as a cost function network in Figure 6.3a.*

Following Proposition 6.7, we know that variables M and N are VAC-integral with value a.

The optimal solutions for P, which all have a total cost of 1, are as follows:

- $M = a$, $X = a$, $Y = b$, $Z = c$, $N = a$
- $M = a$, $X = b$, $Y = a$, $Z = c$, $N = a$

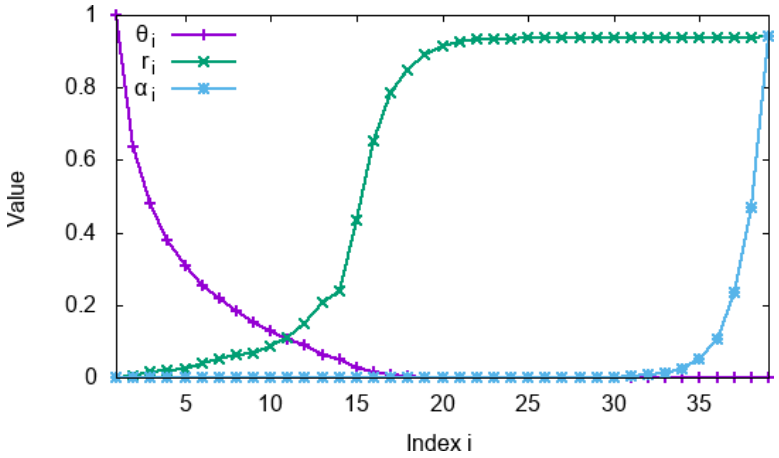


Figure 6.4: Evolution of θ_t, r_t and α_t over iterations.

- $M = a$, $X = a$, $Y = c$, $Z = b$, $N = a$
- $M = a$, $X = b$, $Y = c$, $Z = a$, $N = a$

Here, we should notice two things. First one is that M and N are both assigned to their VAC-integral value a in all optimal solutions. This observation, despite having no theoretical implications as discussed in Section 6.2.2, remains useful. Second one is that either Y or Z has to be assigned to c in any optimal solution. The value c of these two variables does not appear in $Bool_1(P)$, however, it appears in $Bool_2(P)$. This observation will be relevant in Section 6.5.

In Figure 6.4, we see the evolution of relevant parameters over VAC iterations $t \in \{1, \ldots, 39\}$ for the cnd1threeL1_1228061 instance from the Worms benchmark. These include the threshold θ_t, the ratio of VAC-integral variables r_t, and the value $\alpha_t = r_t/\theta_t$ shown by purple, green and blue curves, respectively. Note that the θ_t's are normalised, ranging from 1 to $\frac{\theta_{39}}{\theta_1}$, in order for the plot to be readable. The ratio of VAC-integral variables start from 0 and goes until 0.94 in the last iteration, when $\theta_{39} = 1$. Note that the α_t has the same range as r_t, although a different behaviour.

6.4 Branching Heuristics based on VAC-integrality

For a branch-and-bound algorithm, the order in which variables are assigned has a crucial impact on the performance. In general, a branching decision should help the solver quickly prune sub-trees which contain no improving solutions, by creating sub-problems with increased dual bound in all branches [Achterberg 2009].

Based on this observation and the connection of VAC-integrality to integrality explained in Section 6.2, we observe that branching on a VAC-integral variable v will create a branch which must contain the VAC-integral value a of v. Since a is the only value in the domain of v in $Bool(P)$ and its unary cost does not

change by branching, $Bool(P\ |_{v=a})$ is identical to $Bool(P)$, so the dual bound is not improved in this branch. Therefore, it makes sense to avoid branching on VAC-integral variables, although, as a heuristic we cannot expect this to always be the best choice.

To implement this, we find the set of VAC-integral variables implied by Proposition 6.7, i.e., those that have singleton domain in $Bool(P)$ and only allow branching on the rest. The choice among the rest of the variables is made using whatever branching heuristic the solver uses normally. In the case of ToulBar2, which we use in our implementation, that is DOM+WDEG [Boussemart *et al.* 2004] together with the last conflict heuristic [Lecoutre *et al.* 2009].

The motivation behind this idea is the possibility that it might help us detect a conflict during the search procedure earlier. If we assume that the partial solution for the VAC-integral part is consistent with the optimal solution, when we branch on a VAC-integral variable, we can only choose its VAC-integral value which has a cost of zero. In this case, we gain no new information. On the other hand, if we insist on choosing a non-zero-cost value, then we immediately know that we are going in a bad direction and thus increasing the cost. Therefore, branching on a non-VAC-integral variable gives us a chance to find the optimal solution, and makes the propagation stronger. This is because, if the partial solution for the VAC-integral part is not consistent with the optimal solution, branching on non-VAC-integral variables will (if the propagation is strong enough) change the set of VAC-integral variables, or, at least change the values with zero unary cost. On the contrary, branching on VAC-integral variables will still make no change to the problem if we use the zero-cost values.

When only VAC-integral variables remain, we assign them all at the same time and check that the lower bound did not increase (see premature termination of VAC in [Cooper *et al.* 2010]). If so, we update the upper bound if a better solution was found, unassign VAC-integral variables, and keep branching with the default heuristic.

For efficiency reasons, during search, EDAC [de Givry *et al.* 2005] is established before enforcing VAC in ToulBar2. If the VAC property cannot be enforced at a given search node due to premature termination of VAC, then VAC-integrality is unavailable for that node and again we rely on the default branching heuristic. We tried to exploit the last valid VAC-integrality information collected along the current search branch, but it did not improve the results.

6.4.1 Exploiting Larger Zero-cost Partial Assignments

Definition 6.3 requires there is a unique VAC-integral value $a \in D(v_i)$ for each VAC-integral variable. The partial assignment of unique values to their corresponding variables implies a zero-cost lower-bound increase as said before. Thus, our branching heuristic will avoid branching on these variables. We could search for other potentially-larger assignments with the same zero-cost property. A simple way to do that is to test a particular value assignment and keep the variables not in

conflict with it, *i.e.,* with no cost violations related to them or with their neighbours. We choose first to test the assignment based on *EAC values*, which are maintained by EDAC [Heras & Larrosa 2006]. An EAC value is defined like a VAC-integral value but it is not required to be unique in the domain. If a variable is kept, it means that its EAC value is fully compatible, *i.e.,* has zero-cost, with all the EAC values of its neighbours. We call it a *Full EAC value*. This approach can be combined with VAC-integrality. By restricting the EAC values to belong to the AC closure of $Bool(P)$ when the problem is made VAC, we ensure that any VAC-integral value is also Full EAC. The opposite is not true (see Figure 6.1). Thus, the set of Full EAC values can be larger. We perform the Full EAC test in an incremental way (using a variable's revise queue based on EAC value changes) at every node of the search tree, before choosing the next non Full EAC variable to branch on.

In order to improve this approach further, as soon as a new solution is found during search, EDAC will prefer to select the corresponding solution value as its EAC value for each unassigned variable if this value belongs to the current set of feasible EAC values. By doing so, we may exploit larger zero-cost partial assignments found previously during search. Notice that our branching heuristic is related to the min-conflicts repair algorithm [Minton *et al.* 1992] as it will only branch on variables in conflict with respect to a given assignment. Exploiting the best solution found so far for value heuristics has been shown to perform well on several constraint optimisation problems when combined with restarts [Demirovic *et al.* 2018]. We use such a value ordering heuristic inside the HBFS algorithm of ToulBar2 in all our experiments.

6.5 Relaxation-Aware Sub-Problem Search (RASPS)

One problem that branch-and-bound faces, especially in depth-first order, is that without a good upper bound it may explore large parts of the search tree that contain only poor quality solutions.

Here, we propose to use integrality information to try to quickly generate solutions that are close to the optimum. We describe a primal heuristic that we call Relaxation-Aware Sub-Problem Search (RASPS), which runs in preprocessing. We simply fix all VAC-integral variables to their values, prune values from the rest of the variables that are pruned in $AC(Bool(P))$, and then solve using the EDAC lower bound the resulting subproblem to optimality or until a resource bound is met[2]. In order to choose the set of VAC-integral variables, we use the dual solutions constructed in iterations of VAC before the last, hence examine $Bool_\theta(P)$ for an appropriate θ.

Although the idea of the heuristic is pretty straightforward, the key issue is to choose the threshold value (the θ) (recall Section 6.3) to construct $Bool_\theta(P)$, as it has an impact on the quality of the upper bound produced and the time spent for this. To determine the threshold value for the RASPS, we observe the curves of

[2]In our implementation, we set an upper bound of 1000 backtracks for solving the subproblem.

the threshold θ_t, the ratio of VAC-integral variables r_t, and the value $\alpha_t = r_t/\theta_t$, collected during VAC iterations. The idea is that, once the ratio of VAC-integral variables saturates, θ_t continues to decrease. As a result, α_t starts increasing more quickly which is the desired cutoff point. To identify that point, we track the curve of α_t over the VAC iterations and choose the threshold value when the angle of the curve reaches 10 degrees (see Figure 6.4). The cutoff angle of 10 degrees was chosen experimentally, as we found that this exhibited the best performance and greatest robustness empirically.

Example 6.12. *Let $P = \langle V, D, C, k\rangle$ be a WCSP where $V = \{M, X, Y, Z, N\}$, $D = \{\{a, b\}, \{a, b\}, \{a, b, c\}, \{a, b, c\}, \{a, b\}\}$, C is the set of cost functions and k is an arbitrary upper bound. This instance is presented as a cost function network in Figure 6.5a. The same instance was used in Example 6.11.*

In Figures 6.5b and 6.5e, we see $Bool(P)$ constructed with $\theta = 1$ and $\theta = 2$, respectively, and in Figures 6.5c and 6.5f we see their AC closures.

We obtain the sub-WCSP to be solved by RASPS by doing two things:

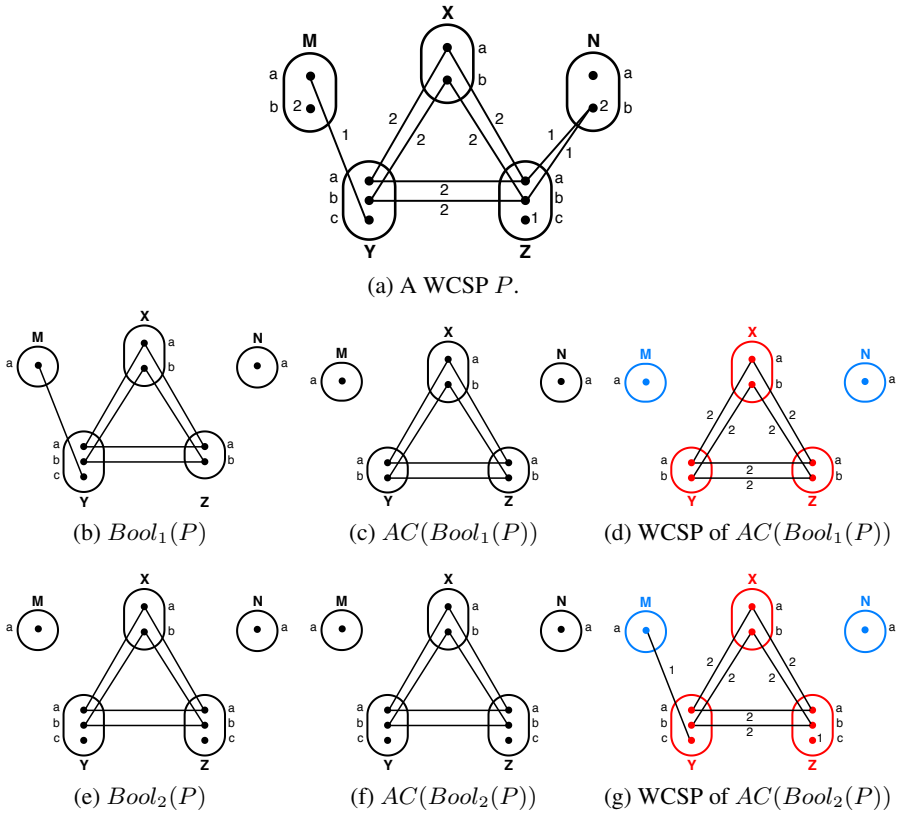
1. *Pick a threshold value θ to construct $Bool_\theta(P)$*
2. *Plug back the costs in the original WCSP P into $AC(Bool_\theta(P)$*

Once we have the sub-WCSP, we can categorise variables as VAC-integral (blue) and others (red). The set of blue variables correspond to the integral (easy) part of the problem and they are immediately assigned to their VAC-integral value, whereas that of red variables correspond to the combinatorial (difficult) part.

Two candidate subproblems can be seen in Figures 6.5d and 6.5g. Obviously, $\theta = 2$ is a better choice for the threshold as we know from Example 6.11 that either Y or Z has to be assigned to c in the optimal solution. With $\theta = 1$, we miss that.

This idea is related to the COMBILP method of Haller et al. [Haller *et al.* 2018], described earlier, which uses strict arc consistency to decompose the problem into an integral partition and the combinatorial partition and successively refines them until they agree. They do this by identifying the Strictly AC variables on the dual solution produced. The combinatorial part is solved exactly and if that solution can be combined with the easy part without incurring extra cost, it reports optimality. Otherwise, it moves some variables from the easy part to the combinatorial part and iterates until it identifies exactly the set of integral variables in the relaxation that appear in an optimal solution.

Compared to COMBILP, RASPS solves a simpler combinatorial subproblem because of the larger VAC-integral set and the remaining pruned domains. Then, it only aims to produce a good initial upper bound and leaves proving optimality to the branch-and-bound solver. Even more closely related is the RINS heuristic of Danna et al. [Danna *et al.* 2005]. It also searches for primal bounds by extending the integral part of the relaxation. In contrast to RASPS, it permits values of the incumbent solution and may be invoked in nodes other than the root. However,



(a) A WCSP P.

(b) $Bool_1(P)$ (c) $AC(Bool_1(P))$ (d) WCSP of $AC(Bool_1(P))$

(e) $Bool_2(P)$ (f) $AC(Bool_2(P))$ (g) WCSP of $AC(Bool_2(P))$

Figure 6.5: Stratified VAC and RASPS examples for two different thresholds. Blue variables are VAC-integral. See Example 6.12.

it has no way of distinguishing among integral variables as RASPS does with its choice of $\theta > 1$. We have experimented with RASPS during search but have so far not found it worthwhile.

6.6 Experimental Results

We have implemented VAC-integrality and RASPS inside ToulBar2, an open-source exact branch-and-bound WCSP solver in C++ [3]. All computations were performed on a single core of Intel Xeon E5-2680 v3 at 2.50 GHz and 256 GB of RAM with a 1-hour CPU time limit. No initial upper bounds were used, as is the default of the solver.

6.6.1 Benchmark Description

We performed experiments on probabilistic and deterministic graphical models coming from different communities [Hurley *et al.* 2016]. We considered a large set

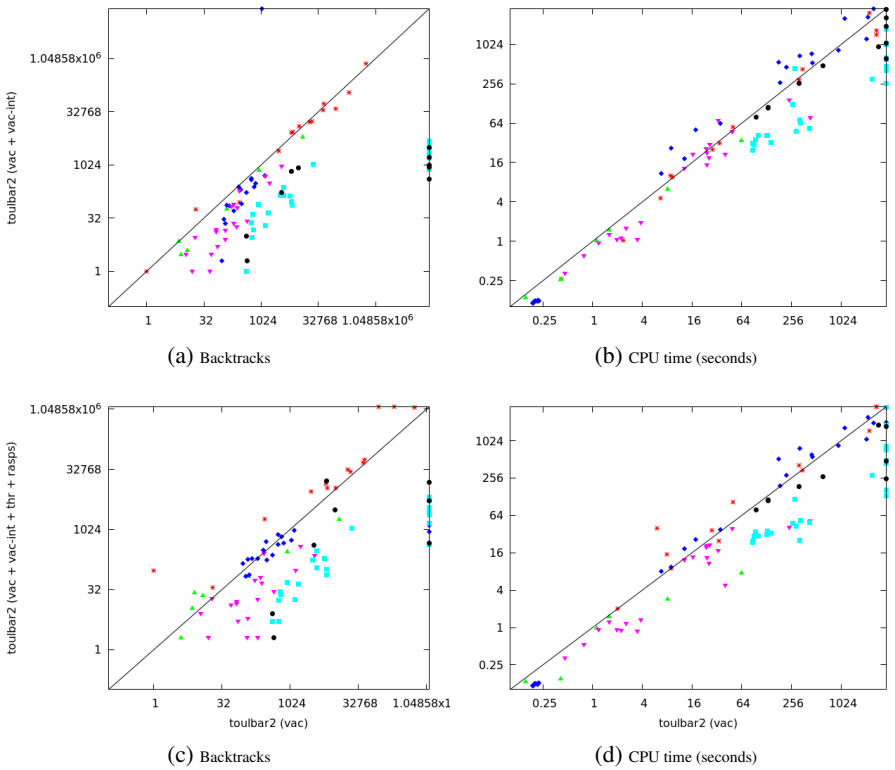
[3] `https://github.com/toulbar2/toulbar2`, branch *fural/strictac* from version 1.0.1.

of 431 instances[4] which are all binary. It includes 251 instances (170 Auction, 16 CELAR, 10 ProteinDesign, 55 Warehouse) from the Cost Function Library[5], 129 instances (108 DBN, 21 ProteinFolding) from the *Probabilistic Inference Challenge* (PIC 2011)[6], 30 "Worms" instances [Kainmueller *et al.* 2014] where COMBILP is state-of-the-art [Haller *et al.* 2018], and 21 Computational Protein Design (CPD) large instances for which ToulBar2 is state-of-the-art [Allouche *et al.* 2014, Ouali *et al.* 2020]. We discarded Max-CSP, Max-SAT, Constraint Programming (CP), and Computer Vision (CVPR) instances which are either unweighted (all costs equal to 1), or non-binary, or being too easy (small search tree) or unsolved by all the tested approaches including MRF and ILP solvers [Hurley *et al.* 2016].

6.6.2 Comparison with VAC

First, we compared our new heuristics with default VAC maintained during search (option -A=999999 for all tested methods). We skipped Auction and DBN as they

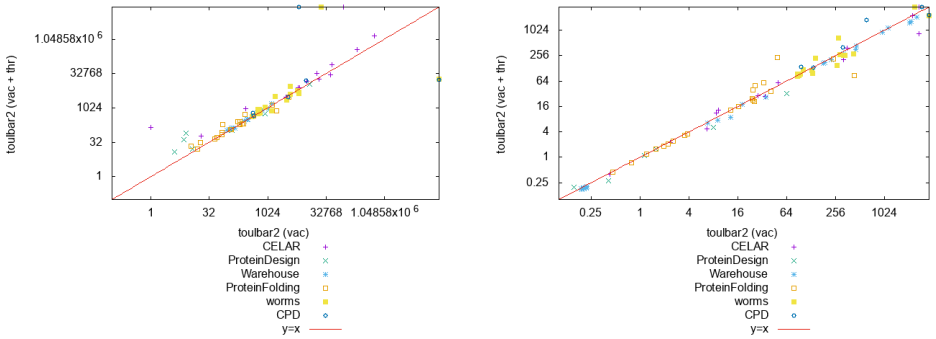
[4] genoweb.toulouse.inra.fr/~degivry/evalgm
[5] forgemia.inra.fr/thomas.schiex/cost-function-library
[6] www.cs.huji.ac.il/project/PASCAL



(a) Backtracks (b) CPU time (seconds)

(c) Backtracks (d) CPU time (seconds)

Figure 6.6: Comparison with ToulBar2 using VAC during search. CELAR: ✳, CPD: ●, ProteinDesign: ▲, ProteinFolding: ▼, Warehouse: ◆, Worms: ■.



(a) Backtracks (b) CPU time (seconds)

Figure 6.7: Comparison with and without threshold for VAC during search.

do not have VAC-integral variables.

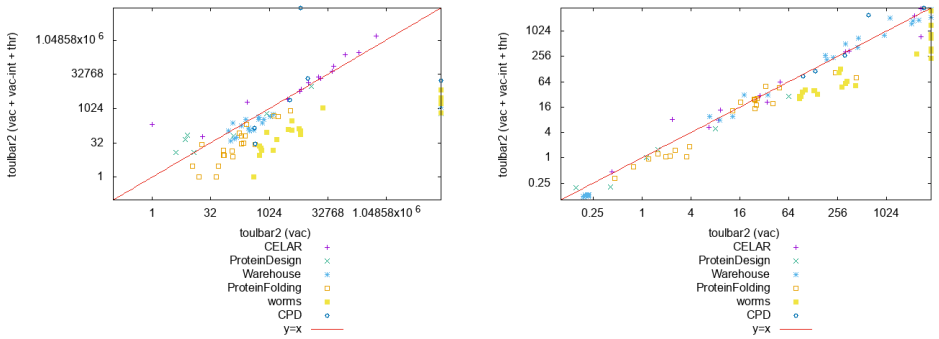
In Figure 6.6a, we show a scatter plot comparing the number of backtracks between VAC and VAC exploiting VAC-integral variable heuristic. The size of the search tree is significantly reduced thanks to VAC-integrality for most instance families. Notice the logarithmic axes. The improvement in terms of CPU time (Figure 6.6b) is less important but still significant for CPD, ProteinFolding, Worms, and some CELAR instances. However, we found several Warehouse instances where it was significantly slower using VAC-integrality. In this case, we found the explanation was a larger number of VAC iterations per search node (8 times more in average) corresponding to small lower bound improvements at small threshold values (θ near 1) that did not reduce the search tree sufficiently (only by a mean factor 2.2 on difficult Warehouse instances).

In order to avoid such pathological cases, we placed a bound on the minimum threshold value θ for VAC iterations during search. We selected the same limit as for RASPS (*e.g.,* θ_{30} for Worms/cnd1threeL1_1228061).

We found that using this threshold mechanism alone speeds up Warehouse resolution and does not significantly deteriorate the results in the other families (see Figure 6.7). Furthermore we obtained consistent results when combining with VAC-integrality, reducing the number of backtracks and CPU time for several families while being equivalent for the others (see Figure 6.8).

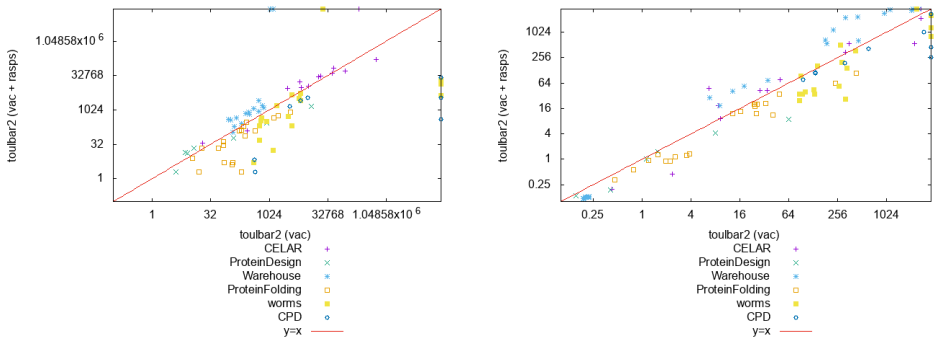
Next, we analysed the impact of applying the RASPS upper-bounding procedure in preprocessing. We limit RASPS to 1000 backtracks. Again, our new heuristic RASPS significantly reduces the search effort in terms of backtracks and time, except for Warehouse and some CELAR. For Warehouse, the upper bounds found did not reduce the total number of backtracks. For CELAR scen06_r, it reduces backtracks by 3.4 and solving time by 4.2. For Worms, it was more than 10 times faster for some instances (see Figure 6.9).

Finally, we combine the two heuristics, VAC-integrality and RASPS, with VAC threshold limit and show the results compared to VAC alone in Figure 6.6c and 6.6d.



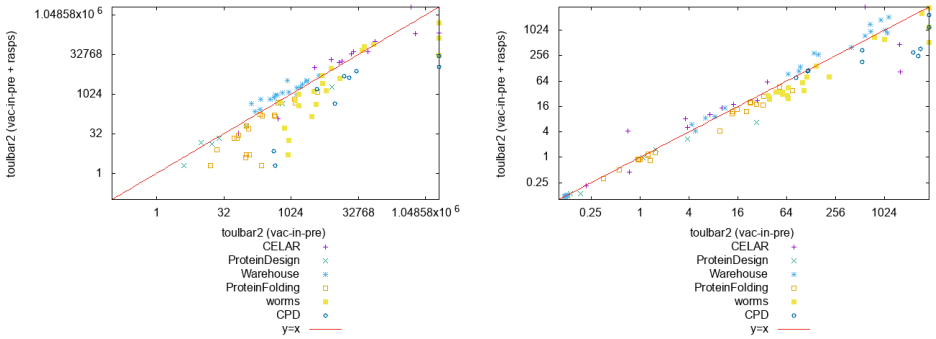
(a) Backtracks (b) CPU time (seconds)

Figure 6.8: Comparison with and without VAC-integrality and threshold for VAC during search.



(a) Backtracks (b) CPU time (seconds)

Figure 6.9: Comparison with and without RASPS in preprocessing and VAC during search.



(a) Backtracks (b) CPU time (seconds)

Figure 6.10: Comparison with and without RASPS for VAC in preprocessing and EDAC during search.

We keep this best configuration in the rest of the paper.

6.6.3 Comparison with VAC-in-preprocessing and COMBILP

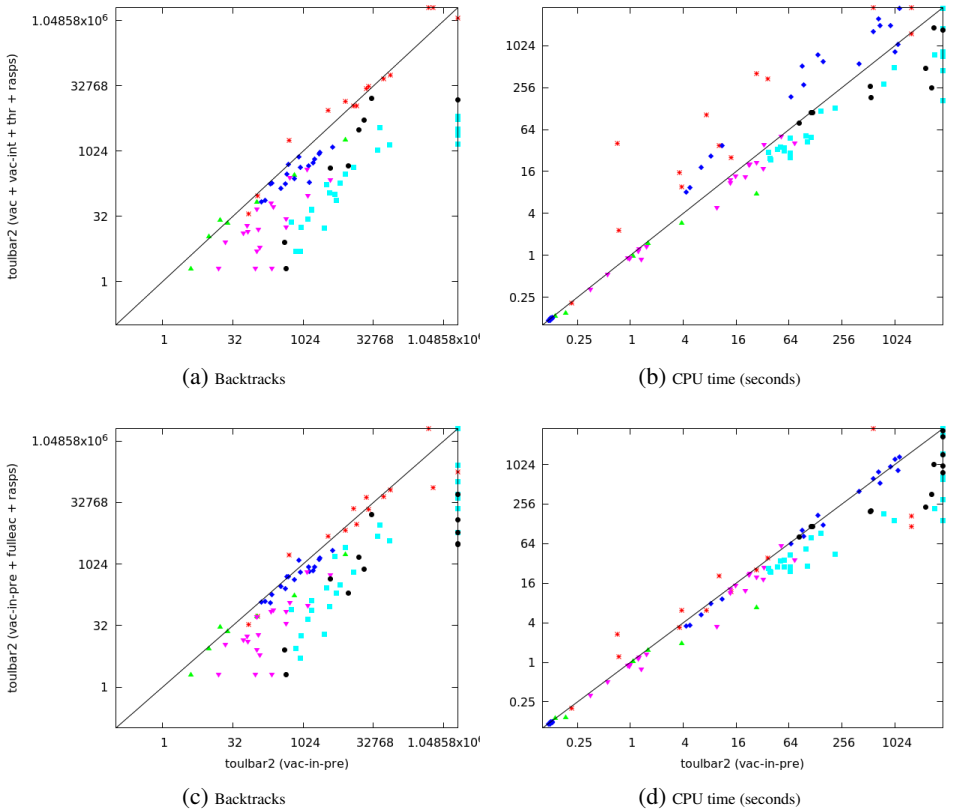
One might expect using VAC only in preprocessing to be the fastest option, as it is the default for ToulBar2 and significantly outperforms VAC during search in most cases [Hurley *et al.* 2016]. However, for certain instance families, we manage to outperform it.

When VAC is used only in the preprocessing, using RASPS in addition considerably improves runtimes except for Warehouse and some CELAR (see Figure 6.10). If we add VAC-integrality and RASPS when using VAC during search, we manage to outperform VAC in preprocessing for all families except CELAR and Warehouse, where the overhead of VAC is too high (see Figure 6.11a and 6.11b).

Moreover, if we compare methods using VAC in preprocessing only, then exploiting our simpler Full EAC branching heuristic and RASPS performs even better in most cases, being as good as default ToulBar2 on Warehouse instances (55 instances solved in average in 128 seconds) and comparable on CELAR (our approach solved graph13 and scen06 one-order-of-magnitude faster, but could not solve graph11 compared to default VAC in preprocessing, see Figure 6.11c, 6.11d).

See in Figure 6.12, anytime profiles on the CELAR graph11 instance [Cabon *et al.* 1999]. For ColorSeg, InPainting, ObjectSeg, Coloring and ProteinFolding, using RINS in preprocessing in addition to VAC also results in better CPU times compared to VAC alone. VAC-integrality results in a significant decrease in CPU times when used with VAC during search, for EHI, QCP, DBN, ImageAlignment and ProteinFolding.

Next, we compare ToulBar2 and COMBILP (which uses the same ToulBar2 as its internal ILP solver) with different lower bound techniques, showing the advantages of exploiting VAC-integrality or Full EAC and RASPS extensions.



(a) Backtracks (b) CPU time (seconds)

(c) Backtracks (d) CPU time (seconds)

Figure 6.11: Comparison with default ToulBar2. CELAR: ✳, CPD: ●, ProteinDesign: ▲, ProteinFolding: ▼, Warehouse: ◆, Worms: ■.

In Figure 6.13a, we see a cactus plot[7] for the Worms benchmark where there are 30 instances. We solved these instances with different combinations of solvers and heuristics with a CPU time limit of 1 hour. CombiLP was reported to solve 25 of these instances in [Haller *et al.* 2018] within 1 hour CPU time. Here, we compare CombiLP with parameters used in [Haller *et al.* 2018] (VAC in preprocessing and EDAC during search), as well as our version of ToulBar2 plugged in it. In addition to those, we have standalone ToulBar2 either with VAC in preprocessing and EDAC during search, with or without Full EAC, or VAC-integrality-aware branching, and RASPS options. ToulBar2 alone can go up to 25 instances. However, by plugging our version of ToulBar2 in CombiLP, we manage to solve 26 of these instances, which makes 1 more than [Haller *et al.* 2018]. Another important detail is that, although it is costly to use VAC throughout the search tree, it becomes better with VAC-integrality and RASPS. Still, it was slightly dominated by Full EAC.

This simpler heuristic performed even better on the CPD benchmark (Figure 6.13b). Our Full EAC heuristic with RASPS got the best results, solving 13

[7] It shows on the x-axis the number of instances solved for a time limit given in y-axis.

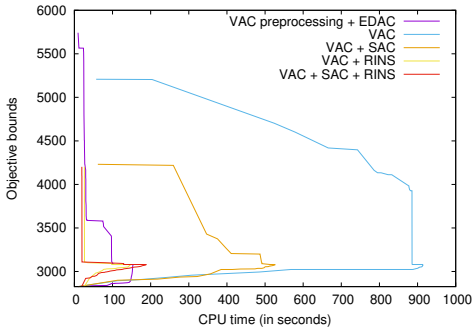
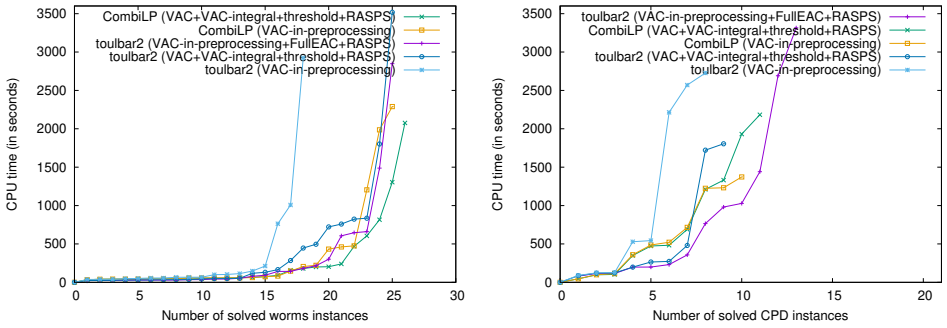


Figure 6.12: CELAR graph11 instance.



(a) Worms instances.

(b) CPD instances.

Figure 6.13: Comparison with COMBILP on Worms and CPD instances.

instances, compared to VAC-integrality and RASPS which solves 9, and only 8 by default ToulBar2. COMBILP using VAC during search with VAC-integrality and RASPS solved 11 instances, instead of 10 without these options and VAC in preprocessing.

6.7 Conclusion

We revisited the Strict Arc Consistency property which was recently used in an iterative relaxation solver. We identified properties that make it easier to use within a branch-and-bound solver and in particular in conjunction with the VAC algorithm. This property allows us to integrate information about the relaxation that VAC computes to be used in heuristics. We presented three new heuristics that exploit this information, two for branching and the other for finding good quality upper bounds. In an experimental evaluation, these heuristics showed great performance in some families of instances, improving on the previous state of the art. VAC-integrality identifies a single zero-cost satisfiable partial assignment in a particular CSP $Bool(P)$ of the original problem P. Other CSP techniques such as neighbourhood substitutability [Freuder 1991] could be used to detect larger

tractable sub-problems. The integral subproblem can also be viewed as a particularly easy tractable class, where each variable has a single value. Therefore, another possible direction is to detect subproblems that are tractable for more sophisticated reasons.

CHAPTER 7

Conclusion

Achievements

In this thesis, we made contributions to the state of the art in learning the structure of BNs, namely the Bayesian Network Structure Learning problem (BNSL), and answering Most Probable Explanation (MPE) and minimisation queries on Bayesian networks (BNs) and Cost Function Networks (CFNs), which are both NP-hard.

The common, and the most important purpose of all the implementations presented in this work has been to achieve a better trade-off between inference strength and speed of inference, whatever the problem is. It is true that when the inference mechanisms are lightweight, it becomes easier to explore a plethora of search nodes per second. Unfortunately, with a lightweight inference, one is also more likely take into consideration only limited information about the current state of the search. On the other hand, a strong inference mechanism provides a more insightful perspective over the consequences of decisions taken during the search, which can indeed lead to more effective decisions later on. However, in most cases encountered in real-life applications, this operation gets so costly that, inevitably we question if it is worth the effort. All of this motivated us to seek a balance between these two key notions: speed and strength.

For BNSL, existing algorithms opted for either maximal strength of inference, like the algorithms based on Integer Programming (IP) and branch-and-cut, or maximal speed of inference, like the algorithms based on Constraint Programming (CP). Our first focus was on how to perform extensive value pruning faster. A previous work [Hoffmann & van Beek 2013] provided a GAC propagator for the global acyclicity constraint with a time complexity of $O(n^3d^2)$. Given the large domain sizes BNSL problems inherently have, having a quadratic factor of d is, to say the least, catastrophic. We managed to design a GAC propagator with complexity $O(n^3d)$, and implemented it in our solver ELSA, which is based on the state-of-the-art CP-based solver CPBayes v1.1 [Van Beek & Hoffmann 2015]. The results presented in Chapter 3 proved that as problem sizes grow, so does the time gained thanks to GAC.

Continuing with BNSL, we next put a spotlight on lower bound generation. Given the strong lower bounds provided by the state-of-the-art IP-based method GOBNILP [Cussens 2011, Barlett & Cussens 2013], we sought ways to implement a similar idea in our CP-based solver in order to perform a more efficient search. In Chapter 4, we specified properties of a specific class of inequalities, called cluster

inequalities, which lead to an algorithm that performs much stronger inference than that based on CP, much faster than that based on IP.

The major obstacle with BNSL problems is the exponential growth of domain sizes with every additional variable. This is inevitable to a certain extent, therefore it is only reasonable to look for ways to represent domains in a more compact manner. To this end, we came up with a Binary Decision Tree (BDT) implementation. BDTs allowed us to run queries over domain values much faster, and led to an additional performance boost.

For minimisation queries in Cost Function Networks (CFNs), we identified a weakness in the use of linear programming relaxations by a specific class of solvers, which includes the award-winning open source ToulBar2 solver [Cooper *et al.* 2010]. We proved that this weakness can lead to suboptimal branching decisions and show how to detect maximal sets of such decisions, which can then be avoided by the solver. This allowed ToulBar2 to tackle problems previously solvable only by hybrid algorithms.

Perspectives

A potential improvement lies in the way we solve $LP_{\mathcal{W}}$, the LP introduced in Chapter 4. Existing nearly linear time algorithms for solving positive LPs [Allen-Zhu & Orecchia 2015], or another solver which is suited to the clausal constraints we have might be worth trying. Additionally, we would try getting even closer to the VAC algorithm, although there is the limitation that currently the VAC algorithm is not generalised for clausal constraints. In case we decide to try this, whether the decision trees would help is an interesting question.

Another direction is definitely a more sophisticated implementation of the BDTs. The information gain heuristic, especially for certain datasets as we saw in Chapter 5, may turn out to be too costly to be employed throughout the construction of a BDT. In this case, switching off the information gain heuristic at a certain depth, or not using it at all for some variables, is envisageable.

There is also the question of whether computing parent set scores during search would help or not. A large part of the computational cost is not even present in the runtime of ELSA, since the scores are already computed before we ever launch ELSA. Doing this can take a very long time, even more than 24 hours in some cases. If we bring it into ELSA, it will indeed slow down the solver, but it might improve the overall time needed to compute scores and find the optimal structure.

Yet another research direction is probably an unexpected one for the reader who went through all the chapters until here, as it was never mentioned at all: investigating the IP-based methods developed for the well-known Travelling Salesman Problem (TSP). The ILP formulation of BNSL without the cluster constraints (CCs) is basically the Assignment Problem (AP). Likewise, TSP without the subtour elimination constraints (SECs) is also equivalent to the AP. The number of CCs and SECs being exponential in the number of variables, along with the fact that the LP

relaxation of AP is very strong, branch-and-cut is a popular choice for IP-based implementations both for BNSL and TSP. Another common point between CCs and SECs is their semantics. CCs eliminate *all* cycles whereas SECs eliminate *all but one*: the cycles that cover the entire set of variables. So in a way, SECs are a subset of CCs, and any method developed to handle SECs can be relevant for CCs.

This is the end of my doctoral thesis, originally written in English. The following chapters are the exact translation into French of the original seven chapters, done by the Pro version of DeepL [1] translator, *and not by myself*, although I have corrected some inevitable mistakes. The one and only aim of this translated part is to cater to those who are more comfortable to read in French. Therefore, the French part *should not be taken as a reference*.

[1] https://www.deepl.com/translator

CHAPTER 1

Introduction

Tous les problèmes de la vie réelle que nous rencontrons impliquent une combinaison de plusieurs difficultés : trop de variables, trop de contraintes, un niveau d'incertitude arbitraire, un temps limité pour trouver une solution et la liste est longue. Si nous essayons de résoudre de tels problèmes avec un papier, un stylo et un bon raisonnement, il ne nous faudra pas beaucoup de temps pour voir que c'est loin d'être faisable. D'autre part, si nous demandons à un ordinateur d'effectuer exactement la même procédure que celle que nous avions initialement prévu de faire à la main, nous serons très probablement déçus de voir que la qualité de la solution obtenue est plutôt faible. Lorsque nous pensons avoir enfin trouvé la bonne approche censée nous donner la solution optimale, l'ordinateur peut avoir d'autres plans et mettre des siècles à trouver une seule solution réalisable.

L'optimisation discrète est un sous-domaine des mathématiques appliquées qui traite des problèmes où les variables de décision prennent un ensemble fini de valeurs (presque toujours entières). La plupart des décisions dans la vie réelle, notamment dans l'industrie, impliquent des décisions discrètes telles que la planification des employés [Ağralı *et al.* 2017], la planification des ventes et des opérations [Taşkın *et al.* 2015], et la planification de la production [Güngör *et al.* 2018]. La *Programmation en nombres entiers* (PI) et la *Programmation par contraintes* (PC) sont deux des paradigmes les plus courants pour aborder de tels problèmes.

Un autre défi souvent rencontré dans les problèmes de la vie réelle, outre l'exigence d'intégralité, est l'abondance de variables. Le problème à ce stade est que, chaque fois que nous devons définir une certaine fonction sur tant de variables, la description de cette fonction croît exponentiellement avec le nombre de variables. Cela rend la tâche de *raisonnement* nécessaire à la résolution des problèmes d'optimisation discrète beaucoup plus difficile.

Les *modèles graphiques* sont un type de structure de données et des outils très utiles pour le raisonnement. Les modèles graphiques non dirigés tels que les *Réseaux de fonction de coût*, alias *Problèmes de satisfaction de contraintes pondérées* (WCSP), et les *Champs aléatoires de Markov* (MRF) peuvent être utilisés pour donner une représentation factorisée d'une fonction, dans laquelle les sommets d'un graphique représentent les variables de la fonction et les (hyper)arêtes représentent les facteurs. Les facteurs peuvent être, par example, *fonctions de coût*, auquel cas le modèle graphique représente une factorisation d'une fonction de coût, ou *tableaux de probabilité locaux*, auquel cas le modèle représente une distribution de probabilité conjointe non normalisée [Koller & Friedman 2009].

Les deux modèles, WCSP et MRF, sont équivalents sous une transformation

$-\log$, donc la requête NP-complète de minimisation des coûts dans WCSP est équivalente à la requête d'affectation maximum a posteriori (MAP) dans MRF. Ce problème d'optimisation a des applications dans de nombreux domaines, comme l'analyse d'images [Li 2009, Geman & Graffigne 1986, Li 1994], la reconnaissance vocale [Gravier *et al.* 1999, Gravier *et al.* 2000], et la bioinformatique [Allouche *et al.* 2014].

Parfois, l'optimisation est un moyen (vital) plutôt qu'une fin. Cela se produit dans le contexte des réseaux Bayésiens.

Les réseaux Bayésiens (RB) [Pearl 1988] sont des modèles graphiques probabilistes qui représentent des distributions de probabilité conjointes sur des variables aléatoires. Souvent, les relations de dépendance entre les variables aléatoires ne sont pas connues, et déterminer cette relation analytiquement est loin d'être trivial. Dans ce cas, la méthode habituelle consiste à prendre en entrée un ensemble d'observations conjointes des variables et à trouver une structure BN, c'est-à-dire le graphe acyclique dirigé (DAG) sous-jacent, où chaque nœud correspond à une variable aléatoire et chaque arête dirigée encode une dépendance conditionnelle directe, et qui explique le mieux les données. C'est ce que l'on appelle le problème de l'*apprentissage de la structure du réseau bayésien (BNSL) à partir de données discrètes*. La difficulté ici est que la taille de l'espace de recherche des DAGs est exponentielle en nombre de variables aléatoires. Par conséquent, une approche intelligente doit être développée.

De nombreuses méthodes approximatives [Behjati & Beigy 2020, Dai *et al.* 2020, Liu *et al.* 2017, Scutari *et al.* 2019] et exactes ont été développées pour BNSL au cours des dernières décennies. Les approches exactes comprennent la programmation dynamique [Silander & Myllymäki 2006, Malone *et al.* 2011a], la recherche heuristique [Yuan & Malone 2013, Fan & Yuan 2015], la programmation par contraintes [Van Beek & Hoffmann 2015], la satisfiabilité maximale [Berg *et al.* 2014, Cussens 2008], la recherche par branchement de type breadth-first [Campos & Ji 2011, Fan *et al.* 2014, Malone *et al.* 2011b] et la programmation linéaire en nombres entiers [Bartlett & Cussens 2017, Cussens *et al.* 2017].

Une façon d'apprendre des BN à partir de données est la méthode de score-and-search [Heckerman *et al.* 1995], qui est la méthode que nous utilisons principalement dans ce travail. Cette méthode utilise un ensemble précalculé de scores pour formuler le BNSL comme un problème d'optimisation. Pour chaque variable, il existe un ensemble prédéfini d'ensembles parents candidats avec leurs scores associés. Avec ces informations, il est possible de définir une fonction de coût décomposable. On essaie ensuite de trouver une solution qui minimise cette fonction de coût, qui est une somme de scores locaux, et qui renvoie un réseau acyclique.

BNSL est par essence un problème d'affectation : à chaque variable aléatoire, nous affectons un ensemble de parents. S'il n'y avait pas de contrainte d'acyclicité, BNSL serait aussi simple que d'assigner à chaque variable aléatoire son ensemble parent candidat le moins coûteux. Cependant, l'acyclicité est obligatoire, d'où la difficulté, sinon cette dissertation n'existerait pas.

L'objectif principal de cette thèse est de développer un ensemble de méthodes afin de contribuer au traitement des instances de BNSL sans limite sur l'arité des ensembles de parents. Presque tous les algorithmes que nous avons conçus à cette fin et présentés ici sont inspirés de l'algorithme bien connu de Virtual Arc Consistency (VAC) qui est largement utilisé dans le cadre du WCSP. Nous constatons que notre solveur ELSA (la deuxième Elsa qui a vu le jour au cours de ce doctorat) utilisant tous ces algorithmes améliore l'état de l'art. En prime, nous présentons également quelques heuristiques qui conduisent à une utilisation plus efficace de VAC au sein des solveurs branch-and-bound pour WCSPs.

Ce manuscrit est organisé comme suit :

- Le chapitre 2 donne le contexte nécessaire sur les notions pertinentes telles que la programmation par contraintes, la programmation linéaire en nombres entiers, les réseaux de fonctions de coût, les réseaux Bayésiens et les stratégies de recherche, ainsi que les travaux existants connexes.

- Le chapitre 3 fournit un algorithme d'inférence plus efficace pour la contrainte d'acyclicité, plus précisément un algorithme de cohérence d'arc généralisé avec une complexité asymptotique et des performances pratiques nettement améliorées.

- Le chapitre 4 présente un algorithme qui calcule une approximation d'une borne inférieure bien connue pour BNSL, qui présente des performances pratiques nettement meilleures. Plus précisément, nous donnons un algorithme en temps polynomial pour détecter un sous-ensemble de toutes les coupes de clusters violées augmentant de manière prouvée la borne inférieure et un algorithme gourmand pour augmenter la borne inférieure en utilisant les coupes de clusters trouvées.

- Le chapitre 5 démontre comment nous pouvons exploiter la structure des domaines d'ensembles parents en utilisant des arbres de décision binaires pour les maintenir et les manipuler efficacement.

- Le chapitre 6 introduit deux heuristiques qui permettent de bénéficier des informations sur la relaxation linéaire du problème que produit l'algorithme VAC.

- Le chapitre 7 conclut et indique les futures directions de recherche.

CHAPTER 2

Préliminaires

Contents

Ce chapitre présente les concepts de base de la programmation par contraintes, de la programmation linéaire en nombres entiers, des problèmes de satisfaction de contraintes pondérées et des réseaux Bayésiens que le lecteur rencontrera tout au long de cette thèse.

2.1 Programmation par Contraintes

La programmation par contraintes (Constraint Programming (CP)) [Rossi *et al.* 2006] est un paradigme visant à résoudre des problèmes combinatoires. Elle a des applications réussies dans de nombreux domaines, notamment l'ordonnancement [Baptiste *et al.* 2001, Beck *et al.* 2011, Trilling *et al.* 2006], la planification [Van Beek & Chen 1999], le routage de véhicules [Shaw 1998, De Backer *et al.* 2000, Guimarans

et al. 2011], et la bioinformatique [Barahona & Krippahl 2008, Backofen *et al.* 1999]. Les premières tentatives de représentation de problèmes contraints sous forme de réseaux remontent aux années 1970 [Montanari 1974], ainsi que les stratégies de résolution [Freuder 1978, Freuder 1982, Borning 1981] et les solveurs basés sur les contraintes [Lauriere 1978].

Dans le cadre du CP, nous avons des variables de décision, auxquelles il faut attribuer une valeur à partir d'un domaine fini, et un ensemble de contraintes qui imposent des restrictions sur les valeurs que peuvent prendre ces variables de décision. La tâche consiste à attribuer une valeur à chaque variable de décision de manière à ce que toutes les contraintes soient satisfaites. Les variables de décision ainsi que les valeurs qu'elles peuvent prendre et les contraintes qui leur sont imposées correspondent à un problème, auquel on peut chercher une solution réalisable, auquel cas il s'agit d'un problème de décision que nous appelons un *Problème de satisfaction de contraintes*, ou la meilleure solution, auquel cas il s'agit d'un problème d'optimisation que nous appelons un *Problème d'optimisation de contraintes*.

2.1.1 Problèmes de Satisfaction de Contraintes

Un problème de satisfaction de contraintes (CSP) est un triplet $\langle V, D, C\rangle$, où

- $V = \{v_1, \ldots, v_n\}$ est un ensemble de n variables de décision.
- $D = \{D(v_1), \ldots, D(v_n)\}$ est un ensemble de fonctions, faisant correspondre les variables de décision à leurs domaines.

  La taille du domaine $D(v)$ d'une variable de décision v est notée d_v. Chaque domaine $D(v) = \{S_1, \ldots, S_{d_v}\}$ consiste en des valeurs que v peut éventuellement prendre. La taille maximale du domaine dans un CSP est notée $d = \max_{v \in V} d_v$.
- C est un ensemble de contraintes.

Soit $J \subseteq V$ un sous-ensemble des variables, et $\ell(J)$ désigne le produit cartésien $\prod_{v_i \in J} D(v_i)$ des domaines des variables dans J. Une *affectation* $T \in \ell(J)$ est un mappage de chaque $v \in J$ à une valeur $S \in D(v)$. Une *assignation complète* est une affectation à V. Si une affectation fait correspondre v à $S \in D(v)$, nous la désignons par (v, S). Une *contrainte* $c \in C$ est une paire $\langle V_s, F\rangle$, où $V_s \subseteq V$ est la *scope* de la contrainte et F est un *prédicat* sur $\prod_{v \in V_s} D(v)$ qui accepte les affectations à V_s qui *satisfont* la contrainte. Pour une affectation T à $V'_s \supseteq V_s$, laissons $T(V_s)$ être la restriction de T à V_s. Nous disons que T satisfait à $c = \langle V_s, F\rangle$ si $T(V_s)$ satisfait à c. Un problème est satisfait par T si T satisfait toutes les contraintes. Trouver une solution qui satisfait toutes les contraintes est une tâche NP-complète. On dit qu'un CSP est $vide$ si au moins un de ses domaines est l'ensemble vide.

Les CSP sont typiquement résolues par une recherche arborescente, en utilisant des propagateurs pour réduire les domaines à chaque nœud et éviter les parties de

l'arbre de recherche dont il est prouvé qu'elles ne contiennent aucune solution. Ces techniques seront bientôt présentées en détail dans la section 2.1.4.

Example 2.1. *Le célèbre puzzle de placement de nombres sudoku est en fait un CSP. Dans sa forme la plus célèbre, on a une grille de forme carrée de* 9×9 *divisée en* 3×3 *sous-grilles, dont nous avons neuf. Cette grille est initialement partiellement remplie et le but est de remplir toutes les cellules avec un chiffre compris entre 1 et 9, tout en s'assurant que les chiffres apparaissant dans chaque ligne, chaque colonne et chaque sous-grille sont tous différents.*

Soit $v_{ij} \in V$ *désigne la cellule à la ligne* $i \in \{1, \ldots, 9\}$ *et à la colonne* $j \in \{1, \ldots, 9\}$. *Ce sont les variables de décision de ce CSP particulier. Alors, nous avons* $D(v_{ij}) = \{1, \ldots, 9\}$ *pour tous les* i *et* j. *Pour chaque sous-réseau* $k\ dans\{1, \ldots, 9\}$, *laissons* $V_k \subset V$ *être l'ensemble des cellules dans celui-ci. Remarquez que nous avons* $|V_k| = 9$, $\bigcap_k V_k = \varnothing$ *et* $\bigcup_k V_k = V$.

Pour chaque ligne i, *colonne* j *et sous-grille* k, *nous avons les contraintes suivantes :*

$$
\begin{aligned}
C_i &: v_{ij} \neq v_{ij'} && \forall lesi, j, j' \neq j \in \{1, \ldots, 9\} \\
NC_j &: v_{ij} \neq v_{i'j} && \forall i, i' \neq i, j \in \{1, \ldots, 9\} \\
NC_k &: v_{ij} \neq v_{i'j'} && \forall v_{ij}, v_{i'j'} \in V_k, i' \neq i, j' \neq j, k \in \{1, \ldots, 9\}
\end{aligned}
$$

$C = C_i \cup C_j \cup C_k$ *est l'ensemble des contraintes à satisfaire.*

2.1.2 Problèmes d'Optimisation de Contraintes

Les CSP étant des problèmes de décision, la technologie CP peut également être utilisée efficacement pour résoudre des problèmes d'optimisation en ajoutant une fonction objectif à minimiser ou maximiser. Nous avons alors un problème d'optimisation par contraintes (COP) qui est un quadruplet $\langle V, D, C, f \rangle$ où V est un ensemble de variables, D est l'ensemble des domaines de ces variables, C est un ensemble de contraintes sur ces variables, tout comme dans un CSP, et enfin, $f : \ell(V) \to \mathbb{R}$ est la fonction objective qui évalue la qualité d'une solution donnée en faisant correspondre une affectation $T \in \ell(V)$ à une valeur réelle.

Example 2.2. *Le célèbre problème du sac à dos peut être représenté par un COP. Soit* $I = \{1, \ldots, n\}$ *un ensemble d'éléments, dont chaque élément* i *a un poids particulier* w_i *et une valeur* p_i. *Le but est de choisir un ensemble d'articles tel que leur valeur totale soit maximisée tout en s'assurant que leur poids total ne dépasse pas une capacité prédéfinie* K.

Soit $\{v_1, \ldots, v_n\}$ *les variables de décision avec* $D(v_i) = \{0, 1\}$. *Nous avons* $v_i = 1$ *si nous décidons de prendre l'article* i *et* $v_i = 0$ *sinon. Nous formulons la contrainte de capacité comme suit :*

$$\sum_{i \in I} v_i w_i \leq K$$

v_{11}	v_{12}
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9

v_{11}	v_{21}
2	1
2	3
2	4
2	5
2	6
2	7
2	8
2	9

v_{11}	v_{22}
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9

Table 2.1: Une petite partie de toutes les contraintes de table pour la formulation CSP du sudoku présentée dans l'exemple 2.1. On voit les contraintes entre v_{11} et v_{12} (même ligne) pour $v_{11} = 1$, entre v_{11} et v_{21} (même colonne) pour $v_{11} = 2$, et entre v_{11} et v_{22} (même sous-grille) pour $v_{11} = 1$.

Enfin, la fonction objective devient :

$$\max_{T \in \mathbb{1}^n} T(v_i) p_i$$

où T est une affectation, $T(v_i) \in \{0, 1\}$ est la valeur que prend v_i dans cette affectation, et $\mathbb{1}^n$ est l'ensemble de tous les vecteurs d'incidence à n dimensions.

2.1.3 Principaux Types de Contraintes

Dans cette section, nous allons montrer différentes façons de représenter les contraintes, ainsi qu'un example pour voir comment chacune d'entre elles peut être appliquée à la formulation CSP du sudoku que nous avons vue dans l'exemple 2.1.

2.1.3.1 Contraintes de Table

Les contraintes de table énumèrent toutes les combinaisons de valeurs autorisées pour les variables de leur portée. Les combinaisons de valeurs qui n'apparaissent pas sont simplement interdites.

Example 2.3 (sudoku)**.** *Nous avons vu dans l'exemple 2.1 comment formuler le sudoku comme un CSP. Ici, nous allons voir comment les contraintes peuvent être représentées comme des contraintes de table.*

Pour toutes les paires de variables $\{x_{ij}, x_{i'j'}\}$ qui sont concernées par une contrainte parce qu'elles sont dans la même ligne, colonne ou sous-grille, nous énumérons toutes les combinaisons de valeurs autorisées entre ces deux variables. Nous avons 9 lignes, 9 colonnes et 9 sous-grilles, chacune d'entre elles contenant 9 variables. Chaque fois que nous regardons un ensemble de 9 variables, nous pouvons choisir $\binom{9}{2} = 36$ paires différentes. Pour chaque paire de variables, nous

avons $9x9 - 9 = 72$ *combinaisons de valeurs autorisées. Cela nous donne un total de* $(9 + 9 + 9) \times 36 \times 72 = 69984$ *combinaisons de valeurs à considérer.*

Dans le tableau 2.1, nous avons une petite partie de ces 69984 *combinaisons de valeurs. Dans celui de gauche, nous voyons les combinaisons de valeurs pour la paire* $\{v_{11}, v_{12}\}$*, où nous avons* $v_{11} = 1$*. Nous voyons que l'affectation où les deux valent 1 n'apparaît pas dans le tableau, ce qui signifie simplement qu'elle est interdite. De même, dans le tableau du milieu et celui de droite, nous avons une paire de variables qui se trouvent respectivement dans la même colonne et dans la même sous-grille.*

2.1.3.2 Contraintes Arithmétiques

Les contraintes d'égalité et d'inégalité arithmétiques apparaissent naturellement dans de nombreux problèmes de la vie réelle. Bien que les contraintes de tout degré puissent être traitées dans le cadre de la CP, nous nous concentrons ici sur les contraintes linéaires. Elles sont essentiellement de la forme

$$\sum_{i=1}^{n} k_i \times v_i \leq K$$
$$N \sum_{i=1}^{n} k_i \times v_i = K$$
$$N \sum_{i=1}^{n} k_i \times v_i \geq K$$

où v_i sont des variables de décision, k_i et K sont des nombres réels, et n est un entier positif arbitraire.

Example 2.4 (sudoku)**.** *Redéfinissons les variables de décision pour le sudoku. Pour chaque variable* v_{ij} *que nous avions auparavant et chaque valeur* $S \in \{1, \ldots, 9\}$ *qu'elle peut prendre, nous allons définir une nouvelle variable de décision* x_{ijS}*. Cette nouvelle variable sera égale à un si la cellule de la ligne* i *et de la colonne* j *est affectée de* S*, et à zéro sinon. Notez que nous aurons* $9 \times 9 \times 9 = 729$ *de ces variables. De même, nous remplacerons l'ensemble* V_k *par* X_k*, l'ensemble des* x_{ijS} *tels que la cellule à la ligne* i *et à la colonne* j *appartienne à la sous-grille* k*. Par conséquent, nos contraintes deviennent :*

$$C_i : x_{ijS} + x_{ij'S} \leq 1 \qquad \forall i, j, j' \neq j, S \in \{1, \ldots, 9\}$$
$$C_j : x_{ijS} + x_{i'jS} \leq 1 \qquad \forall i, i' \neq i, j, S \in \{1, \ldots, 9\}$$
$$C_k : x_{ijS} + x_{i'j'S} \leq 1 \qquad \forall x_{ijS}, x_{i'j'S} \in X_k, k, S \in \{1, \ldots, 9\}$$
$$C_{ij} : \sum_{S=1}^{9} x_{ijS} = 1 \qquad \forall i, j \in \{1, \ldots, 9\}$$

Pour chaque paire de variables dans chaque ligne i, et chaque valeur S, nous devons écrire $\binom{9}{2} \times 9 = 324$ contraintes. Nous avons également le même nombre de contraintes pour chaque colonne j et chaque sous-grille k. Par conséquent, nous avons un total de $3 \times 9 \times 324 = 8748$ contraintes. En plus de celles-ci, nous avons également les contraintes dans C_{ij} pour nous assurer que chaque cellule ij se voit attribuer exactement un chiffre, et nous avons 81 de ces contraintes.

2.1.3.3 Contraintes Logiques

Les contraintes logiques nous permettent de capturer les relations entre les variables à l'aide d'instructions *and* et *or*. Chaque variable de décision correspond à un *littéral* qui peut être *vrai* ou *faux*. Ces littéraux sont assemblés dans des clauses, où ils peuvent être positifs (x) ou négatifs ($\overline{x}$) afin de formuler une contrainte.

Example 2.5 (sudoku)**.** *Redéfinissons une fois de plus les variables de décision pour le sudoku. Cette fois, les x_{ijS} sont des variables propositionnelles, c'est-à-dire qu'elles sont vraies si la cellule de la ligne i et de la colonne j est affectée de S et fausses sinon. Nous avons alors les contraintes suivantes :*

$$
\begin{aligned}
C_i &: \overline{x}_{ijS} \vee \overline{x}_{ij'S} && \forall i, j, j' \neq j, S \in \{1, \ldots, 9\} \\
C_j &: \overline{x}_{ijS} \vee \overline{x}_{i'jS} && \forall i, i' \neq i, j, S \in \{1, \ldots, 9\} \\
C_k &: \overline{x}_{ijS} \vee \overline{x}_{i'j'S} && \forall x_{ijS}, x_{i'j'S} \in X_k, k, S \in \{1, \ldots, 9\} \\
C_{ij} &: \bigvee_{S \in \{1,\ldots,9\}} x_{ijS} && \forall i, j \in \{1, \ldots, 9\}
\end{aligned}
$$

Comme dans l'exemple 2.4, pour chaque paire de variables, chaque valeur et chaque ligne i, nous devons écrire un total de $\binom{9}{2} \times 9 = 324$ contraintes. Nous avons 9 lignes, 9 colonnes et 9 sous-grilles, donc avoir 324 contraintes pour chacune d'elles nous donne un total de $(9 + 9 + 9) \times 324 = 8748$ contraintes. En outre, nous avons les contraintes dans C_{ij} avec $|C_{ij}| = 81$ pour nous assurer que chaque cellule ij se voit attribuer exactement un chiffre.

Il convient de noter que tous les solveurs ne peuvent pas gérer les contraintes logiques dont l'arité est supérieure à 2. Dans le cas où le solveur utilisé requiert des contraintes binaires comme celles de C_i, C_j et C_k dans l'exemple 2.5, il faudrait transformer les contraintes dans C_{ij} en utilisant le codage dual [Larrosa & Dechter 2000]. Cela impliquerait toutefois d'introduire un assez grand nombre de variables auxiliaires et de remplacer les contraintes originales par un plus grand nombre de contraintes binaires, ce qui donne lieu à une formulation plus vaste et plus complexe.

2.1.3.4 Contraintes Globales

Une *contrainte globale* est une contrainte définie de manière informelle sur un nombre non fixe de variables de décision. Elle capture la relation entre ces variables en

une seule fois, ce qui nous évite d'exprimer explicitement des milliers de contraintes. Cela facilite non seulement la tâche de modélisation, mais permet également aux solveurs CP d'avoir une vue plus compacte de la structure du problème.

Example 2.6 (sudoku)**.** *Rappelons la formulation initiale du CSP présentée dans l'exemple 2.1, avec des variables de décision v_{ij}, et les contraintes binaires non égales c_i, c_j et c_k. Soit V_i, V_j et V_k l'ensemble des variables de chaque ligne i, colonne j et sous-grille k, respectivement. Nous pouvons remplacer les contraintes binaires pour chaque ligne i, colonne j et sous-grille k par la contrainte globale bien connue AllDifferent comme suit :*

$$\begin{aligned} C_i &: AllDifferent(V_i) && \forall i \in \{1, \ldots, 9\} \\ C_j &: AllDifferent(V_j) && \forall j \in \{1, \ldots, 9\} \\ C_k &: AllDifferent(V_k) && \forall k \in \{1, \ldots, 9\} \end{aligned}$$

Ici, nous avons 27 contraintes au lieu de 78732 comme dans les examples précédents utilisant d'autres types de contraintes.

2.1.4 Procédure de Recherche dans CP

Dans CP, la recherche s'effectue en prenant des décisions et en mettant à jour les domaines en fonction de ces décisions. Une décision consiste essentiellement à attribuer à une variable non renseignée une valeur de son domaine actuel. Une fois que cela est fait, il est très probable que certaines valeurs des autres variables deviennent incompatibles par rapport à certaines contraintes, qui doivent être détectées et supprimées.

Dans les sections suivantes, nous supposons que nous avons un COP, et non un CSP, et que le COP en question est un problème de minimisation.

2.1.4.1 Propagation des Contraintes

Le type de propagation de contraintes que nous considérons dans cette thèse est appelé *élagage* des domaines de variables. La tâche d'élagage consiste à parcourir les domaines pour détecter et supprimer les valeurs *inconsistantes*. Une valeur $S \in D(v)$ est incohérente si elle ne peut apparaître dans aucune solution complète basée sur la solution partielle actuelle. De même, une valeur $S \in D(v)$ est *consistante* s'il existe une solution complète où v est affecté à S.

Déterminer si une valeur est inconsistante ou non est généralement une tâche NP-hardarde [Rossi *et al.* 2006]. C'est pourquoi, les solveurs CP essaient de simplifier cette tâche en élaguant les valeurs par rapport à chaque contrainte séparément. Clairement, si une valeur est incohérente par rapport à une contrainte, alors elle est également incohérente par rapport à la CSP ; il est donc sûr de la supprimer. Cependant, il n'est pas nécessairement toujours vrai qu'une valeur qui est incohérente par rapport à la CSP l'est aussi par rapport à chacune des contraintes. Cela nous amène

à un point où nous devons faire un compromis entre la vitesse (c'est-à-dire le temps d'exécution) et la force (c'est-à-dire le nombre de valeurs incohérentes supprimées) de l'élagage. Certaines contraintes permettent d'obtenir un meilleur compromis avec un élagage important et une efficacité de calcul élevée.

Les contraintes globales réalisent un bon compromis de ce point de vue. Leur capacité à considérer la *grande image*, c'est-à-dire toutes les variables à la fois et non un petit sous-ensemble d'entre elles à la fois, leur permet de supprimer de grandes parties, et la plupart du temps *toutes*, des valeurs incohérentes. Ils sont peut-être la clé la plus importante du succès de la CP, car leurs algorithmes sur mesure effectuent une très forte propagation et une réduction de l'espace de recherche.

La tâche d'élaguer toutes les valeurs incohérentes est comme le saint graal en CP. Par conséquent, la conception d'algorithmes spécialisés pour les contraintes globales afin d'accomplir cette tâche, c'est-à-dire atteindre *cohérence d'arc*, a été un sujet de recherche très populaire au cours des dernières décennies.

Definition 2.7 (Cohérence d'arc généralisée)**.** *Soit $P = \langle V, D, C\rangle$ un CSP.*

Une variable $v \in V$ est dite être generalised arc consistent (GAC) par rapport à une contrainte $c \in C = \langle V_s, F\rangle$, si pour chaque valeur $S \in D(v)$, il existe au moins une affectation $T \in \ell(V_S)$ avec $T(v) = S$ et $F(T) = vrai$.

Une variable $v \in V$ est dite GAC par rapport à l'ensemble du CSP P si elle est GAC par rapport à toutes ses contraintes dans C.

Une CSP P est dite GAC si toutes ses variables dans V sont GAC.

Example 2.8. *Reprenons l'exemple du sudoku et comparons la quantité de valeurs incohérentes que nous parvenons à élaguer lorsque nous utilisons des contraintes binaires non égales et lorsque nous utilisons la contrainte globale AllDifferent.*

Supposons que dans la première ligne, les six dernières variables $\{v_{14}, v_{15}, v_{16}, v_{17}, v_{18}, v_{19}\}$ se voient déjà attribuer un chiffre de l'ensemble $\{4, 5, 6, 7, 8, 9\}$, et que les domaines actuels des trois premières variables sont les suivants :

$$D(v_{11}) = \{1, 2, 3\} \qquad D(v_{12}) = \{2, 3\} \qquad D(v_{13}) = \{2, 3\}$$

Examinons maintenant la formulation CSP où nous avons les contraintes binaires non égales ci-dessous :

$$v_{11} \neq v_{12} \qquad v_{11} \neq v_{13} \qquad v_{12} \neq v_{13}$$

Avec ces contraintes, la seule situation dans laquelle nous sommes autorisés à élaguer les valeurs est une variable ayant un domaine singleton. Or, les domaines actuels donnés ci-dessus ont tous au moins deux valeurs. Dans ce cas, nous sommes malheureusement incapables d'élaguer les valeurs, car pour chaque contrainte, nous pouvons toujours trouver une affectation partielle qui la satisfait. Par exemple, les affectations partielles $\{(v_{11}, 2), (v_{12}, 3)\}$ et $\{(v_{12}, 3), (v_{13}, 2)\}$ satisfont les contraintes $v_{11} \neq v_{12}$ et $v_{12} \neq v_{13}$, respectivement. Cependant, lorsque ces affectations partielles sont réunies, v_{11} et v_{13} ont la même valeur, ce qui n'est pas autorisé. Avec les contraintes binaires, nous sommes incapables de voir que $2 \in D(v_{11})$ est en fait une valeur incohérente (il en est de même pour $3 \in D(v_{11})$).

Si nous avions $AllDifferent(v_{11}, v_{12}, v_{13})$ au lieu des contraintes binaires, nous aurions accès à l'un des propagateurs de $AllDifferent$ nous permettant d'élaguer une grande partie des valeurs incohérentes. Les alternatives incluent, sans s'y limiter, un célèbre algorithme à faible complexité [Régin 1994] qui est basé sur la théorie de l'appariement [Lovász & Plummer 2009] et qui applique le GAC sur $AllDifferent$, et, un algorithme légèrement "plus faible" mais plus rapide [López-Ortiz et al. 2003] qui applique la cohérence du domaine. En appliquant le GAC sur $AllDifferent$, nous pourrions supprimer 2 et 3 du domaine de v_{11}, puisque de toute évidence, ces deux chiffres doivent être partagés entre v_{12} et v_{13}.

2.1.4.2 La Recherche par Retours-arrières

La recherche par retours-arrières est l'une des trois principales méthodes de résolution des CSP et des COP, les deux autres étant la recherche locale et la programmation dynamique ; [Rossi *et al.* 2006]. Il s'agit d'une méthode complète, ce qui signifie qu'elle garantit de trouver une solution si elle existe.

La recherche par retour en arrière est une procédure itérative qui repose sur la prise de *décisions*. En partant du nœud racine, nous prenons une décision pour exactement une variable conformément à notre *stratégie de branchement*, qui peut être l'une des trois suivantes :

1. *Enumération* : Nous créons une branche pour chaque valeur S dans le domaine de la variable v et dans toutes les solutions explorées dans le sous-arbre de cette branche, v se voit attribuer S. Avec cette stratégie, nous nous ramifions sur une variable à chaque niveau, de sorte qu'au moment où nous sommes à la profondeur $n = |V|$, toutes les variables se sont vu attribuer une valeur.

2. *Points de choix binaires* : À chaque nœud, nous créons deux branches pour une variable v et une valeur $S \in D(v)$. Soit nous assignons S à v, soit nous supprimons S de $D(v)$, c'est-à-dire $v = S$ ou $v \neq S$.

3. *Domain Splitting* : Avec cette stratégie, nous créons également deux branches, mais cette fois pour les décisions $v \leq S$ et $v > S$.

Avant de procéder à l'exploration du sous-arbre d'une nouvelle branche, nous déterminons d'abord s'il vaut la peine de l'explorer. Déjà parce que chaque fois que nous prenons une décision, nous rendons potentiellement incohérent un certain ensemble de valeurs. Par conséquent, nous devons nous assurer que l'affectation partielle actuelle peut être étendue à une affectation complète. En outre, nous devons savoir si cette nouvelle branche promet une solution meilleure que celle que nous avons trouvée jusqu'à présent. À cette fin, à chaque nœud, nous effectuons la propagation et le *bounding*. Une recherche par retours-arrières sans ces deux sous-programmes n'est rien d'autre qu'une énumération complète. Nous voulons que la partie de l'arbre de recherche que nous explorons soit aussi petite que

possible en raison de l'explosion combinatoire inhérente à la plupart des problèmes d'optimisation.

La partie évaluation consiste à trouver une évaluation optimiste et une évaluation pessimiste de la fonction objectif. Une *borne inférieure* (resp. une *borne supérieure*) est la plus petite (resp. la plus grande) valeur possible que la fonction objectif peut atteindre. Plus les limites sont serrées, plus l'effort de recherche est léger, car la recherche se termine lorsque les deux limites se rencontrent.

Une approche valide pour générer une borne inférieure (resp. supérieure) est la méthode gloutonne qui additionne le coût le plus petit (resp. le plus grand) d'une valeur dans le domaine de chaque variable. Il s'agit en effet d'un moyen très facile de calculer des bornes, mais ces bornes ont tendance à être très faibles, comme on pouvait s'y attendre. Pour la génération initiale des bornes supérieures, il est très courant d'utiliser un algorithme de recherche locale qui trouve une solution approximative au problème à résoudre. Une fois la recherche lancée, la borne supérieure est mise à jour chaque fois qu'une nouvelle solution est trouvée. En ce qui concerne la borne inférieure, une heuristique de base de données de modèles [Felner *et al.* 2004], entre autres algorithmes heuristiques, peut être utilisée.

Deux autres outils cruciaux pour réduire la taille de l'arbre de recherche sont l'heuristique *ordre des variables* et l'heuristique *ordre des valeurs*. L'ordre dans lequel les variables sont assignées a un impact énorme sur les performances des algorithmes de recherche retours-arrières [Gent *et al.* 1996]. Cependant, trouver l'ordre qui donne lieu à une taille minimale de l'arbre de recherche est au moins aussi difficile que de résoudre le problème lui-même [Liberatore 2000], ce qui est la raison pour laquelle il est très courant d'utiliser des heuristiques d'ordonnancement des variables. Ces heuristiques incluent MinDom [Haralick & Elliott 1980], dom/ddeg [Bessiere & Régin 1996], dom/wdeg [Boussemart *et al.* 2004], l'heuristique basée sur le dernier conflit [Lecoutre *et al.* 2009] et l'heuristique basée sur l'impact [Refalo 2004]. Il ne s'agit toutefois que d'approches génériques, il est donc toujours bon de trouver des moyens d'exploiter les structures spécifiques au problème afin d'obtenir de meilleurs résultats. L'ordre des valeurs est également important, car, si un problème a une solution et qu'une valeur correcte est choisie pour chaque variable, nous pouvons trouver une solution en suivant un chemin direct sans avoir à revenir en arrière. Une approche possible consiste à donner la priorité aux valeurs qui maximisent le nombre d'alternatives pour les affectations futures ; [Haralick & Elliott 1980].

2.2 Programmation Linéaire en Nombres Entiers

Bien que nous n'employions pas directement la programmation linéaire en nombres entiers (ILP) dans cette thèse, notre travail y fait de nombreuses références intéressantes du point de vue de la CP. Représenter un problème d'optimisation dans un cadre CP ou ILP revient à parler deux langues différentes, chacune ayant ses propres forces et faiblesses. Tout comme parler deux langues dans la vie réelle

est quelque chose de formidable, comprendre profondément la CP et la ILP ainsi que saisir les correspondances entre elles peut s'avérer d'une grande utilité dans la pratique, comme nous le verrons également dans cette thèse.

2.2.1 Programmation Linéaire

Un programme linéaire (PL) est le problème consistant à trouver

$$\min_x \{c^T x \mid x \in \mathbb{R}^n \wedge Ax \geq b \wedge x \geq 0\}$$

où c est un vecteur colonne de dimension n, b est un vecteur colonne de dimension m, A est une matrice m x n, et x est un vecteur colonne de variables n. Une ligne A_i pour $i \in \{1, \ldots, m\}$ correspond à une contrainte linéaire individuelle i et une colonne A_j pour $j \in \{1, \ldots, n\}$ à une variable x_j. Une solution réalisable de ce problème est celle qui satisfait $x \in \mathbb{R}^n \wedge Ax \geq b \wedge x \geq 0$, où $\geq$ est une comparaison par éléments. D'autre part, une solution optimale est une solution faisable qui minimise la fonction objectif $c^T x$.

La solution optimale d'un LP peut être trouvée à l'aide du célèbre algorithme Simplex [Murty 1983]. Bien que la complexité théorique de l'algorithme Simplex soit exponentielle, en pratique, sa complexité moyenne sous diverses distributions de probabilité est polynomiale [Schrijver 1998].

Le dual d'une LP P sous la forme ci-dessus est une autre LP D :

$$\max_y \{b^T y \mid y \in \mathbb{R}^m \wedge A^T y \leq c \wedge y \geq 0\}$$

où A, b, c sont comme précédemment et y est le vecteur de m variables duales. Chaque ligne (contrainte) dans le primal correspond à une colonne (variable) dans le dual. De manière équivalente, chaque contrainte dans le dual correspond à une variable dans le primal. Étant donné que, sans perte de généralité, le primal est une minimisation et le dual une maximisation, la valeur objective de toute solution réalisable duale est une borne inférieure de l'optimum du primal (P). Lorsque P est satisfaisable, son dual est également satisfaisable et les valeurs de leurs optima se rencontrent. Nous notons également que le dual du dual est la primale elle-même.

Au point où les solutions de P et D se rencontrent, non seulement leurs optima ont la même valeur, mais aussi ce que nous appelons les *conditions de relâchement complémentaires* tiennent. Pour une solution primale faisable donnée $\hat{x}$, laissez $slack^P_{\hat{x}}(i) = A_i\hat{x} - b_i$ être le slack de la contrainte i. De même, pour une solution duale réalisable donnée $\hat{y}$, laissez $slack^D_{\hat{y}}(j) = c - A^T\hat{y}$ être le slack de la contrainte j. Le slackness complémentaire stipule que dans les solutions optimales x^*, y^*, soit la valeur d'une variable x_j dans P, soit $slack^D_{y^*}(j)$ dans D doit être 0. De même, soit la valeur d'une variable duale y_i dans D, soit $slack^P_{x^*}(i)$ dans P doit être 0.

Étant donné une solution duale réalisable $\hat{y}$, $slack^D_{\hat{y}}(j)$ est le *coût réduit* de la variable primale $j, rc_{\hat{y}}(j)$. Le coût réduit $rc_{\hat{y}}(j)$ est le montant minimum par lequel

l'objectif dual augmenterait sur $b^T\hat{y}$ si x_j était forcé d'être non nul dans le primal. D'une certaine manière, le coût réduit d'une variable peut être interprété comme une sous-approximation du coût marginal : le coût que nous devrions ajouter à la borne inférieure actuelle si nous voulions absolument que cette variable soit non nulle.

2.2.2 Stratégies de Résolution

Comme la plupart des problèmes de la vie réelle impliquent des décisions discrètes, disposer uniquement de variables de décision continues est loin d'être pratique. C'est pourquoi, nous devons modéliser de tels problèmes sous forme de programme linéaire en nombres entiers (ILP), qui est un LP où nous remplaçons la contrainte $x \in \mathbb{R}^n$ par $x \in \mathbb{Z}^n$. Trouver la solution optimale d'un ILP est cependant une tâche NP-hardarde [Schrijver 1998]. Ce défi particulier, ainsi que le fait que les ILP constituent un outil très pratique pour modéliser toutes sortes de problèmes, ont suscité une attention considérable au cours des dernières décennies et aujourd'hui, nous disposons de plusieurs techniques reconnues pour les résoudre.

Dans les sections suivantes, nous supposons que nous avons un ILP qui est un problème de minimisation.

2.2.2.1 Branch-and-Bound

Le branch-and-bound [Little *et al.* 1963] est une énumération systématique des solutions candidates. Les solutions candidates sont explorées dans un arbre de recherche à racines où les nœuds représentent une solution (partielle) et les branches correspondent aux décisions qui attribuent une valeur à une variable ou qui réduisent l'intervalle de domaine d'une variable. Avant d'explorer le sous-arbre d'une nouvelle branche, une estimation de la borne inférieure basée sur cette décision est calculée. Si cette estimation de la borne inférieure est supérieure ou égale à la borne supérieure actuelle, le sous-arbre est immédiatement écarté car il ne promet pas de meilleure solution. D'autre part, si l'estimation de la borne inférieure est inférieure à la borne inférieure actuelle, le sous-arbre est exploré plus avant. Chaque fois qu'une nouvelle solution réalisable est trouvée, la borne supérieure est mise à jour comme la valeur de cette solution. La recherche se termine lorsque la borne supérieure et la borne inférieure se rejoignent.

Cela nous indique que l'efficacité de la procédure de branch-and-bound dépend fortement de la bonne qualité de l'estimation des bornes inférieures et supérieures, tout comme dans la recherche par retours-arrières dans CP. S'il n'y a pas de mécanisme de délimitation, l'algorithme deviendrait simplement une recherche exhaustive parmi un nombre exponentiel de solutions alternatives. De grandes parties de l'arbre de recherche peuvent être évitées grâce à des bornes inférieures et supérieures serrées.

La technique la plus couramment utilisée pour générer des bornes inférieures pour une ILP consiste à résoudre sa *relaxation linéaire*. La relaxation linéaire

d'une ILP est une LP obtenue en relaxant toutes les contraintes d'intégralité et en permettant ainsi aux variables intégrales de prendre des valeurs réelles.

À chaque nœud de l'arbre de recherche, la relaxation linéaire de l'ILP actuelle est résolue. Si la valeur de la fonction objectif à l'optimum de la LP est supérieure ou égale à la borne supérieure actuelle, le nœud est fouillé. Sinon, une variable ayant une valeur fractionnaire dans la solution optimale est choisie et une nouvelle branche est créée à partir du nœud actuel. Cette branche correspond à une nouvelle contrainte à ajouter dans l'ILP : soit la variable est supérieure ou égale au plus petit nombre entier qui est supérieur à sa valeur fractionnaire, soit la variable est inférieure ou égale au plus grand nombre entier qui est inférieur à sa valeur fractionnaire. Par example, si une variable x s'avère être $3,4$ dans la LP, alors les branches candidates sont $x \leq 3$ et $x \geq 4$.

La recherche se termine lorsque la relaxation linéaire de l'ILP courante a une solution optimale où toutes les variables sont affectées d'une valeur intégrale.

2.2.2.2 Branch-and-Cut

Le branch-and-cut [Padberg & Rinaldi 1987] est une méthode bien connue de résolution des ILP. Elle est généralement utilisée lorsque le nombre de contraintes est trop important (exponentiel) pour être traité efficacement par le solveur. Par conséquent, l'astuce consiste à ignorer initialement des parties ou la totalité des contraintes et à introduire un sous-ensemble de celles-ci dans l'ILP au cours de la procédure de recherche lorsqu'elles s'avèrent capables de *découper* des parties de l'*espace de recherche*.

Nous présentons par la suite les notions de géométrie relatives à la résolution de LP et ILP.

Dans une LP avec n variables, chaque contrainte sous la forme d'une inégalité correspond à un *milieu* dans un espace euclidien à n dimensions, qui est divisé par un *hyperplan* constitué des points où le côté droit et le côté gauche de cette inégalité sont égaux. L'intersection d'un nombre fini de demi-espaces définit un *polytope convexe*, et ce polytope est l'ensemble de toutes les solutions réalisables, c'est-à-dire notre espace de recherche.

Un *face* d'un polytope convexe est toute intersection du polytope avec un demi-espace tel qu'aucun des points intérieurs du polytope ne se trouve sur la limite du demi-espace. Étant donné un polytope convexe de n-dimensions, ses *facettes* sont ses faces de $(n-1)$-dimensions, et ses *vertices* sont ses faces de 0-dimensions. Les sommets sont également appelés *points extrêmes*. La solution optimale d'un LP se trouve dans l'un des sommets de son polytope convexe [Wolsey 2020].

Lorsque nous examinons le polytope convexe de la relaxation linéaire d'une ILP, il est probable que les sommets ne correspondent pas à des solutions intégrales, et il est encore moins probable que la solution optimale en soit une. Si nous avions le *coque convexe* de (c'est-à-dire le plus petit ensemble convexe contenant) les points à l'intérieur du polytope convexe qui correspondent à une solution intégrale, alors nous pourrions simplement exécuter l'algorithme du Simplex sur celui-ci

$S_1 \in D(v_1)$	$c_{v_1}(S_1)$
a	0
b	2

Table 2.2: c_{v_1}

$S_2 \in D(v_2)$	$c_{v_2}(S_2)$
a	1
b	0

Table 2.3: c_{v_2}

$S_1 \in D(v_1)$	$S_2 \in D(v_2)$	$c_{v_1v_2}(S_1, S_2)$
a	a	0
a	b	1
b	a	0
b	b	3

Table 2.4: $c_{v_1v_2}$

Table 2.5: Tableau des coûts pour l'exemple 2.10.

et avoir fini de résoudre l'ILP. Cependant, dans la plupart des cas, il existe un nombre exponentiel d'inégalités, c'est-à-dire de *inégalités définissant une facette*, nécessaires pour décrire cette coque convexe, et leur caractérisation n'est pas une tâche simple.

Le but du branch-and-cut est de couper progressivement des parties de l'espace de recherche de la relaxation linéaire de l'ILP afin de le réduire de plus en plus vers la coque convexe des solutions intégrales. Le découpage est effectué en introduisant un sous-ensemble de l'ensemble original de contraintes de l'ILP qui ont été initialement omises. Une question clé de cette méthode est d'identifier les contraintes qui sont violées, c'est-à-dire les *inégalités violées*, car seules les contraintes violées sont capables de découper l'espace de recherche. La tâche consistant à extraire les inégalités violées d'un ensemble de *inégalités valides* est appelée *problème de séparation* et généralement, le problème de séparation d'un problème NP-hard est également NP-hard [Wolsey 2020].

2.3 Problèmes de Satisfaction de Contraintes Pondérées

Ce que nous rencontrons assez souvent dans les problèmes de la vie réelle, c'est qu'il existe des solutions meilleures et pires, à savoir nos préférences, plutôt que de simples solutions faisables et infaisables. Le cadre du *problème de satisfaction de contraintes pondérées* nous permet d'exprimer efficacement nos préférences sous la forme de *fonctions de coût*.

Definition 2.9. Une *fonction de coût* $c_{V_S} : \ell(V_S) \to \mathbb{N}_{\geq 0}$ est définie sur l'étendue $V_S \subseteq V$ et elle associe un coût *entier* non négatif à chaque affectation $T \in \ell(V_S)$.

Les fonctions de coût peuvent être représentées sous la forme d'un tableau ainsi que d'un réseau de fonctions de coût (CFN), comme dans l'exemple 2.10. La *arity* d'une fonction de coût c_{V_S} est la cardinalité de son étendue, $|V_S|$, et dans ce travail, nous appelons les fonctions de coût d'arité *zero*, *one*, *two* respectivement *nullary* (ou *constant*), *unary* et *binary*. La fonction de coût constante $c_\varnothing$ correspond au coût que nous payons quelle que soit l'affectation complète $T \in \ell(V)$ que nous avons.

Example 2.10. *Soit v_1, v_2 deux variables avec des domaines $D(v_1) = \{a, b\}$ et $D(v_2) = \{a, b\}$. Soit $c_{v_1}(a) = 0$, $c_{v_1}(b) = 2$, $c_{v_2}(a) = 1$, $c_{v_2}(b) = 0$ les valeurs*

de coût unaires, et $c_{v_1v_2}(a,a) = 0$, $c_{v_1v_2}(a,b) = 1$, $c_{v_1v_2}(b,a) = 0$, $c_{v_1v_2}(b,b) = 3$ *sont les valeurs de coût binaires. Les coûts sous forme de tableaux peuvent être vus dans le tableau 2.5.*

Le CFN de cet example est visible dans la Figure 2.1. Chaque variable $v \in \{v_1, v_2\}$ *correspond à une cellule. Chaque valeur* $S \in D(v)$ *correspond à un point dans la cellule. Un coût unaire* $c_v(S)$ *est écrit à côté du point uniquement s'il est non nul. S'il existe un coût binaire non nul* $c_{vv'}$ *entre* (v, S) *et* (v', S')*, alors une arête est tracée entre les points correspondant à ces affectations.*

Definition 2.11. Un *Problème de satisfaction de contraintes pondérées (WCSP)* [Cooper *et al.* 2010] est un quadruple $\langle V, D, C, k\rangle$ où V est un ensemble de n variables $V = \{v_1, \ldots, v_n\}$, chaque variable $v_i \in V$ a un domaine de valeurs possibles $D(v_i) \in D$, comme dans CSP. C est un ensemble de fonctions de coût, et k est un entier positif ou l'infini servant de borne supérieure.

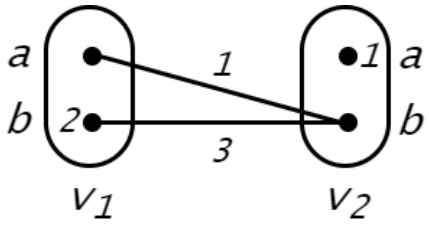


Figure 2.1: Le CFN de l'exemple 2.10.

Les WCSP généralisent les CSP car ils peuvent représenter le même ensemble de solutions réalisables avec un coût infini $c_{V_S}(T)) = k$ pour les affectations interdites $T \in \ell(V_S)$, mais définissent en plus un coût pour les affectations réalisables. Sans perte de généralité, nous supposons que tous les WCSP contiennent une fonction de coût nulle $c_\varnothing$, qui représente une constante dans la fonction objectif, une fonction de coût unaire pour chaque variable, et, au plus une fonction de coût pour tous les autres périmètres. Puisque tous les coûts sont non négatifs, $c_\varnothing$ est une borne inférieure valide sur le coût des solutions réalisables du WCSP.

Si la plus grande arité de toute fonction de coût dans un WCSP est 2, alors nous disons qu'il s'agit d'un *binary WCSP*. À partir de maintenant, pour des raisons de simplicité, nous ne parlerons que des WCSP binaires. Cependant, toutes les définitions et propriétés que nous présentons peuvent facilement être généralisées à des ariés supérieurs.

Étant donné un WCSP P, la tâche consiste à trouver une affectation complète, c'est-à-dire une solution $T \in \ell(V)$ qui minimise la somme de toutes les fonctions de coût, désignée par $c_P(T) = c_\varnothing + \sum_{v\in V} c_v(T(v)) + \sum_{c_{vv'}\in C} c_{vv'}(T(v), T(v'))$, et telle que $c_P(T) < k$. La valeur minimale de cette somme est désignée par $opt(P)$. Ce problème est NP-hard [Cooper & Schiex 2004].

2.3.1 Equivalence dans les WCSPs

Un WCSP est généralement résolu avec une procédure de branch-and-bound, où le maintien d'une bonne borne inférieure est crucial. $c_\varnothing$ étant la borne inférieure d'un WCSP, on peut transférer les coûts des fonctions de coût unaires et binaires vers $c_\varnothing$, augmentant ainsi la borne inférieure, en effectuant des *transformations préservant l'équivalence* que nous verrons en détail dans la section 2.3.2.

Étant donné deux WCSP P_1, P_2 avec le même ensemble de variables et de portées, nous disons qu'ils ont la même structure. Si $c_{P_1}(T(V)) = c_{P_2}(T(V))$ pour tout $T \in \ell(V)$ alors P_1 et P_2 sont *équivalents* et ils sont *reparamétrages* l'un de l'autre. Nous disons qu'une reparamétrisation est *meilleure* si elle a un $c_\varnothing$ supérieur. Trouver une reparamétrisation optimale avec le plus grand $c_\varnothing$ possible en utilisant des coûts entiers est une tâche NP-hardarde [Cooper & Schiex 2004] et il est donc courant d'approximer cette tâche.

Il a été démontré [Cooper *et al.* 2010] que la reparamétrisation optimale, qui maximise $c_\varnothing$ en utilisant des coûts rationnels, est donnée par le dual du LP suivant, appelé le *polytope local* du WCSP :

$$\min c_\varnothing + \sum_{v\in V, S\in D(v)} c_v(S)x_{vS} + \sum_{c_{vv'}\in C, S\in D(v), S'\in D(v')} c_{vv'}(S,S')y_{vSv'S'} \tag{2.1a}$$

s.t.

$$\sum_{S\in D(v)} x_{vS} = 1 \qquad \forall v \in V \tag{2.1b}$$

$$x_{vS} = \sum_{S'\in D(v')} y_{vSv'S'} \qquad \forall v \in V, S \in D(v), c_{vv'} \in C \tag{2.1c}$$

$$0 \le x_{vS} \le 1 \qquad \forall v \in V, S \in D(v) \tag{2.1d}$$

$$0 \le y_{vSv'S'} \le 1 \qquad \forall c_{vv'} \in C, S \in D(v), S' \in D(v') \tag{2.1e}$$

Cette LP est la relaxation de la formulation ILP d'un WCSP, où x_{vS} est une variable binaire qui vaut 1 si la variable WCSP v est affectée S, et 0 sinon. De même, $y_{vSv'S'}$ est une variable binaire qui vaut 1 si x_{vS} et $x_{v'S'}$ valent tous deux 1, ce qui indique que v est affecté à S et v' est affecté à S' en même temps. Les contraintes 2.1b garantissent que chaque variable v se voit attribuer exactement une valeur, tandis que les contraintes 2.1c garantissent qu'une affectation où v se voit attribuer S est choisie dans le cas où $x_{vS} = 1$. La fonction objective 2.1a est la décomposition du coût total d'une affectation.

À partir de la solution optimale y^* du dual du LP ci-dessus, la reparamétrisation est extraite des coûts réduits $rc_{y^*}(x_{vS})$ et $rc_{y^*}(y_{vSv'S'})$ de chaque fonction de coût unaire et binaire, respectivement, en fixant $c_v(S)$ à $rc_{y^*}(x_{vS})$, $c_{vv'}(S,S')$ à $rc_{y^*}(y_{vSv'S'})$, et $c_\varnothing$ à la valeur de la fonction objectif de y^*. En raison de la correspondance entre les reparamétrages et les solutions du dual de cette LP, nous utilisons les deux de manière interchangeable.

Puisque la recherche du $c_\varnothing$ maximal est NP-hard, elle est généralement approchée en appliquant différents types de *cohérence d'arc souple* que nous verrons dans la section 2.3.3. Ces cohérences d'arcs souples sont renforcées en effectuant des transformations préservant l'équivalence.

Algorithm 19: Principales procédures utilisées pour effectuer des transformations préservant l'équivalence dans les WCSP binaires.

Require $\alpha \leq min\{c_{vv'}(T(v), T(v')) \colon T \in \ell(\{v, v'\})$ **and** $T(v) = S\}$
Procedure `Project`$(\{v, v'\}, v, S, \alpha)$**:**
  $c_v(S) \leftarrow c_v(S) \oplus \alpha$
  foreach $T \in \ell(\{v, v'\})$ *such that* $T(v) = S$ **do**
    $c_{vv'}(T(v), T(v')) \leftarrow c_{vv'}(T(v), T(v')) \ominus \alpha$
Require $\alpha \leq c_v(S)$
Procedure `Extend`$(v, S, \{v, v'\}, \alpha)$**:**
  foreach $T \in \ell(\{v, v'\})$ *such that* $T(v) = S$ **do**
    $\beta \leftarrow c_\varnothing \oplus c_v(T(v)) \oplus c_{v'}(T(v'))$
    $c_{vv'}(T(v), T(v')) \leftarrow ((c_{vv'}(T(v), T(v')) \oplus \beta) \ominus \beta) \oplus \alpha$
  $c_v(S) \leftarrow c_v(S) \ominus \alpha$
Require $\alpha \leq min\{c_v(S) \colon S \in D(v)\}$
Procedure `UnaryProject`(v, α)**:**
  foreach $S \in D(v)$ **do**
    $c_v(S) \leftarrow ((c_v(S) \oplus c_\varnothing) \ominus c_\varnothing) \ominus \alpha$
  $c_\varnothing \leftarrow c_\varnothing \oplus \alpha$

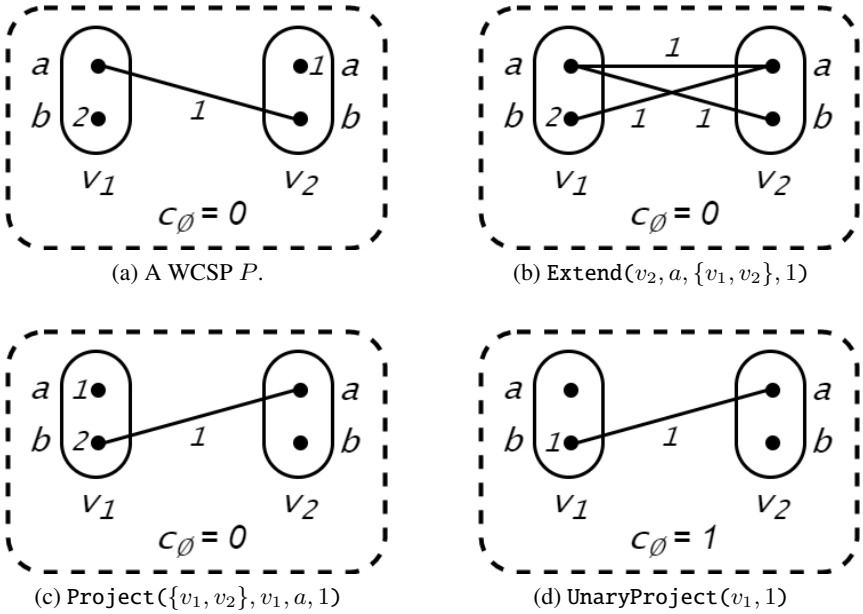
2.3.2 Transformations Préservant l'Équivalence

Une *transformation préservant l'équivalence* (EPT) est une opération qui transforme un WCSP en un WCSP équivalent avec la même distribution de coûts pour toutes les affectations $T \in \ell(V)$.

L'algorithme 19 donne trois transformations préservant l'équivalence de base. `Project` projette les poids d'une fonction de coût binaire $c_{vv'}$, vers une fonction de coût unaire c_v. `Extend`, au contraire, projette les poids d'une fonction de coût unaire vers une fonction de coût binaire. `UnaryProject` projette les poids d'une fonction de coût unaire vers la fonction de coût constante $c_\varnothing$. Tous les déplacements de coûts sont effectués avec les opérateurs $\oplus$ et $\ominus$. $\oplus$ est l'opérateur d'agrégation qui est soit l'opérateur d'addition habituel $+$, soit l'opérateur d'addition avec plafond $+_k$ défini comme suit

$$a +_k b = min\{a + b, k\} \qquad \forall a, b \in \{0, 1, \ldots, k\}$$

L'inverse partiel de l'opérateur d'agrégation $\oplus$ est désigné par $\ominus$, et il est défini comme suit



(a) A WCSP P.
(b) `Extend`$(v_2, a, \{v_1, v_2\}, 1)$
(c) `Project`$(\{v_1, v_2\}, v_1, a, 1)$
(d) `UnaryProject`$(v_1, 1)$

Figure 2.2: EPTs réalisés sur un WCSP P.

$$a \ominus b = a - b \qquad \forall a, b \in \{0, 1, \ldots, k-1\}, b \leq a$$
$$k \ominus b = k \qquad \forall b \in \{0, 1, \ldots, k\}$$

$\ominus$ est utilisé dans `Extend` et `UnaryProject` pour s'assurer que les valeurs interdites restent interdites après tout EPT.

Example 2.12. *Considérons le WCSP P présenté sur la Figure 2.2a. Nous avons deux variables v_1 et v_2, qui ont toutes deux le même domaine $D(v_1) = D(v_2) = \{a, b\}$. Initialement, la fonction de coût constant $c_\varnothing$, c'est-à-dire que la borne inférieure est 0. Nous effectuons des EPT sur P dans l'ordre suivant :*

- *De la Figure 2.2a à la Figure 2.2b :* `Extend`$(v_2, a, \{v_1, v_2\}, 1)$

  Nous transférons un coût de 1 de (v_2, a) à chaque affectation sur $\{v_1, v_2\}$ où v_2 est affecté a, à savoir $\{(v_1, a), (v_2, a)\}$ et $\{(v_1, b), (v_2, a)\}$.

- *De la Figure 2.2b à la Figure 2.2c :* `Project`$(\{v_1, v_2\}, v_1, a, 1)$

  Nous transférons un coût de 1 à (v_1, a) de chaque affectation sur $\{v_1, v_2\}$ où v_1 est affecté a, à savoir $\{(v_1, a), (v_2, a)\}$ et $\{(v_1, a), (v_2, b)\}$.

- *De la Figure 2.2c à la Figure 2.2d :* `UnaryProject`$(v_1, 1)$

  Nous transférons un coût de 1 à $c_\varnothing$ de chaque valeur de v_1.

En conséquence, nous obtenons un autre WCSP apparaissant dans la Figure 2.2d, qui est équivalent à P (tout comme les WCSP des figures 2.2b et 2.2c), mais qui a une meilleure borne inférieure avec $c_\varnothing = 1$.

2.3.3 Cohérence d'Arc Souple

$c_\varnothing$, ou de manière équivalente l'optimum du dual du polytope local (2.1) d'un WCSP peut être approximé en appliquant différents types de *cohérence d'arc souple*. Nous utilisons le mot "souple" car dans les WCSP nous n'avons pas de contraintes "dures" comme dans les CSP. Certains de ces algorithmes produisent des bornes inférieures fortes, avec malheureusement un coût de calcul élevé. Inversement, certains autres ont une faible complexité, mais ils fournissent des bornes inférieures plus faibles. Il s'agit là d'un autre point où nous devons faire un compromis entre la force et la rapidité de l'inférence.

Definition 2.13. Un WCSP est *conforme aux nœuds* [Cooper *et al.* 2010] si pour tous les $v \in V$,

1. $\forall S \in D(v), c_v(S) \oplus c_\varnothing < k$
2. $\exists S \in D(v)$ tel que $c_v(S) = 0$.

Example 2.14. *Le WCSP de la Figure 2.2c n'est pas cohérent avec les nœuds puisqu'il n'existe aucune valeur dans $D(v_1)$ avec un coût unaire nul. Cependant, celui de la Figure 2.2d est cohérent avec les nœuds, puisqu'il existe au moins une valeur dans $D(v_1)$, a, avec un coût unaire nul.*

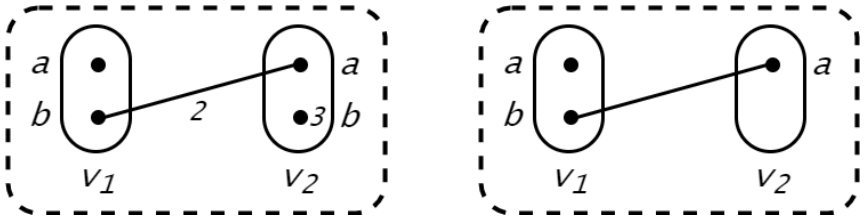
La cohérence des nœuds (NC) est très intuitive dans le sens où, si toutes les valeurs d'une variable ont un coût non nul, évidemment, quelle que soit la valeur que nous attribuons à cette variable, nous devrons payer un certain coût non nul. Ce coût est au moins aussi grand que le plus petit coût de toutes les valeurs, donc nous pouvons aussi bien le transférer à la borne inférieure. Appliquer la NC est une tâche simple qui peut être réalisée en temps $O(nd)$ et c'est en fait ce que fait `UnaryProject`.

Nous nous concentrons principalement sur la cohérence des arcs virtuels (VAC) qui calcule des bornes inférieures de haute qualité [Cooper *et al.* 2010] mais à un coût significatif, elle est donc principalement utilisée dans le prétraitement, plutôt que dans chaque nœud de l'arbre de recherche. Dans le chapitre 6, nous découvrirons les moyens d'employer efficacement VAC afin de bénéficier de ses bornes inférieures de bonne qualité tout au long de la recherche.

Avant de passer à VAC, redéfinissons maintenant *cohérence d'arc* du point de vue du WCSP, car cela sera pertinent. Cette fois-ci, nous laissons tomber le mot "généralisé" puisque nous nous concentrons ici uniquement sur les WCSP binaires de toute façon.

Definition 2.15. Un CSP P est *arc consistent* (AC) s'il remplit les conditions suivantes :

1. Pour chaque contrainte binaire $c_{vv'} \in C$ et pour chaque valeur $S \in D(v)$, il existe une valeur $S' \in D(v')$ telle que l'affectation partielle $\{(v, S), (v', S')\}$ satisfait $c_{vv'}$. Nous disons que la valeur $S' \in D(v')$ est un *support* pour la valeur $S \in D(v)$.



(a) P (b) $Bool(P)$

Figure 2.3: WCSP P et $Bool(P)$ mentionnés dans l'exemple 2.17.

2. Pour chaque contrainte binaire $c_{vv'} \in C$ et pour chaque valeur $S' \in D(v')$, il existe une valeur $S \in D(v)$, c'est-à-dire un *support*, telle que l'affectation partielle $\{(v, S), (v', S')\}$ satisfait $c_{vv'}$.

La fermeture arc cohérente $AC(P)$ est l'unique CSP qui résulte de la suppression des valeurs des domaines qui violent la propriété arc cohérente propriété. $AC(P)$ est équivalente à P, c'est-à-dire qu'elle possède exactement le même ensemble de solutions. En particulier, si $AC(P)$ est vide (a des domaines vides), P est insatisfaisable.

De même, un WCSP P est AC s'il remplit les conditions suivantes :

1. Pour chaque fonction de coût binaire $c_{vv'} \in C$ et pour chaque valeur $S \in D(v)$, il existe une valeur $S' \in D(v')$ telle que $c_{vv'}(S, S') = 0$.

2. Pour chaque fonction de coût binaire $c_{vv'} \in C$ et pour chaque valeur $S' \in D(v')$, il existe une valeur $S \in D(v)$ telle que $c_{vv'}(S, S') = 0$.

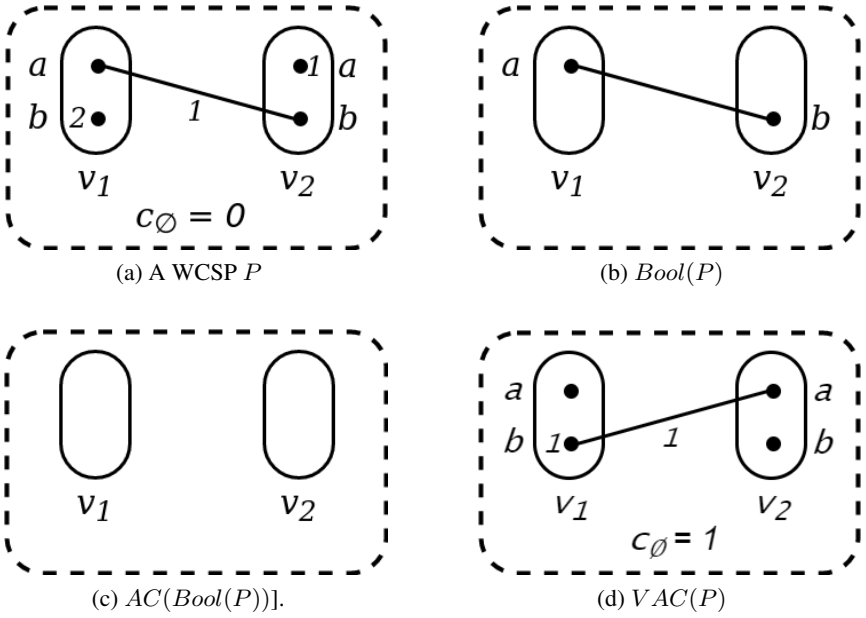
Definition 2.16. Soit $P = \langle V, D, C, k\rangle$ un WCSP. Alors $Bool(P) = \langle V, \overline{D}, \overline{C}\rangle$ est le CSP où, pour tout $v \in V$, S apparaît dans $\overline{D}(v)$ si et seulement si $c_v(S) = 0$, et, pour tout $v, v' \in V$, $S \in D(v)$, $S' \in D(v')$, $\{(v, S), (v', S')\} \in \overline{C}$ si et seulement si $c_{vv'}(S, S') = 0$.

En d'autres termes, étant donné un CSP P, $Bool(P)$ est le CSP classique où les valeurs qui ont un coût unaire supérieur à 0 sont supprimées, et où les affectations binaires ayant un coût supérieur à 0 deviennent interdites.

Example 2.17. *Considérons le WCSP P présenté sur la Figure 2.3a. Nous avons deux variables v_1 et v_2, qui ont toutes deux le même domaine $D(v_1) = D(v_2) = \{a, b\}$. Les coûts non nuls sont $c_{v_1v_2}(b, a) = 2$ et $c_{v_2}(b) = 3$.*

Dans $Bool(P)$ présenté dans la Figure 2.3b, on voit que $b \notin \overline{D}(v_2)$, c'est-à-dire que $\overline{D}(v_2) = \{a\}$. De même, l'affectation $\{(v_1, b), (v_2, a)\}$ qui a un coût de 2 dans P est simplement interdite dans $Bool(P)$. Notez que nous traçons un bord entre deux valeurs lorsque l'affectation correspondante est interdite.

Par construction, $Bool(P)$ admet exactement les solutions de P avec un coût $c_\varnothing$, puisque toutes les assignations qui ont un coût non nul dans toute fonction de



(a) A WCSP P (b) $Bool(P)$

(c) $AC(Bool(P))$]. (d) $VAC(P)$

Figure 2.4: Enforcing VAC on a WCSP P.

coût de P sont mappées à des tuples interdits dans $Bool(P)$. Ainsi, si $Bool(P)$ est insatisfaisable, cela indique $c_\varnothing < opt(P)$. En d'autres termes, la requête "*La CSP $Bool(P)$ a-t-elle une solution?*" est équivalente à la requête "*La CSP P a-t-elle une solution avec un coût total de $c_\varnothing$?*".

Definition 2.18. Un CSP P est *virtual arc consistent* (VAC) si la fermeture de cohérence d'arc du CSP $Bool(P)$ est non vide [Cooper *et al.* 2010]. La fermeture VAC $VAC(P)$ d'un CSP est un point fixe qui n'est pas nécessairement unique.

Si $AC(Bool(P))$ est vide, ou non vide mais toujours insatisfaisable, alors $Bool(P)$ est insatisfaisable et donc $c_\varnothing < opt(P)$.

Example 2.19. *Le WCSP P de la Figure 2.4a n'est pas VAC, car son $Bool(P)$, présenté dans la Figure 2.4b a une fermeture AC vide, c'est-à-dire qu'il n'est pas AC. C'est parce que $a\ dans D(v_1)$ n'a pas de support dans $D(v_2)$, et vice versa. Lorsque nous supprimons ces valeurs incohérentes, nous nous retrouvons avec des domaines vides, c'est-à-dire un effacement de domaine. Nous savons que lorsque $AC(Bool(P))$ est vide, il existe une séquence d'EPT qui augmente de $c_\varnothing$ lorsqu'elle est appliquée à P [Cooper et al. 2010]. Nous voyons le WCSP résultant dans la Figure 2.4d, avec un $c_\varnothing$ amélioré qui est 1.*

2.3.4 Méthodes de Solution Éxactes pour les WCSPs

Les méthodes de solution exacte pour les WCSP sont principalement basées sur le branch-and-bound, bien qu'il existe quelques exceptions à cette norme : les solveurs

MaxSAT guidés par le noyau [Morgado *et al.* 2014], la décomposition de Benders basée sur la logique [Davies & Bacchus 2011], la génération de coupes [Sontag *et al.* 2008], pour n'en citer que quelques-unes. Pour en revenir au branch-and-bound, exprimer l'optimisation WCSP comme une ILP (c'est-à-dire le polytope local (2.1) de WCSP) et utiliser un solveur pour cela est une approche qui vient immédiatement à l'esprit. Cependant, les solveurs ILP doivent résoudre exactement la relaxation linéaire du problème pour obtenir une borne à chaque nœud de l'arbre de branchement, une opération trop coûteuse pour l'échelle des problèmes rencontrés dans de nombreuses applications réelles. Une alternative plus rapide consiste à trouver une transformation de cohérence d'arc souple d'un WCSP maximisant $c_\varnothing$, en utilisant des coûts rationnels au lieu de coûts entiers, ce qui est fait par l'algorithme *OSAC (Optimal Soft Arc Consistency)* [Cooper *et al.* 2010].

À ce stade, les progrès de l'inférence MAP (Maximum a Posteriori) pour les champs aléatoires de Markov (MRF) sont également pertinents, en raison du fait que chaque WCSP correspond à un CFN et que les CFN ont une relation étroite avec les GM probabilistes. En exploitant la propriété logarithmique $\log(x \times y) = \log(x) + \log(y)$, il est possible de transformer tout GM probabiliste en CFN en utilisant une transformation $-\log$. Ce CFN définit une fonction de coût conjointe qui est en fait une approximation contrôlée du $-\log()$ de la fonction de probabilité conjointe du GM probabiliste. Étant donné que la fonction logarithmique est monotone, la requête de minimisation des coûts dans WCSP est équivalente à la requête d'affectation MAP dans MRF ; citecooper2020graphical. Par conséquent, les solveurs doubles développés pour l'affectation MAP dans les MRF peuvent également être utilisés. Par example, *TRW-S (Sequential Tree-Reweighted Message Passing)* [Schoenemann *et al.* 2014] est un algorithme conçu pour résoudre la relaxation LP du problème de minimisation de l'énergie dans les MRF, dont les points fixes satisfont aux conditions VAC, et qui s'est révélé très efficace dans de nombreuses applications pratiques [Werner 2007].

Les solveurs dédiés les plus performants utilisent des algorithmes qui résolvent en fait la relaxation linéaire du WCSP *approximativement* et donc *potentiellement sous-optimalement*. Des algorithmes comme VAC, OSAC et TRW-S produisent des solutions réalisables au dual de la relaxation linéaire de la WCSP, qui peuvent être utilisées comme bornes inférieures. Pour la perte de précision qu'ils abandonnent, ces algorithmes gagnent considérablement en efficacité de calcul. Cependant, VAC, OSAC et TRW-S étant des algorithmes complexes, ils ne sont généralement utilisés que dans le prétraitement, plutôt que dans l'ensemble de la procédure de recherche. Pendant la recherche, d'autres alternatives "plus légères" comme *EDAC (Existential Directional Arc Consistent)* [de Givry *et al.* 2005] ont tendance à être l'option par défaut. EDAC est robuste et était en fait l'algorithme en temps polynomial le plus fort pour obtenir une cohérence d'arc douce avant que OSAC et VAC n'entrent en jeu [Cooper *et al.* 2010].

Pour les implémentations que nous présenterons au chapitre 6, nous utilisons

ToulBar2[1] [Cooper *et al.* 2010], un solveur exact à code source ouvert qui s'est révélé extrêmement performant en tant que solveur *boîte noire* [Hurley *et al.* 2016]. ToulBar2 effectue une recherche hybride meilleure en premier (HBFS), combinant les qualités de la recherche en profondeur d'abord et de la recherche meilleure en premier [Allouche *et al.* 2015]. Il utilise un certain nombre d'algorithmes sophistiqués, comme la recherche par boosting avec élimination des variables [Larrosa & Dechter 2003], qui sont soit utilisés par défaut (mais peuvent toujours être désactivés), soit rendus facultatifs pour l'utilisateur. Les fonctionnalités par défaut incluent l'algorithme EDAC pour produire des bornes inférieures de bonne qualité pendant la recherche, ainsi que les heuristiques dom/wdeg et last-conflict pour l'ordre des variables.

2.4 Réseaux Bayésiens

Lorsque nous avons une fonction sur un grand ensemble de variables, la description de cette fonction croît exponentiellement avec le nombre de variables. Les GM probabilistes sont des outils puissants qui nous permettent d'exprimer les distributions de probabilité conjointes sur un grand ensemble de variables aléatoires de manière compacte. *Réseau bayésien* (BN) est un GM probabiliste qui capture les dépendances conditionnelles entre un ensemble de variables aléatoires via un graphe acyclique dirigé (DAG). Chaque sommet du DAG correspond à une variable aléatoire et les arêtes dirigées codent les dépendances conditionnelles.

Les BN fournissent une représentation compacte d'une distribution de probabilité conjointe donnée en décomposant la description de la fonction de probabilité. Prenons un example pour comprendre comment ils procèdent. Nous avons pris un example existant [Koller & Friedman 2009] comme base et l'avons modifié pour qu'il corresponde mieux à l'expérience personnelle de l'auteur : avoir vécu l'épidémie de Covid-19 et souffrir de rhinite allergique chaque printemps. Dans cet example, nous avons six variables aléatoires avec un ensemble prédéfini de valeurs qu'elles peuvent prendre :

- *Saison* $\in \{Printemps, \text{Été}, Automne, Hiver\}$
- *Nombre de cas quotidiens (rapports nationaux Covid-19)* $\in \{\text{Élevé}, Moyen, Bas\}$
- *Rhume des foins* $\in \{Vrai, Faux\}$
- *Covid-19* $\{Vrai, Faux\}$
- *Perte de goût* $\{Vrai, Faux\}$
- *Fièvre élevée* $\{Vrai, Faux\}$

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

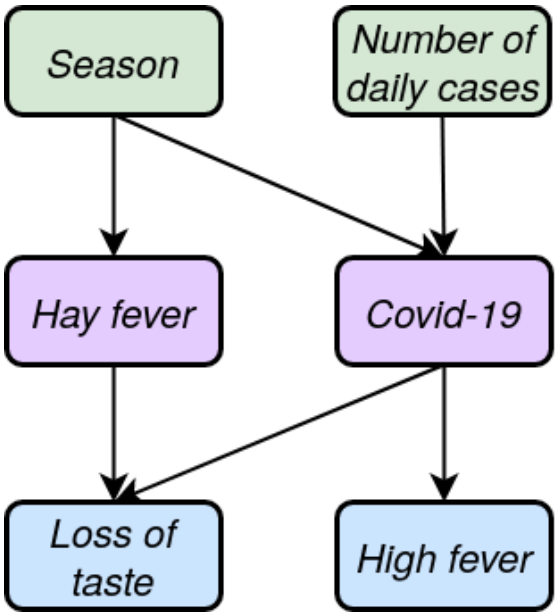


Figure 2.5: Relations de dépendance entre les variables représentées sous la forme d'un graphe dirigé. On observe que ce graphe se trouve être acyclique.

Nous avons $4\times3\times2\times2\times2\times2 = 192$ combinaisons possibles de valeurs de ces six variables. La tâche de définir une distribution de probabilité conjointe sur 192 événements possibles est fastidieuse. Il s'agit même d'un tout petit example avec seulement 6 variables ayant des domaines de très petite taille. Si nous voulons être en mesure de traiter des paramètres ayant une taille plus réaliste, nous devons trouver un moyen de simplifier cette tâche. Grâce aux BN, nous pouvons décomposer des distributions de probabilité complexes et les définir comme un produit de facteurs. Considérons les relations de dépendance entre ces variables aléatoires comme le montre la Figure 2.5.

Dans la Figure 2.5, nous avons un graphe acyclique dirigé (DAG) dont les sommets correspondent aux variables aléatoires {*Saison, Nombre de cas quotidiens, Rhume des foins, Covid-19, Perte de goût, Forte fièvre*}. Nous associons à chaque sommet une distribution de probabilité conditionnelle (DPC), de sorte que le DAG avec les DPC locales donne une représentation décomposée de la distribution de probabilité originale. Grâce à cette représentation, nous pouvons calculer, par example, la probabilité de l'événement {*printemps, taux moyen, pas de rhume des foins, pas de Covid-19, perte de goût, pas de forte fièvre*} comme un produit de ces six facteurs :

- *P(Saison = Printemps)*
- *P(Cas quotidiens = Moyen)*
- *P(Rhume des foins = Faux | Saison = Printemps)*

- *P(Covid-19 = Faux | Saison = Printemps, Cas quotidiens = Moyen)*
- *P(Perte de goût = Vrai | Fièvre des foins = Faux, Covid-19 = Faux)*
- *P(Fièvre élevée = Faux | Covid-19 = Faux)*

Ce calcul nécessite $3 + 2 + 4 + (4 \times 3) + (2 \times 2) + 2 = 27$ de paramètres indépendants : beaucoup plus petit comparé à $192 - 1 = 191$.

2.4.1 Apprentissage de la Structure des Réseaux Bayésiens

Dans ce travail, nous nous intéressons à l'apprentissage d'un RB à l'aide de données discrètes. Ceci est connu sous le nom d'apprentissage de structure de réseau bayésien (BNSL) dans la littérature scientifique. Imaginez que depuis le premier verrouillage de Covid-19 en France, nous collectons des données au format suivant :

Person ID	Date	Saison	Nombre de cas quotidiens	Rhume des foins	Covid-19	Perte de goût	Fièvre élevée
1	17/03/2020	Printemps	Élevé	Non	Non	Non	Non
2	17/03/2020	Printemps	Élevé	Non	Oui	Non	Oui
...	...	...	...	...	...	...	...
1	18/03/2020	Printemps	Élevé	Non	Oui	Oui	Oui
2	18/03/2020	Printemps	Élevé	Non	Oui	Non	Oui
...	...	...	...	...	...	...	...
1	01/06/2020	Été	Bas	Oui	Non	Oui	Non
2	01/06/2020	Été	Bas	Non	Non	Non	Non
...	...	...	...	...	...	...	...

Table 2.6: Un ensemble d'observations quotidiennes sur un groupe de personnes prenant en compte six critères : saison, nombre de cas quotidiens de Covid-19, avoir le rhume des foins, avoir le Covid-19, présence d'une perte de goût, présence d'une forte fièvre.

Nous prenons les données, qui sont un ensemble d'observations en entrée, et nous essayons de construire le BN le plus simple qui a le plus de chances de reproduire ces données. L'apprentissage du NE est un processus en deux étapes : Premièrement, nous apprenons la structure du BN, en d'autres termes, le DAG sous-jacent, et deuxièmement, nous apprenons les probabilités conditionnelles en utilisant les relations de dépendance intégrées dans la structure. Pour le BN de la Figure 2.5, nous apprenons la probabilité de chaque valeur de *Perte de goût* étant donné *Fièvre des foins* et *Covid-19*.

L'apprentissage des DPC peut se faire en temps polynomial et on sait que le BN appris en appliquant ces deux étapes en séquence est globalement optimal : [Heckerman *et al.* 1995]. Une fois que le BN est appris à partir de données, il peut être utilisé comme un outil puissant pour *l'inférence* [Koller & Friedman 2009], par example pour trouver l'affectation la plus probable à une variable aléatoire étant donné un ensemble d'observations sur d'autres. C'est ce qu'on appelle trouver l'explication la plus probable (EPP), et l'EPP est au cœur de nombreux problèmes de la vie réelle :

trouver le diagnostic le plus probable compte tenu des symptômes et des résultats de tests médicaux [Wasyluk *et al.* 2001], les prévisions météorologiques compte tenu des observations météorologiques [Kennett *et al.* 2001, Cofiño *et al.* 2002], et bien d'autres problèmes dans les domaines de la biotechnologie [Allouche *et al.* 2013, Liu *et al.* 2016, Xing *et al.* 2017], de l'économie [Demirer *et al.* 2006, Gemela 2001], sociologie [Nedevschi *et al.* 2006, Sticha *et al.* 2006], vision par ordinateur [Yuille & Kersten 2006, Luo *et al.* 2005, Stassopoulou *et al.* 1998, Suk *et al.* 2008, Zhang & Ji 2011], analyse du risque [Baksh *et al.* 2018, Johansson & Falkman 2008, Lee & Lee 2006, Sỳkora *et al.* 2018, Trucco *et al.* 2008], et ainsi de suite [Sierra *et al.* 2018].

Cependant, l'apprentissage de la structure, le DAG, est NP-hard, même si le nombre de parents pour chaque variable aléatoire est limité à 2 [Chickering *et al.* 2004]. Cette tâche difficile implique de trouver le graphe optimal dans un espace de recherche superexponentiel, qui s'étend drastiquement avec chaque variable aléatoire supplémentaire.

2.4.2 Méthode de Recherche par Scores

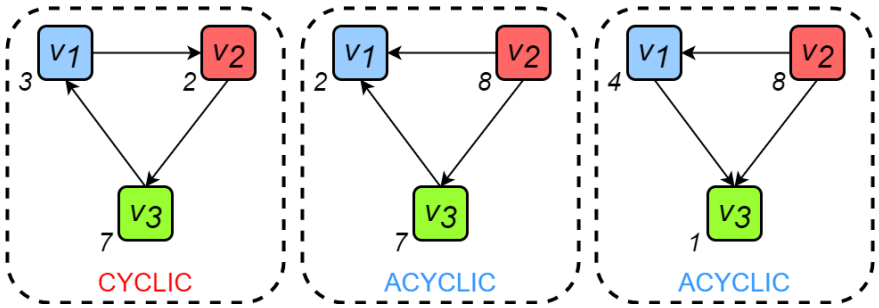
Étant donné un ensemble V de variables aléatoires avec $|V| = n$, une variable v a $2^{(n-1)}$ d'ensembles parents candidats. Sans parler de trouver le NE optimal, le simple fait d'attribuer un score à chacun de ces ensembles parents prendrait un temps considérable. À ce stade, il est tout à fait naturel de se demander comment traiter ces ensembles de parents candidats dont le nombre est exponentiel. Heureusement, dans la majorité des cas, il est possible de montrer que de nombreux ensembles parents ne peuvent pas apparaître dans un NE optimal [de Campos & Ji 2010]. Par conséquent, ces ensembles parents peuvent être omis dès le début. Cependant, le nombre d'ensembles parents candidats a tendance à être important même après l'élimination de ces ensembles parents. Ce problème peut être résolu en limitant la taille maximale des ensembles parentaux pour lesquels les scores sont calculés [de Campos & Ji 2010, de Campos *et al.* 2018]. La limitation de la taille des ensembles parentaux empêche malheureusement le BN optimal trouvé d'être *globalement optimal*, puisque nous omettons évidemment des parties de l'espace de recherche.

L'approche que nous utilisons ici pour apprendre un NE à partir de données est la méthode *score-and-search*. Étant donné un ensemble de données discrètes multivariées $I = \{I_1, \ldots, I_N\}$, une *fonction de score* $\sigma(G \mid I)$ mesure la qualité du BN avec la structure sous-jacente G. Le problème BNSL demande de trouver une structure G qui minimise $\sigma(G \mid I)$ pour une certaine fonction de scoring σ, ce qui se trouve être un problème NP-hard [Chickering 1995].

Plusieurs fonctions de notation ont été proposées à cette fin, notamment BDeu [Buntine 1991, Heckerman *et al.* 1995] et BIC [Schwarz 1978, Lam & Bacchus 1994]. Ces fonctions sont *décomposables* et peuvent être exprimées comme la somme de scores locaux qui ne dépendent que de l'ensemble des parents (dorénavant, *ensemble des parents*) de chaque sommet : $\sigma_F(G \mid I) = \sum_{v \in V} \sigma_F^v(parents(v) \mid I)$

v	$S \in PS(v)$	$\sigma^v(S)$
v_1	$\{v_2, v_3\}$	2
	$\{v_2\}$	4
	$\{v_3\}$	3
	$\{\}$	10
v_2	$\{v_1, v_3\}$	1
	$\{v_1\}$	2
	$\{v_3\}$	6
	$\{\}$	8
v_3	$\{v_1, v_2\}$	1
	$\{v_1\}$	4
	$\{v_2\}$	7
	$\{\}$	9

Table 2.7: Ensembles parentaux candidats et leurs scores pour chaque variable de l'exemple 2.20.



(a) Somme des scores = 12. (b) Somme des scores = 17. (c) Somme des scores = 13.

Figure 2.6: Trois alternatives de réseau pour les variables et leurs ensembles parents candidats dans l'exemple 2.20. La première présente un cycle et est donc infaisable ; celle du milieu est faisable avec un score total sous-optimal, et la dernière est le réseau optimal.

pour $F \in \{BDeu, BIC\}$.

Dans ce cadre, nous calculons d'abord les scores locaux, puis la structure du score minimal. Nous désignons par $PS(v)$ l'ensemble des ensembles parents candidats de $v \in V$. Dans ce qui suit, nous supposons que les scores locaux sont précalculés et donnés en entrée, comme cela est courant dans les travaux axés sur BNSL. Nous omettons également de mentionner explicitement I ou F, car ils sont constants pour la résolution de tout ensemble de données donné.

Example 2.20. *Dans le tableau 2.7, nous avons un ensemble $V = \{v_1, v_2, v_3\}$ de trois variables avec leurs ensembles parents candidats et leurs scores. Dans les figures 2.6a, 2.6b et 2.6c, trois alternatives de réseau sont présentées pour ces variables avec leur somme de scores résultante. Le réseau de la Figure 2.6a a un très bon score total, mais n'est malheureusement pas acyclique. Les réseaux des figures 2.6b et 2.6c sont tous deux faisables, c'est-à-dire des réseaux acycliques, ce dernier étant le réseau optimal.*

2.4.3 GOBNILP

L'un des algorithmes de pointe pour BNSL est GOBNILP (Globally Optimal Bayesian Network learning using Integer Linear Programming) [Cussens 2011, Barlett & Cussens 2013]. Il utilise un algorithme spécialisé de branchement et de coupe dans un solveur ILP. À chaque nœud de l'arbre de branches et de limites, il génère des coupes qui améliorent la relaxation linéaire.

GOBNILP formule le problème BNSL comme l'ILP 0/1 ci-dessous :

$$\min \sum_{v \in V, S \in PS(v)} \sigma^v(S) x_{vS} \tag{2.2a}$$

$$s.t. \sum_{S \in PS(v)} x_{vS} = 1 \qquad \forall v \in V \tag{2.2b}$$

$$\sum_{v \in W, S \in PS^{-W}(v)} x_{vS} \geq 1 \qquad \forall W \subseteq V \tag{2.2c}$$

$$x_{vS} \in \{0, 1\} \qquad \forall v \in V, S \in PS(v) \tag{2.2d}$$

Cette ILP possède une variable 0/1 x_{vS} pour chaque ensemble parent candidat S de chaque sommet v, où $x_{vS} = 1$ signifie que S est l'ensemble parent de v. L'objectif (2.2a) encode directement la décomposition de la fonction de notation. La contrainte (2.2b) affirme qu'exactement un ensemble de parents est sélectionné pour chaque variable aléatoire. Enfin, les contraintes (2.2c), appelées *inégalités de cluster*, garantissent que pour chaque sous-ensemble, c'est-à-dire cluster $W \subseteq V$, il existe au moins une variable $v \; dans W$ qui se voit attribuer un ensemble parent S à partir de l'ensemble $PS^{-W}(v) \subseteq PS(v)$ de ses ensembles parents candidats qui *ne* coupent W. Ils sont basés sur la propriété suivante [Jaakkola *et al.* 2010] des graphes acycliques dirigés :

Theorem 2.21. *Soit G un graphe dirigé sur des sommets V et soit $parents(v)$ les parents du sommet v dans le graphe. G est acyclique si et seulement si pour chaque sous-ensemble non vide, c'est-à-dire cluster $W \subseteq V$ il existe au moins un sommet $v \in W$ avec $parents(v) \cap W = \{\}$.*

Un cluster W est un *violated cluster* lorsqu'il ne satisfait pas la condition ci-dessus, c'est-à-dire qu'il n'existe aucune variable $v \in W$ dont les parents sont *tous*

en dehors de W. L'inégalité de cluster (2.2c) pour le cluster W est violée lorsque W est un cluster violé.

Comme il existe un nombre exponentiel d'inégalités de cluster, GOBNILP génère uniquement celles qui sont violées et qui améliorent donc la relaxation linéaire actuelle. Elles sont appelées *cluster cuts*. Il s'agit en soi d'un problème de séparation qui se trouve être NP-hard [Cussens *et al.* 2017], que GOBNILP encode et résout également comme un ILP. Il est intéressant de noter que ces inégalités sont des inégalités définissant les facettes du polytope BNSL [Cussens *et al.* 2017], ce qui permet d'améliorer la relaxation de manière significative.

2.4.4 CPBayes

CPBayes [Van Beek & Hoffmann 2015] est une méthode basée sur CP pour BNSL. Elle utilise le modèle CP suivant :

$$\begin{aligned} \min_{v \in V, S \in PS(v)} \quad & \sum_{v \in V} \sigma^v(S) \\ s.t. \quad & acyclic(V) \end{aligned}$$

Dans ce modèle, chaque $v \in V$ est appelé un *variable d'ensemble parent* avec un ensemble d'ensembles parents candidats $PS(v)$, c'est-à-dire son domaine. Une connexion immédiate entre les modèles GOBNILP et CPBayes est que les variables ILP $x_{vS}, \forall S \in PS(v)$ sont le codage direct [Walsh 2000] des variables d'ensemble parent du modèle CP. Par conséquent, nous les utilisons de manière interchangeable, par example, nous pouvons nous référer à la valeur S dans $D(v)$ comme x_{vS}. Toute affectation complète $T \in \ell(V)$ encode un DAG : à partir de la valeur $S = T(v)$ de chaque variable $v \in V$ nous extrayons les arêtes dirigées qui sont présentes dans le digraphe. Enfin, sur tout l'ensemble V des variables, nous avons une contrainte d'acyclicité globale. Par conséquent, toute solution réalisable de ce modèle correspond à un DAG.

CPBayes exploite les relations de symétrie et de dominance présentes dans le problème, la mise en cache des sous-problèmes et une base de données de modèles pour calculer les bornes inférieures, adaptées de la recherche heuristique [Fan & Yuan 2015]. Il exécute un algorithme de recherche par retour en arrière où, à chaque niveau de l'arbre de recherche, il instancie une variable d'ensemble parent et propage les valeurs de domaine pour supprimer celles qui sont incohérentes avec l'affectation partielle actuelle. Pour la contrainte d'acyclicité globale, il n'utilise pas de propagateur GAC, mais plutôt un vérificateur de satisfiabilité en temps polynomial qui détecte chaque fois qu'une affectation partielle actuelle est cyclique. Nous verrons ce vérificateur de satisfiabilité plus en détail dans les chapitres 3 et 4.

Il a été démontré que CPBayes est compétitif avec GOBNILP dans de nombreux ensembles de données [Van Beek & Hoffmann 2015]. Contrairement à GOBNILP, les mécanismes d'inférence de CPBayes sont très légers, ce qui lui permet d'explorer plusieurs ordres de grandeur de nœuds supplémentaires par unité de temps, même en

tenant compte du fait que le calcul des bases de données de motifs avant la recherche peut parfois consommer un temps considérable. D'autre part, le mécanisme léger de délimitation basé sur les motifs ne peut prendre en compte que des informations limitées sur l'état actuel de la recherche. Plus précisément, il peut prendre en compte l'ordonnancement total actuel impliqué par le DAG en cours de construction, mais aucune information qui a été dérivée sur les ensembles parents potentiels de chaque sommet, c'est-à-dire les domaines actuels des variables des ensembles parents.

Remarque. CPBayes et GOBNILP se situent aux deux extrêmes du compromis entre la force d'inférence et la vitesse d'inférence : GOBNILP produit des bornes inférieures de haute qualité alors que CPBayes a un taux exceptionnel de nœuds de recherche découverts par seconde. Dans cette thèse, notre objectif principal est de parvenir à un meilleur compromis entre ces deux extrêmes. Nous prenons le solveur CPBayes comme base et y incorporons plusieurs algorithmes, afin de mieux traiter la contrainte d'acyclicité globale et les grands domaines. Tout au long des chapitres de ce manuscrit, nous ajouterons chacun de ces algorithmes un par un, et à la toute fin, nous appellerons le solveur résultant ELSA (**E**xact **L**earning of bayesian network **S**tructure using **A**cyclicity reasoning).

CHAPTER 3

Cohérence d'Arc Généralisée pour la Contrainte Globale d'Acyclicité

Contents

3.1 Introduction

La contrainte d'acyclicité globale pour BNSL est difficile à implémenter efficacement. van Beek et Hoffmann ont donné un algorithme qui détecte l'insatisfaisabilité de la contrainte d'acyclicité en temps $O(n^2d)$ [Hoffmann & van Beek 2013], où n est le nombre de variables aléatoires et d est la plus grande taille du domaine. Ils ont ensuite utilisé ce vérificateur de satisfiabilité pour construire un propagateur qui applique la cohérence d'arc généralisée (GAC) par sondage, c'est-à-dire en détectant l'insatisfiabilité après avoir attribué chaque valeur individuelle et en élaguant les valeurs qui conduisent à l'insatisfiabilité. Cela donne un propagateur GAC de complexité $O(n^3d^2)$. Cependant, ils ont montré plus tard que ce propagateur GAC était malheureusement trop coûteux pour être pratique [Van Beek & Hoffmann 2015].

En utilisant ce vérificateur d'acyclicité, nous montrons que nous pouvons appliquer la GAC en temps $O(n^4d)$. Étant donné que d est généralement beaucoup plus grand que n, cela représente une amélioration significative par rapport à $O(n^3d^2)$. Nous montrons par la suite que nous pouvons encore réduire la complexité à $O(n^3d)$.

De plus, nous proposons un ensemble de techniques qui n'améliorent pas la complexité asymptotique mais améliorent les performances.

3.2 Vérificateur d'Acyclicité de van Beek et Hoffman

`acycChecker` présenté dans l'algorithme 20 prend ce qui suit comme entrée :

- $prefix$: un sous-ensemble de variables de l'ensemble parent déjà placé dans la liste ordonnée.

  Notez que nous traitons $prefix$ à la fois comme un ensemble et une liste ordonnée, selon le cas.

- V : l'ensemble de toutes les variables de l'ensemble parent.

- D : les domaines des variables dans V.

  Notez que les domaines sont les domaines actuels au moment où `acycChecker` est appelé. Par conséquent, par exemple, le domaine d'une variable instanciée est un domaine singleton constitué de la valeur qui lui est attribuée.

Algorithm 20: Vérificateur d'acyclicité.

```
1  Procedure acycChecker(prefix, V, D):
2      O ← prefix
3      support ← [ ]
4      changes ← true
5      i ← size(prefix)
6      while changes do
7          changes ← false
8          foreach v ∈ V \ O do
9              if ∃S ∈ D(v) s.t. S ⊆ O then
10                 O_i ← v
11                 support_v ← S
12                 changes ← true
13                 i ← (i + 1)
14     return O
```

Son exactitude est basée sur la propriété suivante [Jaakkola *et al.* 2010] des graphes acycliques dirigés.

Theorem 3.1. *Soit G un graphe dirigé sur des sommets V et soit $parents(v)$ les parents du sommet v dans le graphe. G est acyclique si et seulement si pour chaque sous-ensemble non vide $W \subseteq V$ il existe au moins un sommet v $dansW$ avec $parents(v) \cap W = \{\}$.*

`acycChecker` construit avidement une liste ordonnée O des variables[1], de telle sorte que si la variable v est plus tardive dans la liste ordonnée que v', alors $v \notin parents(v')$. Pour ce faire, elle essaie de choisir un ensemble parent S pour un sommet non encore ordonné de telle sorte que S soit entièrement contenu dans l'ensemble des sommets précédemment ordonnés. L'ordre dans lequel les sommets non encore ordonnés sont considérés n'a pas d'importance, c'est-à-dire que l'algorithme est correct quel que soit l'ordre utilisé. Toutefois, sans perte de généralité, nous choisissons de considérer les variables dans un ordre lexicographique dans notre implémentation.

Si en fait toutes les affectations donnent des graphes cycliques, `acycChecker` finira par atteindre un point où tous les sommets restants forment un cycle entre eux, et il retournera une liste ordonnée partiellement construite. S'il existe au moins une affectation complète qui donne un graphe acyclique, il sera possible par le théorème 3.1 de sélectionner à partir d'une variable dans $V \setminus O$ un ensemble parent qui n'intersecte pas $V \setminus O$, donc qui est un sous-ensemble de O. La valeur S choisie pour chaque variable dans la ligne 9 donne également un témoin d'un tel graphe acyclique.

Theorem 3.2. `acycChecker` *retourne une liste ordonnée complète si et seulement s'il existe une instanciation des variables qui donne un DAG acyclique. Il le fait en* $O(n^2d)$ *temps.*

Proof. $\Rightarrow$ Soit $i \in \{0, \ldots, (n-1)\}$ l'itération de la boucle while qui doit être exécutée. `acycChecker` a placé i de variables dans O et examine les variables dans $V \setminus O$. Supposons qu'il existe une solution acyclique avec l'ensemble actuel de domaines. Par le théorème 3.1, nous savons qu'il existe une variable $v \in (V \setminus O)$ avec une valeur $S \in D(v)$ telle que $S \cap (V \setminus O) = \{\}$. Par conséquent, à chaque itération i, nous pouvons trouver une variable à placer dans O.

$\Leftarrow$ Supposons maintenant que l'ensemble actuel de domaines ne peut donner aucune solution acyclique. Cela signifie qu'il existe au moins un cycle $\{v_1 \rightarrow \ldots \rightarrow v_m \rightarrow v_1\}$. Soit $W = \{v_1, \ldots, v_m\}$ l'ensemble des variables impliquées dans ce cycle. Sans perte de généralité, supposons que `acycChecker` a placé toutes les variables de $V \setminus W$ dans la liste ordonnée, nous avons donc $prefix = V \setminus W$. Cependant, par le théorème 3.1, nous savons qu'il n'existe aucune variable v dans W avec une valeur S dans son domaine telle que $S \cap W = \{\}$. L'échec survient donc.

Dans le pire des cas, la boucle while (ligne 6) et la boucle for (ligne 8) s'exécuteront n fois, et la vérification du domaine (ligne 9) s'exécutera d fois. Il en résulte une complexité $O(n^2d)$. □

Example 3.3 (Vérificateur d'acyclicité sur un petit problème)**.** *Considérons un problème BNSL dont les domaines sont indiqués dans le tableau 3.1. Soit O un vecteur qui est initialement vide. Soit* O_i *la variable* $v \in V$ *qui se trouve à la position* i *dans* O*.*

[1] Comme $prefix$, nous traitons O à la fois comme une liste ordonnée et un ensemble, selon le cas.

Variable	Domaine
v_0	$\{v_1\}$
	$\{v_2\}$
	$\{v_2, v_4\}$
v_1	$\{v_2\}$
	$\{v_3\}$
	$\{v_4\}$
	$\{v_2, v_3\}$
	$\{v_3, v_4\}$
v_2	$\{\}$
v_3	$\{v_0, v_4\}$
	$\{v_1, v_2\}$
v_4	$\{v_0\}$

Table 3.1: Le problème BNSL utilisé comme exemple courant dans ce chapitre.

À la première itération de la boucle while commençant à la ligne 6, `acycChecker` *examine v_0 pour voir s'il peut le placer dans la liste ordonnée. Étant donné qu'il n'y a aucune valeur dans $D(v_0)$ qui lui permette d'être la première dans la liste ordonnée, il passe à la variable suivante, v_1, puis à v_2. v_2 a l'ensemble vide dans son domaine et par conséquent, O_0 devient v_2. À l'itération suivante, il regarde à nouveau v_0. Voyant qu'elle peut maintenant ajouter v_0 à la liste ordonnée puisqu'il peut avoir v_2 comme parent qui a déjà été ajouté à O, elle fixe $O_1 = v_0$. En continuant, il regarde v_1 et voit qu'il a un ensemble parent candidat, $\{v_2\}$, qui est un sous-ensemble de l'actuel O, il ajoute donc v_1 à O et O_2 devient v_1. Il continue de la même manière jusqu'à ce qu'il obtienne la liste ordonnée $\{v_2, v_0, v_1, v_3, v_4\}$.*

3.3 Intuition Derrière le Propagateur GAC

Supposons que `acycChecker` se termine sans détecter l'insatisfaisabilité. Alors, il produit et renvoie une *liste ordonnée valide* $O = \{O_0, \ldots, O_i, \ldots, O_{n-1}\}$ de variables qu'il obtient en choisissant à chaque étape i une variable avec une valeur d'ensemble parent dans son domaine qui est un sous-ensemble des variables choisies aux étapes $0 \ldots (i-1)$.

Rappelons qu'à partir d'une liste ordonnée valide O, nous pouvons construire un support pour la contrainte : à chaque position i, choisissez une valeur d'ensemble parent S dans le domaine de $v = O_i$ telle que $S \subseteq \{O_0, \ldots, O_{i-1}\}$. Nous disons

qu'une telle valeur est cohérente avec la liste ordonnée O. Cependant, le choix de S n'a aucun impact sur le fait que l'affectation construite soit un support. Par conséquent, tout choix fera l'affaire et donc une liste ordonnée valide fournit un support pour toutes les valeurs qui sont cohérentes avec lui.

Considérons maintenant $S' \in D(v)$ qui est inconsistant avec O, c'est-à-dire que $S' \not\subseteq \{O_0, \ldots, O_{i-1}\}$, nous devons donc sonder pour voir s'il est supporté. Par sonde, nous entendons une séquence d'opérations : supposer $v = S'$, propager, élaguer $S' \in D(v)$ si la propagation échoue. Nous savons que pendant la sonde, rien ne force `acycChecker` à s'écarter de $\{O_0, \ldots, O_{i-1}\}$. Ainsi, lors d'une sonde réussie, il *n'est pas incorrect* pour `acycChecker` de construire une nouvelle liste ordonnée valide en plaçant les premiers $i-1$ éléments $\{O_0, \ldots, O_{i-1}\}$ de la même manière, et en essayant de placer une autre variable en premier, et non v, à la position i. S'il existe une autre variable v' avec $S' \in D(v')$ telle que $S' \subseteq \{O_0, \ldots, O_{i-1}\}$, alors `acycChecker` peut construire un autre liste ordonnée valide O' qui est identique à O dans les $i-1$ premières positions $\{O_0, \ldots, O_{i-1}\}$ et dans lequel v est *poussé vers le bas*, c'est-à-dire que v est placé à la position i. c'est-à-dire que v est placé à une position postérieure à i. Alors toutes les valeurs $S \in D(v)$ compatibles avec O' sont supportées.

Algorithm 21: Vérificateur d'acyclicité - La version prenant v' et i' en entrée et s'assurant que v' est placé après la position i'.

```
1  Procedure acycChecker-Constrained(prefix, V, D, v', i'):
2      O ← prefix
3      support ← {}
4      changes ← true
5      i ← size(prefix)
6      while changes do
7          changes ← false
8          foreach v ∈ V \ O do
9              if v = v' and i ≤ i' then
10                 continue
11             if ∃S ∈ D(v) s.t. S ⊆ O then
12                 O_i ← v
13                 support_v ← S
14                 changes ← true
15                 i ← (i + 1)
16     return O
```

Cette observation offre une amélioration potentielle des performances par rapport au propagateur de sondage : nous n'avons besoin de sonder que les valeurs qui ne sont pas supportées par la liste ordonnée produit lors de la première exécution de `acycChecker`. De plus, au fur et à mesure que nous l'exécutons dans différentes sondes, il produit de nouvelles liste ordonnées qui supportent un ensemble de valeurs,

Algorithm 22: GAC propagator probing with orders.

```
1  Procedure Acyclicity-GAC-n4d(V, D):
2      O ← acycChecker(∅, V, D)
3      if O ⊊ V then
4          return false
5      foreach v ∈ V do
6          i ← O^{-1}(v)
7          prefix ← {O_0, ..., O_{i-1}}
8          Mark each S ∈ D(v) s.t. S ⊆ prefix
9          while i < n and O is not empty do
10             v' ← v
11             i' ← i
12             O ← acycChecker-Constrained(prefix, V, D, v', i')
13             i ← O^{-1}(v)
14             prefix ← {O_0, ..., O_{i-1}}
15             Mark each S ∈ D(v) s.t. S ⊆ prefix
16     foreach v ∈ V, S ∈ D(v) do
17         if v = S is unmarked then
18             Prune v = S
19     return true
```

qui est un sur-ensemble des valeurs précédemment supportées. Par conséquent, nous pouvons diminuer le nombre de sondes qui sont nécessaires pour trouver un support pour toutes les valeurs. Pour être plus précis, nous réduisons le nombre de sondes de $O(nd)$ (exécuter `acycChecker` pour chaque variable et valeur) à $O(n^2)$ (exécuter `acycChecker` pour chaque variable et position). `Acyclicity-GAC-n4d`, présenté dans l'algorithme 22, exploite cette perspicacité. Il s'assure d'abord que `acycChecker` peut produire une liste ordonnée valide O. Ensuite, pour chaque variable v, il construit une nouvelle liste ordonnée O' à partir de O de sorte que v soit poussé vers le bas autant que possible. Enfin, il élague toutes les valeurs de l'ensemble des parents $S\ dans D(v)$ qui sont incompatibles avec O'.

3.4 Le Propagateur GAC

Dans `Acyclicity-GAC-n4d` présenté dans l'Algorithme 22, nous appelons d'abord `acycChecker` à la ligne 2 pour construire une liste ordonnée initiale valide. S'il ne peut pas produire une liste ordonnée valide complète, nous retournons immédiatement un échec. Ensuite, dans la boucle for qui commence à la ligne 5, nous prenons chaque variable v une par une et dans la boucle while qui commence à la ligne 9, nous essayons de pousser v dans la liste ordonnée initial autant que

possible. Nous le faisons en appelant une[2] version de `acycChecker`, appelée `acycChecker-Constrained` car nous ne l'utilisons plus[3], à la ligne 12 avec deux paramètres d'entrée supplémentaires. Ce sont v' et i' et nous les utilisons pour nous assurer que v n'est pas antérieur à la position $i+1$ dans la liste ordonnée. Nous utilisons également la notation $O^{-1}(v)$ pour obtenir l'indice de v dans la liste ordonnée O. Notez également qu'ici, le $prefix$ donné à `acycChecker-Constrained` n'est pas nécessairement l'ensemble vide, mais l'ensemble des variables précédant v dans O.

Chaque fois que nous parvenons à faire descendre une variable, la liste ordonnée, le préfixe et la position i de la variable sont mis à jour. Étant donné le nouveau $prefix$, nous marquons chaque valeur $S \in D(v)$ qui est un sous-ensemble de celui-ci comme cohérente. À la fin de toutes les itérations, nous élaguons les valeurs qui n'ont jamais été marquées.

Theorem 3.4. `Acyclicity-GAC-n4d` *applique la GAC sur la contrainte d'acyclicité en* $O(n^4d)$.

Proof. Nous savons que si une valeur $S \in D(v)$ d'une variable $v \in V$ est cohérente, alors il existe une liste ordonnée valide Q qui la supporte. Ici, nous montrons qu'à partir de toute liste ordonnée valide O qui ne supporte pas S, on peut construire une autre liste ordonnée valide O' qui le supporte. Nous montrons également que pour chaque variable v, il existe une liste ordonnée valide qui supporte toutes ses valeurs cohérentes et que `Acyclicity-GAC-n4d` découvre toujours une telle liste ordonnée valide.

Soit $v \in V$ et $S \in D(v)$. Laissez également $O = \{O_0, \ldots, O_{(n-1)}\}$ et $Q = \{Q_0, \ldots,$
$Q_{(n-1)}\}$ être deux liste ordonnées valides tels que O ne supporte pas S alors que Q le fait. Supposons que $O_i = Q_j = v$ et laissons $O_p = \{O_0, \ldots, O_{(i-1)}\}$, $Q_p = \{Q_0, \ldots, Q_{(j-1)}\}$ être les *prefix* de v, c'est-à-dire les ensembles de variables qui précèdent v dans les liste ordonnées O et Q, respectivement. De la même manière, laissez $O_s = \{O_{i+1}, \ldots, O_{(n-1)}\}$, $Q_s = \{Q_{j+1}, \ldots, Q_{(n-1)}\}$ être les *suffix* de v, c'est-à-dire les ensembles de variables qui succèdent à v dans les liste ordonnées O et Q, respectivement.

Soit O' la liste ordonnée O_p suivi de Q_p, suivi de v, suivi de O_s, en ne gardant que la première occurrence de chaque variable lorsqu'il y a des doublons. O' est une liste ordonnée valide : O_p est témoigné par l'affectation qui témoigne de O, Q_p par l'affectation qui témoigne de Q, v par S (comme dans Q) et O_s par l'affectation qui témoigne de O. Il supporte également S, comme requis.

Passer de O à O' peut se faire par étapes : Nous parcourons les variables de Q_p une par une, à savoir $Q_0, \ldots, Q_{(j-1)}$, et la première variable que nous atteignons qui vient après v dans O nous pouvons l'insérer avant v. Que cette première variable soit Q_p^*. Nous pouvons alors transformer O en O'^1, dans lequel nous avons O_p suivi

[2] La différence provient des deux paramètres supplémentaires v' et i' et des lignes 9 et 10.

[3] La raison en sera claire dans la section 3.4.1.

de $\{Q_p^*\}$, suivi de $\{v\}$, suivi de $O_s \setminus \{Q_p^*\}$. Nous pouvons continuer à pousser v vers le bas de cette manière jusqu'à ce que nous obtenions O'^2 et ainsi de suite et finalement O'.

Une fois que nous ne pouvons plus pousser v vers le bas, cela signifie que toutes les valeurs cohérentes sont supportées, sinon nous serions capables de transformer la liste ordonnée actuel en une autre liste ordonnée valide pour supporter cette valeur. Par conséquent, la liste ordonnée final obtenu nous permettra de supprimer toutes les valeurs incohérentes et de conserver toutes celles qui sont cohérentes.

Nous répétons la boucle while à la ligne 9 au plus n fois pour chacune des n variables, où nous appelons `acycChecker` avec une complexité $O(n^2d)$, ce qui donne $O(n^4d)$. □

Example 3.5 (Exécution de GAC sur un petit problème)**.** *Considérons un problème BNSL avec des domaines comme indiqué dans le tableau 3.1. Nous allons maintenant exécuter* `Acyclicity-GAC-n4d` *présenté dans l'Algorithme 22 sur ce problème. À partir de l'exemple 3.3, nous savons que* `acycChecker` *à la ligne 2 renvoie* $O = \{v_2, v_0, v_1, v_3, v_4\}$*. Soit* O_i *la variable* $v \in V$ *qui se trouve à la position* i *dans* O*.*

Pousser les variables vers le bas dans la liste ordonnée. *Pendant la boucle for qui commence à la ligne 5, nous considérons chaque variable* $v \in V$ *une par une et essayons de voir si nous pouvons la pousser vers le bas dans la liste ordonnée. À la première itération, nous regardons la variable* v_0*. Elle se trouve à la position* $i = 1$*, et son préfixe est* $\{v_2\}$*. À la ligne 15, nous marquons* $S = \{v_2\} \in D(v_0)$ *comme cohérent et appelons* `acycChecker` *pour construire une liste ordonnée en utilisant* $\{v_2\}$ *comme préfixe de sorte que* v_0 *ne soit pas placé avant la position 2.* O *devient* $\{v_2, v_1, v_0, v_3, v_4\}$*. A l'itération suivante, à la ligne 15, nous marquons* $S = \{v_1\} \in D(v_0)$ *comme cohérent et demandons à* `acycChecker-Constrained` *de construire une liste ordonnée en utilisant* $\{v_2, v_1\}$ *comme préfixe de sorte que* v_0 *ne soit pas placé avant la position 3.* O *devient* $\{v_2, v_1, v_3, v_0, v_4\}$*. Il n'y a pas de valeur supplémentaire* $S \in D(v_0)$ *que nous pouvons marquer comme cohérente. Nous pouvons encore essayer de pousser* v_1 *plus bas car nous avons actuellement* $i = 3 < 4$*. Nous demandons à* `acycChecker-Constrained` *de construire une liste ordonnée utilisant* $\{v_2, v_1, v_3\}$ *comme préfixe de sorte que* v_0 *ne soit pas placé avant la position 4. Il ne parvient pas à construire une telle liste ordonnée car* v_0 *est un parent nécessaire pour* v_4*. Par conséquent, la boucle while pour* v_0 *se termine et nous élaguons la valeur* $S = \{v_2, v_4\} \in D(v_0)$ *puisqu'elle n'a jamais été marquée comme cohérente au cours des itérations.*

3.4.1 Réduire la Complexité à $O(n^3d)$

En suivant la preuve du théorème 3.4, étant donné une liste ordonnée initial valide O, il suffit de trouver une variable dans le suffixe de v que nous pouvons insérer avant v. Par conséquent, nous n'avons pas besoin de demander à `acycChecker-Constrained` de construire une nouvelle liste ordonnée valide à

chaque fois. Cette amélioration nous conduit à `Acyclicity-GAC-n3d` présenté dans l'algorithme 23. Comme on peut le voir, l'appel à `acycChecker-Constrained` n'est plus appelé à l'intérieur de la boucle while commençant à la ligne 9. Au lieu de cela, nous trouvons simplement une variable dans le suffixe, à savoir l'ensemble $O \setminus (prefix \cup \{v\})$, qui peut avoir un ensemble parent qui est un sous-ensemble du préfixe actuel. Par conséquent, la complexité de la boucle while diminue de $O(n^2d)$ à $O(nd)$.

Example 3.6 (Exécution du propagateur GAC amélioré `Acyclicity-GAC-n3d` sur un petit problème)**.** *Considérons un problème BNSL avec des domaines comme indiqué dans le tableau 3.1. D'après les exemples précédents, nous savons que le propagateur* `Acyclicity-GAC-n3d` *à la ligne 2 dans l'Algorithme 23 renvoie* $O = \{v_2, v_0, v_1, v_3, v_4\}$.

Pousser les variables vers le bas dans la liste ordonnée. *À la première itération de la boucle for commençant à la ligne 5, nous examinons la variable* v_0*. Elle se trouve à la position* $i = 1$*, et son préfixe est* $\{v_2\}$*. À la ligne 5, nous recherchons une variable dans* $O \setminus (prefix \cup \{v\})$*, qui est* $\{v_1, v_3, v_4\}$*, que nous pouvons insérer juste avant* v_0*.* v_1 *est une telle variable, car elle peut avoir* v_2 *comme parent. Par conséquent, nous ajoutons* v_1 *au préfixe (ligne 13) et le préfixe actuel devient* $\{v_2, v_1\}$*. À l'itération suivante de la boucle for, nous examinons l'ensemble* $\{v_3, v_4\}$ *pour voir s'il existe une variable que nous pouvons insérer juste avant* v_0*. Nous choisissons* v_3 *car elle peut avoir* $\{v_1, v_2\}$ *comme ensemble parent. Nous ajoutons ensuite* v_3 *au préfixe (ligne 13) et le préfixe actuel devient* $\{v_2, v_1, v_3\}$*. Enfin, à l'itération suivante, nous regardons la dernière variable du suffixe actuel,* v_4*, pour voir si nous pouvons l'insérer juste avant* v_0*. Ce n'est pas faisable, car le seul parent que* v_4 *peut avoir est* v_0*, il doit donc suivre* v_0 *dans la liste ordonnée.*

À la fin de ces itérations de la boucle for, nous avons obtenu une nouvelle liste ordonnée valide où v_0 *vient le plus tard possible :* v_2, v_1, v_3, v_0, v_4*. Observez que nous n'avons jamais maintenu cette liste ordonnée explicitement dans le pseudo-code, car nous avons seulement besoin de connaître l'ensemble des variables apparaissant dans le préfixe final. Maintenant, nous pouvons élaguer les valeurs* $S \in D(v_0)$ *qui ne sont pas un sous-ensemble de* $\{v_2, v_1, v_3\}$*. Cette valeur se trouve être* $\{v_2, v_4\}$.

Nous avons modifié les paramètres d'entrée de CPBayes afin de donner en entrée une solution initiale trouvée par des méthodes de recherche locale BNSL de pointe [4]. En pratique, nous exécutons l'algorithme génétique MINOBS [Lee & van Beek 2017b] avec les paramètres par défaut pendant 5 secondes (resp. 300 secondes) pour les petites (resp. grandes) instances. La meilleure liste ordonnée trouvé par MINOBS est donné comme solution primale initiale à notre solveur.

[4] Remplaçant la recherche locale par ascension de collines existante dans CPBayes.

Algorithm 23: Propagateur GAC pour l'acyclicité

```
1  Procedure Acyclicity-GAC-n3d(V, D):
2      O ← acycChecker(∅, V, D)
3      if O ⊊ V then
4          return false
5      foreach v ∈ V do
6          changes ← true
7          i ← O^{-1}(v)
8          prefix ← {O_0, ..., O_{i-1}}
9          while changes do
10             changes ← false
11             foreach w ∈ O \ (prefix ∪ {v}) do
12                 if ∃S ∈ D(w) s.t. S ⊆ prefix then
13                     prefix ← prefix ∪ {w}
14                     changes ← true
15         Prune {S | S ∈ D(v) ∧ S ⊈ prefix}
16     return true
```

3.5 Optimisation du Code

Nous avons apporté deux modifications principales à l'implémentation de `Acyclicity-GAC-n3d` dans l'Algorithme 23 afin d'améliorer les performances pratiques.

La première est que lorsque nous créons une liste ordonnée valide en appelant `acycChecker` à la ligne 2, nous sauvegardons également le support de chaque variable pour cette liste ordonnée dans un vecteur séparé. Ensuite, à la ligne 12, avant de vérifier s'il existe une valeur de l'ensemble parent $S \in D(w)$ qui est un sous-ensemble du préfixe actuel, nous voyons d'abord si le support actuel de w est cohérent. De cette façon, nous évitons parfois de parcourir l'ensemble du domaine d'une variable w.

Example 3.7 (Sauter la traversée du domaine)**.** *Considérons le problème BNSL avec des domaines comme indiqué dans le tableau 3.1. Soit* $support$ *un vecteur qui est initialement vide. Soit* $support_v$ *la valeur* $S \in D(v)$ *qui est supportée par* O*.*

Utilisation des supports enregistrés dans `acycChecker`**.** *Lors de l'exécution de* `acycChecker` *appelée à la ligne 2, chaque fois que nous ajoutons une nouvelle variable* v *à* O*, nous sauvegardons également la valeur support* S *qui nous permet d'effectuer cet ajout (ligne 11 dans l'Algorithme 20). À partir de l'exemple 3.3, nous savons que* `acycChecker` *renvoie la liste ordonnée* $O = \{v_2, v_0, v_1, v_3, v_4\}$ *pour ce problème. Le vecteur de support correspondant est* $\{\{v_2\}, \{v_2\}, \{\}, \{v_1, v_2\}, \{v_0\}\}$*.*

Supposons maintenant que nous exécutons le propagateur GAC dans l'algorithme 23 et que nous itérons sur la variable v_0*. Elle se trouve à la position* $i = 1$ *dans* O *et actuellement nous avons* $prefix = \{v_2\}$*. Nous essayons de voir s'il existe une*

variable w dans $O \setminus (prefix \cup \{v_0\})$ que nous pouvons insérer avant v_0. Nous regardons la variable v_1 et au lieu de parcourir toutes les valeurs de son domaine, nous vérifions d'abord si $support_{v_1} = \{v_2\}$ est un sous-ensemble du préfixe actuel. Nous avons de la chance et c'est le cas, nous pouvons donc immédiatement l'ajouter au préfixe et passer à l'itération suivante.

Algorithm 24: Détection des variables de queue.

```
1 tailVars ← ∅
2 tailVarsSupport ← ∅
3 foreach i ∈ {(n − 1), . . . , 0} do
4     v ← O_i
5     if v ⊆ tailVarsSupport then
6         break
7     tailVars ← tailVars ∪ {v}
8     tailVarsSupport ← tailVarsSupport ∪ support_v
```

La deuxième amélioration nous permet de sauter la vérification des domaines des variables qui peuvent manifestement être poussées à la fin de la liste ordonnée. Elle est basée sur l'observation suivante :

Observation 3.8. *Si une variable v est placée à la dernière position d'une liste ordonnée valide O, c'est-à-dire $O_{n-1} = v$, alors toutes les valeurs S dans $D(v)$ sont supportées.*

Sur la base de cette observation, nous n'avons pas besoin de sonder les valeurs de la variable en position $n - 1$. De plus, nous pouvons essayer de voir s'il existe d'autres variables qui peuvent être échangées avec elle. Pour ce faire, nous remontons de O_{n-1} une à une. Nous ajoutons O_i à l'ensemble des variables *tail* si elle n'apparaît pas dans l'ensemble $support_{n-1} \cup support_{n-2} \cdots \cup support_{i+1}$. Dès que nous rencontrons la première variable apparaissant dans l'union des valeurs supportées, nous terminons. Ensuite, pendant la propagation, nous sautons les variables ajoutées à l'ensemble des variables de queue, puisque nous savons que dans leurs domaines il n'y a pas de valeur à élaguer. La détection des variables de queue est présentée dans l'Algorithme 24. Notez que la détection de cet ensemble n'est pas parfaite. Il existe peut-être d'autres supports qui donneraient un plus grand ensemble de variables de queue, mais cela ne pose pas de problème car il ne s'agit que d'une heuristique.

Example 3.9 (Variables de queue)**.** *À partir de l'exemple 3.3, nous savons que* `acycChecker` *à la ligne 2 de l'algorithme 23 renvoie $O = \{v_2, v_0, v_1, v_3, v_4\}$ et enregistre un vecteur de support $\{\{v_2\}, \{v_2\}, \{\}, \{v_1, v_2\}, \{v_0\}\}$.*

Nous initialisons les ensembles $tailVars$ et $tailVarsSupport$ comme l'ensemble vide. Nous ajoutons $O_4 = v_4$ à l'ensemble $tailVars$ et son support $\{v_0\}$ à l'ensemble $tailVarsSupport$. Ensuite, nous examinons $O_3 = v_3$. Il n'apparaît pas

dans $tailVarsSupport$, nous l'ajoutons donc à $tailVars$ et son support $\{v_1, v_2\}$ à $tailVarsSupport$. À l'étape suivante, nous examinons $O_2 = v_1$. v_1 apparaît dans $tailVarsSupport = \{v_0, v_1, v_2\}$, donc nous terminons et retournons l'ensemble $\{v_3, v_4\}$ comme dernières variables.

Pendant la boucle for qui commence à la ligne 5 dans l'Algorithme 23, nous sautons les variables apparaissant dans $tailVars$.

3.6 Résultats Expérimentaux

3.6.1 Benchmarks et Paramètres

Les jeux de données utilisés dans ce chapitre, ainsi que dans le chapitre 4 et le chapitre 5, proviennent du dépôt d'apprentissage machine de l'UCI[5], du dépôt de réseaux bayésiens[6], et du paquet d'apprentissage et d'inférence de réseaux bayésiens[7]. Nous avons 51 jeux de données moyens avec $|V| < 64$, 18 grands jeux de données avec $64 \leq |V| < 128$, et 23 très grands jeux de données avec $|V| \geq 128$. Pour l'instant, nous omettons les 23 très grands jeux de données jusqu'au prochain chapitre.

Les scores locaux ont été calculés à partir des jeux de données à l'aide du code de B. Malone[8]. Les scores BDeu et BIC ont été utilisés pour les jeux de données moyens (moins de 64 variables) et uniquement le score BIC pour les grands jeux de données (plus de 64 variables). Le nombre maximal de parents a été limité à 5 pour les grands jeux de données (sauf pour `accidents.test` avec un maximum de 8), une valeur élevée qui permet même d'apprendre des structures complexes [Scanagatta *et al.* 2015]. Par exemple, `jester.test` a 100 variables aléatoires, une taille d'échantillon de 4 $116 et des valeurs d'ensemble parentales de 770950$. Pour les jeux de données moyens, aucune restriction n'a été appliquée sauf pour certains scores BDeu (limiter les ensembles à 6 ou 8 pour terminer le calcul des scores locaux en 24 heures de temps CPU [Lee & van Beek 2017a]).

Pour les expériences, nous avons modifié la source C++ de CPBayes v1.1 juste pour nous permettre de l'exécuter avec des jeux de données ayant plus de 64 variables. Pour nos propres méthodes, nous prenons cette version de CPBayes comme base, y intégrons notre propagateur GAC, et appelons le solveur résultant ELSA ^{gac}. Dans le chapitre 4 et le chapitre 5, nous intégrerons séquentiellement notre mécanisme de borne inférieure et nos domaines basés sur les arbres de décision binaires à ELSA ^{gac}, jusqu'à obtenir ELSA que nous avons rendu public.

Tous les calculs ont été effectués sur un seul cœur d'Intel Xeon E5-2680 v3 à 2,50 GHz et 256 Go de RAM avec une limite de temps CPU d'une heure pour les 51 jeux de données moyens, ainsi que 3 des grands jeux de données : `kdd.ts`, `kdd.test`, et `kdd.valid`. Pour les 15 autres grands jeux de données, nous avions

[5] `http://archive.ics.uci.edu/ml`
[6] `http://www.bnlearn.com/bnrepository`
[7] `https://ipg.idsia.ch/software.php?id=132`
[8] `http://urlearning.org`

une limite de temps CPU de 10 heures. Pour la phase de prétraitement, nous avons utilisé deux paramètres différents en fonction de la taille du problème $n = |V|$: $l_{min} = 20, l_{max} = 26, r_{min} = 50, r_{max} = 500$ si $n \leq 64$, sinon $l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$, où l_{min}, l_{max} sont les tailles des limites inférieures des partitions et r_{min}, r_{max} sont le nombre de recommencements de la recherche locale. Tous ces paramètres s'appliquent également aux expériences réalisées pour les chapitres 4 et 5.

3.6.2 Evaluation

Pour l'évaluation, nous comparons CPBayes et ELSA ^{gac} en termes de temps de recherche total en secondes, et de nombre total de nœuds de recherche. Dans le tableau 3.2, nous présentons ces mesures de CPBayes et ELSA ^{gac} pour résoudre chaque jeu de données de manière optimale. En termes de temps de recherche, nous ne constatons ni une nette amélioration ni une baisse pour les plus petits jeux de données, cependant, pour ELSA ^{gac}, il s'améliore de 48% pour `bnetflix.ts`, 43% pour `bnetflix.test` et 35% pour `bnetflix.valid`, par rapport à CPBayes. L'observation la plus frappante est la réduction constante du nombre de nœuds de recherche, même si la réduction *rate* n'est pas aussi élevée que celle des temps de recherche. Nous explorons 21% moins de nœuds pour `bnetflix.ts`, 16% pour `bnetflix.test` et 21% pour `bnetflix.valid`, par rapport à CPBayes. Cela indique que le temps moyen que nous passons à un nœud de recherche augmente avec la GAC, mais la surcharge de la GAC est clairement rentable lorsque les jeux de données deviennent plus grands.

3.7 Conclusion

Nous avons présenté un nouveau propagateur GAC pour la contrainte d'acyclicité globale pour BNSL qui s'exécute en temps $O(n^3 d)$, ainsi que plusieurs idées améliorant ses performances pratiques. Bien que nous explorions un plus petit nombre moyen de nœuds par seconde, la forte inférence fournie par GAC conduit sans aucun doute à des décisions plus efficaces pendant la recherche, réduisant la taille globale de l'arbre de recherche. Cette réduction globale annule les efforts supplémentaires entraînés par GAC pour les petits jeux de données, et l'emporte pour les plus grands. Un algorithme à faible complexité fournissant des bornes inférieures serrées peut nous aider à explorer une partie encore plus petite de l'arbre de recherche, et multiplier le gain que nous obtenons de GAC. Dans le prochain chapitre, nous nous concentrerons sur cet aspect.

Jeu de données	$\|V\|$	$\sum \|d_v\|$	CPBayes time	CPBayes nodes	ELSA ^{gac} time	ELSA ^{gac} nodes
mildew1000_BIC	35	126	0	0	0	0
lympho_BIC	19	143	0	0	0	0
water1000_BIC	32	159	0	1974	0	**1941**
mildew1000_BDe	35	166	0.2	16127	**0.1**	**9267**
hailfinder100_BIC	56	167	0	0	0	0
tumour_BIC	18	219	0	0	0	0
barley1000_BIC	48	244	**1.2**	99485	1.3	**99403**
shuttle_BIC	10	264	0	0	0	0
hepatitis_BIC	20	266	0	60	0	**30**
tumour_BDe	18	274	0	0	0	0
lung-cancer_BIC	57	292	**2.7**	97071	2.8	**95654**
lympho_BDe	19	345	0	163	0	**72**
hailfinder500_BIC	56	418	2.6	128134	**2.5**	**115682**
carpo100_BIC	60	423	27.8	1262025	**27.6**	**1234813**
horse_BDe	28	490	**0.7**	65969	0.8	**64879**
horse_BIC	28	490	0	758	0	**581**
hepatitis_BDe	20	501	0	87	0	**49**
insurance1000_BIC	27	506	0	168	0	**94**
adult_BIC	15	547	0	19	0	**18**
zoo_BIC	17	554	0	45	0	**16**
spectf_BIC	45	610	3.5	145928	**3.3**	**143400**
sponge_BIC	45	618	**3.2**	178474	3.7	**177891**
flag_BIC	29	741	0.6	40155	**0.5**	**37607**
vehicle_BIC	19	763	0	78	0	**28**
adult_BDe	15	768	0	43	0	**35**
insurance1000_BDe	27	792	0	211	0	**129**
shuttle_BDe	10	812	0	0	0	0
bands_BIC	39	892	0.4	16083	0.4	**15856**
alarm1000_BIC	37	1002	**145.7**	11497094	147.8	**11096565**
segment_BIC	20	1053	0	0	0	0
flag_BDe	29	1324	**14.0**	877874	14.7	**852999**
voting_BIC	17	1848	0	46	0	**39**
voting_BDe	17	1940	0	47	0	**35**
autos_BIC	26	2391	0	123	0	**86**
zoo_BDe	17	2855	0	84	0	**50**
vehicle_BDe	19	3121	0	321	0	**176**
letter_BIC	17	4443	0	83	0	**75**
soybean_BIC	36	5926	1.5	22394	1.5	**21775**
msnbc.ts	17	6298	0	73	0	**43**
segment_BDe	20	6491	0	90	0	**71**
mushroom_BIC	23	13025	0	59	0	**40**
wdbc_BIC	31	14613	**337.5**	5931690	337.8	**5913953**
msnbc.test	17	16594	0	94	0	**51**
letter_BDe	17	18841	0	95	0	**55**
msnbc.valid	17	20673	0	0	0	0
nltcs.ts	16	22156	0	0	0	0
autos_BDe	26	25238	0.1	563	0.1	**340**
kdd.ts	64	43584	†	-	†	-
nltcs.valid	16	47097	0	39	0	**28**
nltcs.test	16	48303	0	58	0	**32**
steel_BIC	28	93026	931.5	4805144	**874.6**	**4798161**
kdd.test	64	152873	†	-	†	-
kdd.valid	64	197546	†	-	†	-
mushroom_BDe	23	438185	5.7	183	**5.6**	**118**
plants.ts	69	164640	†	-	†	-
baudio.ts	69	371117	†	-	†	-
bnetflix.ts	100	446406	456.2	424503	**236.3**	**331636**
plants.test	111	520148	†	-	†	-
jester.ts	100	531961	†	-	†	-
accidents.ts	100	568160	†	-	†	-
plants.valid	111	684141	†	-	†	-
jester.test	100	770950	†	-	†	-
baudio.test	100	1016403	†	-	†	-
bnetflix.test	100	1103968	2475.1	1035226	**1402.0**	**862913**
baudio.valid	69	1235928	†	-	†	-
bnetflix.valid	111	1325818	146.8	48792	**94.4**	**38227**
accidents.test	100	1425966	†	-	†	-
jester.valid	100	1463335	†	-	†	-
accidents.valid	100	1617862	†	-	†	-

Table 3.2: Comparaison de CPBayes et ELSA ^{gac} en termes de temps de recherche total en secondes, et de nombre total de noeuds de recherche. La limite de temps pour les jeux de données au-dessus de la ligne est de 1h, pour le reste 10h. Les jeux de données sont triés par taille totale croissante du domaine.

CHAPTER 4

Classe Polynomiale d'Inégalités de Clusters Violés

Contents

4.1 Introduction

Dans ce chapitre, nous dérivons une borne inférieure qui est moins coûteuse en termes de calcul que celle calculée par GOBNILP. L'un des problèmes qui entravent les performances de CPBayes est qu'il calcule des bornes inférieures relativement pauvres à des niveaux plus profonds de l'arbre de recherche. Intuitivement, au fur et à mesure que les domaines de variables de l'ensemble parent se réduisent en supprimant les valeurs qui sont incompatibles avec l'ordre actuel, le calcul de la borne inférieure rejette davantage d'informations sur l'état actuel du problème, ce qui le rend plus faible. Nous abordons ce problème et obtenons ainsi un meilleur compromis entre la force et la rapidité de l'inférence en adaptant l'approche branch-and-cut de GOBNILP. Cependant, au lieu de trouver toutes les inégalités de cluster violées qui peuvent améliorer la borne inférieure LP, nous n'en identifions qu'un sous-ensemble.

Nous donnons dans la Section 4.3 un algorithme en temps polynomial qui découvre une classe de coupes de clusters qui améliorent de manière prouvable la relaxation linéaire. Dans la section 4.4, nous donnons un algorithme glouton pour résoudre la relaxation linéaire, inspiré par des algorithmes similaires pour MaxSAT et WCSPs, en particulier l'algorithme VAC. Nous incorporons tous ces algorithmes

dans ELSA ^{gac} et appelons le solveur résultant ELSA ^{cluster}. Enfin, dans la section 4.5, nous démontrons que ELSA ^{cluster} présente une performance significativement améliorée, tant au niveau de la taille de l'arbre de recherche exploré que du temps d'exécution.

4.2 Clusters Violés

Comme nous l'avons vu au chapitre 2, nous utilisons le modèle CP de CPBayes pour formuler le problème BNSL. Nous avons un ensemble V de *variables d'ensembles parents* correspondant aux variables aléatoires, dont chaque v a un ensemble d'ensembles parents candidats, $PS(v)$. Chacun de ces ensembles parents candidats $S \in PS(v)$ a un *score* associé $\sigma^v(S)$, c'est-à-dire un coût. Nous essayons de trouver le réseau optimal, c'est-à-dire acyclique et de moindre coût, dans lequel nous attribuons à chaque variable $v \in V$ un ensemble parent $S \in PS(v)$. Cet objectif est le même que celui du modèle ILP (4.1) utilisé par GOBNILP:

$$\min \sum_{v \in V, S \in PS(v)} \sigma^v(S) x_{vS} \tag{4.1a}$$

$$s.t. \sum_{S \in PS(v)} x_{vS} = 1 \qquad \forall v \in V \tag{4.1b}$$

$$\sum_{v \in W, S \in PS^{-W}(v)} x_{vS} \geq 1 \qquad \forall W \subseteq V \tag{4.1c}$$

$$x_{vS} \in \{0,1\} \qquad \forall v \in V, S \in PS(v) \tag{4.1d}$$

Rappelons maintenant le théorème suivant : [Jaakkola *et al.* 2010], sur lequel les contraintes (4.1c) pour chaque $W \subseteq V$, désigné par $cons(W)$, et `acycChecker` (réapparaissant dans l'Algorithme 25 par souci d'auto-confinement) sont basées :

Theorem 4.1. *Soit G un graphe dirigé sur des sommets V et soit $parents(v)$ les parents du sommet v dans le graphe. G est acyclique si et seulement si pour chaque sous-ensemble non vide, c'est-à-dire cluster $W \subseteq V$ il existe au moins un sommet $v \in W$ avec $parents(v) \cap W = \{\}$.*

W est un *cluster violé* si l'ensemble des parents de chaque sommet $v \cap dansW$ intersecte W. D'autre part, rappelez-vous que `acycChecker` retourne un ordre de toutes les variables si l'ensemble actuel des domaines des variables de l'ensemble parent peut produire un graphe acyclique, ou un ordre partiellement complété si la contrainte est insatisfaisable, indiquant qu'il existe des clusters violés.

4.3 Détection Restreinte de Clusters

Considérons la relaxation linéaire de l'ILP (4.1), restreinte à un sous-ensemble $\mathcal{W}$ de toutes les inégalités de cluster valides, c'est-à-dire avec la contrainte (4.1d)

Algorithm 25: Vérificateur d'acyclicité.

```
1  Procedure acycChecker(prefix, V, D):
2      O ← prefix
3      support ← {}
4      changes ← true
5      i ← size(prefix)
6      while changes do
7          changes ← false
8          foreach v ∈ V \ O do
9              if ∃S ∈ D(v) s.t. S ⊆ O then
10                 O_i ← v
11                 support_v ← S
12                 changes ← true
13                 i ← (i + 1)
14     return O
```

remplacée par $0 \leq x_{vS} \leq 1 \forall v \in V, S \in PS(v)$ et avec la contrainte (4.1c) restreinte uniquement aux clusters dans $\mathcal{W}$. Nous désignons cette LP par $LP_{\mathcal{W}}$. Nous exploitons la propriété suivante de cette LP.

Theorem 4.2. *Soit $\hat{y}$ une solution faisable duale de $LP_{\mathcal{W}}$ avec l'objectif dual o. De plus, laissons $varsof(W)$ désigner les variables 0/1 impliquées dans $cons(W)$. Alors, si W est un cluster tel que $W \notin \mathcal{W}$ et que le coût réduit rc de toutes les variables $varsof(W)$ est supérieur à 0, il existe une solution duale faisable $\hat{y}$ de $LP_{\mathcal{W}\cup\{W\}}$ avec un objectif dual $o' \geq o + minrc(W)$ où*

$$minrc(W) = \min_{x_{vS} varsof(W)} rc_{\hat{y}}(x_{vS}) \tag{4.2}$$

Proof. La seule différence entre $LP_{\mathcal{W}}$ et $LP_{\mathcal{W}\cup\{W\}}$ est la contrainte supplémentaire $cons(W)$ dans le primal et la variable duale correspondante y_W. Dans le dual, y_W n'apparaît que dans les contraintes duales des variables $varsof(W)$ et dans l'objectif, toujours avec le coefficient 1. Sous la solution duale réalisable $\hat{y} \cup \{y_W = 0\}$, ces contraintes ont un relâchement d'au moins $minrc(W)$, par la définition du coût réduit. Par conséquent, nous pouvons fixer $\hat{y} = \hat{y} \cup \{y_W = minrc(W)\}$, qui reste faisable et a pour objectif $o' = o + minrc(W)$, comme requis. □

Le théorème 4.2 donne une classe de coupes de clusters, que nous appelons RC-clusters, pour les clusters à coût réduit, garantie pour améliorer la borne inférieure. Fait important, cela ne nécessite qu'une solution faisable, peut-être sous-optimale.

Ce théorème est fortement lié à l'algorithme VAC pour les WCSPs [Cooper *et al.* 2010], puisqu'il utilise également l'idée d'effectuer la propagation sur le sous-ensemble de domaines qui ont un coût réduit 0. Lorsque nous avons parlé d'VAC au chapitre 2, nous avons dit qu'en demandant à

"*La CSP $Bool(P)$ a-t-elle une solution ?*"

est équivalent à demander

"TexteLe WCSP P a-t-il une solution avec un coût total de $c_\varnothing$?"

Si la réponse à la première question est "non", alors il existe une séquence d'EPTs qui augmentent $c_\varnothing$ lorsqu'ils sont appliqués à P. De même, ici, en demandant

"*Il existe un réseau acyclique où chaque variable $v \in V$ se voit attribuer un ensemble parent $S \in PS(v)$ tel que $rc_{\hat{y}}(x_{vS}) = 0$?*"

est équivalent à demander

"TexteIl existe un réseau acyclique dont le coût total est égal à la borne inférieure courante?"

De nouveau, si la réponse à la première question est "non", alors la borne inférieure actuelle peut être améliorée, comme nous l'avons vu dans la preuve du théorème 4.2. Contrairement à l'algorithme VAC, notre méthode est plus légère, car elle n'effectue que la propagation sur la contrainte d'acyclicité, mais elle peut donner des bornes plus mauvaises. Le mécanisme de mise à jour des limites dans la preuve du théorème 4.2 est également plus simple que VAC et ressemble davantage à la "phase de noyau disjoint" dans les solveurs MaxSAT guidés par noyau [Morgado *et al.* 2013].

Variable	Domaine	Coût
v_0	$\{v_2\}$	0
v_1	$\{v_2, v_4\}$	0
	$\{\}$	6
v_2	$\{v_1, v_3\}$	0
	$\{\}$	10
v_3	$\{v_0\}$	0
	$\{\}$	5
v_4	$\{v_2, v_3\}$	0
	$\{v_3\}$	1
	$\{v_2\}$	2
	$\{\}$	3

Table 4.1: Le problème BNSL utilisé comme exemple courant dans ce chapitre.

Example 4.3 (Exemple courant)**.** *Considérez un problème BNSL avec des domaines comme indiqué dans le tableau 4.1 et laissez $\mathcal{W} = \emptyset$. Alors, $\hat{y} = 0$ laisse le coût*

réduit de chaque variable à exactement son coefficient objectif primaire. Le $\hat{x}$ correspondant attribue 1 aux variables dont le coût réduit est de 0 et 0 à tout le reste. Ce sont toutes deux des solutions optimales, avec un coût 0 et $\hat{x}$ est intégral, donc c'est aussi une solution de l'ILP correspondante. Cependant, ce n'est pas une solution de la BNSL, car elle contient plusieurs cycles, dont $C = \{v_0, v_2, v_3\}$. L'inégalité de cluster $cons(W)$ est violée dans le primal et nous permet d'augmenter la borne duale.

Nous considérons le problème de la découverte de RC-clusters dans le modèle CP de CPBayes. Tout d'abord, nous introduisons la notation $LP_{\mathcal{W}}(D)$ qui est $LP_{\mathcal{W}}$ avec la contrainte supplémentaire $x_{vS} = 0$ pour chaque $S \notin D(v)$. Inversement, $D^{rc}_{\mathcal{W},\hat{y}}$ est l'ensemble des domaines moins les valeurs dont la variable correspondante dans $LP_{\mathcal{W}}(D)$ a un coût réduit non nul sous $\hat{y}$, c'est-à-dire, $D^{rc}_{\mathcal{W},\hat{y}} = D'$ où $D'(v) = \{S \mid S \in D(v) \wedge rc_{\hat{y}}(x_{vS}) = 0\}$ avec $rc(x_{vS})$ étant le coût réduit de x_{vS} dans $LP_{\mathcal{W}}(D)$ sous $\hat{y}$. Avec cette notation, pour les valeurs $S \notin D(v)$, c'est-à-dire que S est élagué dans $D(v)$, $x_{vS} = 1$ est infaisable dans $LP_{\mathcal{W}}(D)$, donc effectivement $rc_{\hat{y}}(x_{vS}) = \infty$.

Theorem 4.4. *Étant donné une collection de clusters $\mathcal{W}$, un ensemble de domaines D et $\hat{y}$, une solution duale faisable de $LP_{\mathcal{W}}(D)$, il existe un RC-cluster $W \notin \mathcal{W}$ si et seulement si $D^{rc}_{\mathcal{W},\hat{y}}$ n'admet pas une affectation acyclique.*

Proof. $\Rightarrow$ Soit W un tel cluster. Puisque pour tous les $x_{vS} \in varsof(W)$, aucun de ceux-ci n'est dans $D^{rc}_{\mathcal{W},\hat{y}}$, donc $cons(W)$ est violé et donc il n'y a pas d'affectation acyclique.

$\Leftarrow$ Considérons une fois encore `acycChecker`, dans l'algorithme 25. Lorsqu'il ne parvient pas à trouver un témoin d'acyclicité, il a atteint un point où $O \subsetneq V$ et pour les variables restantes $C = V \setminus O$, tous les ensembles parents autorisés coupent W. Donc, si `acycChecker` est appelé avec $D^{rc}_{\mathcal{W},\hat{y}}$, toutes les valeurs dans $varsof(W)$ ont un coût réduit supérieur à 0, donc W est un RC-cluster. □

Le théorème 4.4 montre que détecter l'insatisfaisabilité de $D^{rc}_{\mathcal{W},\hat{y}}$ est suffisant pour trouver un RC-cluster. Sa preuve donne également un moyen d'extraire un tel cluster de `acycChecker`.

L'algorithme 26 montre comment le théorème 4.2 et le théorème 4.4 peuvent être utilisés pour calculer une borne inférieure. On lui donne l'ensemble actuel de domaines D et le pool actuel de clusters $\mathcal{W}$ en entrée. Il résout d'abord le dual de $LP_{\mathcal{W}}(D)$, potentiellement de manière sous-optimale. Ensuite, il utilise `acycChecker` de manière itérative pour déterminer s'il existe un RC-cluster W sous la solution duale actuelle $\hat{y}$. Si ce cluster est vide, il n'y a plus de RC-clusters, et il se termine et retourne une borne inférieure égale au coût de $\hat{y}$ sous $LP_{\mathcal{W}}(D)$ et un pool de clusters mis à jour. Sinon, il minimise W (voir la section 4.3.1), l'ajoute au pool de clusters et résout le LP mis à jour. Pour ce faire, il appelle $dualImprove$, qui résout $LP_{\mathcal{W}}(D)$ en exploitant le fait que seule l'inégalité de cluster $cons(W)$ a été ajoutée.

Algorithm 26: Calcul de la borne inférieure avec RC-clusters

```
1 Procedure lowerBoundRC(V, D, 𝒲):
2     ŷ ← dualSolve(LP_𝒲(D))
3     while true do
4         W ← V \ acycChecker(V, D^rc_{𝒲,ŷ})
5         if W = ∅ then
6             return ⟨cost(ŷ), 𝒲⟩
7         W ← minimise(W)
8         𝒲 ← 𝒲 ∪ {W}
9         ŷ ← dualImprove(ŷ, LP_𝒲(D), W)
```

Example 4.5. *En poursuivant notre exemple, considérons le comportement de* `acycChecker` *à la ligne 4 de l'Algorithme 26 avec les domaines* $D^{rc}_{\emptyset,\hat{y}}$ *après la solution duale initiale* $\hat{y} = 0$*. Puisque l'ensemble vide a un coût réduit non nul pour toutes les variables,* `acycChecker` *ne parvient pas à placer une variable dans l'ordre et renvoie* $ordre = \{\}$ *comme cluster, donc* $C = V$*. Nous reportons la discussion de la minimisation à la section 4.3.1, sinon pour observer que* W *peut être minimisé à* $C_1 = \{v_1, v_2\}$ *à la ligne 7. Au LP primal, nous ajoutons* $cons(C_1)$ *qui implique les variables* $x_{v_1\{\}}$ *et* $x_{v_2\{\}}$ *dont les coûts réduits sont respectivement de 6 et 10. Nous fixons la variable duale de* C_1 *à 6 dans la nouvelle solution duale* $\hat{y}_1$*. Les coûts réduits de* $x_{1\{\}}$ *et de* $x_{v_2\{\}}$ *sont diminués de 6 et, fait important,* $rc_{\hat{y}_1}(x_{v_1\{\}}) = 0$*. Dans l'itération suivante de* `lowerBoundRC`*,* `acycChecker` *est invoqué sur* $D^{rc}_{\{C_1\},\hat{y}_1}$ *et renvoie le cluster* $\{v_0, v_2, v_3, v_4\}$*. Celui-ci est minimisé à* $C_2 = \{v_0, v_2, v_3\}$*. Les ensembles parents dans les domaines de ces variables qui n'intersectent pas* C_2 *sont* $x_{v_2\{\}}$ *et* $x_{v_3\{\}}$*, dont les coûts réduits sont de* $10-6 = 4$ *et* 5*, respectivement. Par conséquent,* $minrc(C_2) = 4$*, nous ajoutons donc* $cons(C_2)$ *à la primale et nous fixons la variable duale de* C_2 *à 4 dans* $\hat{y}_2$*. Cela porte l'objectif dual à 10. Le coût réduit de* $x_{v_2\{\}}$ *est 0, donc dans l'itération suivante,* `acycChecker` *s'exécute sur* $D^{rc}_{\{C_1,C_2\},\hat{y}_2}$ *et réussit avec l'ordre* $\{v_2, v_0, v_3, v_4, v_1\}$*, donc la borne inférieure ne peut pas être améliorée davantage. Il se trouve que c'est également le coût de la structure optimale.*

Theorem 4.6. `lowerBoundRC` *se termine mais n'est pas confluent.*

Proof. Elle se termine car il existe un nombre fini d'inégalités de cluster et chaque itération en génère une. À l'extrême, toutes les inégalités de cluster sont dans $\mathcal{W}$ et le test à la ligne 5 réussit, mettant fin à l'algorithme.

Pour voir qu'il n'est pas confluent, considérons un exemple avec 3 clusters $C_1 = \{v_1, v_2\}, C_2 = \{v_2, v_3\}$ et $C_3 = \{v_3, v_4\}$ et supposons que le coût réduit minimum pour chaque cluster est unitaire et provient de $x_{v_2\{4\}}$ et $x_{v_3\{1\}}$, c'est-à-dire, la première valeur a un coût réduit minimum pour C_1 et C_2 et la seconde pour C_2 et C_3. Ensuite, si la minimisation génère d'abord C_1, le coût réduit de $x_{v_3\{1\}}$ n'est pas affecté par $dualImprove$, il peut donc ensuite découvrir C_3, pour obtenir

une borne inférieure de 2. D'autre part, si la minimisation génère d'abord C_2, les coûts réduits des deux $x_{v_2\{4\}}$ et $x_{v_3\{1\}}$ sont diminués à 0 par $dualImprove$, donc ni C_1 ni C_3 ne sont des RC-clusters sous la nouvelle solution duale et l'algorithme se termine avec une borne inférieure de 1. □

Theorem 4.7. `lowerBoundRC` *s'exécute en* $O(n^2 dmin(UB - LB, \sum_{v \in V} d_v))$ *temps, où LB est la borne inférieure actuelle, et UB est la borne supérieure actuelle, et* d_v *est la taille du domaine de la variable* v*.*

Proof. Le terme $n^2 d$ provient de `acycChecker` qui est appelé à chaque itération de la boucle while. Le nombre d'itérations est limité par le nombre de fois où l'on peut augmenter la borne inférieure. Il existe deux cas extrêmes :

1. Nous faisons le coût réduit d'exactement une valeur $S \in D(v)$ à la fois, et nous avons au plus $\sum_{v \in V} d_v$ de ces valeurs
2. Nous augmentons la borne inférieure de 1 jusqu'à ce que nous atteignions la borne supérieure actuelle

Le nombre d'itérations est limité par le minimum de ces deux. □

4.3.1 Minimisation du Cluster

Il est crucial pour la qualité de la borne inférieure produite par `lowerBoundRC` que les RC-clusters découverts par `acycChecker` soient minimisés, comme le montre l'exemple suivant. Empiriquement, l'omission de la minimisation a rendu la borne inférieure inefficace.

Example 4.8. *Supposons que nous essayons d'utiliser* `lowerBoundRC` *sans minimisation de cluster. Nous utilisons alors le cluster donné par* `acycChecker`*,* $C_1 = \{v_0, v_1, v_2, v_3, v_4\}$*. Nous avons* $minrc(C_1) = 3$*, donné à partir de la valeur de l'ensemble parent vide de toutes les variables. Cela amène le coût réduit de* $x_{v_4\{\}}$ *à 0. Il procède ensuite à la recherche du cluster* $C_2 = \{v_0, v_1, v_2, v_3\}$ *avec* $minrc(C_2) = 2$ *et diminue le coût réduit de* $x_{v_3\{\}}$ *à 0, puis* $C_3 = \{v_0, v_1, v_2\}$ *avec* $minrc(C_3) = 1$*, ce qui amène le coût réduit de* $x_{v_1\{\}}$ *à 0. À ce stade,* `acycChecker` *réussit avec l'ordre* $\{v_4, v_3, v_1, v_2, v_0\}$ *et* `lowerBoundRC` *renvoie une borne inférieure de 6, contre 10 avec la minimisation. L'ordre produit par* `acycChecker` *est également en désaccord avec la structure optimale.*

Par conséquent, lorsque nous obtenons un RC-cluster W à la ligne 4 de l'algorithme 26, nous voulons extraire un RC-cluster minimal (par rapport à l'inclusion de l'ensemble) de W, c'est-à-dire un cluster $C' \subseteq C$, tel que pour tout $\emptyset \subset C'' \subset C'$, C'' n'est pas un cluster.

Les problèmes de minimisation comme celui-ci sont traités avec une instanciation appropriée de QuickXPlain [Junker 2004]. Ces algorithmes trouvent un sous-ensemble minimal de contraintes, et non de variables. Nous pouvons poser ce problème comme un problème de minimisation d'ensemble de contraintes en

Algorithm 27: Trouver un sous-ensemble minimal RC-cluster de W

```
1  Procedure minimiseCluster(V, D, C):
2      N = ∅
3      while C ≠ ∅ do
4          Pick c ∈ C
5          C ← C \ {c}
6          C' ← V \ acycChecker(N ∪ W, D)
7          if C' = ∅ then
8              N ← N ∪ {c}
9          else
10             C ← C' \ N
11     return N
```

traitant implicitement une variable comme la contrainte "cette variable est assignée à une valeur" et en traitant l'acyclicité comme une contrainte dure.

Cependant, la propriété d'être un RC-cluster n'est pas monotone. Par exemple, considérons les variables $\{v_1, v_2, v_3, v_4\}$ et $\hat{y}$ telles que les domaines restreints aux valeurs avec un coût réduit de 0 sont $\{\{v_2\}, \{\{\{v_1\}\}, \{\{\{v_4\}\}, \{\{\{v_3\}\}$, respectivement. Alors $\{v_1, v_2, v_3, v_4\}$, $\{v_1, v_2\}$ et $\{v_3, v_4\}$ sont des RC-clusters. Mais $\{v_1, v_2, v_3\}$ ne l'est pas car l'unique valeur du domaine de v_3 ne coupe pas $\{v_1, v_2, v_3\}$. Nous minimisons plutôt l'ensemble de variables qui n'admet pas de solution acyclique et donc *contenu* un RC-cluster. Un ensemble minimal insatisfaisant qui contient un cluster est un RC-cluster, ce qui nous permet donc d'utiliser un algorithme de minimisation pour les prédicats monotones, tels que les variantes de QuickXPlain. Nous nous concentrons sur RobustXPlain, qui est appelé dans la littérature SAT l'algorithme basé sur la suppression pour minimiser les sous-ensembles insatisfaisants [Marques-Silva & Mencía 2020]. L'idée principale de l'algorithme est de choisir itérativement une variable et de la classer comme apparaissant dans tous les sous-ensembles minimaux de W, auquel cas nous la marquons comme nécessaire, ou non, auquel cas nous la rejetons. Pour détecter si une variable apparaît dans tous les sous-ensembles minimaux insatisfaisants, il suffit de tester si l'omission de cette variable donne un ensemble sans sous-ensembles insatisfaisants, c'est-à-dire sans clusters violés. Ceci est donné en pseudocode dans Algorithme 27. Cela exploite une caractéristique subtile de `acycChecker` telle que décrite dans l'Algorithme 25 : s'il est appelé avec un sous-ensemble de V, il n'essaie pas de placer les variables manquantes dans l'ordre et permet aux ensembles parents d'utiliser ces variables manquantes. Omettre des variables de l'ensemble donné à `acycChecker` revient à omettre la contrainte selon laquelle une valeur doit être attribuée à ces variables. La complexité de `minimiseCluster` est de $O(n^3 d)$, où $n = |V|$ et $d = \max_{v \in V} |D(v)|$.

4.4 Résolution du Cluster par LP

La résolution d'un programme linéaire se fait en temps polynomial, donc en principe, $dualSolve$ peut être implémenté à l'aide de n'importe laquelle des bibliothèques de logiciels commerciaux ou libres disponibles pour cela. Cependant, la résolution de ce LP à l'aide d'un solveur LP général est trop coûteuse dans ce contexte. À titre d'exemple, la résolution du jeu de données `steel_BIC` avec notre solveur modifié a nécessité 25 016 nœuds de recherche et 45 secondes de recherche, et a généré 5869 RC-clusters. Environ 20% du temps de recherche a été consacré à la résolution du LP à l'aide de l'algorithme glouton que nous décrivons dans cette section. CPLEX a mis environ 70 secondes pour résoudre une fois $LP_{\mathcal{W}}$ avec ces inégalités de cluster. Nous pensons que cela est dû au fait que nous avons près de 100000 $x_{v,S}$ variables (voir la colonne 3 du tableau 4.2 pour cet jeu de données.) et que le nombre de variables impliquées dans chaque inégalité de cluster est important. Bien que ce point de données ne soit pas la preuve que la résolution exacte de la LP est trop coûteuse, c'est un indicateur assez fort. Nous n'avons pas non plus exploré les algorithmes en temps quasi linéaire pour résoudre les LP positives : [Allen-Zhu & Orecchia 2015].

Notre algorithme glouton est dérivé du théorème 4.2. Observons d'abord que $LP_{\mathcal{W}}$ avec $\mathcal{W} = \emptyset$, c'est à dire seulement avec les contraintes (4.1b) a une solution duale optimale $\hat{y}_0$ qui assigne la variable duale y_v de $\sum_{S \in PS(v)} x_{vS} = 1$ à $\min_{S \in PS(v)} \sigma^v(S)$. Cela laisse au moins un des $x_{vS}, S \in PS(v)$ avec un coût réduit de 0 pour chaque $v \in V$. $dualSolve$ commence avec $\hat{y}_0$ et itère ensuite sur $\mathcal{W}$. Étant donné $\hat{y}_{i-1}$ et un cluster W, il définit $\hat{y}_i = \hat{y}_{i-1}$ si W n'est pas un RC-cluster. Sinon, il augmente la borne inférieure de $c = minrc(W)$ et fixe $\hat{y}_i = \hat{y}_{i-1} \cup \{y_W = c\}$. Il reste à spécifier l'ordre dans lequel nous parcourons $\mathcal{W}$.

Nous trions les clusters par taille croissante $|C|$, en brisant les liens par coût minimal décroissant de toutes les valeurs de l'ensemble des parents originaux dans $varsof(W)$. Cela favorise la recherche de coupes de clusters non chevauchants avec un coût minimum élevé. Dans la section 4.5, nous donnons des preuves expérimentales que cela calcule de meilleures limites inférieures.

$dualImprove$ peut être implémenté en écartant les informations précédentes et en appelant $dualSolve(LP_{currentclusters}(D))$. Au lieu de cela, il utilise le RC-cluster W pour mettre à jour la solution sans revisiter les clusters précédents.

En termes d'implémentation, nous stockons $varsof(W)$ pour chaque cluster, et non $cons(W)$. Pendant $dualSolve$, nous maintenons les coûts réduits des variables plutôt que la solution duale, sinon le calcul de chaque coût réduit nécessiterait d'itérer sur toutes les inégalités de cluster qui contiennent une variable. Plus précisément, nous maintenons $\Delta_{v,S} = \sigma^v(S) - rc_{\hat{y}}(x_{vS})$. Afin de tester si un cluster W est un RC-cluster, nous devons calculer $minrc(W)$. Pour accélérer cette opération, nous associons à chaque cluster stocké une *paire de supports* (v, S) correspondant au dernier coût minimum trouvé. Si $rc_{\hat{y}}(v, S) = 0$, le cluster n'est pas un RC-cluster et est ignoré. De plus, les domaines de l'ensemble parent sont triés par score croissant $\sigma^v(S)$, donc $S \succ S' \iff \sigma^v(S) > \sigma^v(S')$.

Nous maintenons également le montant maximum de coût transféré à la borne inférieure, $\Delta_v^{max} = \max_{S \in D(v)} \Delta_{v,S}$ pour chaque $v \in V$. Nous arrêtons d'itérer sur $D(v)$ dès que $\sigma^v(S) - \Delta_v^{max}$ est supérieur ou égal au minimum actuel car $\forall S' \succ S, \sigma^v(S') - \Delta_{v,b} \geq \sigma^v(S) - \Delta_v^{max}$. Nous fournissons une analyse empirique détaillée sur la productivité des clusters dans la section 4.5.2.3.

4.5 Résultats Expérimentaux

Pour les expériences de ce chapitre, nous prenons ELSA ^{gac} comme base, intégrons tous les algorithmes présentés ici, et appelons le solveur résultant ELSA ^{cluster}. Nous produisons une autre variante appelée ELSA ^{chrono}, qui est la même que ELSA ^{cluster} à l'exception du fait qu'elle laisse les clusters dans l'ordre chronologique, plutôt que l'ordre heuristique présenté dans la section 4.4. Pour résumer, nous effectuons une analyse comparative entre les solveurs suivants :

- GOBNILP v1.6.3 utilisant SCIP v3.2.1 avec CPLEX v12.8.0
- CPBayes v1.1 compatible avec $n > 64$
- ELSA ^{gac}
- ELSA ^{cluster}
- ELSA ^{chrono}

Nous avons le même ensemble de 51 jeux de données moyens ($|V| < 64$) et 18 grands ($64 \leq |V| < 128$) que dans le chapitre précédent. Pour les jeux de données moyens, ainsi que pour `kdd.ts`, `kdd.test` et `kdd.valid`, nous avions une limite de temps CPU d'une heure, et une limite de temps CPU de 10 heures pour les 15 grands jeux de données restants. En outre, dans la section 4.5.2, nous partageons et étudions en détail les résultats pour les 18 grands jeux de données, et les 23 très grands jeux de données ($|V| > 128$) que nous avons exécutés avec une limite de temps de 90 heures.

4.5.1 Evaluation

Jeu de données	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
carpo100_BIC	60	423	**0.6**	79.8 (27.8)	79.9 (27.6)	52.6 (0.1)	56.9 (0.1)
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	181.1 (147.8)	34.4 (1.0)	39.4 (3.9)
flag_BDe	29	1324	4.36	14.9 (14)	15.7 (14.7)	**1.0 (0.2)**	1.6 (0.7)
wdbc_BIC	31	14613	99.77	390.6 (337.5)	396.1 (337.8)	**56 (2.4)**	63.7 (4.7)
kdd.ts	64	43584	**327.63**	†	†	1452.3 (274.6)	2257.9 (960.0)
steel_BIC	28	93026	†	981.2 (931.5)	924.8 (874.6)	**124.2 (71.8)**	196.6 (139.6)
kdd.test	64	152873	1521.73	†	†	**1594.3 (224.4)**	1863.6 (246.1)
mushroom_BDe	23	438185	†	131.2 (5.7)	**130.8 (5.6)**	182.6 (58.9)	225.3 (66.6)
bnetflix.ts	100	446406	†	673.4 (456.2)	**453.2 (236.3)**	2103.1 (1900.9)	2482.1 (2727.0)
plants.test	111	520148	†	†	†	**28049.6 (26312.9)**	†
jester.ts	100	531961	†	†	†	**21550.5 (21003.7)**	29868.0 (29251.5)
accidents.ts	100	568160	**1273.96**	†	†	2302.2 (930)	2887.7 (1280.2)
plants.valid	111	684141	†	†	†	**17801.6 (14080.2)**	†
jester.test	100	770950	†	†	†	**30186.8 (29455)**	†
bnetflix.test	100	1103968	†	2725.6 (2475.1)	**1651.7 (1402)**	10333.1 (10096.5)	10079.1 (9799.5)
bnetflix.valid	111	1325818	†	511.7 (146.8)	**457.7 (94.4)**	10871.7 (10527.7)	10749.1 (10363.3)
accidents.test	100	1425966	4975.64	†	†	**3641.7 (680.7)**	4290.7 (866.0)

Table 4.2: Comparaison de ELSA ^{cluster} contre GOBNILP, CPBayes, ELSA ^{gac}, et ELSA ^{chrono}, en termes de temps total d'exécution (et de recherche) en secondes. La limite de temps pour les jeux de données blue à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. Pour CPBayes ainsi que pour toutes les variantes de ELSA, nous indiquons entre parenthèses le temps passé en recherche, une fois le prétraitement terminé. Le signe † indique un dépassement de délai.

Nous comparons le temps d'exécution pour résoudre chaque jeu de données à l'optimalité avec GOBNILP, CPBayes, ELSA ^{gac}, ELSA ^{cluster}, et ELSA ^{chrono} dans le Tableau 4.2. Le temps de prétraitement pour CPBayes ainsi que toutes les variantes de ELSA est à peu près le même ; c'est pourquoi pour eux, nous reportons entre parenthèses et comparons le temps passé pendant la recherche uniquement.

Dans le tableau 4.2, nous excluons les jeux de données qui ont été résolus dans le temps imparti par GOBNILP et qui ont un temps de recherche inférieur à 10 secondes pour CPBayes et toutes les variantes de ELSA. Nous excluons également 8 jeux de données qui n'ont été résolus jusqu'à l'optimalité par aucune méthode. Cela nous laisse 17 résultats "plus intéressants" à analyser ici sur un total de 69. Les résultats dans le même format pour les 69 jeux de données peuvent être consultés dans le Tableau 4.5 à la fin de ce chapitre.

Comparaison avec GOBNILP. Il a déjà été prouvé que le CPBayes était compétitif par rapport au GOBNILP. Nos résultats dans le tableau 4.2 le confirment tout en montrant qu'aucun n'est clairement meilleur. En ce qui concerne notre solveur ELSA ^{cluster}, tous les jeux de données résolus dans le temps imparti par GOBNILP sont résolus, contrairement à CPBayes. En plus de cela, ELSA ^{cluster} résout 9 jeux de données de plus de manière optimale que GOBNILP.

Comparaison avec ELSA ^{gac}. ELSA ^{cluster} prend plusieurs ordres de grandeur de temps de recherche en moins que ELSA ^{gac} pour résoudre de manière optimale la plupart des jeux de données, la seule exception étant `bnetflix.valid`. Pour `bnetflix.valid`, le nombre de nœuds de recherche est passé de 38227 à 6834 ($\downarrow 82\%$) ; pour `bnetflix.valid`, le temps de recherche est passé de 457,7 à 10 871,7 ($\uparrow 2275\%$). Nous devons noter que la taille moyenne des domaines pour cet jeu de données est de près de 12 000. Par conséquent, le nombre total d'itérations de valeur de domaine effectuées au cours de `lowerBoundRC` est également particulièrement important : 244 154 531. Pour en revenir au tableau général, ELSA ^{cluster} a prouvé l'optimalité pour 8 jeux de données de plus que ELSA ^{gac} dans le temps imparti.

Comparaison avec ELSA ^{chrono}. Nous constatons que l'heuristique d'ordonnancement des clusters améliore de manière significative les limites calculées par notre algorithme dual LP glouton. Comparé au fait de ne pas ordonner les clusters, nous constatons une amélioration du temps de recherche pour tous les jeux de données (sauf pour `bnetflix.test` et `bnetflix.valid`, et la diminution est négligeable). En plus de cela, 3 jeux de données supplémentaires ont été résolus de manière optimale.

4.5.2 Autre Analyse sur les Grands et Très Grands Jeux de Données

Nous avons effectué une exécution plus longue avec une limite de 90 heures de temps CPU pour 18 grands et 23 très grands jeux de données sur un cluster d'Intel Xeon E5-2683 v4 à 2,10 GHz et 128 Go de RAM. Nous avons utilisé les paramètres suivants :

$l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$ pour les tailles min/max des limites inférieures des partitions l_{min}, l_{max}, et le nombre min/max de redémarrages de la recherche locale r_{min}, r_{max}. *E.g.,* nous avons exécuté `./elsa -B 20 -r 15 -R 30 -t 324000 kdd.test.jkl` pour résoudre `kdd.test` dans le temps imparti. Nous avons comparé ELSA ^{cluster} avec GOBNILP avec les paramètres par défaut, sauf l'écart d'optimalité fixé à zéro, et CPBayes.

4.5.2.1 Qualité des Bornes Inférieures et Temps par Nœud

Nous avons mesuré la qualité de la borne inférieure initiale lb trouvée au nœud racine de l'arbre de recherche comme la distance relative $\frac{o-lb}{o}$ par rapport à la meilleure solution o trouvée par n'importe quelle méthode dans la limite des 90 heures de temps CPU. Les résultats sont donnés dans le tableau 4.3.

GOBNILP a généralement produit les meilleures limites inférieures. CPBayes a obtenu les pires et ELSA ^{cluster} a bien progressé vers GOBNILP mais est resté en dessous sauf quelques exceptions (GOBNILP s'est arrêté au noeud racine à cause de la mémoire épuisée pour `accidents.valid`, `dna`, `kosarek.valid`, `msweb.test`, et `msweb.valid`).

Dans la Figure 4.1, nous montrons le temps CPU moyen par noeud de recherche lorsque les informations étaient disponibles pour un sous-ensemble des 41 grands et très grands jeux de données. Nous trions les données en augmentant le temps CPU indépendamment pour chaque solveur. Ici, GOBNILP est plus lent de plusieurs ordres de grandeur que CPBayes et notre approche. ELSA ^{cluster} était généralement plus lent que CPBayes, jusqu'à $457,4$ fois pour `bnetflix.test`, sauf sur quelques cas, *e.g.,* ELSA ^{cluster} était $9,7$ plus rapide que CPBayes sur `pumsb_star.test`.

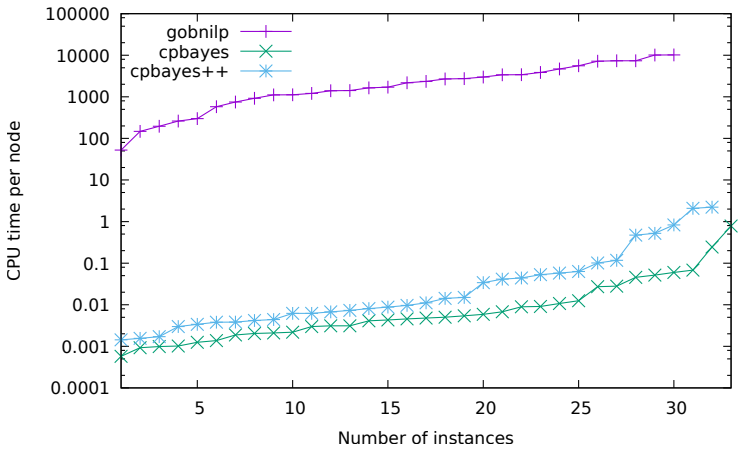


Figure 4.1: Temps PCU (en secondes) par nœud de recherche sur un sous-ensemble des 41 grands et très grands jeux de données.

Dans la Figure 4.2, nous comparons le nombre de nœuds dans l'arbre de recherche pour CPBayes et ELSA ^{cluster} sur des jeux de données résolus par les deux. Comme le nombre de jeux de données couramment résolus est trop faible

Problem	$\|V\|$	$\sum_{v\in\mathbf{V}} \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}
kdd.ts	64	43584	⋆**0.03%**	2.51%	⋆0.50%
kdd.test	64	152873	⋆**0.02%**	⋆1.75%	⋆0.20%
plants.ts	69	164640	⋆**2.42%**	35.01%	7.28%
kdd.valid	64	197546	⋆**0.13%**	2.60%	⋆0.49%
baudio.ts	100	371117	**2.00%**	11.34%	⋆3.63%
pumsb_star.ts	163	394992	⋆**0.19%**	24.69%	0.46%
tretail.ts	135	435976	⋆0.17%	0.48%	**0.07%**
bnetflix.ts	100	446406	**3.23%**	⋆20.86%	⋆8.73%
plants.test	69	520148	**2.58%**	34.32%	⋆5.98%
jester.ts	100	531961	**4.20%**	13.03%	⋆6.44%
kosarek.ts	190	556189	⋆**0.00%**	2.13%	0.44%
accidents.ts	111	568160	⋆**0.00%**	8.36%	⋆0.38%
plants.valid	69	684141	**2.57%**	34.61%	⋆6.66%
msweb.ts	294	732213	**0.25%**	1.83%	0.62%
diabetes-5000	413	754563	**0.26%**	0.67%	0.67%
jester.test	100	770950	**4.76%**	13.29%	⋆6.37%
tretail.test	135	897478	**2.88%**	4.44%	3.16%
baudio.test	100	1016403	**2.22%**	11.92%	⋆3.10%
pumsb_star.test	163	1034955	23.79%	68.12%	**21.61%**
tretail.valid	135	1087404	**0.67%**	2.60%	2.60%
bnetflix.test	100	1103968	**5.49%**	⋆24.48%	⋆11.69%
kosarek.test	190	1192386	**1.06%**	4.38%	2.60%
baudio.valid	100	1235928	**2.37%**	12.09%	⋆3.32%
pumsb_star.valid	163	1271525	33.69%	75.89%	**21.63%**
kosarek.valid	190	1312600	**2.20%**	5.84%	5.84%
bnetflix.valid	100	1325818	**5.38%**	⋆25.26%	⋆11.24%
accidents.test	111	1425966	⋆**0.00%**	20.03%	⋆3.71%
jester.valid	100	1463335	**5.64%**	17.03%	⋆7.63%
msweb.test	294	1597487	0.53%	3.81%	**0.37%**
accidents.valid	111	1617862	**3.57%**	34.77%	7.83%
dna.ts	180	1849860	**1.07%**	5.53%	1.52%
msweb.valid	294	1869374	**0.78%**	3.59%	3.59%
tmovie.ts	500	1918883	**35.15%**	-	-
pigs-5000	441	1984359	**24.50%**	-	-
dna.test	180	2019003	**1.07%**	5.66%	⋆1.62%
book.ts	500	2071722	**7.56%**	-	-
dna.valid	180	2561134	3.90%	9.92%	**3.72%**
tmovie.valid	500	2610026	**49.06%**	-	-
tmovie.test	500	2778556	**53.78%**	-	-
book.test	500	2794588	**15.08%**	-	-
book.valid	500	3020475	**20.24%**	-	-

Table 4.3: Qualité des bornes inférieures initiales sur 18 grands et 23 très grands jeux de données. La meilleure méthode est indiquée en gras. '⋆' : optimum prouvé, '-' : aucune borne inférieure signalée.

pour les grands et très grands jeux de données, nous effectuons cette comparaison sur les jeux de données moyens avec moins de 64 variables. Les points sous la ligne $y = x$ indiquent un arbre de recherche plus petit pour ELSA ^{cluster}. La réduction de la taille de l'arbre de recherche est très claire.

4.5.2.2 Qualité des Solutions et Preuves d'Optimalité

Nous avons mesuré la qualité d'une solution BNSL s comme la distance relative $\frac{s-o}{o}$ à la meilleure solution o trouvée par n'importe quelle méthode dans la limite des 90 heures de temps CPU. Les résultats sont donnés dans le tableau 4.4. Rappelons que CPBayes et ELSA ^{cluster} ont tous deux une procédure de recherche locale en prétraitement. Elle aide la recherche à trouver de bonnes solutions initiales.

Concernant les preuves d'optimalité, ELSA ^{cluster} a résolu 17 jeux de données, GOBNILP en a résolu 9, et CPBayes 4 seulement. GOBNILP s'est exécuté hors mémoire (128 Go) sur 11 jeux de données alors que ELSA ^{cluster} a pris moins de 8 Go sur tous les repères.

4.5.2.3 Impact du Seuil d'Activité du Pool de Clusters

En pratique, sur de très grands jeux de données, $97,6\%$ des clusters improductifs sont détectés par les paires de support et $8,6\%$ des domaines actuels sont visités pour le reste. Pour conserver un pool de clusters à mémoire limitée, nous écartons les clusters fréquemment improductifs. Nous rejetons les grands clusters dont le ratio productif $\frac{\#productif}{\#productif+\#inproductif}$ est inférieur à $\frac{1}{1{,}000}$. Les clusters de taille 10

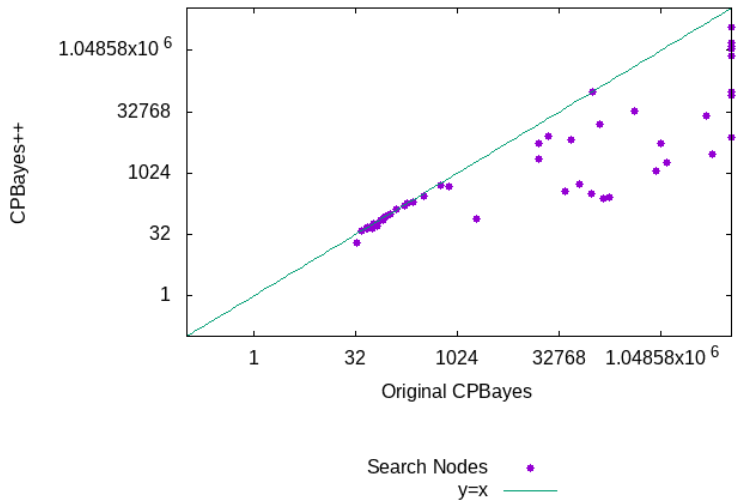


Figure 4.2: Nombre de nœuds d'arbre de recherche pour CPBayes et ELSA ^{cluster} sur des jeux de données moyens avec moins de 64 variables.

Problem	$\|V\|$	$\sum_{v \in \mathbf{V}} \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}
kdd.ts	64	43584	⋆**0.00%**	0.17%	⋆**0.00%**
kdd.test	64	152873	⋆**0.00%**	⋆**0.00%**	⋆**0.00%**
plants.ts	69	164640	⋆**0.00%**	3.02%	1.48%
kdd.valid	64	197546	⋆**0.00%**	0.16%	⋆**0.00%**
baudio.ts	100	371117	1.62%	1.29%	⋆**0.00%**
pumsb_star.ts	163	394992	⋆**0.00%**	4.76%	4.76%
tretail.ts	135	435976	⋆**0.00%**	0.29%	0.29%
bnetflix.ts	100	446406	0.01%	⋆**0.00%**	⋆**0.00%**
plants.test	69	520148	0.01%	2.61%	⋆**0.00%**
jester.ts	100	531961	10.29%	0.50%	⋆**0.00%**
kosarek.ts	190	556189	⋆**0.00%**	1.62%	1.62%
accidents.ts	111	568160	⋆**0.00%**	0.37%	⋆**0.00%**
plants.valid	69	684141	1.74%	2.40%	⋆**0.00%**
msweb.ts	294	732213	9.87%	**0.00%**	**0.00%**
diabetes-5000	413	754563	2.86%	**0.00%**	**0.00%**
jester.test	100	770950	7.45%	0.18%	⋆**0.00%**
tretail.test	135	897478	6.93%	0.03%	**0.00%**
baudio.test	100	1016403	13.69%	0.92%	⋆**0.00%**
pumsb_star.test	163	1034955	102.62%	**0.00%**	**0.00%**
tretail.valid	135	1087404	17.60%	0.38%	**0.00%**
bnetflix.test	100	1103968	0.41%	⋆**0.00%**	⋆**0.00%**
kosarek.test	190	1192386	†15.70%	**0.00%**	**0.00%**
baudio.valid	100	1235928	12.46%	0.49%	⋆**0.00%**
pumsb_star.valid	163	1271525	155.68%	**0.00%**	**0.00%**
kosarek.valid	190	1312600	†25.15%	**0.00%**	**0.00%**
bnetflix.valid	100	1325818	7.07%	⋆**0.00%**	⋆**0.00%**
accidents.test	111	1425966	⋆**0.00%**	1.71%	⋆**0.00%**
jester.valid	100	1463335	7.86%	0.68%	⋆**0.00%**
msweb.test	294	1597487	†14.99%	**0.00%**	**0.00%**
accidents.valid	111	1617862	†433.17%	0.32%	**0.00%**
dna.ts	180	1849860	†16.82%	**0.00%**	**0.00%**
msweb.valid	294	1869374	†14.82%	**0.00%**	**0.00%**
tmovie.ts	500	1918883	**0.00%**	-	-
pigs-5000	441	1984359	†**0.00%**	-	-
dna.test	180	2019003	†22.33%	0.51%	⋆**0.00%**
book.ts	500	2071722	**0.00%**	-	-
dna.valid	180	2561134	†25.77%	**0.00%**	**0.00%**
tmovie.valid	500	2610026	**0.00%**	-	-
tmovie.test	500	2778556	**0.00%**	-	-
book.test	500	2794588	†**0.00%**	-	-
book.valid	500	3020475	†**0.00%**	-	-

Table 4.4: Qualité de la solution sur 18 grands et 23 très grands jeux de données. La meilleure méthode est indiquée en gras. '⋆' : optimum trouvé, '†' : hors mémoire, '-' : aucune solution trouvée.

ou moins sont toujours conservés car ils sont souvent plus productifs et leur nombre est borné.

Nous avons réalisé une dernière expérience pour voir si le seuil d'activité a un impact sur la performance globale de notre approche. La figure 4.3 donne le nombre total de coupes de clusters dans le pool à la fin de la recherche pour 17 jeux de données résolus. Soit a_W le nombre de fois où le cluster W a amélioré la borne inférieure. Nous avons constaté que le fait de garder W dans le pool de clusters pendant $100 \times a_W$ ou jusqu'à $100000 \times a_W$ nœuds de recherche ne change pas l'effort de recherche global en termes de nombre total de nœuds de recherche pour résoudre un problème (Figure 4.4). Cependant, il peut augmenter le temps CPU, mais cela peut également être dû aux fluctuations de performance du cluster (Figure 4.5).

Par défaut, nous choisissons un seuil d'activité de $\frac{1}{1{,}000}$, *i.e.* gardant chaque cluster W pendant $1,000 \times a_W$ noeuds de recherche. Les clusters de taille inférieure ou égale à 10 ont toujours été conservés. De tels clusters sont souvent plus productifs et leur nombre est borné, voir par exemple les activités de cluster dans `kdd.test` (Figure 4.6). Sur 41 grands et très grands jeux de données, le nombre de coupes trouvées en prétraitement varie de 328 (`accidents.test`) à 4011 (`kosarek.valid`) avec une moyenne de $2011,4$. À la fin de la recherche, ou à la limite des 90 heures de temps CPU, il y avait de $85 (\texttt{pumsb\_star.valid}) \text{à} 163710$ (`baudio.ts`) clusters dans le pool avec une moyenne de $26638,3$ clusters.

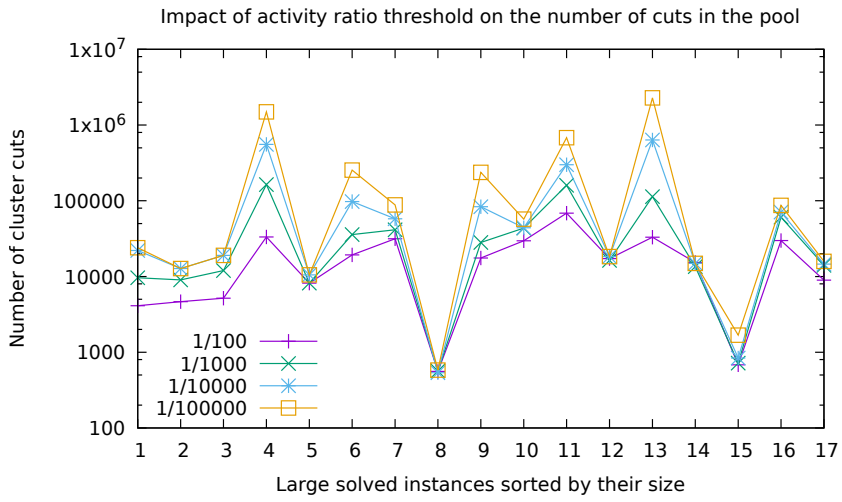


Figure 4.3: Nombre de coupes de clusters dans le pool à la fin de la recherche lors de la résolution de grands jeux de données en fonction du seuil de ration d'activité.

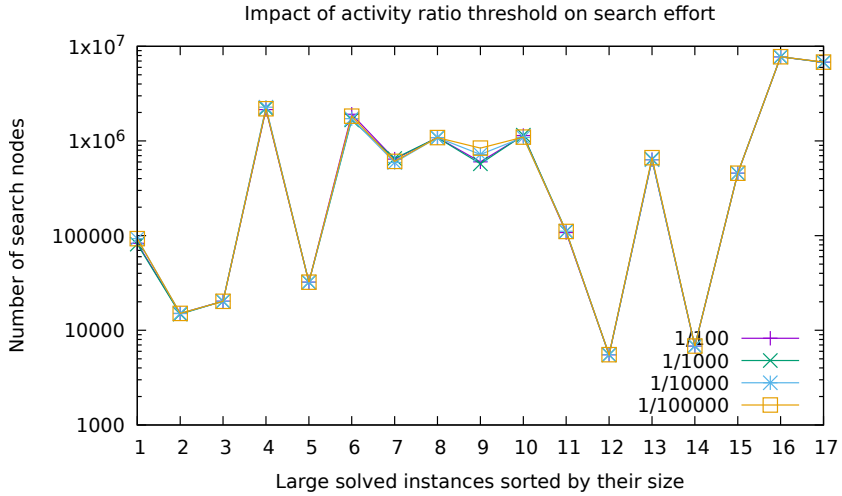


Figure 4.4: Nombre de nœuds de recherche pour la résolution de grands jeux de données en fonction du seuil de rationnement de l'activité.

4.6 Conclusion

Nous avons présenté un nouvel ensemble de techniques d'inférence pour BNSL utilisant la programmation par contraintes, centré sur l'expression de la contrainte d'acyclicité. Ces nouvelles techniques exploitent et améliorent les travaux précédents sur les relaxations linéaires de la contrainte d'acyclicité et du propagateur

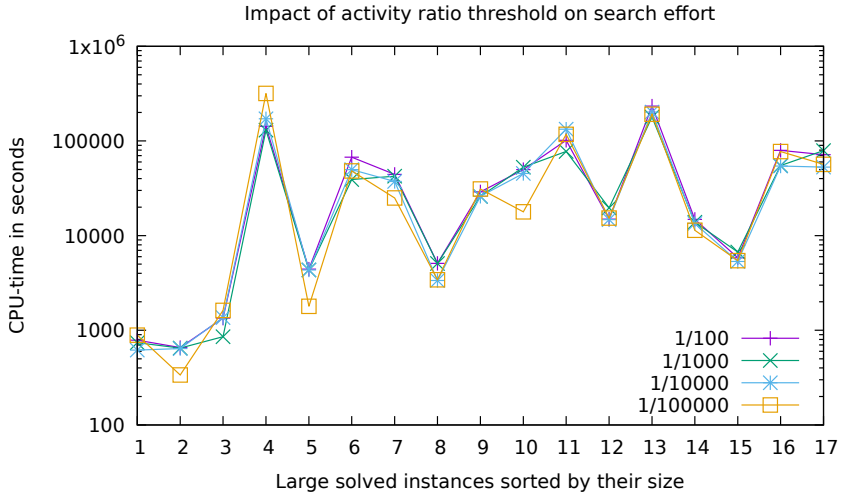


Figure 4.5: Temps PCU pour la résolution de grands jeux de données en fonction du seuil du ratio d'activité.

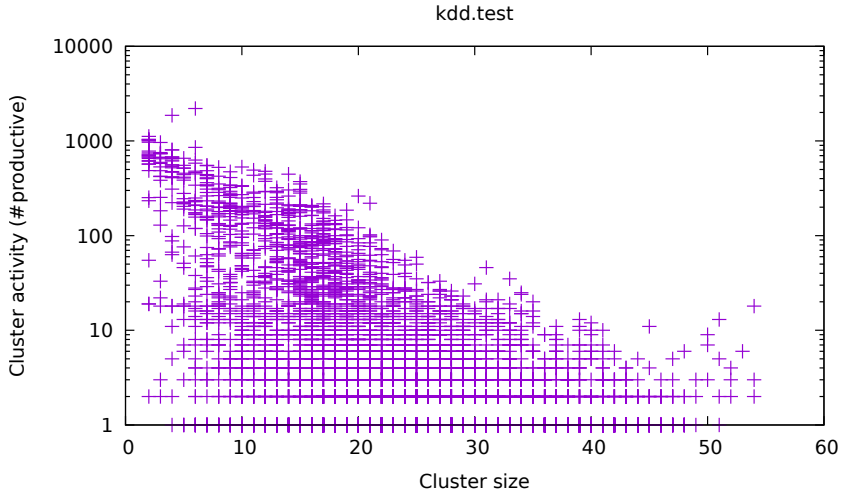


Figure 4.6: Activité du cluster ($\#productive$) en fonction de la taille pour $9,018$ de coupes conservées dans le pool du cluster à la fin de la recherche, après $14,840$ de nœuds de recherche en 613 secondes sur `kdd.test` avec 64 variables aléatoires.

associé. Le solveur résultant ELSA ^{cluster} explore un compromis différent sur l'axe de la force d'inférence par rapport à la vitesse, avec GOBNILP à un extrême et CPBayes à l'autre. Nous avons montré expérimentalement que le compromis que nous obtenons est meilleur que les deux extrêmes, puisque notre solveur ELSA ^{cluster} surpasse à la fois GOBNILP et CPBayes.

Le principal obstacle à une meilleure évolutivité vers de plus grands jeux de données est le fait que la taille des domaines croît de manière exponentielle avec le nombre de variables, ce qui est dans une certaine mesure inévitable (comme le cas de `bnetflix.valid`). Par conséquent, dans le prochain chapitre, nous nous concentrerons sur l'exploitation de la structure de ces domaines pour améliorer les performances.

Table 4.5: Comparaison de ELSA ^{cluster} contre GOBNILP, CPBayes, ELSA ^{gac}, et ELSA ^{chrono}, en termes de temps d'exécution (et de recherche) total en secondes. La limite de temps pour les jeux de données blue à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. Pour CPBayes ainsi que pour toutes les variantes de ELSA, nous indiquons entre parenthèses le temps passé en recherche, une fois le prétraitement terminé. Le signe † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
mildew1000_BIC	35	126	0.37	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
lympho_BIC	19	143	0.16	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
water1000_BIC	32	159	0.16	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
mildew1000_BDe	35	166	0.5	0.5 (0.2)	0.5 (0.1)	0.4 (0.1)	0.6 (0.1)
hailfinder100_BIC	56	167	0.43	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
tumour_BIC	18	219	0.18	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
barley1000_BIC	48	244	0.29	1.5 (1.2)	1.5 (1.3)	1.6 (1.4)	1.9 (1.6)
shuttle_BIC	10	264	1.81	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hepatitis_BIC	20	266	0.63	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.4 (0.0)
tumour_BDe	18	274	0.39	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
lung-cancer_BIC	57	292	0.57	3.8 (2.7)	3.9 (2.8)	1.1 (0.0)	1.4 (0.0)
lympho_BDe	19	345	0.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
hailfinder500_BIC	56	418	0.36	3.4 (2.6)	3.3 (2.5)	1.2 (0.5)	1.2 (0.4)
carpo100_BIC	60	423	**0.6**	79.8 (27.8)	79.9 (27.6)	52.6 (0.1)	56.9 (0.1)
horse_BDe	28	490	0.71	1.2 (0.7)	1.4 (0.8)	0.6 (0.0)	0.6 (0.0)
horse_BIC	28	490	0.99	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.5 (0.0)
hepatitis_BDe	20	501	0.93	0.4 (0.0)	0.4 (0.0)	0.5 (0.0)	0.4 (0.0)
insurance1000_BIC	27	506	**0.51**	32.4 (0.0)	32.6 (0.0)	32.8 (0.0)	33.9 (0.0)
adult_BIC	15	547	0.33	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

Table 4.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
zoo_BIC	17	554	0.42	0.1 (0.0)	0.0 (0.0)	0.1 (0.0)	0.1 (0.0)
spectf_BIC	45	610	1.52	4.3 (3.5)	4.1 (3.3)	**0.8 (0.0)**	1.2 (0.1)
sponge_BIC	45	618	1.45	5.0 (3.2)	5.5 (3.7)	1.8 (0.0)	2.0 (0.0)
flag_BIC	29	741	1.28	1.1 (0.6)	1.0 (0.5)	0.4 (0.0)	0.5 (0.0)
vehicle_BIC	19	763	1.48	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
adult_BDe	15	768	0.67	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
insurance1000_BDe	27	792	**0.7**	34.0 (0.0)	34.2 (0.0)	34.3 (0.0)	37.5 (0.0)
shuttle_BDe	10	812	4.03	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
bands_BIC	39	892	3.96	0.7 (0.4)	0.7 (0.4)	0.6 (0.2)	0.5 (0.2)
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	181.1 (147.8)	34.4 (1.0)	39.4 (3.9)
segment_BIC	20	1053	4.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
flag_BDe	29	1324	4.36	14.9 (14)	15.7 (14.7)	**1.0 (0.2)**	1.6 (0.7)
voting_BIC	17	1848	1.87	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
voting_BDe	17	1940	1.91	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.1 (0.0)
autos_BIC	26	2391	**11.83**	17.2 (0.0)	17.3 (0.0)	19.2 (0.0)	18.3 (0.0)
zoo_BDe	17	2855	19.02	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
vehicle_BDe	19	3121	5.68	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
letter_BIC	17	4443	3.95	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
soybean_BIC	36	5926	**40.19**	45.2 (1.5)	45.4 (1.5)	50.8 (3.0)	54.3 (5.7)
msnbc.ts	17	6298	8.61	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.3 (0.0)
segment_BDe	20	6491	29.15	0.4 (0.0)	0.4 (0.0)	0.5 (0.1)	0.6 (0.1)
mushroom_BIC	23	13025	1561.82	4.1 (0.0)	4.2 (0.0)	4.3 (0.2)	4.4 (0.2)
wdbc_BIC	31	14613	99.77	390.6 (337.5)	396.1 (337.8)	**56.0 (2.4)**	63.7 (4.7)
msnbc.test	17	16594	57.96	0.4 (0.0)	0.4 (0.0)	0.5 (0.1)	0.5 (0.1)

Table 4.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
letter_BDe	17	18841	111.73	0.4 (0.0)	0.4 (0.0)	0.7 (0.3)	0.7 (0.3)
msnbc.valid	17	20673	23.14	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
nltcs.ts	16	22156	15.3	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
autos_BDe	26	25238	715.89	146.6 (0.1)	146.9 (0.1)	**145.8 (0.8)**	156.3 (0.8)
kdd.ts	64	43584	**327.63**	†	†	1452.3 (274.6)	2257.9 (960.0)
nltcs.valid	16	47097	40.91	0.9 (0.0)	0.9 (0.0)	1.0 (0.1)	1.0 (0.1)
nltcs.test	16	48303	56.58	1.0 (0.0)	1.0 (0.0)	1.2 (0.3)	1.2 (0.3)
steel_BIC	28	93026	†	981.2 (931.5)	924.8 (874.6)	**124.2 (71.8)**	196.6 (139.6)
kdd.test	64	152873	**1521.73**	†	†	1594.3 (224.4)	1863.6 (246.1)
kdd.valid	64	197546	†	†	†	†	†
mushroom_BDe	23	438185	†	131.2 (5.7)	**130.8 (5.6)**	182.6 (58.9)	225.3 (66.6)
plants.ts	69	164640	†	†	†	†	†
baudio.ts	69	371117	†	†	†	†	†
bnetflix.ts	100	446406	†	673.4 (456.2)	**453.2** (236.3)	2103.1 (1900.9)	2482.1 (2727.0)
plants.test	111	520148	†	†	†	**28049.6 (26312.9)**	†
jester.ts	100	531961	†	†	†	**21550.5 (21003.7)**	29868.0 (29251.5)
accidents.ts	100	568160	**1273.96**	†	†	2302.2 (930)	2887.7 (1280.2)
plants.valid	111	684141	†	†	†	**17801.6 (14080.2)**	†
jester.test	100	770950	†	†	†	**30186.8 (29455)**	†
baudio.test	100	1016403	†	†	†	†	†
bnetflix.test	100	1103968	†	2725.6 (2475.1)	**1651.7** (1402.0)	10333.1 (10096.5)	10079.1 (9799.5)
baudio.valid	69	1235928	†	†	†	†	†
bnetflix.valid	111	1325818	†	511.7 (146.8)	**457.7** (94.4)	10871.7 (10527.7)	10363.3 (10749.1)
accidents.test	100	1425966	4975.64	†	†	**3641.7 (680.7)**	4290.7 (866.0)

Table 4.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{gac}	ELSA ^{cluster}	ELSA ^{chrono}
jester.valid	100	1463335	†	†	†	†	†
accidents.valid	100	1617862	†	†	†	†	†

CHAPTER 5

Arbres de Décision Binaires pour le BNSL

Contents

5.1 Introduction

Rappelons la complexité asymptotique des algorithmes que nous avons vus dans les chapitres 3 et 4 :

- `Acyclicity-GAC-n3d`: $O(n^3d)$
- `lowerBoundRC`: $O(n^2dmin(UB - LB, \sum_{v \in V} d_v))$

Celles-ci ont toutes un facteur de d, c'est-à-dire la plus grande taille de domaine dans un problème BNSL. Nous savons qu'en théorie, d est limité par $2^{(n-1)}$, et en pratique, il est limité par un certain nombre d'entiers m, qui est effectivement plus petit que $2^{(n-1)}$, mais beaucoup plus grand que n dans la plupart des cas.

Cette limitation des performances pratiques de nos algorithmes nous a amenés à réfléchir à des moyens d'exploiter les caractéristiques spécifiques de la BNSL. La caractéristique la plus frappante est sans doute le fait que les valeurs du domaine ne sont pas des valeurs arbitraires, mais des *ensembles*. Par conséquent, nous pouvons utiliser les structures de données existantes pour les ensembles d'ensembles afin d'obtenir de meilleures performances. Nous devons toutefois reconnaître que les options ne sont *pas* illimitées, car la structure de données en question doit être flexible pour permettre les suppressions et les récupérations de valeurs, capable de prendre en compte les informations de coût et de répondre à des requêtes telles que

- Existe-t-il une valeur d'ensemble parent qui est un sous-ensemble d'un ensemble donné S ?
- Existe-t-il une valeur d'ensemble parent qui est un sous-ensemble de S avec un coût associé de 0 ?
- Existe-t-il une valeur d'ensemble parent qui est un sur-ensemble d'un ensemble donné S ?

Afin de répondre à l'une de ces requêtes, si nous itérons sur chaque valeur du domaine d'une variable, selon notre chance, nous pouvons nous arrêter à la première ou à la dernière valeur rencontrée. Par conséquent, en particulier pour les grands problèmes BNSL, répondre à ces requêtes avec une meilleure complexité que $O(d)$ peut potentiellement représenter un gain de temps énorme. Dans ce chapitre, nous utilisons des arbres de décision binaires à cette fin. À la fin, nous constatons que même une mise en œuvre plutôt naïve peut entraîner une amélioration significative des performances pour certains jeux de données.

5.2 Arbres de Décision Binaires

Un arbre de décision binaire (BDT) est un outil d'aide à la décision qui modélise un ensemble de décisions et leurs résultats sous la forme d'un arbre binaire enraciné. Ils fournissent un moyen d'exécuter des requêtes avec des instructions de contrôle conditionnel. Les branches d'un BDT correspondent aux étapes de prise de décision qui mènent à un résultat souhaité.

Les BDT sont un instrument populaire dans l'apprentissage automatique (ML) [Cardie 1993, Elaidi *et al.* 2018] avec de nombreuses utilisations pratiques, une typique étant *classification* [Jensen & Arnspang 1999]. La classification est la tâche consistant à trier un jeu de données d'entrée en fonction de certaines caractéristiques données. Chaque nœud d'un BDT représente une caractéristique à classer, et chaque

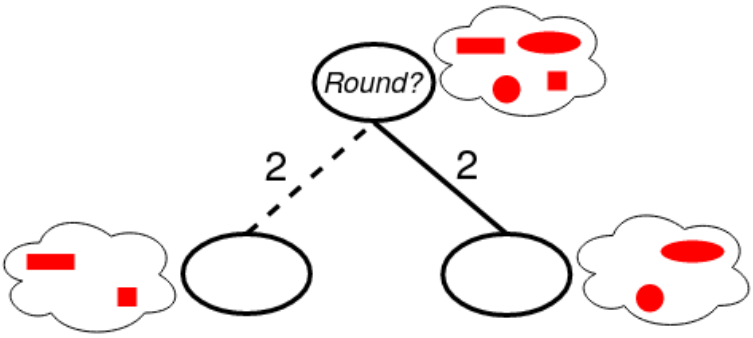


Figure 5.1: Un BDT qui n'est pas complètement développé.

branche représente une valeur que le nœud peut prendre. Comme on peut s'y attendre, les branches d'une BDT peuvent représenter soit *vrai* soit *faux*.

Dans la Figure 5.1, nous avons un BDT où nous classons un ensemble de formes. Ces formes peuvent appartenir à une *classe* telle que *cercle, ellipse, carré, rectangle*. Chaque ellipse désigne un noeud et à côté de chaque noeud, nous avons un nuage montrant l'ensemble d'entrée de ce noeud. Si un noeud classe son ensemble d'entrée par rapport à une caractéristique, alors nous écrivons cette caractéristique à l'intérieur du noeud. Nous créons une branche en pointillés à partir d'un noeud s'il y a au moins un élément dans son ensemble d'entrée qui *ne possède pas* cette caractéristique, et de la même manière, nous créons une branche pleine à partir d'un noeud s'il y a au moins un élément dans son ensemble d'entrée qui *a* cette caractéristique. À chaque branche, nous associons également le nombre d'éléments qui circulent vers le nœud enfant.

La classification, et donc le branchement, s'arrête lorsque l'ensemble d'entrée d'un noeud est *suffisamment pur* par rapport à la quantité d'erreur autorisée. Par exemple, si nous autorisons une erreur de 1%, il suffit que 99 éléments sur 100 soient dans la même classe. Le nœud où la classification s'arrête est appelé un *nœud feuille*. Nous désignons les nœuds feuilles par une ellipse vide.

En ML, le processus de construction d'un BDT est appelé *training* et il est effectué à l'aide d'un *training dataset*. Une fois la phase de training terminée et la structure de la BDT entièrement connue, elle est ensuite utilisée pour classer l'ensemble des données. Un défi commun dans ce contexte particulier est de trouver une structure BDT qui *ne sur-ajuste pas* les données. L'overfitting est une situation dans laquelle un BDT s'adapte parfaitement aux données d'apprentissage mais ne parvient pas à généraliser les données non vues. Une raison possible pour laquelle l'overfitting peut se produire est que le BDT est *complètement développé*. Nous disons qu'un BDT est complètement développé lorsqu'il classifie les données d'apprentissage en ensembles complètement purs, c'est-à-dire des ensembles où tous les éléments appartiennent à la même classe, modulo l'erreur que nous autorisons.

Le BDT de la Figure 5.1 n'est pas complètement développé, car les nœuds feuilles les plus à droite et les plus à gauche n'ont pas d'ensembles purs. Ils

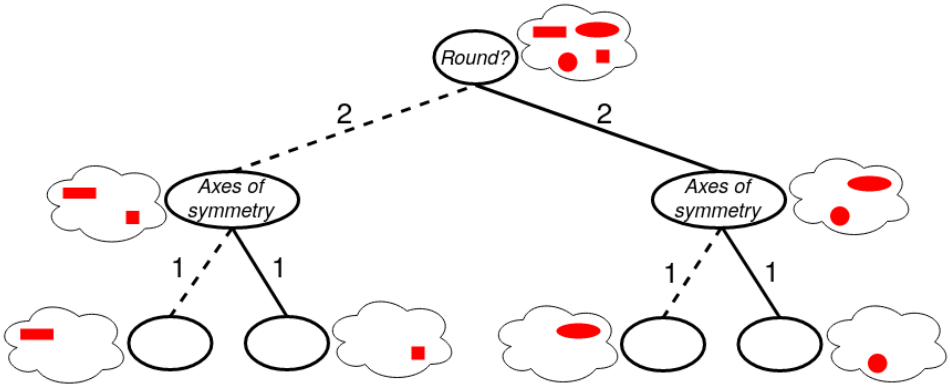


Figure 5.2: Un BDT entièrement développé.

contiennent tous deux des formes qui appartiennent à des classes différentes. Donc, ici, une certaine quantité d'erreur, c'est-à-dire d'impureté, est autorisée, et du point de vue de l'overfitting, c'est plutôt positif. Le BDT de la Figure 5.2 effectue une étape de classification supplémentaire pour vérifier si les axes de symétrie des formes sont de même longueur ou non. Nous voyons qu'il est complètement développé, qu'il a une erreur nulle et qu'il est peut-être plus enclin à sur-adapter les données.

Bien qu'un BDT entièrement développé ne soit pas nécessairement préféré dans le cadre de ML, dans notre cas, c'est ce que nous voulons avoir afin de représenter le domaine $D(v)$ d'une variable d'ensemble parent v. Nous voulons classer chaque valeur du domaine, c'est-à-dire l'ensemble parent $S \in D(v)$ en $d_v = |D(v)|$ nombreux groupes, c'est-à-dire un groupe par chaque ensemble parent. La raison en sera claire dans la section suivante.

5.3 Utilisation des BDTs pour Représenter les Valeurs de Domaine en BNSL

Le BDT de chaque variable v est constitué de $n = |V|$ niveaux. Les $n - 1$ premiers niveaux sont pour chaque variable dans $V \setminus \{v\}$. L'approche naïve consiste à se brancher sur les variables à chaque niveau dans un ordre lexicographique, cependant, un ordre différent peut également être utilisé, et de plus, l'ordre des variables *n'a pas à être* identique pour chaque branche. Le dernier niveau est constitué des noeuds feuilles, dans chacun desquels il y a exactement une valeur $S \in D(v)$. Pour chaque valeur $S \in D(v)$, il existe un chemin unique, c'est-à-dire une séquence de branches menant du noeud racine au noeud feuille. Cela implique également que pour chaque valeur $S \in D(v)$, il existe *moins une arête* qui n'est utilisée que par cette valeur. Il s'agit d'une propriété importante qui nous permet d'ajouter et de supprimer des valeurs si nécessaire.

Chaque noeud η possède les attributs suivants :

- ID
- Index variable
- Faux noeud enfant η_f
- True child node η_t (nœud enfant vrai)
- Faux compteur d'arêtes
- Vrai compteur d'arêtes

La racine est créée avant que toutes les autres ne soient explorées en profondeur d'abord. L'indice variable u d'un noeud η est son classificateur. S'il existe au moins une valeur d'ensemble parent $S \in D_\eta(v) \subseteq D(v)$, où $D_\eta(v)$ est l'ensemble des valeurs prises en entrée par η, alors un vrai enfant η_t est créé pour η. S, ainsi que toutes les autres valeurs d'ensemble parent qui contiennent u, sont transférées à η_t. Nous désignons l'ensemble de ces valeurs par $D^u_\eta(v)$. Le nombre de valeurs de l'ensemble parent transférées à η_t est conservé comme le "compteur de bords vrais" de η. De même, s'il existe une valeur d'ensemble parent $S \notin D_\eta(v)$, alors un faux enfant η_f est créé pour η. Nous désignons l'ensemble de ces valeurs par D^{-u}_η. Toutes les valeurs de l'ensemble des parents dans D^{-u}_η sont transférées à η_f et le compteur de faux bords de η est fixé à $|D^{-u}_\eta|$.

Variable	Domaine
v_0	$\{v_1\}$
	$\{v_2\}$
	$\{v_3\}$
	$\{v_1, v_2\}$

Table 5.1: Le problème BNSL utilisé tout au long de ce chapitre. Nous avons $V = \{v_0, v_1, v_2, v_3\}$ et $n = |V| = 4$, et nous nous concentrons sur le domaine de la variable v_0.

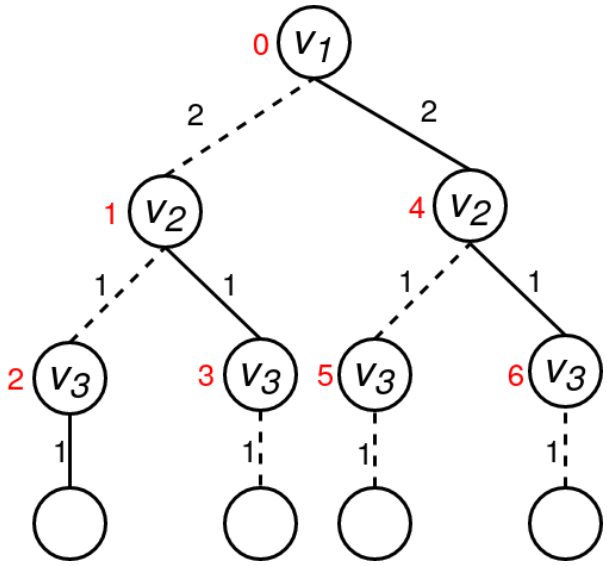


Figure 5.3: BDT de v_0 du problème BNSL présenté dans le tableau 5.1

Example 5.1. *Dans le tableau 5.1, nous présentons le domaine d'une variable v_0, qui appartient à un problème BNSL à 4 variables $V = \{v_0, v_1, v_2, v_3\}$. $D(v_0)$ contient 4 valeurs et dans la Figure 5.3, nous voyons le BDT de v_0. Cette BDT a 1 niveau pour chaque variable dans $V \setminus \{v_0\}$, et un niveau supplémentaire pour les noeuds feuilles, que nous appelons également sink. Le nœud racine prend les 4 valeurs en entrée et les classe en deux groupes : celles qui contiennent v_1 et celles qui ne contiennent pas v_1. En conséquence, un faux enfant est créé pour la racine, et, les valeurs $\{v_2\}$ et $\{v_3\}$ sont transférées à ce nouveau faux enfant, augmentant le compteur de faux bord à 2. Ensuite, à partir du faux enfant, $\{v_3\}$ descend plus bas via le bord pointillé, et $\{v_2\}$ via le bord plein. Enfin, chacune de ces valeurs est transférée à un noeud feuille avec une arête appropriée (en pointillés ou pleine). Par la suite, un enfant vrai pour la racine est également créé, et, les valeurs $\{v_1\}$ et $\{v_1, v_2\}$ sont transférées à ce nouvel enfant vrai, augmentant le compteur d'arêtes vraies à 2. Nous continuons de cette manière jusqu'à ce que toutes les valeurs trouvent leur chemin vers un noeud feuille. Les chiffres rouges à côté de chaque noeud indiquent son ID, ainsi que l'ordre dans lequel chaque noeud est créé. Nous remarquons que ce BDT atteint sa plus grande largeur, qui est de 4, à l'avant-dernier niveau. Par convention, la plus grande largeur d'un BDT est simplement sa largeur.*

5.4 Opérations Principales dans les BDTs

5.4.1 Création de BDTs

Au début de la recherche, nous créons un BDT pour chaque variable $v \in V$ que nous maintenons dynamiquement pendant toute la recherche. C'est-à-dire que chaque

Algorithm 28: Construction d'un BDT et exploration de chaque nœud η en partant de la racine.

```
1  Procedure constructBDT(v):
2      η ← root
3      D_η(v) ← D(v)
4      exploreNode(η, D_η(v), 0, {})
5  Procedure exploreNode(η, D_η(v), level, seenVars):
6      η.varIndex = chooseVar(V \ (seenVars ∪ {v}))
7      η.falseCounter ← d_v^{-η.varIndex}
8      η.trueCounter ← d_v^{η.varIndex}
9      if level = nbLevels − 2 then
10         if η.falseCounter > 0 then
11             η_f ← sink
12         if η.trueCounter > 0 then
13             η_t ← sink
14         return
15     seenVars ← seenVars ∪ {η.varIndex}
16     if η.falseCounter > 0 then
17         η_f ← newNode()
18         exploreNode(η_f, D_η^{-η.varIndex}(v), level + 1,seenVars)
19     if η.trueCounter > 0 then
20         η_t ← newNode()
21         exploreNode(η_t, D_η^{η.varIndex}(v), level + 1, seenVars)
```

fois qu'une valeur de domaine $S \in D(v)$ est élaguée ou restaurée, le BDT de v sera mis à jour en conséquence.

Voyons d'abord comment nous construisons un BDT. Dans l'algorithme 28, nous voyons la sous-routine qui construit le BDT de $v \in V$. Nous avons `constructBDT`, où nous créons un nœud racine, dont l'indice de variable doit encore être défini. Ensuite, nous déclenchons une procédure récursive appelée `exploreNode` qui a 6 paramètres d'entrée :

- Un nœud BDT η dont l'indice variable sera défini et les enfants seront créés
- L'ensemble des valeurs de l'ensemble des parents d'entrée $D_\eta(v) \subseteq D(v)$
- Le niveau, c'est-à-dire la profondeur de η
- L'ensemble des variables sur lesquelles nous avons jusqu'à présent bifurqué jusqu'à atteindre η, c'est-à-dire $seenVars$.

Pour l'appel initial à `exploreNode`, les paramètres d'entrée ci-dessus sont $root$, $D(v)$ c'est-à-dire le domaine entier, 0, l'ensemble vide, respectivement.

Dans $chooseVar$ qui est appelé à la ligne 6, bien que nous puissions utiliser l'heuristique que nous voulons, pour l'instant nous supposons que nous choisissons la première variable $v_i \in V \setminus (seenVars \cup \{v\})$ comme la prochaine variable à brancher. Nous définissons l'indice de variable de η à v_i, et nous divisons $D_\eta(v)$ en deux ensembles : l'ensemble $D_\eta^{v_i}(v)$ des valeurs S telles que $v_i \in S$ et l'ensemble $D_\eta^{-v_i}(v)$ des valeurs S^- telles que $v_i \notin S^-$. Nous fixons ensuite le compteur de faux bords et le compteur de vrais bords de η à $d_v^{-v_i}$ et $d_v^{v_i}$, respectivement.

Maintenant que nous savons sur quelle variable nous allons nous brancher, nous fixons l'indice de variable de η à $nodeVar$. Si à la ligne 9, nous constatons que η est en fait au dernier niveau avant le puits, nous fixons immédiatement ses enfants au puits, à condition que les compteurs d'arêtes correspondants soient positifs. Sinon, nous continuons à développer la BDT d'une manière profondeur-première. Si le compteur de faux bord de η est positif, nous créons un nouveau faux enfant pour lui et l'explorons. Pour cela, nous donnons en entrée à `exploreNode` le faux nœud enfant η_f, $D_\eta^{-nodeVar}(v)$ qui est effectivement $D_{\eta_f}(v)$, la profondeur du niveau suivant, et l'ensemble des variables vues dans lequel nous avons maintenant ajouté $nodeVar$. Une fois que la fausse branche est complètement développée, nous créons et explorons le vrai enfant à condition que le compteur de vrais bords de η soit positif.

5.4.2 Maintien des BDTs

Notre solveur étant basé sur CPBayes, plusieurs mécanismes d'élagage basés sur les coûts de symétrie et sur la symétrie sont utilisés. En outre, chaque fois que nous revenons en arrière au cours de la recherche, certaines des valeurs élaguées sont restaurées. Par conséquent, entre le moment où les BDT sont créés et le moment où nous devons effectivement exécuter des requêtes sur eux, il est très probable qu'il y ait eu plusieurs changements de ce type et nous devons nous assurer que les BDT correspondent exactement au domaine actuel. Nous faisons cela via les procédures `pruneParentSet` et `restoreValue` présentées dans les algorithmes 29 et 30.

`pruneParentSet` suit le chemin indiqué par un ensemble parent donné S de la racine au puits. Chaque fois que l'indice variable d'un noeud η que nous atteignons apparaît dans S, nous diminuons le compteur d'arêtes vraies de 1 et allons au vrai enfant η_t, et, lorsque ce n'est pas le cas, nous diminuons le compteur d'arêtes fausses de 1 et allons au faux enfant η_f. Notez que la mise à jour des compteurs se traduira par des compteurs nuls pour certains nœuds, dont la branche correspondante n'est utilisée que par S. `restoreValue` fait exactement la même chose que `pruneParentSet`, sauf qu'il augmente d'une unité les compteurs qu'il rencontre. Une autre observation importante à faire à ce stade est que, nous ne *jamais supprimer* aucun nœud. Nous rendons seulement un noeud *inaccessible* en rendant le compteur de la branche qui y mène nul. Nous pouvons voir les compteurs comme des *capacités d'écoulement*, et si le vrai (faux) compteur est zéro, nous ne pouvons tout simplement pas aller vers le vrai (faux) enfant.

Algorithm 29: Prune un ensemble de valeurs parentales $S \in D(v)$.

```
1  Procedure pruneParentSet(S):
2      η ← root
3      while η ≠ sink do
4          if η.varIndex ∈ S then
5              η.trueCounter ← η.trueCounter − 1
6              η ← η_t
7              continue
8          if η.varIndex ∉ S then
9              η.falseCounter ← η.falseCounter − 1
10             η ← η_f
11             continue
```

Algorithm 30: Restaurer un ensemble de valeurs parentales $S \in D(v)$.

```
1  Procedure restoreValue(S):
2      η ← root
3      while η ≠ sink do
4          if η.varIndex ∈ S then
5              η.trueCounter ← η.trueCounter + 1
6              η ← η_t
7              continue
8          if η.varIndex ∉ S then
9              η.falseCounter ← η.falseCounter + 1
10             η ← η_f
11             continue
```

5.4.3 Naviguer à Travers les BDTs

Nous avons très souvent besoin de demander si un nœud particulier que nous rencontrons a un faux enfant ou un vrai enfant. `hasTrueChild` et `hasFalseChild`, présentés dans l'Algorithme 31, répondent à ces questions. `hasTrueChild` renvoie $true$ si un enfant vrai η_t a été créé pour η, *et* le compteur vrai est positif, *et* η_t est *unmasked*. De même, `hasFalseChild` renvoie $true$ si un faux enfant η_f a été créé pour η, *et* le compteur de faux est positif, *et* η_f est *unmasked*. Par défaut, tous les nœuds sont démasqués. Nous utiliserons la fonction de masquage dans la section 5.6.

Algorithm 31: Vérifier si un noeud η a un enfant vrai ou faux et s'il est atteignable.

```
1 Procedure hasTrueChild(η):
2     if η_t exists and η.trueCounter > 0 and η_t unmasked then
3         return true
4     return false
5 Procedure hasFalseChild(η):
6     if η_f exists and η.falseCounter > 0 and η_f unmasked then
7         return true
8     return false
```

5.5 BDT dans le Propagateur GAC

Rappelons d'abord comment fonctionne le propagateur GAC. Chaque fois que nous itérons sur une variable donnée dans un ordre initial valide, nous essayons de voir dans quelle mesure nous pouvons la pousser vers le bas dans l'ordre vers la fin. Pour voir si nous pouvons pousser une variable plus bas, nous vérifions s'il existe une valeur dans son domaine qui est un sous-ensemble du préfixe actuel (voir ligne 12 dans l'Algorithme 32). C'est là que les BDT entrent en jeu avec la procédure `hasValueThatIsSubsetOf` présentée dans l'Algorithme 33.

Nous savons que dans le BDT d'une variable $w \in V$, nous avons un chemin unique pour chaque valeur de l'ensemble parent $S \in D(w)$. En complément, nous savons qu'il n'existe pas de chemin de la racine au puits pour $S \notin D(w)$. Par conséquent, il suffit de vérifier s'il existe un chemin de la racine au puits qui respecte le préfixe donné. Nous partons de la racine, et nous essayons tous les chemins alternatifs pour atteindre le puits. Le fait que nous soyons autorisés à suivre la branche pointillée ou pleine d'un nœud η que nous rencontrons dépend de l'indice de variable de η. Si $\eta.varIndex \in prefix$, nous pouvons suivre les deux branches, sinon exclusivement la branche en pointillés. Si le noeud que nous tentons d'atteindre en suivant une branche particulière, c'est-à-dire η_t ou η_f est atteignable, nous nous rendons à ce noeud et continuons de la même manière. Si,

Algorithm 32: Propagateur GAC pour l'acyclicité, utilisant les caractéristiques BDT.

```
1  Procedure Acyclicity-GAC-n3d(V, D):
2      O ← acycChecker(∅, V, D)
3      if O ⊊ V then
4          return false
5      foreach v ∈ V do
6          changes ← true
7          i ← O^{-1}(v)
8          prefix ← {O_0, ..., O_{i-1}}
9          while changes do
10             changes ← false
11             foreach w ∈ O \ (prefix ∪ {v}) do
12                 if BDT[w].hasValueThatIsSubsetOf (prefix) then
13                     prefix ← prefix ∪ {w}
14                     changes ← true
15         foreach u ∉ (prefix ∪ {v}) do
16             BDT[v].pruneParentSetsWith (u)
17     return true
```

toutefois, à un nœud η avant que nous atteignions le puits, il n'y a pas de nœud enfant atteignable où nous sommes autorisés à aller, `reachSink` appelé depuis η renvoie false. Si toutes les alternatives échouent ainsi, `reachSink` appelé depuis la racine renvoie false. Sinon, dès qu'un de ces chemins alternatifs que nous suivons atteint effectivement le puits, nous cessons immédiatement d'essayer les alternatives et retournons true.

Il convient ici de souligner le fait qu'un BDT fusionne les préfixes communs, grâce auxquels nous évitons d'explorer le sous-arbre entier d'un noeud dès que nous nous rendons compte qu'il ne contient aucune valeur qui soit un sous-ensemble d'un ensemble donné. Par exemple, supposons que nous voulions vérifier s'il existe une valeur $S \subseteq \{2, 3, 5\}$. Supposons également que nous avons $root.varIndex = 1$, $root_f.varIndex = root_t.varIndex = 2$. Nous ne prenons pas la peine d'explorer le sous-arbre enraciné à $root_t$, car toutes les valeurs de ce sous-arbre contiennent 1.

Une fois qu'une variable v est poussée vers le bas autant que possible dans l'ordre valide, nous procédons à la suppression des valeurs incohérentes de son domaine. Ce sont les valeurs $S \in D(v)$ telles que pour tout $u \notin prefix$, c'est-à-dire tout u qui vient après v dans l'ordonnancement final, $u \in S$. Ceci est réalisé par `pruneParentSetsWith` présenté dans l'algorithme 34.

Dans `pruneParentSetsWith`, nous trouvons *tout* les noeuds avec $varIndex = u$ dans le BDT de v. En partant de la racine, nous suivons les branches pleines et en pointillés de tout noeud dont l'indice de variable est différent de u. Chaque fois que nous rencontrons un noeud η avec $\eta.varIndex = u$, nous mettons son

Algorithm 33: Procédure permettant de vérifier si une variable possède au moins une valeur dans son domaine qui est un sous-ensemble du préfixe donné. $nbNodes$ est le nombre total de noeuds dans le BDT de la variable en question.

```
1  Procedure hasValueThatIsSubsetOf(prefix):
2      return reachSink (root, prefix)
3  Procedure reachSink(η, prefix):
4      if η is sink then
5          return true;
6      if η.varIndex ∈ prefix then
7          if η.hasTrueChild() and reachSink (η_t, prefix) then
8              return true;
9          if η.hasFalseChild() and reachSink (η_f, prefix) then
10             return true;
11         return false;
12     else
13         if η.hasFalseChild() and reachSink (η_f, prefix) then
14             return true;
15         return false;
```

Algorithm 34: Réduire les ensembles parentaux contenant une variable particulière.

```
1  Procedure pruneParentSetsWith(u):
2      reachNodeWithVarIndex(u, root)
3      updateCounters()
4  Procedure reachNodeWithVarIndex(u, node):
5      if η.varIndex = u then
6          if η.hasTrueChild() then
7              η.trueCounter ← 0
8          return
9      if η.hasTrueChild() then
10         reachNodeWithVarIndex(u, η_t)
11     if η.hasFalseChild() then
12         reachNodeWithVarIndex(u, η_f)
```

compteur vrai à 0, et arrêtons de descendre à partir de là. En fixant $\eta.trueCounter$ à 0, nous rendons non seulement η_t inaccessible, mais aussi le puits pour toutes les valeurs utilisant cette branche. Une fois que le compteur vrai de tous les nœuds avec $varIndex = u$ est fixé à 0, nous mettons à jour les compteurs à la ligne 3. `updateCounters` détecte tous les noeuds qui sont désormais devenus inaccessibles, et met à zéro les compteurs de vrais et de faux de ces noeuds. Ensuite, elle parcourt la BDT de bas en haut, et s'assure que pour chaque noeud η, la somme $\eta.trueCounter + \eta.falseCounter$ est égale au compteur de la branche menant à η.

5.6 BDT dans les Algorithmes de Cluster

Algorithm 35: Calcul de la borne inférieure avec RC-clusters

```
1  Procedure lowerBoundRC(V, D, 𝒲):
2      resetMasks()
3      ŷ ← dualSolve(LP_𝒲(D))
4      unmaskValues(D^rc_{𝒲,ŷ})
5      while true do
6          W ← V \ acycChecker(V, D^rc_{𝒲,ŷ})
7          if W = ∅ then
8              return ⟨cost(ŷ), 𝒲⟩
9          W ← minimise(W)
10         𝒲 ← 𝒲 ∪ {W}
11         ŷ ← dualImprove(ŷ, LP_𝒲(D), W)
12         unmaskValues(D^rc_{𝒲,ŷ})
```

Comme nous l'avons vu au chapitre 4, nous utilisons `acycChecker` à la fois dans `lowerBoundRC`, l'algorithme détectant un ensemble de coupes de clusters qui améliorent de manière prouvable la relaxation linéaire, et `minimiseCluster`, l'algorithme minimisant les clusters trouvés dans `lowerBoundRC`.

Un détail important à retenir cependant est que dans ces algorithmes, nous nous concentrons sur les variables dont le coût réduit est de zéro. Nous utilisons à ce stade la fonction *masking* que nous avons mentionnée précédemment. Dans `lowerBoundRC` présenté dans l'algorithme 35, juste avant de résoudre le dual de $LP_{\mathcal{W}}(D)$, nous appelons $resetMasks$ pour masquer *tous* les nœuds (sauf la racine) du BDT de toutes les variables afin de les rendre inaccessibles. Puis, une fois que nous avons $\hat{y}$, nous démasquons les noeuds qui sont utilisés dans les chemins pour les valeurs apparaissant dans $D^{rc}_{\mathcal{W},\hat{y}}$: l'ensemble des valeurs avec un coût réduit de zéro dans $\hat{y}$. De même, à chaque itération de la boucle while commençant à la ligne 5, le nouveau cluster W que nous découvrons crée un ensemble non vide de valeurs de l'ensemble parent dont le coût réduit devient nul. Par conséquent, nous

démasquons également les nœuds utilisés dans les chemins de ces valeurs.

Un autre point où nous faisons usage des BDT est lorsque nous recherchons la prochaine variable à ajouter dans l'ensemble O dans `acycChecker`. À la ligne 8 de l'algorithme 36, nous vérifions si la variable v sur laquelle nous itérons a une valeur dans son domaine qui est un sous-ensemble des membres actuels de l'ensemble O. Pour ce faire, nous faisons un appel à `hasValueThatIsSubsetOf`, que nous avons vu dans la section 5.5.

Algorithm 36: Vérificateur d'acyclicité

```
1  Procedure acycChecker(prefix, V, D^rc_{W,ŷ}):
2      O ← prefix
3      changes ← true
4      i ← size(prefix)
5      while changes do
6          changes ← false
7          foreach v ∈ V \ O do
8              if v hasValueThatIsSubsetOf (O) then
9                  O_i ← v
10                 changes ← true
11                 i ← (i + 1)
12     return O
```

5.7 Amélioration du Temps d'Exécution

5.7.1 Comment les BDTs Affectent le Temps d'Exécution

Le temps d'exécution est proportionnel à diverses mesures de la taille d'un BDT. Par exemple, la construction d'une BDT est proportionnelle au nombre de nœuds qu'elle aura finalement, compte tenu de l'ordre des variables utilisé, car évidemment, des ordres différents donnent une structure de BDT différente. D'autre part, trouver si un sous-ensemble d'un ensemble spécifique est contenu dans l'ensemble des ensembles représentés par l'arbre, comme dans `hasValueThatIsSubsetOf`, est proportionnel en $largeur \times profondeur$ *dans le pire des cas*.

Minimiser une métrique n'optimise pas nécessairement toutes les métriques et, en tout cas, trouver l'ordonnancement qui optimise n'importe quelle métrique est NP-hard [Laurent & Rivest 1976]. Il existe en effet des méthodes exactes pour résoudre ce problème NP-hard [Bessiere *et al.* 2009, Avellaneda 2019], néanmoins, la question de savoir si cela vaut la peine d'investir du temps ou non dans notre contexte reste pour l'instant une orientation future. En attendant, nous employons un algorithme heuristique afin d'espérer réduire la taille des BDT.

5.7.2 Gain d'Information

L'une des méthodes largement utilisées pour l'apprentissage d'une BDT repose sur la notion de *gain d'information*. Commençons par définir formellement et en termes généraux ce qu'elle est.

Definition 5.2 (Gain d'information)**.** ***Gain d'information (GI)*** *est la réduction attendue de l'entropie, c'est-à-dire la quantité d'information résultant d'une décision prise. Cette réduction d'un état à l'autre se calcule comme suit*

$$IG(\mathcal{T}, x) = H(\mathcal{T}) - H(\mathcal{T}|x) \tag{5.1}$$

où $\mathcal{T}$ est un ***ensemble d'apprentissage*** *que nous voulons classer, $H(\mathcal{T})$ est son* ***entropie****, et $H(\mathcal{T}|x)$ est son* ***entropie conditionnelle*** *étant donné un* ***attribut*** *x. On les calcule comme suit*

$$H(\mathcal{T}) = -\sum_i P(t_i) \times \log P(t_i) \tag{5.2a}$$

$$rH(\mathcal{T}|x) = \sum_{a \in vals(x)} \frac{|\mathcal{T}^{x=a}|}{|\mathcal{T}|} H(\mathcal{T}^{x=a}) \tag{5.2b}$$

$$= \sum_{a \in vals(x)} P(x=a) H(\mathcal{T}^{x=a}) \tag{5.2c}$$

$P(t_i)$ est la ***probabilité*** *d'un élément de l'ensemble $\mathcal{T}$ appartenant à une classe i, par exemple la probabilité d'une forme appartenant à la classe des carrés dans l'ensemble donné au BDT de la Figure 5.4, qui est de 0,25. Notez que le logarithme est à la base 2.*

$\mathcal{T}^{x=a}$ est l'ensemble des éléments dans $\mathcal{T}$ pour lesquels l'attribut x a la valeur a. Par conséquent, le rapport $\frac{|\mathcal{T}^{x=a}|}{|\mathcal{T}|}$ est également la ***probabilité*** *d'un élément ayant $x = a$, $P(x = a)$. Par exemple, nous avons $P(Round? = 1) = \frac{|\mathcal{T}^{Round?=1}|}{|\mathcal{T}|} = \frac{2}{4} = 0.5$ pour l'ensemble donné au BDT dans la Figure 5.4. $H(\mathcal{T}^{x=a})$ est la* ***entropie*** *de l'ensemble d'apprentissage* ***conditionné sur*** *$x = a$.*

Dans notre contexte, nous calculons le IG pour chaque variable $v \in V$ par rapport à une autre variable u. $\mathcal{T}$ correspond à l'ensemble de tous les *ensembles parents possibles (non candidats)* de v. Nous n'avons que deux classes : les ensembles parents qui apparaissent dans l'ensemble des *ensembles parents candidats*, c'est-à-dire le domaine $D(v)$, et ceux qui n'apparaissent pas. L'attribut correspond au fait de contenir, ou de ne pas contenir, une autre variable donnée u.

Example 5.3. *Rappelons le domaine de v_0 dans le petit problème BNSL avec $V = \{v_0, v_1, v_2, v_3\}$ et $n = 4$ que nous avons vu plus tôt dans ce chapitre, maintenant présenté dans le tableau 5.2. L'ensemble de tous les ensembles parents possibles de v_0, noté $\mathcal{T}_{v_0}$, est de taille $2^{n-1} = 8$ et se compose de ce qui suit :*

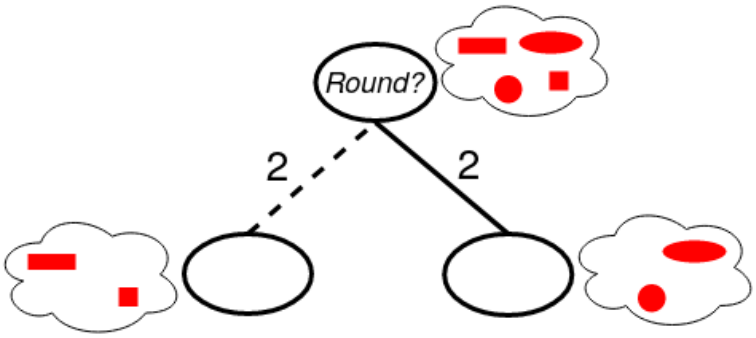


Figure 5.4: Le BDT pour les formes de la section 5.2.

	Valeur de la variable et du domaine
N	$\{v_1\}$
v_0	
N	$\{v_2\}$ $\{v_3\}$ $\{v_1, v_2\}$

Table 5.2: $n = |V| = 4$, $V = \{v_0, v_1, v_2, v_3\}$.

$\{\}$	$\{v_1, v_2\}$
$\{v_1\}$	$\{v_1, v_3\}$
$\{v_2\}$	$\{v_2, v_3\}$
$\{v_3\}$	$\{v_1, v_2, v_3\}$

Calculons d'abord l'entropie du domaine de $D(v_0)$. Nous savons d'après l'équation 5.2a qu'elle est égale à

$$H(\mathcal{T}_{v_0}) = -P(D(v_0)) \log P(D(v_0)) - P(D^-(v_0)) \log P(D^-(v_0))$$

où $P(D(v_0))$ est la probabilité qu'un ensemble parent dans $\mathcal{T}_{v_0}$ apparaisse également dans $D(v_0)$, ce qui est égal à

$$P(D(v_0)) = \frac{d_{v_0}}{|mathcalT_{v_0}|}$$

Rappelons que nous utilisons d_{v_0} pour désigner la taille du domaine de v_0. La probabilité qu'un ensemble parent dans $\mathcal{T}_{v_0}$ ***not*** *apparaisse dans $D(v_0)$ est*

$$P(D^-(v_0)) = \frac{|\mathcal{T}_{v_0}| - d_{v_0}}{|\mathcal{T}_{v_0}|}$$

Par conséquent, nous avons

$$\begin{aligned} H(\mathcal{T}_{v_0}) &= -\frac{d_{v_0}}{|\mathcal{T}_{v_0}|}\log\frac{d_{v_0}}{|\mathcal{T}_{v_0}|} - \frac{|\mathcal{T}_{v_0}| - d_{v_0}}{|\mathcal{T}_{v_0}|}\log\frac{|\mathcal{T}_{v_0}| - d_{v_0}}{|\mathcal{T}_{v_0}|} \\ &= -\frac{4}{8}\log\frac{4}{8} - \frac{8-4}{8}\log\frac{8-4}{8} \\ &= 1 \end{aligned}$$

Concentrons-nous maintenant sur l'attribut de contenir v_3.

L'ensemble de tous les ensembles parents possibles de v_0 contenant v_3, noté $\mathcal{T}_{v_0}^{v_3}$, est de taille $2^{n-2} = 4$ et se compose de ce qui suit :

$$\begin{array}{cc} \{v_3\} & \{v_1, v_3\} \\ \{v_2, v_3\} & \{v_1, v_2, v_3\} \end{array}$$

De même, l'ensemble de tous les ensembles parents possibles de v_0 ***not*** *contenant v_3, noté $\mathcal{T}_{v_0}^{-v_3}$, est également de taille $2^{n-2} = 4$ et se compose de ce qui suit :*

$$\begin{array}{cc} \{\} & \{v_2\} \\ \{v_1\} & \{v_1, v_2\} \end{array}$$

Nous désignons l'ensemble des ensembles parents candidats de v_0 contenant v_3 par $D^{v_3}(v_0)$, et il n'a qu'un seul élément, $\{\{v_3\}\}$, et est de taille $d_{v_0}^{v_3} = 1$.

L'ensemble des ensembles parents candidats de v_0 ***not*** *contenant v_3 est noté $D^{-v_3}(v_0)$, dans lequel on a $\{\{v_1\}, \{v_2\}, \{v_1, v_2\}\}$, et qui est de taille $d_{v_0}^{-v_3} = 3$.*

Maintenant que nous avons toutes ces informations, calculons les entropies $H(\mathcal{T}_{v_0}^{v_3})$ et $H(\mathcal{T}_{v_0}^{-v_3})$ en utilisant l'équation 5.2a :

$$\begin{aligned} H(\mathcal{T}_{v_0}^{v_3}) &= -\frac{d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|}\log\frac{d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|} - \frac{|\mathcal{T}_{v_0}^{v_3}| - d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|}\log\frac{|\mathcal{T}_{v_0}^{v_3}| - d_{v_0}^{v_3}}{|\mathcal{T}_{v_0}^{v_3}|} \\ &= -\frac{1}{4}\log\frac{1}{4} - \frac{3}{4}\log\frac{3}{4} \\ &= 0.811 \\ H(\mathcal{T}_{v_0}^{-v_3}) &= -\frac{d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|}\log\frac{d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|} - \frac{|\mathcal{T}_{v_0}^{-v_3}| - d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|}\log\frac{|\mathcal{T}_{v_0}^{-v_3}| - d_{v_0}^{-v_3}}{|\mathcal{T}_{v_0}^{-v_3}|} \\ &= -\frac{3}{4}\log\frac{3}{4} - \frac{1}{4}\log\frac{1}{4} \\ &= 0.811 \end{aligned}$$

Maintenant que nous avons toutes les composantes, nous pouvons calculer $IG(\mathcal{T}_{v_0}, v_3)$:

$$\begin{aligned} IG(\mathcal{T}_{v_0}, v_3) &= H(\mathcal{T}_{v_0}) - \frac{|\mathcal{T}_{v_0}^{v_3}|}{|\mathcal{T}_{v_0}|} H(\mathcal{T}_{v_0}^{v_3}) - \frac{|\mathcal{T}_{v_0}^{-v_3}|}{|\mathcal{T}|} H(\mathcal{T}_{v_0}^{-v_3}) \\ &= 1 - \frac{4}{8} 0,811 - \frac{4}{8} 0,811 \\ N &= 0.189 \end{aligned}$$

Voici la formule générale IG pour une variable $v \in V$ en termes BNSL :

$$IG(\mathcal{T}_v, u) = H(\mathcal{T}_v) - \frac{|\mathcal{T}_v^u|}{|\mathcal{T}_v|} H(\mathcal{T}_v^u) - \frac{|\mathcal{T}_v^{-u}|}{|\mathcal{T}_v|} H(\mathcal{T}_v^{-u}) \tag{5.3a}$$

$$= H(\mathcal{T}_v) \tag{5.3b}$$

$$- \frac{|\mathcal{T}_v^u|}{|\mathcal{T}_v|} \left[- \frac{d_v^u}{|\mathcal{T}_v^u|} \log \frac{d_v^u}{|\mathcal{T}_v^u|} - \frac{|\mathcal{T}_v^u| - d_v^u}{|\mathcal{T}_v^u|} \log \frac{|\mathcal{T}_v^u| - d_v^u}{|\mathcal{T}_v^u|} \right] \tag{5.3c}$$

$$- \frac{|\mathcal{T}_v^{-u}|}{|\mathcal{T}_v|} \left[- \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} - \frac{|\mathcal{T}_v^{-u}| - d_v^{-u}}{|\mathcal{T}_v^{-u}|} \log \frac{|\mathcal{T}_v^{-u}| - d_v^{-u}}{|\mathcal{T}_v^{-u}|} \right] \tag{5.3d}$$

$$= H(\mathcal{T}_v) \tag{5.3e}$$

$$+ \frac{1}{|\mathcal{T}_v|} \left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|} + (|\mathcal{T}_v^u| - d_v^u) \log \frac{|\mathcal{T}_v^u| - d_v^u}{|\mathcal{T}_v^u|} \right] \tag{5.3f}$$

$$+ \frac{1}{|\mathcal{T}_v|} \left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} + (|\mathcal{T}_v^{-u}| - d_v^{-u}) \log \frac{|\mathcal{T}_v^{-u}| - d_v^{-u}}{|\mathcal{T}_v^{-u}|} \right] \tag{5.3g}$$

5.7.3 Simplifier la Formule IG

La taille de l'ensemble d'apprentissage $\mathcal{T}_v$ pour une variable $v \in V$ est de 2^{n-1} (et de façon similaire 2^{n-2} pour $\mathcal{T}_v^u$ et $\mathcal{T}_v^{-u}$). Au fur et à mesure que n augmente, non seulement il devient encombrant de calculer ces termes avec les puissances de 2 présentes dans la formule IG 5.3a, mais en plus nous finissons par dépasser la précision des types à virgule flottante. Afin de surmonter ce problème, nous exploitons le fait que pour un nombre suffisamment grand de n, $\frac{2^{n-2}-d_v^u}{2^{n-2}} \simeq 1$ et $\log \frac{2^{n-2}-d_v^u}{2^{n-2}} \simeq 0$. Par conséquent, le deuxième terme à l'intérieur des grandes parenthèses dans 5.3f et 5.3g est réduit à zéro.

Réfléchissons maintenant à la façon dont nous pouvons factoriser le terme $\frac{1}{|\mathcal{T}_v|} = \frac{1}{2^{n-1}}$ à l'extérieur des grandes parenthèses dans 5.3f et 5.3g. Si nous considérons $2^{n-1} \log \frac{2^{n-1}-d_v^u}{2^{n-1}}$, nous ne pouvons pas nécessairement dire qu'elle est approximativement égale à 0, car le logarithme va à 0 au même rythme que 2^{n-1} croît, de sorte que leurs taux de croissance s'annulent. C'est peut-être encore vrai, mais ce n'est pas aussi évident. Quoi qu'il en soit, nous avons déjà réussi à simplifier un peu la formule, alors écrivons-la avant de poursuivre :

$$IG(\mathcal{T}_v, u) = H(\mathcal{T}_v) \tag{5.4}$$
$$+ \frac{1}{|\mathcal{T}_v|}\left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|}\right] \tag{5.5}$$
$$+ \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] \tag{5.6}$$

Le fait que nous ayons besoin de l'IG d'une variable juste pour le comparer à celui d'une autre nous permet de factoriser $\frac{1}{2^{n-1}}$. Supposons qu'en plus de u, nous ayons une autre variable (attribut) w. Lorsque nous voulons vérifier si $IG(\mathcal{T}_v, u)$ est supérieur à $IG(\mathcal{T}_v, w)$, nous avons l'inégalité suivante :

$$IG(\mathcal{T}_v, u) - IG(\mathcal{T}_v, w) \overset{?}{\geq} 0 \iff \tag{5.7}$$
$$H(\mathcal{T}_v) + \frac{1}{|\mathcal{T}_v|}\left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|}\right] + \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] \tag{5.8}$$
$$-H(\mathcal{T}_v) - \frac{1}{|\mathcal{T}_v|}\left[d_v^w \log \frac{d_v^w}{|\mathcal{T}_v^w|}\right] - \frac{1}{|\mathcal{T}_v|}\left[d_v^{-w} \log \frac{d_v^{-w}}{|\mathcal{T}_v^{-w}|}\right] \overset{?}{\geq} 0 \iff \tag{5.9}$$
$$\frac{1}{|\mathcal{T}_v|}\left[d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|}\right] + \frac{1}{|\mathcal{T}_v|}\left[d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|}\right] \tag{5.10}$$
$$-\frac{1}{|\mathcal{T}_v|}\left[d_v^w \log \frac{d_v^w}{|\mathcal{T}_v^w|}\right] - \frac{1}{|\mathcal{T}_v|}\left[d_v^{-w} \log \frac{d_v^{-w}}{|\mathcal{T}_v^{-w}|}\right] \overset{?}{\geq} 0 \iff \tag{5.11}$$
$$d_v^u \log \frac{d_v^u}{|\mathcal{T}_v^u|} + d_v^{-u} \log \frac{d_v^{-u}}{|\mathcal{T}_v^{-u}|} \tag{5.12}$$
$$-d_v^w \log \frac{d_v^w}{|\mathcal{T}_v^w|} - d_v^{-w} \log \frac{d_v^{-w}}{|\mathcal{T}_v^{-w}|} \overset{?}{\geq} 0 \iff \tag{5.13}$$
$$d_v^u \log d_v^u - d_v^u \log |\mathcal{T}_v^u| + d_v^{-u} \log d_v^{-u} - d_v^{-u} \log |\mathcal{T}_v^{-u}| \tag{5.14}$$
$$-d_v^w \log d_v^w + d_v^w \log |\mathcal{T}_v^w| - d_v^{-w} \log d_v^{-w} + d_v^{-w} \log |\mathcal{T}_v^{-w}| \overset{?}{\geq} 0 \iff \tag{5.15}$$

$$d_v^u \log d_v^u - d_v^u(n-2) + d_v^{-u} \log d_v^{-u} - d_v^{-u}(n-2) \tag{5.16}$$
$$-d_v^w \log d_v^w + d_v^w(n-2) - d_v^{-w} \log d_v^{-w} + d_v^{-w}(n-2) \overset{?}{\geq} 0 \iff \tag{5.17}$$

En bref, nous choisissons u plutôt que w lorsque la condition suivante est remplie :

$$d_v^u \log d_v^u - d_v^u(n-2) + d_v^{-u} \log d_v^{-u} - d_v^{-u}(n-2) \geq$$
$$d_v^w \log d_v^w - d_v^w(n-2) + d_v^{-w} \log d_v^{-w} - d_v^{-w}(n-2)$$

Quant au seuil que nous utilisons pour décider quand nous passons à cette comparaison simplifiée, nous voulons qu'il soit suffisamment petit pour éviter le débordement et en même temps suffisamment grand pour que la simplification ne fausse pas trop les résultats. Par exemple, si $n = 5$, nous ne voulons pas simplifier. Nous avons considéré que $n = 32$ était un choix sûr, car il est loin des limites d'un nombre en double précision IEEE 754.

5.8 Travail Connexe

Les BDT ne sont certainement pas le seul moyen de représenter les variables dont les domaines sont des ensembles d'ensembles. Nous pouvons citer les variables ensemblistes dans CSP [Gervet 1997], ainsi que les diagrammes de décision (DDs) tels que les diagrammes de décision binaires (BDDs) [Akers 1978], les diagrammes de décision algébriques (ADDs) [Bahar *et al.* 1997], les BDDs supprimés par zéro (ZDDs) [Minato 1993] comme les outils les plus utilisés.

Le domaine d'une variable d'ensemble est traditionnellement défini par deux ensembles (I, O), en indiquant que le domaine contient tous les ensembles s qui contiennent I, c'est-à-dire $I \subseteq s$, et dont l'intersection avec O est l'ensemble vide, c'est-à-dire $s \cap O = \{\}$. Les variables d'ensemble sont pratiques pour implémenter les contraintes d'inclusion et de disjonction. En revanche, elles ne sont pas très flexibles lorsqu'il s'agit de traiter les contraintes de cardinalité et lexicographiques, c'est pourquoi il y a eu des tentatives pour faire face à cette limitation. L'une d'entre elles est l'ordre Length-Lex, qui fournit un codage direct des informations de cardinalité et de lexicographie, en ordonnant totalement un domaine d'ensembles d'abord par la longueur, puis lexicographiquement : [Gervet & Van Hentenryck 2006]. Pour nous, la principale limitation des variables ensemblistes est que leur domaine est défini par deux ensembles seulement et non comme une énumération de toutes les valeurs possibles, ce qui entraîne une perte de précision importante. Une telle description des domaines serait probablement compatible avec notre algorithme GAC, cependant, elle serait problématique pour `lowerBoundRC`, où nous devons connaître le coût réduit de chaque valeur individuelle.

Les ensembles peuvent également être représentés par des DD. Les DD ont été introduits pour la première fois sous la forme d'un programme de décision binaire [Lee 1959], qui est un type de programme informatique représentant un circuit de commutation. Il avait déjà été démontré que les circuits de commutation peuvent être représentés en algèbre booléenne [Shannon 1938], il s'agissait donc d'une alternative de représentation plus propice au calcul des sorties des circuits de commutation. Les programmes de décision binaires et les BDD sont en quelque sorte équivalents, mais les BDD ont l'avantage de permettre une représentation graphique. Les BDD sont en fait une forme plus compacte des BDT : on peut transformer un BDT en un BDD en parcourant le BDT de bas en haut et en réduisant deux nœuds dont les sous-arbres sont isomorphes en un seul nœud. La taille du BDD est également affectée par l'ordre des variables, ce qui a conduit à la conception

d'une heuristique efficace pour trouver les "bons" ordres : [Rice & Kulhari 2008].

En ce qui concerne les ADD et les ZDD, ils constituent un type particulier de BDD. Les ZDD ont un ordonnancement fixe des variables et, grâce à leur règle de réduction spéciale, ils sont meilleurs pour la compression des ensembles épars. Les ADD, quant à eux, diffèrent des BDD par leur capacité à représenter des fonctions arbitraires à valeurs réelles, contrairement aux BDD qui représentent des fonctions booléennes.

Bien que les DD soient des outils puissants pour divers problèmes combinatoires [Bergman *et al.* 2016, Bergman *et al.* 2014b, Bergman *et al.* 2014a], ils ne sont malheureusement pas adaptés à notre cas. Nous avons besoin d'une structure de données qui nous permette d'ajouter et de restaurer dynamiquement des valeurs pendant la procédure de recherche. Cette question pourrait techniquement être abordée en créant un DDs à partir de zéro à chaque nœud de l'arbre de recherche, par rapport à l'état actuel des domaines. C'est ce que nous avons initialement essayé, mais les résultats préliminaires étaient loin d'être prometteurs. L'autre option consiste à utiliser les algorithmes d'union/intersection existants pour les BDD et les ZDD. Ils peuvent donner de bons résultats, surtout si les DD sont beaucoup plus petits que le domaine de la variable, mais ils risquent de doubler la taille du DD à chaque suppression de valeur. Ils rendent également la gestion de la mémoire plus compliquée, puisque nous devons ajouter ou supprimer des nœuds lorsque nous descendons une branche et lors du backtracking.

Même si la modification des DD était quelque peu pratique, il est toujours nécessaire d'associer un score à chaque valeur de cette structure de données. Les DD sont techniquement capables de le faire, mais modifier les DD chaque fois que nous rendons le coût réduit d'un ensemble de valeurs 0 ferait potentiellement exploser leur taille. Ces deux exigences nous obligent à avoir un chemin unique pour chaque valeur, ce qui n'est le cas dans aucun des DD mentionnés ci-dessus, mais l'est dans les BDT.

5.9 Résultats Expérimentaux

Dans ce chapitre, nous intégrons les domaines basés sur les BDT ainsi que l'heuristique de gain d'information dans ELSA ^{cluster}. Nous effectuons une analyse comparative entre les solveurs suivants :

- GOBNILP v1.6.3 utilisant SCIP v3.2.1 avec CPLEX v12.8.0
- CPBayes v1.1 compatible avec $n > 64$
- ELSA ^{cluster}
- ELSA ^{BDT} : utilise les BDT dans tous les algorithmes mentionnés dans ce chapitre

- ELSA ^{IG} : diffère de ELSA ^{BDT} en utilisant l'heuristique de gain d'information pendant la construction des BDTs

Nous disposons du même ensemble de 51 jeux de données moyens ($|V| < 64$) et 18 grands ($64 \leq |V| < 128$) que dans le chapitre précédent. Pour les jeux de données moyens, ainsi que pour `kdd.ts`, `kdd.test` et `kdd.valid`, nous avions une limite de temps CPU d'une heure, et une limite de temps CPU de 10 heures pour les 15 grands jeux de données restants. Pour l'analyse, nous nous concentrons sur les mêmes 17 jeux de données que dans le chapitre 4, et tous les résultats dans le même format pour les 69 jeux de données peuvent être consultés à la fin de ce chapitre dans les tableaux 5.6, 5.7, et 5.8.

5.9.1 Comparaison Générale Entre tous les Solveurs

Dans le tableau 5.3, nous voyons que ELSA ^{cluster} reste le vainqueur en termes de nombre d'jeux de données résolus de manière optimale dans le temps imparti. Cependant, nous atteignons le temps d'exécution le plus court pour plusieurs jeux de données tels que `steel_BIC`, `kdd.test`, `jester.ts`, et `bnetflix.test`.

5.9.2 Comparaison avec ELSA ^{cluster}

Dans le tableau 5.4, nous voyons qu'avec les BDT (avec ou sans IG), le temps d'exécution est soit amélioré, soit pas particulièrement affecté pour les jeux de données moyens. Cependant, pour les grands jeux de données, ce n'est malheureusement pas le cas. Au lieu de cela, pour ces jeux de données, nous avons l'un des deux cas extrêmes : le temps d'exécution est soit considérablement amélioré, par exemple, `jester.ts` et tous les `bnetflix`, soit considérablement détérioré, par exemple, `accidents.ts`, `plants.valid` et `accidents.test`. Il semble que, certains jeux de données bénéficient tellement de l'accélération des requêtes par les BDT, que la charge de travail apportée par les BDT est annulée. Pour d'autres, en revanche, le gain n'est pas suffisant pour compenser les coûts opérationnels.

5.9.3 Comparaison entre ELSA ^{BDT} et ELSA ^{IG}

Dans le tableau 5.5, nous comparons ces deux solveurs en termes de nombre moyen de nœuds des BDT (noté 🌳), de temps total nécessaire pour les créer (noté ⏱), ainsi que de temps total et de temps de recherche entre parenthèses. Ces résultats nous montrent que le gain d'information est effectivement un algorithme heuristique, c'est-à-dire que nous ne pouvons pas espérer une amélioration constante avec lui. Pour donner un exemple, l'utilisation du gain d'information ne nous garantit pas un BDT plus petit, et il peut même doubler le nombre de nœuds comme c'est le cas pour `flag.BDe`. Elle augmente aussi clairement le temps nécessaire à la construction, mais cette augmentation est microscopique par rapport à l'échelle de ces jeux de données. Précisément, la plus grande différence de temps, qui est de

39,6 secondes, se produit pour `accidents.test`, et avec n'importe quel solveur il nous faut au moins 1 heure pour résoudre ce jeu de données.

Table 5.3: Comparaison de ELSA ^{BDT} et ELSA ^{IG} contre GOBNILP, CPBayes, et ELSA ^{cluster}, en termes de temps d'exécution total en secondes. La limite de temps pour les jeux de données bleus à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. Pour CPBayes ainsi que pour toutes les variantes de ELSA, nous indiquons entre parenthèses le temps passé en recherche, une fois le prétraitement terminé. Le signe † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
carpo100_BIC	60	423	0.6	79.8 (27.8)	52.6 (0.1)	47.2 (0.2)	**47.0 (0.1)**
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	34.4 (1.0)	34.3 (4.5)	34.0 (4.3)
flag_BDe	29	1324	4.36	14.9 (14)	1.0 (0.2)	1.1 (0.3)	**1.0 (0.2)**
wdbc_BIC	31	14613	99.77	390.6 (337.5)	56 (2.4)	**52.7 (2.7)**	59 (2.7)
kdd.ts	64	43584	**327.63**	†	1452.3 (274.6)	1395.1 (254.7)	1379.5 (239.7)
steel_BIC	28	93026	†	981.2 (931.5)	124.2 (71.8)	**114 (62.7)**	125.6 (76.2)
kdd.test	64	152873	1521.73	†	1594.3 (224.4)	1451.0 (108.5)	**1438.7 (90.7)**
mushroom_BDe	23	438185	†	**131.2 (5.7)**	182.6 (58.9)	147.0 (24.1)	149.8 (26.8)
bnetflix.ts	100	446406	†	**673.4 (456.2)**	2103.1 (1900.9)	839.2 (622.3)	897.0 (680.5)
plants.test	111	520148	†	†	**28049.6 (26312.9)**	†	†
jester.ts	100	531961	†	†	21550.5 (21003.7)	21473.0 (20742.7)	**16015.3 (15374.2)**
accidents.ts	100	568160	**1273.96**	†	2302.2 (930.0)	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	†	†	**17801.6 (14080.2)**	†	†
jester.test	100	770950	†	†	**30186.8 (29455)**	†	31207.7 (30339.1)
bnetflix.test	100	1103968	†	2725.6 (2475.1)	10333.1 (10096.5)	1900.8 (1650.4)	**1869.9 (1619.8)**
bnetflix.valid	111	1325818	†	**511.7 (146.8)**	10871.7 (10527.7)	2113.7 (1749.9)	2112.3 (1746.6)
accidents.test	100	1425966	4975.64	†	**3641.7 (680.7)**	17727.7 (14560.9)	20589.3 (17196)

Table 5.4: Comparaison de ELSA ^{cluster}, ELSA ^{BDT} et ELSA ^{IG} en termes de temps d'exécution total et de temps de recherche en secondes. La limite de temps pour les jeux de données bleus à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
carpo100_BIC	60	423	52.6 (0.1)	47.2 (0.2)	**47.0 (0.1)**
alarm1000_BIC	37	1002	34.4 (1.0)	34.3 (4.5)	**34.0 (4.3)**
flag_BDe	29	1324	1.0 (0.2)	1.1 (0.3)	**1.0 (0.2)**
wdbc_BIC	31	14613	56.0 (2.4)	**52.7 (2.7)**	59 (2.7)
kdd.ts	64	43584	1452.3 (274.6)	1395.1 (254.7)	**1379.5 (239.7)**
steel_BIC	28	93026	124.2 (71.8)	**114.0 (62.7)**	125.6 (76.2)
kdd.test	64	152873	1594.3 (224.4)	1451.0 (108.5)	**1438.7 (90.7)**
mushroom_BDe	23	438185	182.6 (58.9)	**147.0 (24.1)**	149.8 (26.8)
bnetflix.ts	100	446406	2103.1 (1900.9)	**839.2 (622.3)**	897.0 (680.5)
plants.test	111	520148	**28049.6 (26312.9)**	†	†
jester.ts	100	531961	21550.5 (21003.7)	21473.0 (20742.7)	**16015.3 (15374.2)**
accidents.ts	100	568160	**2302.2 (930.0)**	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	**17801.6 (14080.2)**	†	†
jester.test	100	770950	**30186.8 (29455)**	†	31207.7 (30339.1)
bnetflix.test	100	1103968	10333.1 (10096.5)	1900.8 (1650.4)	**1869.9 (1619.8)**
bnetflix.valid	111	1325818	10871.7 (10527.7)	2113.7 (1749.9)	**2112.3 (1746.6)**
accidents.test	100	1425966	**3641.7 (680.7)**	17727.7 (14560.9)	20589.3 (17196)

Table 5.5: Comparaison de ELSA ^{BDT} et ELSA ^{IG} en termes de taille moyenne des BDT (🌳) et de temps total passé à créer tous les BDT (⏱). La limite de temps pour les jeux de données bleus à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
carpo100_BIC	60	423	**201.5** 0.0	47.2 (0.2)	375.7 0.0	47.0 (0.1)
alarm1000_BIC	37	1002	**403.8** 0.0	34.3 (4.5)	758.8 0.0	34.0 (4.3)
flag_BDe	29	1324	**450.4** 0.0	1.1 (0.3)	986.2 0.0	1.0 (0.2)
wdbc_BIC	31	14613	**3835.9** 0.0	52.7 (2.7)	7890.6 0.1	59 (2.7)
kdd.ts	64	43584	17665.8 0.1	1395.1 (254.7)	**15955.9** 0.4	**1379.5 (239.7)**
steel_BIC	28	93026	**16073.3** 0.1	**114.0 (62.7)**	40174.0 0.3	125.6 (76.2)
kdd.test	64	152873	62548.6 0.4	1451.0 (108.5)	**51480.7** 1.4	**1438.7 (90.7)**
mushroom_BDe	23	438185	**64656.3** 0.2	147.0 (24.1)	123941.3 1.1	149.8 (26.8)
bnetflix.ts	100	446406	160500.9 3.0	**839.2 (622.3)**	**154816.8** 11.0	897.0 (680.5)
plants.test	111	520148	**114790.5** 1.7	†	138612.7 7.7	†
jester.ts	100	531961	158942.0 3.7	21473.0 (20742.7)	**114897.2** 13.9	**16015.3 (15374.2)**
accidents.ts	100	568160	222061.2 5.5	13921.2 (12328.3)	**208849.9** 19.8	**13555.3 (12008.6)**
plants.valid	111	684141	**149565.8** 2.3	†	178192.2 10.1	†
jester.test	100	770950	206566.4 4.8	†	**155281.7** 20.3	**31207.7 (30339.1)**
bnetflix.test	100	1103968	359724.9 7.1	1900.8 (1650.4)	**348861.9** 28.7	**1869.9 (1619.8)**
bnetflix.valid	111	1325818	417987.3 8.4	2113.7 (1749.9)	**413199.0** 34.7	2112.3 (1746.6)

Table 5.5 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
accidents.test	100	1425966	534753.2	12.2	**17727.7 (14560.9)**	**519021.7**	51.8	20589.3 (17196)

5.10 Conclusion

Dans ce chapitre, nous avons tenté d'aborder un problème inhérent aux problèmes BNSL : les domaines de grande taille. Les résultats nous montrent que les BDT peuvent être des outils puissants pour résoudre peut-être pas tous, mais certainement certains grands problèmes BNSL. Dans le cadre de travaux futurs, on pourrait envisager une mise en œuvre plus efficace des BDT, afin de voir s'ils peuvent être bénéfiques pour une collection étendue de jeux de données. D'autre part, l'heuristique de gain d'information, surtout pour certains jeux de données comme nous l'avons vu, peut entraîner des BDT plus grands. Dans ce cas, il est envisageable de désactiver l'heuristique de gain d'information à une certaine profondeur, ou de ne pas l'utiliser du tout pour certaines variables.

Table 5.6: Comparaison de ELSA ^{BDT} et ELSA ^{IG} avec GOBNILP, CPBayes, et ELSA ^{cluster}, en termes de temps d'exécution total en secondes. La limite de temps pour les jeux de données bleus à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. Pour CPBayes ainsi que pour toutes les variantes de ELSA, nous indiquons entre parenthèses le temps passé en recherche, une fois le prétraitement terminé. Le signe † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
mildew1000_BIC	35	126	0.37	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
lympho_BIC	19	143	0.16	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
water1000_BIC	32	159	0.16	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
mildew1000_BDe	35	166	0.5	0.5 (0.2)	0.4 (0.1)	0.5 (0.1)	0.5 (0.1)
hailfinder100_BIC	56	167	0.43	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
tumour_BIC	18	219	0.18	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
barley1000_BIC	48	244	0.29	1.5 (1.2)	1.6 (1.4)	2.8 (2.6)	2.7 (2.4)
shuttle_BIC	10	264	1.81	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hepatitis_BIC	20	266	0.63	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
tumour_BDe	18	274	0.39	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
lung-cancer_BIC	57	292	0.57	3.8 (2.7)	1.1 (0.0)	1.1 (0.1)	1.1 (0.0)
lympho_BDe	19	345	0.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
hailfinder500_BIC	56	418	0.36	3.4 (2.6)	1.2 (0.5)	1.7 (0.9)	1.6 (0.8)
carpo100_BIC	60	423	0.6	79.8 (27.8)	52.6 (0.1)	47.2 (0.2)	**47.0 (0.1)**
horse_BDe	28	490	0.71	1.2 (0.7)	0.6 (0.0)	0.6 (0.0)	0.5 (0.0)
horse_BIC	28	490	0.99	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
hepatitis_BDe	20	501	0.93	0.4 (0.0)	0.5 (0.0)	0.4 (0.0)	0.4 (0.0)
insurance1000_BIC	27	506	0.51	32.4 (0.0)	32.8 (0.0)	29.4 (0.0)	29.6 (0.0)
adult_BIC	15	547	0.33	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

Table 5.6 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
zoo_BIC	17	554	0.42	0.1 (0.0)	0.1 (0.0)	0.0 (0.0)	0.0 (0.0)
spectf_BIC	45	610	1.52	4.3 (3.5)	0.8 (0.0)	0.9 (0.2)	0.8 (0.1)
sponge_BIC	45	618	1.45	5 (3.2)	1.8 (0.0)	1.8 (0.0)	1.8 (0.0)
flag_BIC	29	741	1.28	1.1 (0.6)	0.4 (0.0)	0.5 (0.1)	0.5 (0.0)
vehicle_BIC	19	763	1.48	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
adult_BDe	15	768	0.67	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
insurance1000_BDe	27	792	0.7	34.0 (0.0)	34.3 (0.0)	31.3 (0.0)	31.8 (0.0)
shuttle_BDe	10	812	4.03	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
bands_BIC	39	892	3.96	0.7 (0.4)	0.6 (0.2)	1.0 (0.6)	0.8 (0.5)
alarm1000_BIC	37	1002	**1.2**	178.9 (145.7)	34.4 (1.0)	34.3 (4.5)	34.0 (4.3)
segment_BIC	20	1053	4.7	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
flag_BDe	29	1324	4.36	14.9 (14)	1.0 (0.2)	1.1 (0.3)	**1.0 (0.2)**
voting_BIC	17	1848	1.87	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
voting_BDe	17	1940	1.91	0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
autos_BIC	26	2391	**11.83**	17.2 (0.0)	19.2 (0.0)	17.1 (0.0)	17 (0.0)
zoo_BDe	17	2855	19.02	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
vehicle_BDe	19	3121	5.68	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
letter_BIC	17	4443	3.95	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
soybean_BIC	36	5926	**40.19**	45.2 (1.5)	50.8 (3)	46.2 (5.6)	45.4 (4.7)
msnbc.ts	17	6298	8.61	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
segment_BDe	20	6491	29.15	0.4 (0.0)	0.5 (0.1)	0.5 (0.1)	0.5 (0.1)
mushroom_BIC	23	13025	1561.82	4.1 (0.0)	4.3 (0.2)	4 (0.1)	4.1 (0.1)
wdbc_BIC	31	14613	99.77	390.6 (337.5)	56 (2.4)	**52.7 (2.7)**	59 (2.7)
msnbc.test	17	16594	57.96	0.4 (0.0)	0.5 (0.1)	0.4 (0.1)	0.5 (0.1)

Table 5.6 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
letter_BDe	17	18841	111.73	0.4 (0.0)	0.7 (0.3)	0.5 (0.1)	0.6 (0.1)
msnbc.valid	17	20673	23.14	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)	0.6 (0.0)
nltcs.ts	16	22156	15.3	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
autos_BDe	26	25238	715.89	146.6 (0.1)	145.8 (0.8)	**144.0 (0.6)**	149.4 (0.7)
kdd.ts	64	43584	**327.63**	†	1452.3 (274.6)	1395.1 (254.7)	1379.5 (239.7)
nltcs.valid	16	47097	40.91	0.9 (0.0)	1.0 (0.1)	1.2 (0.3)	1.2 (0.4)
nltcs.test	16	48303	56.58	1.0 (0.0)	1.2 (0.3)	1.3 (0.3)	1.3 (0.4)
steel_BIC	28	93026	†	981.2 (931.5)	124.2 (71.8)	**114 (62.7)**	125.6 (76.2)
kdd.test	64	152873	1521.73	†	1594.3 (224.4)	1451.0 (108.5)	**1438.7 (90.7)**
kdd.valid	64	197546	†	†	†	†	†
mushroom_BDe	23	438185	†	**131.2 (5.7)**	182.6 (58.9)	147.0 (24.1)	149.8 (26.8)
plants.ts	69	164640	†	†	†	†	†
baudio.ts	69	371117	†	†	†	†	†
bnetflix.ts	100	446406	†	**673.4 (456.2)**	2103.1 (1900.9)	839.2 (622.3)	897.0 (680.5)
plants.test	111	520148	†	†	**28049.6 (26312.9)**	†	†
jester.ts	100	531961	†	†	21550.5 (21003.7)	21473.0 (20742.7)	**16015.3 (15374.2)**
accidents.ts	100	568160	**1273.96**	†	2302.2 (930.0)	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	†	†	**17801.6 (14080.2)**	†	†
jester.test	100	770950	†	†	**30186.8 (29455)**	†	31207.7 (30339.1)
baudio.test	100	1016403	†	†	†	†	†
bnetflix.test	100	1103968	†	2725.6 (2475.1)	10333.1 (10096.5)	1900.8 (1650.4)	**1869.9 (1619.8)**
baudio.valid	69	1235928	†	†	†	†	†
bnetflix.valid	111	1325818	†	**511.7 (146.8)**	10871.7 (10527.7)	2113.7 (1749.9)	2112.3 (1746.6)
accidents.test	100	1425966	4975.64	†	**3641.7 (680.7)**	17727.7 (14560.9)	20589.3 (17196)

Table 5.6 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	GOBNILP	CPBayes	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
jester.valid	100	1463335	†	†	†	†	†
accidents.valid	100	1617862	†	†	†	†	†

Table 5.7: Comparaison de ELSA ^{cluster}, ELSA ^{BDT} et ELSA ^{IG} en termes de temps d'exécution total et de temps de recherche en secondes. La limite de temps pour les jeux de données bleus à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
mildew1000_BIC	35	126	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
lympho_BIC	19	143	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
water1000_BIC	32	159	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
mildew1000_BDe	35	166	0.4 (0.1)	0.5 (0.1)	0.5 (0.1)
hailfinder100_BIC	56	167	0.5 (0.0)	0.5 (0.0)	0.5 (0.0)
tumour_BIC	18	219	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
barley1000_BIC	48	244	1.6 (1.4)	2.8 (2.6)	2.7 (2.4)
shuttle_BIC	10	264	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hepatitis_BIC	20	266	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
tumour_BDe	18	274	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
lung-cancer_BIC	57	292	1.1 (0.0)	1.1 (0.1)	1.1 (0.0)
lympho_BDe	19	345	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
hailfinder500_BIC	56	418	1.2 (0.5)	1.7 (0.9)	1.6 (0.8)
carpo100_BIC	60	423	52.6 (0.1)	47.2 (0.2)	47.0 (0.1)
horse_BDe	28	490	0.6 (0.0)	0.6 (0.0)	0.5 (0.0)
horse_BIC	28	490	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
hepatitis_BDe	20	501	0.5 (0.0)	0.4 (0.0)	0.4 (0.0)
insurance1000_BIC	27	506	32.8 (0.0)	29.4 (0.0)	29.6 (0.0)
adult_BIC	15	547	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
zoo_BIC	17	554	0.1 (0.0)	0.0 (0.0)	0.0 (0.0)
spectf_BIC	45	610	0.8 (0.0)	0.9 (0.2)	0.8 (0.1)

Table 5.7 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
sponge_BIC	45	618	1.8 (0.0)	1.8 (0.0)	1.8 (0.0)
flag_BIC	29	741	0.4 (0.0)	0.5 (0.1)	0.5 (0.0)
vehicle_BIC	19	763	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
adult_BDe	15	768	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
insurance1000_BDe	27	792	34.3 (0.0)	31.3 (0.0)	31.8 (0.0)
shuttle_BDe	10	812	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
bands_BIC	39	892	0.6 (0.2)	1.0 (0.6)	0.8 (0.5)
alarm1000_BIC	37	1002	34.4 (1.0)	34.3 (4.5)	34.0 (4.3)
segment_BIC	20	1053	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
flag_BDe	29	1324	1.0 (0.2)	1.1 (0.3)	1.0 (0.2)
voting_BIC	17	1848	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
voting_BDe	17	1940	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
autos_BIC	26	2391	19.2 (0.0)	17.1 (0.0)	17 (0.0)
zoo_BDe	17	2855	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
vehicle_BDe	19	3121	0.3 (0.0)	0.3 (0.0)	0.3 (0.0)
letter_BIC	17	4443	0.1 (0.0)	0.1 (0.0)	0.1 (0.0)
soybean_BIC	36	5926	50.8 (3)	46.2 (5.6)	45.4 (4.7)
msnbc.ts	17	6298	0.2 (0.0)	0.2 (0.0)	0.2 (0.0)
segment_BDe	20	6491	0.5 (0.1)	0.5 (0.1)	0.5 (0.1)
mushroom_BIC	23	13025	4.3 (0.2)	4 (0.1)	4.1 (0.1)
wdbc_BIC	31	14613	56.0 (2.4)	52.7 (2.7)	59 (2.7)
msnbc.test	17	16594	0.5 (0.1)	0.4 (0.1)	0.5 (0.1)
letter_BDe	17	18841	0.7 (0.3)	0.5 (0.1)	0.6 (0.1)
msnbc.valid	17	20673	0.5 (0.0)	0.5 (0.0)	0.6 (0.0)

Table 5.7 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{cluster}	ELSA ^{BDT}	ELSA ^{IG}
nltcs.ts	16	22156	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
autos_BDe	26	25238	145.8 (0.8)	144 (0.6)	149.4 (0.7)
kdd.ts	64	43584	1452.3 (274.6)	1395.1 (254.7)	1379.5 (239.7)
nltcs.valid	16	47097	1.0 (0.1)	1.2 (0.3)	1.2 (0.4)
nltcs.test	16	48303	1.2 (0.3)	1.3 (0.3)	1.3 (0.4)
steel_BIC	28	93026	124.2 (71.8)	114.0 (62.7)	125.6 (76.2)
kdd.test	64	152873	1594.3 (224.4)	1451.0 (108.5)	1438.7 (90.7)
kdd.valid	64	197546	†	†	†
mushroom_BDe	23	438185	182.6 (58.9)	147.0 (24.1)	149.8 (26.8)
plants.ts	69	164640	†	†	†
baudio.ts	69	371117	†	†	†
bnetflix.ts	100	446406	2103.1 (1900.9)	839.2 (622.3)	897.0 (680.5)
plants.test	111	520148	28049.6 (26312.9)	†	†
jester.ts	100	531961	21550.5 (21003.7)	21473.0 (20742.7)	16015.3 (15374.2)
accidents.ts	100	568160	2302.2 (930.0)	13921.2 (12328.3)	13555.3 (12008.6)
plants.valid	111	684141	17801.6 (14080.2)	†	†
jester.test	100	770950	30186.8 (29455)	†	31207.7 (30339.1)
baudio.test	100	1016403	†	†	†
bnetflix.test	100	1103968	10333.1 (10096.5)	1900.8 (1650.4)	1869.9 (1619.8)
baudio.valid	69	1235928	†	†	†
bnetflix.valid	111	1325818	10871.7 (10527.7)	2113.7 (1749.9)	2112.3 (1746.6)
accidents.test	100	1425966	3641.7 (680.7)	17727.7 (14560.9)	20589.3 (17196)
jester.valid	100	1463335	†	†	†
accidents.valid	100	1617862	†	†	†

Table 5.8: Comparaison de ELSA ^{BDT} et ELSA ^{IG} en termes de taille moyenne des BDT (🌳) et de temps total passé à créer tous les BDT (⏱). La limite de temps pour les jeux de données bleus à la fin est de 10 heures, et pour le reste elle est de 1 heure. Les jeux de données sont triés par taille de domaine totale croissante pour chaque catégorie de limite de temps. † indique un dépassement de délai.

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}			ELSA ^{IG}		
mildew1000_BIC	35	126	🌳 82.9	⏱ 0.0	0.2 (0.0)	🌳 114.9	⏱ 0.0	0.2 (0.0)
lympho_BIC	19	143	🌳 77.3	⏱ 0.0	0.1 (0.0)	🌳 112.9	⏱ 0.0	0.1 (0.0)
water1000_BIC	32	159	🌳 92.6	⏱ 0.0	0.2 (0.0)	🌳 144.2	⏱ 0.0	0.2 (0.0)
mildew1000_BDe	35	166	🌳 92.4	⏱ 0.0	0.5 (0.1)	🌳 147.7	⏱ 0.0	0.5 (0.1)
hailfinder100_BIC	56	167	🌳 103.3	⏱ 0.0	0.5 (0.0)	🌳 154.0	⏱ 0.0	0.5 (0.0)
tumour_BIC	18	219	🌳 82.9	⏱ 0.0	0.1 (0.0)	🌳 161.6	⏱ 0.0	0.1 (0.0)
barley1000_BIC	48	244	🌳 151.8	⏱ 0.0	2.8 (2.6)	🌳 223.0	⏱ 0.0	2.7 (2.4)
shuttle_BIC	10	264	🌳 52.6	⏱ 0.0	0.0 (0.0)	🌳 122.4	⏱ 0.0	0.0 (0.0)
hepatitis_BIC	20	266	🌳 111.8	⏱ 0.0	0.3 (0.0)	🌳 189.1	⏱ 0.0	0.3 (0.0)
tumour_BDe	18	274	🌳 100.5	⏱ 0.0	0.1 (0.0)	🌳 198.3	⏱ 0.0	0.1 (0.0)
lung-cancer_BIC	57	292	🌳 160.8	⏱ 0.0	1.1 (0.1)	🌳 263.4	⏱ 0.0	1.1 (0.0)
lympho_BDe	19	345	🌳 160.7	⏱ 0.0	0.2 (0.0)	🌳 249.4	⏱ 0.0	0.2 (0.0)
hailfinder500_BIC	56	418	🌳 191.3	⏱ 0.0	1.7 (0.9)	🌳 367.3	⏱ 0.0	1.6 (0.8)
carpo100_BIC	60	423	🌳 201.5	⏱ 0.0	47.2 (0.2)	🌳 375.7	⏱ 0.0	47.0 (0.1)
horse_BDe	28	490	🌳 203.7	⏱ 0.0	0.6 (0.0)	🌳 395.9	⏱ 0.0	0.5 (0.0)
horse_BIC	28	490	🌳 219.5	⏱ 0.0	0.4 (0.0)	🌳 391.0	⏱ 0.0	0.4 (0.0)
hepatitis_BDe	20	501	🌳 197.3	⏱ 0.0	0.4 (0.0)	🌳 339.6	⏱ 0.0	0.4 (0.0)

Table 5.8 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
insurance1000_BIC	27	506	226.1 0.0	29.4 (0.0)	394.0 0.0	29.6 (0.0)
adult_BIC	15	547	151.5 0.0	0.0 (0.0)	293.5 0.0	0.0 (0.0)
zoo_BIC	17	554	234.9 0.0	0.0 (0.0)	344.0 0.0	0.0 (0.0)
spectf_BIC	45	610	305.4 0.0	0.9 (0.2)	504.4 0.0	0.8 (0.1)
sponge_BIC	45	618	344.5 0.0	1.8 (0.0)	474.9 0.0	1.8 (0.0)
flag_BIC	29	741	296.9 0.0	0.5 (0.1)	572.2 0.0	0.5 (0.0)
vehicle_BIC	19	763	246.3 0.0	0.2 (0.0)	442.8 0.0	0.2 (0.0)
adult_BDe	15	768	194.6 0.0	0.0 (0.0)	386.3 0.0	0.0 (0.0)
insurance1000_BDe	27	792	307.7 0.0	31.3 (0.0)	600.9 0.0	31.8 (0.0)
shuttle_BDe	10	812	130.6 0.0	0.0 (0.0)	247.4 0.0	0.0 (0.0)
bands_BIC	39	892	394.2 0.0	1.0 (0.6)	661.7 0.0	0.8 (0.5)
alarm1000_BIC	37	1002	403.8 0.0	34.3 (4.5)	758.8 0.0	34.0 (4.3)
segment_BIC	20	1053	186.1 0.0	0.2 (0.0)	648.3 0.0	0.2 (0.0)
flag_BDe	29	1324	450.4 0.0	1.1 (0.3)	986.2 0.0	1.0 (0.2)
voting_BIC	17	1848	457.2 0.0	0.1 (0.0)	847.2 0.0	0.1 (0.0)
voting_BDe	17	1940	450.1 0.0	0.0 (0.0)	893.1 0.0	0.0 (0.0)
autos_BIC	26	2391	686.6 0.0	17.1 (0.0)	1560.6 0.0	17 (0.0)
zoo_BDe	17	2855	923.8 0.0	0.1 (0.0)	1418.7 0.0	0.1 (0.0)
vehicle_BDe	19	3121	848.3 0.0	0.3 (0.0)	1529.5 0.0	0.3 (0.0)

Table 5.8 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
letter_BIC	17	4443	938.5	0.0	0.1 (0.0)	1736.7	0.0	0.1 (0.0)
soybean_BIC	36	5926	1687.3	0.0	46.2 (5.6)	3391.9	0.0	45.4 (4.7)
msnbc.ts	17	6298	1610.7	0.0	0.2 (0.0)	2441.8	0.0	0.2 (0.0)
segment_BDe	20	6491	809.6	0.0	0.5 (0.1)	3167.4	0.0	0.5 (0.1)
mushroom_BIC	23	13025	3871.2	0.0	4 (0.1)	5787.1	0.0	4.1 (0.1)
wdbc_BIC	31	14613	3835.9	0.0	52.7 (2.7)	7890.6	0.1	59 (2.7)
msnbc.test	17	16594	3674.2	0.0	0.4 (0.1)	4974.8	0.0	0.5 (0.1)
letter_BDe	17	18841	2771.6	0.0	0.5 (0.1)	6012.7	0.0	0.6 (0.1)
msnbc.valid	17	20673	4399.8	0.0	0.5 (0.0)	5897.5	0.0	0.6 (0.0)
nltcs.ts	16	22156	3934.4	0.0	0.4 (0.0)	5465.6	0.0	0.4 (0.0)
autos_BDe	26	25238	5943.5	0.0	144 (0.6)	14218.0	0.1	149.4 (0.7)
kdd.ts	64	43584	17665.8	0.1	1395.1 (254.7)	15955.9	0.4	1379.5 (239.7)
nltcs.valid	16	47097	7368.1	0.0	1.2 (0.3)	8458.7	0.1	1.2 (0.4)
nltcs.test	16	48303	7397.0	0.0	1.3 (0.3)	8504.7	0.1	1.3 (0.4)
steel_BIC	28	93026	16073.3	0.1	114.0 (62.7)	40174.0	0.3	125.6 (76.2)
kdd.test	64	152873	62548.6	0.4	1451.0 (108.5)	51480.7	1.4	1438.7 (90.7)
kdd.valid	64	197546	84590.1	0.5	†	61594.7	1.8	†
mushroom_BDe	23	438185	64656.3	0.2	147.0 (24.1)	123941.3	1.1	149.8 (26.8)
plants.ts	69	164640	39108.0	0.5	†	48971.957	2.95	†

Table 5.8 continued from previous page

Instance	$\|V\|$	$\sum \|ps(v)\|$	ELSA ^{BDT}		ELSA ^{IG}	
baudio.ts	69	371117	84552.2 1.9	†	89714.630 10.25	†
bnetflix.ts	100	446406	160500.9 3.0	839.2 (622.3)	154816.8 11.0	897.0 (680.5)
plants.test	111	520148	114790.5 1.7	†	138612.7 7.7	†
jester.ts	100	531961	158942.0 3.7	21473.0 (20742.7)	114897.2 13.9	16015.3 (15374.2)
accidents.ts	100	568160	222061.2 5.5	13921.2 (12328.3)	208849.9 19.8	13555.3 (12008.6)
plants.valid	111	684141	149565.8 2.3	†	178192.2 10.1	†
jester.test	100	770950	206566.4 4.8	†	155281.7 20.3	31207.7 (30339.1)
baudio.test	100	1016403	215677.6 5.0	†	267653.9 26.6	†
bnetflix.test	100	1103968	359724.9 7.1	1900.8 (1650.4)	348861.9 28.7	1869.9 (1619.8)
baudio.valid	69	1235928	242546.6 5.9	†	350898.0 36.1	†
bnetflix.valid	111	1325818	417987.3 8.4	2113.7 (1749.9)	413199.0 34.7	2112.3 (1746.6)
accidents.test	100	1425966	534753.2 12.2	17727.7 (14560.9)	519021.7 51.8	20589.3 (17196)
jester.valid	100	1463335	342123.1 8.0	†	335342.3 38.8	†
accidents.valid	100	1617862	622315.5 14	†	560968.0 50.5	†

CHAPTER 6

VAC-Intégralité

Contents

6.1 Introduction

Dans ce chapitre, nous passons de l'*apprentissage* des réseaux Bayésiens (RB) à l'*raisonnement* avec les modèles graphiques (MG), ce qui inclut également le raisonnement avec les RB. En raison du grand intérêt porté à l'exécution de l'explication la plus probable (EPP) sur les RB, des efforts ont également été déployés pour développer des méthodes efficaces à cet effet. Cela nous amène finalement à la minimisation des coûts dans les problèmes de satisfaction de contraintes pondérées (WCSP), puisque nous savons que chaque WCSP correspond à un réseau de fonction de coût (CFN) et que l'EPM dans les BN est équivalent à la minimisation des coûts dans les CFN sous une transformation $-\log$. Ici, nous apportons des contributions à l'état de l'art des requêtes de minimisation sur les WCSPs. L'objectif principal de ce chapitre est de trouver des moyens de *utiliser plus efficacement l'algorithme VAC (Cohérence d'Arc Virtuelle) dans les solveurs branch-and-bound pour WCSPs*. Ces améliorations nous permettent d'utiliser VAC non seulement dans le prétraitement, mais aussi pendant la recherche.

L'outil technique que nous utilisons pour améliorer VAC est basé sur une propriété développée pour la première fois dans le contexte de la méthode COMBILP [Haller *et al.* 2018], qui résout la relaxation linéaire et utilise la solution de la relaxation pour trouver une décomposition du problème : la partie "facile" correspondant à l'ensemble des variables intégrales dans la relaxation linéaire, et une partie combinatoire contenant les variables affectées de valeurs fractionnaires. Il procède ensuite à la résolution exacte du sous-ensemble combinatoire et si cette solution peut être combinée avec la partie facile sans entraîner de coût supplémentaire, il déclare l'optimalité. Sinon, il déplace certaines variables de la partie facile vers la partie combinatoire et itère. De façon cruciale, ils identifient les variables intégrales en identifiant une condition appelée Strict Arc Consistency (Strict AC) sur la solution duale produite, et peuvent donc être utilisés avec des solveurs LP duaux approximatifs, comme VAC et TRW-S, qui peuvent produire une solution LP duale sous-optimale pour laquelle il n'existe pas de solution primale correspondante.

Nous apportons plusieurs contributions ici. Tout d'abord, nous assouplissons le AC strict, la condition que COMBILP utilise pour détecter l'intégralité. Nous montrons dans la section 6.2 que la condition relaxée admet de plus grands ensembles de variables intégrales et que ces ensembles sont en un sens maximaux. Deuxièmement, nous montrons qu'une classe de points fixes d'un solveur LP comme VAC implique un ensemble spécifique de variables intégrales quelle que soit la solution duale qu'il trouve, même lorsque ces variables ne satisfont pas la condition AC stricte dans cette solution. Cela évite de devoir biaiser le solveur LP vers des solutions qui contiennent des variables strictement AC. Sur le plan pratique, nous introduisons deux techniques simples qui exploitent cette propriété au sein d'un solveur branch-and-bound. La première, présentée dans la section 6.4, modifie l'heuristique de branchement pour éviter le branchement sur les variables strictement AC, car il est peu probable que cela soit informatif. La seconde, présentée dans la section 6.5, est une variante de l'heuristique RINS bien connue en programmation en nombres entiers ([Danna *et al.* 2005]), qui suppose de manière optimiste que l'ensemble des variables Strictement AC auxquelles sont attribuées leurs valeurs intégrales apparaissent effectivement dans la solution optimale et résout un sous-problème restreint pour aider à identifier rapidement les solutions de haute qualité. Dans la section 6.6, nous montrons que l'intégration de ces techniques dans le solveur ToulBar2 [Cooper *et al.* 2010] améliore considérablement les performances par rapport à l'état de l'art dans certaines familles d'instances.

6.2 Cohérence d'Arc Stricte et VAC-intégralité

Savchynskyy et al. ont présenté la Strict Arc Consistency ([Savchynskyy *et al.* 2013]) comme un moyen de partitionner un WCSP en une partie "facile", qui peut être résolue exactement par un solveur LP et une partie combinatoire "dure".

Tout au long de ce chapitre, nous supposons que tous les WCSP sont Node Consistent (NC), c'est-à-dire que $\min_{T \in \ell(V_S)} c_{V_S}(T) = 0$ pour tous les scopes V_S.

Sinon, le WCSP peut être trivialement reparamétré pour le rendre NC et augmenter la borne inférieure.

Definition 6.1 (Cohérence d'arc stricte [Savchynskyy *et al.* 2013])**.** *Une variable $v_i \in V$ est Strictly Arc Consistent (Strictly AC) si elle satisfait aux deux conditions suivantes :*

1. *Il existe une valeur unique $a \in D(v_i)$ telle que $c_{v_i}(a) = 0$,*
2. *Il existe un unique tuple $\{(v_i, a), (v_j, c)\}$ qui satisfait $c_{v_i v_j}(a, c) = 0 \ \ \forall c_{v_i v_j} \in C$.*

La valeur a est appelée la valeur strictement AC de v_i.

Example 6.2. *Soit $P_1 = \langle V, D, C, k\rangle$ un WCSP où $V = \{v_1, v_2\}$, $D = \{\{a, b\}, \{c, d\}\}$, C est l'ensemble des fonctions de coût et k est une borne supérieure arbitraire. Cette instance est présentée comme un réseau de fonctions de coût dans la Figure 6.1a.*

La variable v_1 est Strictement AC puisqu'elle a une valeur unique $a \in D(v_1)$ avec $c_{v_1}(a) = 0$ et qu'il existe un unique tuple à coût nul $\{(v_1, a), (v_2, c)\}$ avec son voisin v_2.

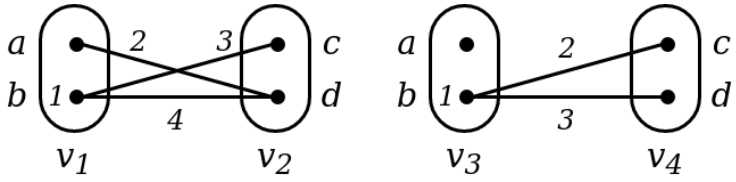
Étant donné un WCSP P et un sous-ensemble V_S de ses variables tel que toutes les variables dans V_S sont Strictement AC, nous pouvons résoudre P restreint à V_S exactement en assignant la valeur Strictement AC à chaque variable. Cette propriété donne une partition naturelle d'un WCSP en l'ensemble des variables Strictement AC et le reste. Cette partition a été utilisée par Savchynskyy et al. [Savchynskyy *et al.* 2013] et dans un algorithme raffiné introduit plus tard [Haller *et al.* 2018]. Ces algorithmes exploitent la solvabilité du sous-ensemble de variables Strictement AC et n'ont besoin de résoudre que le sous-ensemble plus petit non Strictement AC à l'aide d'un solveur combinatoire.

Notre première contribution ici est de noter que la propriété Strict AC est plus forte que nécessaire. En particulier, nous pouvons affaiblir la deuxième condition comme suit :

Definition 6.3 (VAC-intégralité)**.** *Une variable $v_i \in V$ est VAC-intégrale si elle satisfait aux deux conditions suivantes :*

1. *Il existe une valeur unique $a \in D(v_i)$ telle que $c_{v_i}(a) = 0$,*
2. *Il existe au moins un tuple $\{(v_i, a), (v_j, c)\}$ tel que $c_{v_i v_j}(a, c) + c_{v_j}(c) = 0$ $\forall c_{v_i v_j} \in C$.*

La valeur a est la valeur VAC-intégrale de v_i.



(a) P_1 (b) P_2

Figure 6.1: Deux WCSPs P_1 and P_2.

Example 6.4. *Soit $P_2 = \langle V, D, C, k\rangle$ un WCSP où $V = \{v_3, v_4\}$, $D = \{\{a, b\}, \{c, d\}\}$, C est l'ensemble des fonctions de coût et k est une borne supérieure arbitraire. Cette instance est présentée comme un réseau de fonctions de coût dans la Figure 6.1b.*

La variable v_3 est VAC-intégrale puisqu'elle a une valeur unique $a \in D(v_3)$ avec $c_{v_3}(a) = 0$ et avec son voisin v_4, il existe au moins un tuple de coût nul satisfaisant la deuxième condition de la définition 6.3, qui sont $\{(v_3, a), (v_4, c)\}$ et $\{(v_3, a), (v_4, d)\}$.

La différence entre VAC-intégralité[1] et AC strict est que dans VAC-intégrale, la deuxième condition exige que la valeur témoin apparaisse dans au moins un plutôt qu'exactement un tuple de coût 0 dans chaque contrainte d'incident. Le sous-ensemble VAC-intégrale d'un WCSP conserve la principale propriété de l'AC strict, à savoir qu'il est exactement soluble par inspection et que sa solution optimale a un coût 0. La solution optimale, comme dans l'AC strict, attribue simplement à chaque variable VAC-intégrale sa valeur VAC-intégrale. Par définition, cette solution a un coût 0.

Puisque la VAC-intégralité est une relaxation de la CA stricte, chaque variable de la CA stricte est également VAC-intégrale, comme c'est le cas pour v_1 dans l'instance P_1 de la Figure 6.1a. L'inverse ne tient pas, c'est-à-dire que v_3 dans l'instance P_2 de la Figure 6.1b n'est pas Strictement AC. Cependant, cela ne vaut que pour les instances qui sont à un point fixe VAC. P_1 et P_2 sont tous deux VAC puisque la fermeture AC de leur $Bool$ est non vide.

Proposition 6.5. *Si un WCSP P est VAC et qu'une variable v_i est Strictement AC alors il est aussi VAC-intégrale.*

Proof. Soit v_i une variable Strictement AC dans P, et soit v_j l'un de ses voisins. Soit $\{(v_i, a), (v_j, c)\}$ l'unique tuple qui satisfait $c_{v_iv_j}(a, c) = 0$. Supposons que $c_{v_j}(c) > 0$. Alors nous pouvons étendre $c_{v_j}(c)$ à $c_{v_iv_j}(k, b)$ pour tous les $k \in D(v_i)$. Étant donné que $\{(v_i, a), (v_j, c)\}$ était l'unique tuple avec $c_{v_iv_j}(a, c) = 0$, maintenant tous les $c_{v_iv_j}$ sont non nuls après cette extension. Dans ce cas, nous pouvons projeter un coût binaire non nul sur $c_{v_i}(a)$, l'unique valeur avec un coût

[1]Nous changeons le nom de la propriété de "cohérence", qui implique un algorithme qui atteint ladite cohérence, à "intégralité". L'ajout du terme VAC deviendra clair après la Proposition 6.7.

unaire nul. Après cette projection, toutes les valeurs du domaine de v_i ont un coût unaire non nul ; ce qui signifie que nous pouvons projeter un certain coût sur $c_{v_i}(a)$ et augmenter la borne inférieure. Cependant, ceci est contradictoire car P était déjà VAC. En conséquence, nous savons maintenant que, pour une variable strictement AC v_i dans un WCSP P qui est VAC, nous avons non seulement $c_{v_iv_j}(a,c) = 0$, mais aussi $c_{v_j}(c) = 0$. Par conséquent, étant donné que pour une variable VAC-intégrale v_i il peut y avoir un ou plusieurs tuples avec $c_{v_iv_j}(a,c) + c_{v_j}(c) = 0$, l'ensemble des variables VAC-intégrale est au moins aussi grand que l'ensemble des variables Strictement AC et il inclut toutes les variables Strictement AC. □

Cela nous montre que la VAC-intégralité est une propriété moins restrictive que la cohérence d'arc stricte, ce qui signifie que la cohérence d'arc stricte implique la VAC-intégralité alors que le contraire ne se produit pas.

6.2.1 Lien avec la Programmation Linéaire en Nombres Entiers

Rappelons le LP appelé le polytope local d'un WCSP P du chapitre 2 :

$$\min c_\varnothing + \sum_{v_i \in V, a \in D(v_i)} c_{v_i}(a) x_{v_i a} + \sum_{c_{v_iv_j} \in C, a \in D(v_i), c \in D(v_j)} c_{v_iv_j}(a,c) y_{v_i a, v_j c}$$

s.t.

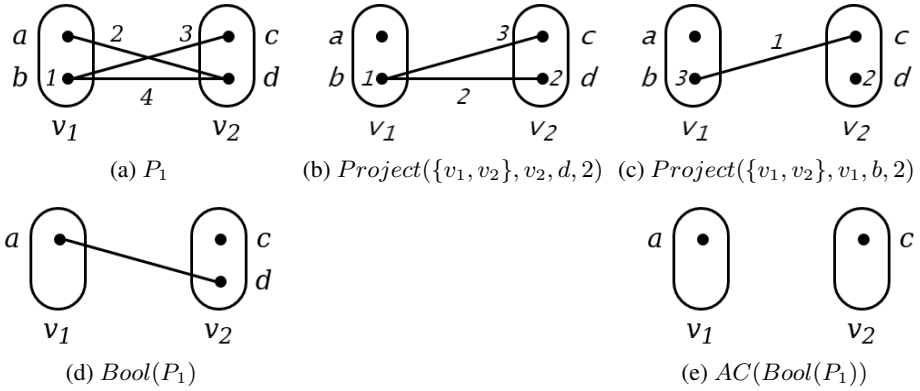
$$\begin{aligned}
&\sum_{a \in D(v_i)} x_{v_i a} = 1 && \forall v_i \in V \\
&x_{v_i a} = \sum_{c \in D(v_j)} y_{v_i a, v_j c} && \forall v_i \in V, a \in D(v_i), c_{v_iv_j} \in C \\
&0 \le x_{v_i a} \le 1 && \forall v_i \in V, a \in D(v_i) \\
&0 \le y_{v_i a, v_j c} \le 1 && \forall c_{v_iv_j} \in C, a \in D(v_i), c \in D(v_j)
\end{aligned} \tag{6.1}$$

Nous savons qu'à partir de la solution optimale de la LP ci-dessus, la reparamétrisation est extraite des *coûts réduits* $r(x_{v_i a})$ et $r(y_{v_i a, v_j b})$ de chaque variable et fonction de coût binaire, respectivement, en fixant $c_{v_i}(a)$ à $r(x_{v_i a})$ et $c_{v_iv_j}(a,c)$ à $r(y_{v_i a, v_j b})$ et en fixant $c_\varnothing$ à l'optimum de la LP (6.1).

Comme pour l'AC strict, VAC-intégralité implique l'intégralité de la solution primale correspondante par relâchement complémentaire.

Proposition 6.6. *Les variables VAC-intégrale dans une solution duale optimale de* (6.1) *correspondent aux variables v pour lesquelles il existe un unique a avec $x_{va} = 1$ et $x_{vb} = 0$ pour $b \neq a$ dans la solution primale optimale correspondante.*

Proof. Étant donné une solution duale optimale du polytope local LP (6.1), pour chaque variable VAC-intégrale v avec une valeur VAC-intégrale a, la solution primale doit avoir $x_{vb} = 0$ pour tous les $b \in D(v) \setminus \{a\}$ par relâchement complémentaire et donc $x_{va} = 1$. □



(a) P_1 (b) $Project(\{v_1, v_2\}, v_2, d, 2)$ (c) $Project(\{v_1, v_2\}, v_1, b, 2)$

(d) $Bool(P_1)$ (e) $AC(Bool(P_1))$

Figure 6.2: Le WCSP P_1 en (a) avec ses différentes reparamétrisations en (b) et (c) obtenues avec des projections de coûts, son $Bool$ en (d) et la fermeture AC de $Bool(P_1)$ en (e).

Notez que dans le cas des solveurs LP dual approximatifs comme VAC et TRW-S, contrairement à OSAC, cette observation ne tient pas : si la solution duale n'est pas optimale, il n'existe pas de solution primale avec le même coût. Nous utilisons plutôt Strict AC et VAC-intégralité comme procurations pour les conditions qui conduiraient à l'intégralité des solutions optimales, tout en conservant la propriété qu'ils admettent des solutions à coût nul.

Une complication avec AC strict et VAC-intégralité est que toute borne inférieure donnée par une solution duale peut en fait être produite par plusieurs solutions duales, mais elles ne donnent pas toutes le même sous-ensemble VAC-intégralité. Une façon de traiter ce problème est de biaiser le solveur LP vers des solutions qui maximisent le sous-ensemble VAC-intégrale [Haller *et al.* 2018]. Nous proposons ici une autre méthode, donnée par l'observation suivante.

Proposition 6.7. *Étant donné un WCSP P qui est VAC et une variable v_i, si dans $AC(Bool(P))$ il tient que $\overline{D}(v_i) = \{a\}$ alors v_i est VAC-intégrale avec la valeur a.*

Proof. Puisque $AC(Bool(P))$ est arc consistent, si une valeur reste dans le domaine de v_i dans $Bool(P)$, elle a un coût unaire 0 dans P et est supportée par des tuples et des valeurs de coût 0 dans toutes les contraintes incidentes. Inversement, si une valeur est supprimée dans $Bool(P)$, soit elle a un coût unaire non nul dans P, soit une certaine quantité non nulle de coût peut être déplacée sur elle ; [Cooper *et al.* 2010]. □

Example 6.8. *Soit $P_1 = \langle V, D, C, k\rangle$ un WCSP où $V = \{v_1, v_2\}$, $D = \{\{a, b\}, \{c, d\}\}$, C est l'ensemble des fonctions de coût et k est une borne supérieure arbitraire. Cette instance est présentée comme un réseau de fonctions de coût dans la Figure 6.2a.*

Les WCSP de la Figure 6.2b et 6.2c sont deux reparamétrisations différentes de P_1 et donc équivalentes à P_1. Elles ont la même borne inférieure $c_\varnothing = 0$ et elles donnent le même coût total pour toute affectation complète $A \in \ell(V)$. Elles sont obtenues séquentiellement en projetant un coût de 2 des fonctions de coût binaires $c_{v_1v_2}(a, d)$ et $c_{v_1v_2}(b, d)$ sur la fonction de coût unaire $c_{v_2}(d)$ (Figure 6.2b), et ensuite, en projetant un coût de 2 des fonctions de coût binaires $c_{v_1v_2}(b, c)$ et $c_{v_1v_2}(b, d)$ sur la fonction de coût unaire $c_{v_1}(b)$ (Figure 6.2c).

Dans P_1, nous sommes en mesure d'identifier uniquement v_1 comme VAC-intégrale alors qu'il existe en fait d'autres reparamétrages qui nous permettraient d'identifier v_2 comme VAC-intégrale également (Figures 6.2b et 6.2c). Cependant, nous n'avons pas besoin de trouver ces reparamétrages car en exécutant l'algorithme VAC sur P_1, qui exécute AC sur $Bool(P_1)$, nous voyons que v_1 et v_2 ont tous deux des domaines singuliers dans $AC(Bool(P_1))$. Par conséquent, ils sont tous deux VAC-intégrale par la Proposition 6.7.

L'effet de la Proposition 6.7 est que la classe des solutions réalisables duales qui ont le même $AC(Bool(P))$ produit le même ensemble de variables VAC-intégrale, même si la plupart de ces solutions ne satisfont pas la Définition 6.3. Par conséquent, cette observation nous permet de construire un sous-ensemble VAC-intégrale plus grand que celui donné en rassemblant les variables qui satisfont la propriété VAC-intégralité étant donné une solution duale. Cela présente l'avantage que nous n'avons pas besoin de modifier le solveur LP pour qu'il soit biaisé vers des solutions duales spécifiques et est facile à utiliser avec VAC, qui maintient explicitement $Bool(P)$. Nous pouvons appliquer le même raisonnement pour trouver des ensembles de variables AC stricts, en utilisant la condition suivante.

Proposition 6.9. *Si un WCSP P est VAC, alors une variable VAC-intégrale v_i est Strictement AC si et seulement si tous ses voisins sont VAC-intégrale.*

Proof. Soit v_i une variable VAC-intégrale.

$\Rightarrow$ Supposons que v_i soit également Strictement AC. Alors, suivant la définition 6.1 et la proposition 6.5, nous savons qu'il existe une valeur unique $a \in D(v_i)$ avec $c_{v_i}(a) = 0$ et un tuple unique $\{(v_i, a), (v_j, c)\}$ qui satisfait $c_{v_iv_j}(a, c) = 0$ et $c_{v_j}(c) = 0$. Étant donné que P est VAC, le $Bool(P)$ est Arc Consistent et il n'y a pas d'élagage supplémentaire à faire. Supposons qu'il existe $c \neq b \in D(v_j)$ tel que $c_{v_j}(c) = 0$. Alors, c devrait apparaître dans le domaine de v_j dans $Bool(P)$ également. Dans ce cas, cependant, c n'aura aucun support dans la variable v_i dans $Bool(P)$, car a est la valeur unique dans le domaine de v_i dans $Bool(P)$ et le tuple $\{(v_i, a), (v_j, c)\}$ est interdit puisqu'il a un coût binaire non nul dans P. Par conséquent, b doit être l'unique valeur dans $D(v_j)$ avec un coût unaire nul. Par conséquent, v_j doit être VAC-intégrale.

$\Leftarrow$ Soit v_j un voisin de v_i. Étant donné que v_i est VAC-intégrale, nous savons qu'il existe au moins un tuple tel que $c_{v_iv_j}(a, c) + c_{v_j}(c) = 0$. Par hypothèse, v_j est VAC-intégrale, cela signifie qu'il a exactement une valeur b telle que $c_{v_j}(c) = 0$. Par conséquent, il ne peut y avoir plus d'un tuple avec $c_{v_iv_j}(a, c) + c_{v_j}(c) = 0$.

Par conséquent, v_i est également Strictement AC lorsque tous ses voisins sont VAC-intégrale. □

6.2.2 Analyse de la Complexité

Il est naturel de se demander si la présence d'un grand sous-ensemble VAC-intégrale rend le problème plus facile à résoudre, dans le sens de la tractabilité à paramètre fixe [Downey & Fellows 2013]. Malheureusement, il s'avère que ce n'est pas le cas. Définissons une classe de WCSPs, dénotée ALMOST-INTEGRAL-WCSP, qui sont VAC avec $n-1$ variables VAC-intégrale. Nous montrons qu'elle est NP-complète, ce qui implique que la WCSP est para-NP-complète pour le paramètre du nombre de variables VAC-intégrale.

Theorem 6.10. ALMOST-INTEGRAL-WCSP *est NP-complet.*

Proof. L'appartenance à NP est évidente puisqu'il s'agit d'une sous-classe de WCSP. Pour la dureté, nous réduisons à partir de WCSP binaire. Soit $P = \langle V, D, C, k\rangle$ un WCSP arbitraire et supposons qu'il est VAC. Nous construisons $P' = \langle V \cup V' \cup \{q\}, D \cup D' \cup D(q), C', k + |C|\rangle$, où V' et D' sont des copies des variables dans V et leurs domaines D, respectivement, y compris les fonctions de coût unaires et q a le domaine $\{a, b\}$ avec $c_q(a) = c_q(b) = 0$. Pour chaque fonction de coût avec portée $\{v_i, v_j\}$ dans C, P' a deux fonctions de coût avec portées $\{v_i, v_j, q\}$ et $\{v_i{}', v_j{}', q\}$.

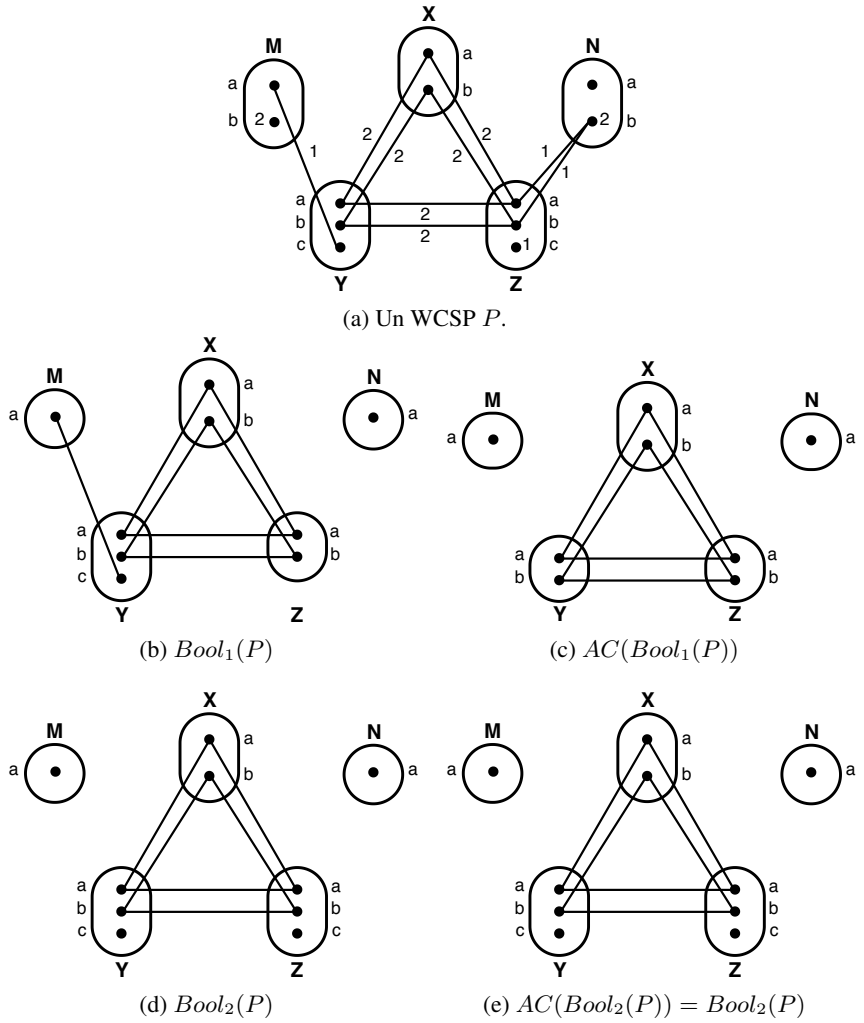
Chaque variable de P possède au moins une valeur de coût unaire 0, puisqu'elle est VAC. Soit cette valeur a pour toutes les variables. Nous définissons les fonctions de coût ternaire comme étant $c_{v_iv_jq}(a, a, a) = 0$, $c_{v_iv_jq}(u, v, a) = k$ lorsque $u \neq a$ ou $v \neq a$, $c_{v_iv_jq}(u, v, b) = c_{v_iv_j}(u, v) + 1$ pour tout u, v. De même, $c_{v_i{}'v_j{}'q}(a, a, b) = 0$, $c_{v_i{}'v_j{}'q}(u, v, b) = k$ lorsque $u \neq a$ ou $v \neq a$, $c_{v_i{}'v_j{}'q}(u, v, a) = c_{v_iv_j}(u, v) + 1$ pour tout u, v.

P' est une instance de ALMOST-INTEGRAL-WCSP avec q la variable non VAC-intégrale. En effet, $c_{v_i}(a) = 0$ pour toutes les variables $v_i \in V \cup V'$ et elle est soutenue par le tuple de coût nul (a, a, a) dans chaque contrainte ternaire. Toutes les autres valeurs apparaissent uniquement dans les tuples ternaires avec un coût non nul ; elles seront donc élaguées dans $AC(Bool(P'))$.

P a une solution optimale de coût c si et seulement si P' a une solution optimale de coût $c + |C|$. En effet, lorsque nous assignons q à a ou b, le problème se décompose en WCSP binaires indépendants sur V et V'. L'un d'entre eux admet l'affectation tout-a, à coût nul, et l'autre est identique à P avec un coût supplémentaire de 1 par fonction de coût.

□

Bien que cette construction utilise des fonctions de coût ternaires, nous pouvons les convertir en binaire en utilisant le codage caché [Bacchus *et al.* 2002]. Cela préserve la cohérence des arcs, donc aussi VAC, et le résultat est donc valable aussi pour les WCSP binaires.



(a) Un WCSP P.

(b) $Bool_1(P)$

(c) $AC(Bool_1(P))$

(d) $Bool_2(P)$

(e) $AC(Bool_2(P)) = Bool_2(P)$

Figure 6.3: Un WCSP P, $Bool(P)$ construit avec $\theta = 1$ et $\theta = 2$ ainsi que leurs fermetures AC.

Ce théorème n'est pas surprenant. Par exemple, Haller *et al.* a mentionné qu'il n'y a aucune raison théorique de s'attendre à ce que les variables Strictement AC maintiennent leur affectation dans toute solution optimale. Ceci fournit simplement une explication formelle spécifique pour cette affirmation.

6.3 VAC Stratifié

La base de toutes les heuristiques que nous présentons dans ce chapitre est l'implémentation de VAC dans ToulBar2 [Cooper *et al.* 2010], qui est limitée aux WCSP binaires. Dans cette implémentation, les coûts binaires non nuls $c_{v_i v_j}(a, c)$ sont stratifiés. Plus précisément, ils sont triés par ordre décroissant et placés dans un nombre fixe

L de seaux. Le coût minimum θ_t de chaque seau $t\ dans\{1,\ldots,L\}$ définit une séquence de seuils $(\theta_1,\ldots,\theta_L)$. À chaque θ_t de 1 à L, il construit le $Bool_{\theta_t}(P)$, qui est le $Bool(P)$ construit en supprimant les valeurs dont le coût unaire est supérieur ou égal à θ_t ainsi qu'en interdisant toutes les relations binaires dont le coût binaire est supérieur ou égal à θ_t, et il itère dessus jusqu'à ce qu'aucun effacement de domaine ne se produise. Après θ_L, il suit un programme géométrique $\theta_{t+1} = \frac{\theta_t}{2}$ jusqu'à $\theta_t = 1$. Le lecteur est renvoyé à [Cooper *et al.* 2010] pour plus de détails.

Pour un θ_t plus petit, $Bool_{\theta_t}(P)$ est plus restreint, c'est-à-dire que la taille des domaines est réduite. L'observation générale et informelle est que l'ensemble des variables VAC-intégrale s'étend à mesure que θ_t devient plus petit, et sature à un certain point, qui est généralement avant $\theta_t = 1$. Cependant, même après cette saturation, la taille des domaines des variables VAC-intégrale ne cesse pas nécessairement de diminuer. Pour l'heuristique que nous présentons dans la section 6.5, nous cherchons à choisir un seuil θ où nous avons un bon compromis entre le nombre de variables VAC-intégrale et les tailles de domaine des variables non-VAC-integral. De cette façon, nous augmentons la taille de la partie facile (VAC-intégrale) et diminuons la complexité de la partie difficile, tout en espérant conserver la plupart (peut-être toutes) les valeurs apparaissant dans la solution optimale. Dans un sens informel, nous considérons que les variables VAC-intégrale qui étaient présentes avec un θ plus élevé sont plus informatives, et plus susceptibles d'apparaître dans une solution optimale. Par exemple, si nous assignons contre la valeur VAC-intégrale pour une valeur élevée θ_t, le coût de la meilleure solution possible est au moins $c_\varnothing + \theta_t$, alors que pour $\theta_{t'} = 1$, le coût de la meilleure solution possible qui est en désaccord avec la valeur VAC-intégrale peut seulement être montré comme étant $c_\varnothing + 1$. Ainsi, plus le θ pour lequel une variable est VAC-intégrale est élevé, moins la relaxation doit être serrée pour que la valeur VAC-intégrale correspondante apparaisse dans une solution optimale.

Example 6.11. *Soit $P = \langle V, D, C, k\rangle$ un WCSP où $V = \{M, X, Y, Z, N\}$, $D = \{\{a,b\},\{a,b\},\{a,b,c\},\{a,b,c\},\{a,b\}\}$, C est l'ensemble des fonctions de coût et k est une borne supérieure arbitraire. Cette instance est présentée comme un réseau de fonctions de coût dans la Figure 6.3a.*

Suite à la Proposition 6.7, nous savons que les variables M et N sont VAC-intégrale avec la valeur a.

Les solutions optimales pour P, qui ont toutes un coût total de 1, sont les suivantes :

- $M = a$, $X = a$, $Y = b$, $Z = c$, $N = a$
- $M = a$, $X = b$, $Y = a$, $Z = c$, $N = a$
- $M = a$, $X = a$, $Y = c$, $Z = b$, $N = a$
- $M = a$, $X = b$, $Y = c$, $Z = a$, $N = a$

Ici, nous devons remarquer deux choses. La première est que M et N sont tous deux affectés à leur valeur VAC-intégrale a dans toutes les solutions optimales.

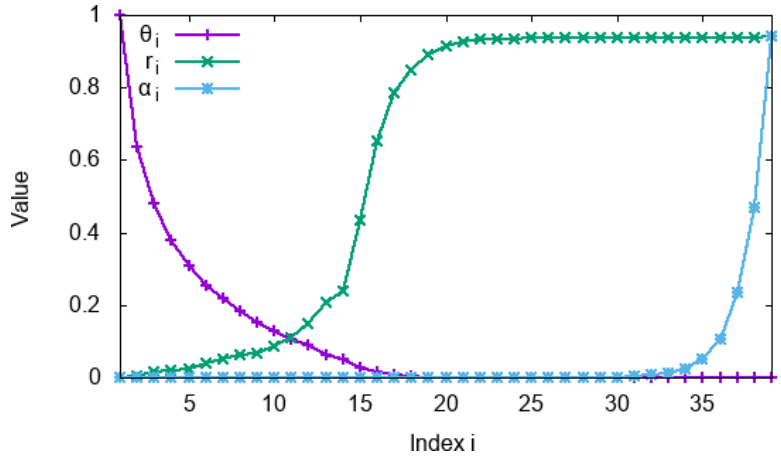


Figure 6.4: Évolution de θ_t, r_t et α_t au fil des itérations.

Cette observation, bien qu'elle n'ait aucune implication théorique comme discuté dans la section 6.2.2, reste utile. La seconde est que soit Y soit Z doivent être assignés à c dans toute solution optimale. La valeur c de ces deux variables n'apparaît pas dans $Bool_1(P)$, cependant, elle apparaît dans $Bool_2(P)$. Cette observation sera pertinente dans la section 6.5.

Dans la Figure 6.4, nous voyons l'évolution des paramètres pertinents sur les itérations VAC $t \in \{1, \ldots, 39\}$ pour l'instance cnd1threeL1_1228061 du benchmark Worms. Il s'agit du seuil θ_t, du rapport des variables VAC-intégrale r_t, et de la valeur $\alpha_t = r_t/\theta_t$ représentée par les courbes violette, verte et bleue, respectivement. Notez que les θ_t sont normalisés, allant de 1 à $\frac{\theta_{39}}{\theta_1}$, afin que le graphique soit lisible. Le rapport des variables VAC-intégrale commence à 0 et va jusqu'à 0,94 dans la dernière itération, lorsque $\theta_{39} = 1$. Notez que le α_t a la même plage que r_t, mais un comportement différent.

6.4 Heuristique de Branchement Basée sur VAC-Intégralité

Pour un algorithme de branchement, l'ordre dans lequel les variables sont assignées a un impact crucial sur les performances. En général, une décision de branchement devrait aider le solveur à élaguer rapidement les sous-arbres qui ne contiennent aucune solution améliorante, en créant des sous-problèmes avec une borne duale accrue dans toutes les branches [Achterberg 2009].

Sur la base de cette observation et de la connexion de la VAC-intégralité à l'intégralité expliquée dans la section 6.2, nous observons que le branchement sur une variable VAC-intégrale v créera une branche qui doit contenir la valeur VAC-intégrale a de v. Puisque a est la seule valeur dans le domaine de v dans $Bool(P)$ et que son coût unaire ne change pas par branchement, $Bool(P\mid_{v=a})$ est identique à $Bool(P)$, donc la borne duale n'est pas améliorée dans cette branche.

Par conséquent, il est logique d'éviter de se brancher sur les variables VAC-intégrale, bien que, en tant qu'heuristique, nous ne pouvons pas nous attendre à ce que ce soit toujours le meilleur choix.

Pour mettre cela en œuvre, nous trouvons l'ensemble des variables VAC-intégrale impliquées par la Proposition 6.7, c'est-à-dire celles qui ont un domaine singleton dans $Bool(P)$ et qui ne permettent de se ramifier que sur le reste. Le choix parmi le reste des variables est fait en utilisant l'heuristique de branchement que le solveur utilise normalement. Dans le cas de ToulBar2, que nous utilisons dans notre implémentation, c'est-à-dire DOM+WDEG [Boussemart *et al.* 2004] ainsi que l'heuristique de dernier conflit [Lecoutre *et al.* 2009].

La motivation derrière cette idée est la possibilité qu'elle nous aide à détecter plus tôt un conflit pendant la procédure de recherche. Si nous supposons que la solution partielle pour la partie VAC-intégrale est cohérente avec la solution optimale, lorsque nous bifurquons sur une variable VAC-intégrale, nous ne pouvons choisir que sa valeur VAC-intégrale qui a un coût de zéro. Dans ce cas, nous ne gagnons aucune nouvelle information. En revanche, si nous insistons pour choisir une valeur dont le coût n'est pas nul, nous savons immédiatement que nous allons dans une mauvaise direction et que nous augmentons ainsi le coût. Par conséquent, le branchement sur une variable non-zero-cost nous donne une chance de trouver la solution optimale, et rend la propagation plus forte. En effet, si la solution partielle de la partie VAC-intégrale n'est pas cohérente avec la solution optimale, le branchement sur des variables non-VAC-intégrale (si la propagation est assez forte) changera l'ensemble des variables VAC-intégrale, ou, au moins, changera les valeurs avec un coût unaire nul. Au contraire, le branchement sur des variables VAC-intégrale n'apportera toujours aucun changement au problème si nous utilisons les valeurs à coût nul.

Lorsqu'il ne reste que des variables VAC-intégrale, nous les assignons toutes en même temps et vérifions que la borne inférieure n'a pas augmenté (voir la terminaison prématurée de VAC dans [Cooper *et al.* 2010]). Si c'est le cas, nous mettons à jour la borne supérieure si une meilleure solution a été trouvée, nous désassignons les variables VAC-intégrale, et nous continuons le branchement avec l'heuristique par défaut.

Pour des raisons d'efficacité, pendant la recherche, EDAC [de Givry *et al.* 2005] est établi avant d'appliquer la propriété VAC dans ToulBar2. Si la propriété VAC ne peut pas être appliquée à un nœud de recherche donné en raison de la fin prématurée de VAC, alors la [de Givry *et al.* 2005] n'est pas disponible pour ce nœud et nous nous basons à nouveau sur l'heuristique de branchement par défaut. Nous avons essayé d'exploiter les dernières informations d'VAC-intégralité valides recueillies le long de la branche de recherche actuelle, mais cela n'a pas amélioré les résultats.

6.4.1 Exploiter les Plus Grandes Affectations Partielles à Coût Nul

La définition 6.3 exige qu'il existe une valeur VAC-intégrale unique $a \in D(v_i)$ pour chaque variable VAC-intégrale. L'affectation partielle de valeurs uniques à

leurs variables correspondantes implique une augmentation de la borne inférieure à coût nul comme dit précédemment. Ainsi, notre heuristique de branchement évitera de se brancher sur ces variables. Nous pourrions rechercher d'autres affectations potentiellement plus importantes avec la même propriété de coût nul. Une façon simple de le faire est de tester une affectation de valeur particulière et de garder les variables qui ne sont pas en *conflict* avec elle, *i.e.,* sans violation de coût liée à elles ou à leurs voisines. Nous choisissons d'abord de tester l'affectation basée sur les *valeurs EAC*, qui sont maintenues par EDAC [Heras & Larrosa 2006]. Une valeur EAC est définie comme une valeur VAC-intégrale mais il n'est pas nécessaire qu'elle soit unique dans le domaine. Si une variable est conservée, cela signifie que sa valeur EAC est totalement compatible, *i.e.,* a coût nul, avec toutes les valeurs EAC de ses voisins. Nous l'appelons une *Full EAC value*. Cette approche peut être combinée avec la VAC-intégralité. En restreignant les valeurs EAC pour qu'elles appartiennent à la fermeture AC de $Bool(P)$ lorsque le problème est rendu VAC, nous nous assurons que toute valeur VAC-intégrale est également Full EAC. L'inverse n'est pas vrai (voir figure 6.1). Ainsi, l'ensemble des valeurs Full EAC peut être plus grand. Nous effectuons le test EAC complet de manière incrémentielle (en utilisant la file d'attente de révision d'une variable basée sur les changements de valeur EAC) à chaque nœud de l'arbre de recherche, avant de choisir la prochaine variable non EAC complète sur laquelle se brancher.

Afin d'améliorer encore cette approche, dès qu'une nouvelle solution est trouvée pendant la recherche, EDAC préférera sélectionner la valeur de la solution correspondante comme valeur EAC pour chaque variable non assignée si cette valeur appartient à l'ensemble actuel des valeurs EAC réalisables. En procédant ainsi, nous pouvons exploiter les affectations partielles à coût nul plus importantes trouvées précédemment pendant la recherche. Remarquez que notre heuristique de branchement est liée à l'algorithme de réparation des min-conflits [Minton *et al.* 1992] car elle ne se branche que sur les variables en conflit par rapport à une affectation donnée. L'exploitation de la meilleure solution trouvée jusqu'à présent pour les heuristiques de valeur s'est avérée performante sur plusieurs problèmes d'optimisation de contraintes lorsqu'elle est combinée avec des redémarrages [Demirovic *et al.* 2018]. Nous utilisons une telle heuristique d'ordre de valeur à l'intérieur de l'algorithme HBFS de ToulBar2 dans toutes nos expériences.

6.5 Relaxation-Aware Sub-Problem Search (RASPS)

L'un des problèmes auxquels est confronté le branch-and-bound, en particulier dans l'ordre de la profondeur, est que sans une bonne borne supérieure, il peut explorer de grandes parties de l'arbre de recherche qui ne contiennent que des solutions de mauvaise qualité.

Ici, nous proposons d'utiliser l'information d'intégralité pour essayer de générer rapidement des solutions proches de l'optimum. Nous décrivons une heuristique primale que nous appelons Relaxation-Aware Sub-Problem Search (RASPS), qui

s'exécute en prétraitement. Nous fixons simplement toutes les variables VAC-intégrale à leurs valeurs, élaguons les valeurs du reste des variables qui sont élaguées dans $AC(Bool(P))$, puis résolvons à l'aide de la borne inférieure de l'EDAC le sous-problème résultant jusqu'à l'optimalité ou jusqu'à ce qu'une limite de ressources soit respectéeDans notre implémentation, nous fixons une borne supérieure de 1000 backtracks pour la résolution du sous-problème.. Afin de choisir l'ensemble des variables VAC-intégrale, nous utilisons les solutions duales construites dans les itérations de VAC avant la dernière, donc examiner $Bool_\theta(P)$ pour un θ approprié.

Bien que l'idée de l'heuristique soit assez simple, la question clé est de choisir la valeur seuil (le θ) (rappelez-vous la section 6.3) pour construire $Bool_\theta(P)$, car cela a un impact sur la qualité de la borne supérieure produite et le temps passé pour cela. Pour déterminer la valeur seuil du RASPS, nous observons les courbes du seuil θ_t, du rapport des variables VAC-intégrale r_t, et de la valeur $\alpha_t = r_t/\theta_t$, recueillies lors des itérations VAC. L'idée est que, une fois que le rapport des variables VAC-intégrale sature, θ_t continue de diminuer. Par conséquent, α_t commence à augmenter plus rapidement, ce qui constitue le point de coupure souhaité. Pour identifier ce point, nous suivons la courbe de α_t sur les itérations VAC et choisissons la valeur seuil lorsque l'angle de la courbe atteint 10 degrés (voir figure 6.4). L'angle de coupure de 10 degrés a été choisi expérimentalement, car nous avons constaté qu'il présentait les meilleures performances et la plus grande robustesse de manière empirique.

Example 6.12. *Soit $P = \langle V, D, C, k\rangle$ un WCSP où $V = \{M, X, Y, Z, N\}$, $D = \{\{a, b\}, \{a, b\}, \{a, b, c\}, \{a, b, c\}, \{a, b\}\}$, C est l'ensemble des fonctions de coût et k est une borne supérieure arbitraire. Cette instance est présentée comme un réseau de fonctions de coût dans la Figure 6.5a. La même instance a été utilisée dans l'exemple 6.11.*

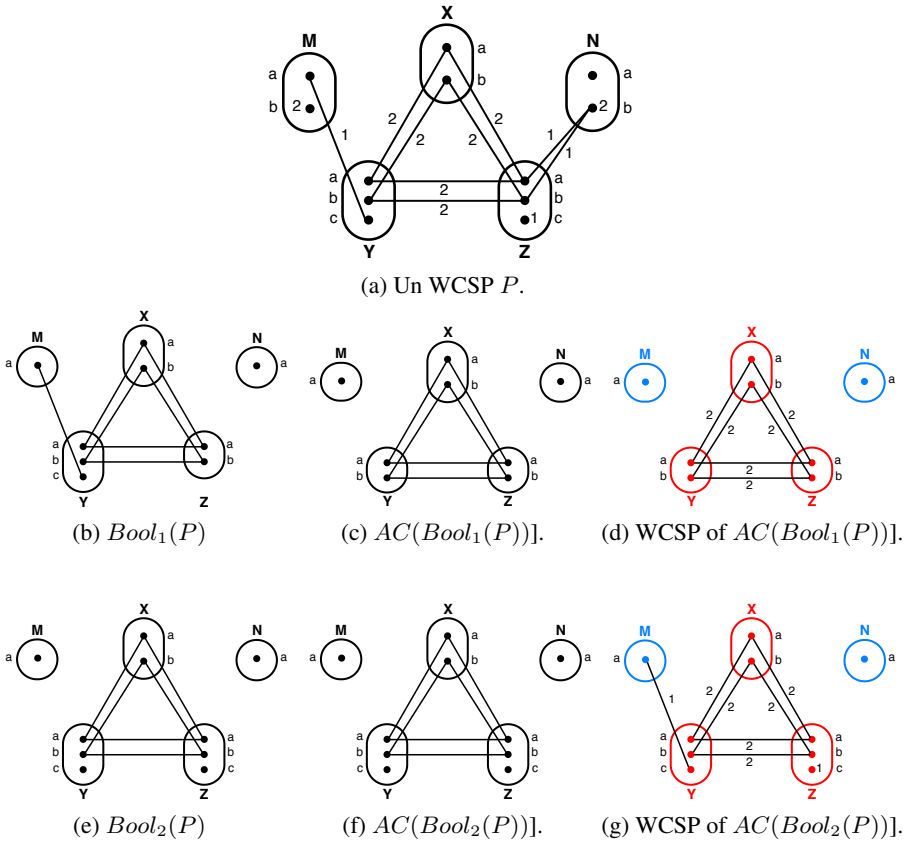
Dans les figures 6.5b et 6.5e, nous voyons $Bool(P)$ construit avec $\theta = 1$ et $\theta = 2$, respectivement, et dans les figures 6.5c et 6.5f nous voyons leurs fermetures AC.

Nous obtenons le sous-WCSP à résoudre par RASPS en faisant deux choses :

1. *Choisir une valeur de seuil θ pour construire $Bool_\theta(P)$.*
2. *Rebranchez les coûts dans le WCSP original P dans $AC(Bool_\theta(P)$*

Une fois que nous avons le sous-WCSP, nous pouvons classer les variables en VAC-intégrale (bleu) et autres (rouge). L'ensemble des variables bleues correspond à la partie intégrale (facile) du problème et elles sont immédiatement affectées à leur valeur VAC-intégrale, tandis que celui des variables rouges correspond à la partie combinatoire (difficile).

Deux sous-problèmes candidats sont visibles sur les figures 6.5d et 6.5g. De toute évidence, $\theta = 2$ est un meilleur choix pour le seuil car nous savons grâce à l'exemple 6.11 que soit Y soit Z doit être assigné à c dans la solution optimale. Avec $\theta = 1$, nous manquons cela.



(a) Un WCSP P.

(b) $Bool_1(P)$ (c) $AC(Bool_1(P))$]. (d) WCSP of $AC(Bool_1(P))$].

(e) $Bool_2(P)$ (f) $AC(Bool_2(P))$]. (g) WCSP of $AC(Bool_2(P))$].

Figure 6.5: Exemples de VAC et de RASPS stratifiés pour deux seuils différents. Les variables bleues sont VAC-intégrale. Voir l'exemple 6.12.

Cette idée est liée à la méthode CombILP de Haller et al. [Haller *et al.* 2018], décrite précédemment, qui utilise la cohérence stricte des arcs pour décomposer le problème en une partition intégrale et une partition combinatoire et les affine successivement jusqu'à ce qu'elles concordent. Pour ce faire, ils identifient les variables AC strictes sur la solution duale produite. La partie combinatoire est résolue exactement et si cette solution peut être combinée avec la partie facile sans entraîner de coût supplémentaire, elle signale l'optimalité. Sinon, il déplace certaines variables de la partie facile vers la partie combinatoire et itère jusqu'à ce qu'il identifie exactement l'ensemble des variables intégrales de la relaxation qui apparaissent dans une solution optimale.

Par rapport à CombILP, RASPS résout un sous-problème combinatoire plus simple en raison de l'ensemble VAC-intégrale plus grand et des domaines élagués restants. Ensuite, il ne vise qu'à produire une bonne borne supérieure initiale et laisse la preuve de l'optimalité au solveur branch-and-bound. Encore plus proche est l'heuristique RINS de Danna et al. [Danna *et al.* 2005]. Elle recherche également des bornes primales en étendant la partie intégrale de la relaxation. Contrairement

à RASPS, elle autorise les valeurs de la solution en place et peut être invoquée dans des nœuds autres que la racine. Cependant, il n'a aucun moyen de faire la distinction entre les variables intégrales comme le fait RASPS avec son choix de $\theta > 1$. Nous avons expérimenté RASPS pendant la recherche mais n'avons pas trouvé jusqu'à présent que cela valait la peine.

6.6 Résultats Expérimentaux

Nous avons implémenté VAC-intégralité et RASPS à l'intérieur de ToulBar2, un solveur WCSP exact de type branch-and-bound open-source en C++ [2]. Tous les calculs ont été effectués sur un seul cœur d'Intel Xeon E5-2680 v3 à 2,50 GHz et 256 Go de RAM avec une limite de temps CPU d'une heure. Aucune borne supérieure initiale n'a été utilisée, comme c'est le cas par défaut du solveur.

6.6.1 Description du Benchmark

Nous avons réalisé des expériences sur des modèles graphiques probabilistes et déterministes provenant de différentes communautés : [Hurley *et al.* 2016]. Nous avons considéré un grand ensemble de 431 instances[3] qui sont toutes binaires. Il comprend 251 instances (170 Enchères, 16 CELAR, 10 ProteinDesign, 55 Entrepôt) de la bibliothèque de fonctions de coût (Cost Function Library), 129 instances (108 DBN, 21 ProteinFolding) du *Probabilistic Inference Challenge* (PIC 2011)[4], 30 instances "Vers" [Kainmueller *et al.* 2014] pour lesquelles CombiLP est à la pointe [Haller *et al.* 2018], et 21 grandes instances de Computational Protein Design (CPD) pour lesquelles ToulBar2 est à la pointe [Allouche *et al.* 2014, Ouali *et al.* 2020]. Nous avons écarté les instances Max-CSP, Max-SAT, de programmation par contraintes (CP) et de vision par ordinateur (CVPR) qui sont soit non pondérées (tous les coûts sont égaux à 1), soit non binaires, soit trop faciles (petit arbre de recherche), soit non résolues par toutes les approches testées, y compris les solveurs MRF et ILP [Hurley *et al.* 2016].

6.6.2 Comparaison avec VAC

Tout d'abord, nous avons comparé nos nouvelles heuristiques avec le VAC par défaut maintenu pendant la recherche (option -A=999999 pour toutes les méthodes testées). Nous avons ignoré Enchères et DBN car ils n'ont pas de variables VAC-intégrale.

Dans la Figure 6.6a, nous montrons un diagramme de dispersion comparant le nombre de retours en arrière entre VAC et VAC exploitant l'heuristique de la variable VAC-intégrale. La taille de l'arbre de recherche est significativement réduite grâce à VAC-intégralité pour la plupart des familles d'instances. Remarquez

[2] `https://github.com/toulbar2/toulbar2`, branche *fural/strictac* de la version 1.0.1.

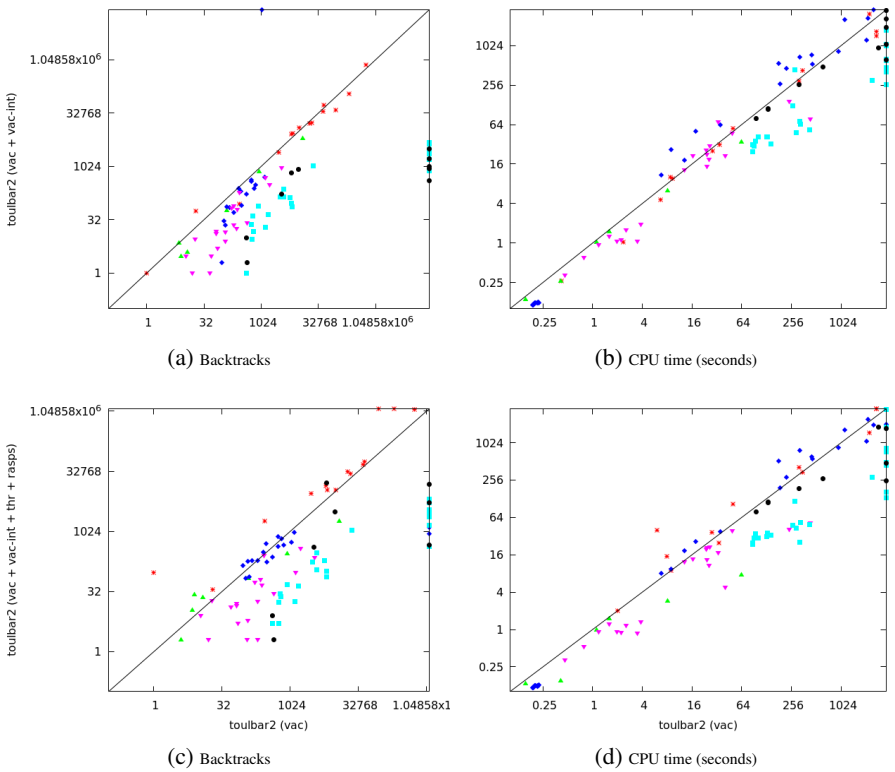
[3] `genoweb.toulouse.inra.fr/~degivry/evalgm`

[4] `www.cs.huji.ac.il/projet/PASCAL`

les axes logarithmiques. L'amélioration en termes de temps CPU (Figure 6.6b) est moins importante mais toujours significative pour CPD, ProteinFolding, Worms, et certaines instances CELAR. Cependant, nous avons trouvé plusieurs instances de l'Entrepôt où il était significativement plus lent en utilisant VAC-intégralité. Dans ce cas, nous avons trouvé que l'explication était un plus grand nombre d'itérations VAC par nœud de recherche (8 fois plus en moyenne) correspondant à de petites améliorations de la borne inférieure à de petites valeurs de seuil (θ proche de 1) qui ne réduisaient pas suffisamment l'arbre de recherche (seulement d'un facteur moyen de $2,2$ sur des instances Warehouse difficiles).

Afin d'éviter de tels cas pathologiques, nous avons placé une limite sur la valeur seuil minimale θ pour les itérations VAC pendant la recherche. Nous avons choisi la même limite que pour RASPS (*e.g.,* θ_{30} pour Worms/cnd1threeL1_1228061).

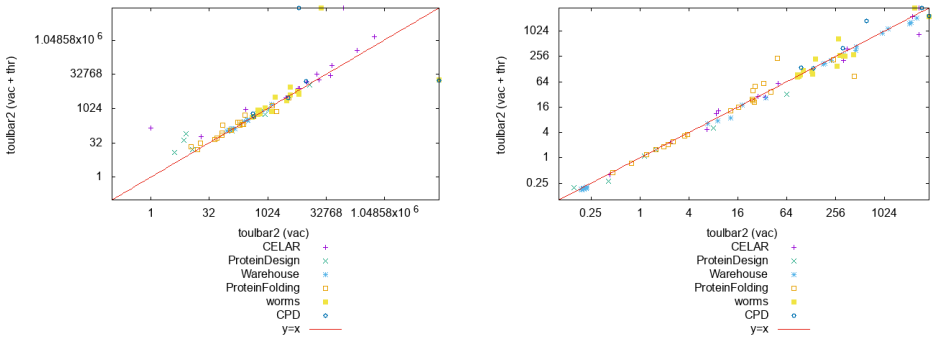
Nous avons constaté que l'utilisation de ce mécanisme de seuil seul accélère la résolution de l'entrepôt et ne détériore pas significativement les résultats dans les autres familles (voir figure 6.7). De plus, nous avons obtenu des résultats cohérents en combinant avec VAC-intégralité, réduisant le nombre de backtracks et le temps



(a) Backtracks (b) CPU time (seconds)

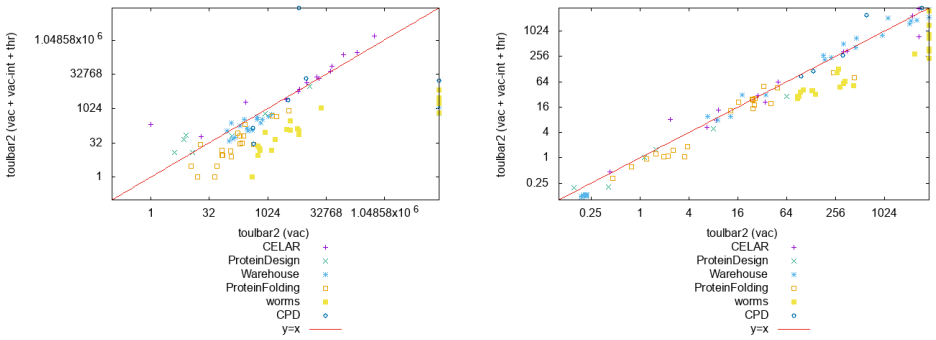
(c) Backtracks (d) CPU time (seconds)

Figure 6.6: Comparaison avec ToulBar2 utilisant VAC pendant la recherche. CELAR: ✳, CPD: ●, ProteinDesign: ▲, ProteinFolding: ▼, Warehouse: ◆, Worms: ■.



(a) Backtracks (b) CPU time (seconds)

Figure 6.7: Comparaison avec et sans seuil pour VAC pendant la recherche.

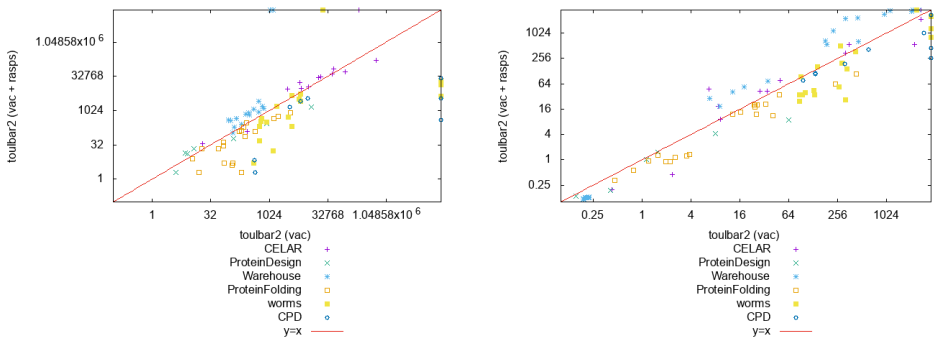


(a) Backtracks (b) CPU time (seconds)

Figure 6.8: Comparaison avec et sans VAC-intégralité et seuil pour VAC pendant la recherche.

CPU pour plusieurs familles tout en étant équivalent pour les autres (voir Figure 6.8).

Ensuite, nous avons analysé l'impact de l'application de la procédure de borne supérieure RASPS dans le prétraitement. Nous limitons RASPS à 1000 backtracks. Encore une fois, notre nouvelle heuristique RASPS réduit significativement l'effort de recherche en termes de backtracks et de temps, sauf pour Warehouse et quelques CELAR. Pour Warehouse, les bornes supérieures trouvées n'ont pas réduit le nombre total de backtracks. Pour CELAR scen06_r, elle réduit les backtracks de $3,4$ et le temps de résolution de $4,2$. Pour Worms, il était plus de 10 fois plus rapide pour certaines instances (voir figure 6.9).



(a) Backtracks (b) CPU time (seconds)

Figure 6.9: Comparaison avec et sans RASPS en prétraitement et VAC pendant la recherche.

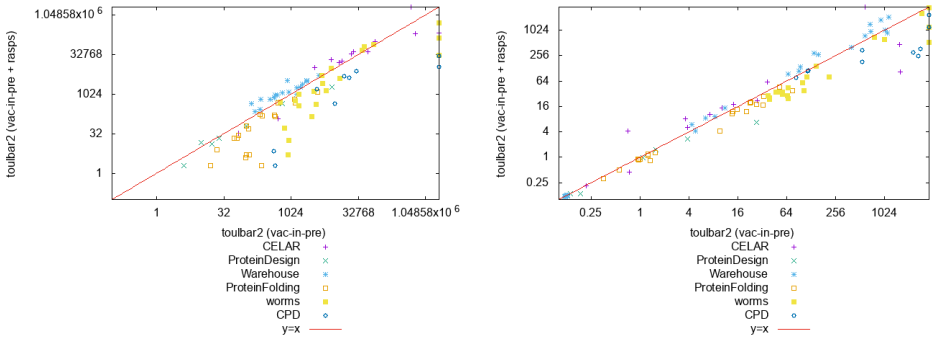
Enfin, nous combinons les deux heuristiques, VAC-intégralité et RASPS, avec le seuil limite de VAC et montrons les résultats comparés à VAC seul dans la Figure 6.6c et 6.6d. Nous conservons cette meilleure configuration dans le reste du document.

6.6.3 Comparaison avec VAC en Prétraitement et COMBILP

On pourrait s'attendre à ce que l'utilisation de VAC uniquement en prétraitement soit l'option la plus rapide, car c'est la valeur par défaut pour ToulBar2 et elle surpasse largement VAC pendant la recherche dans la plupart des cas [Hurley *et al.* 2016]. Cependant, pour certaines familles d'instances, nous parvenons à le surpasser.

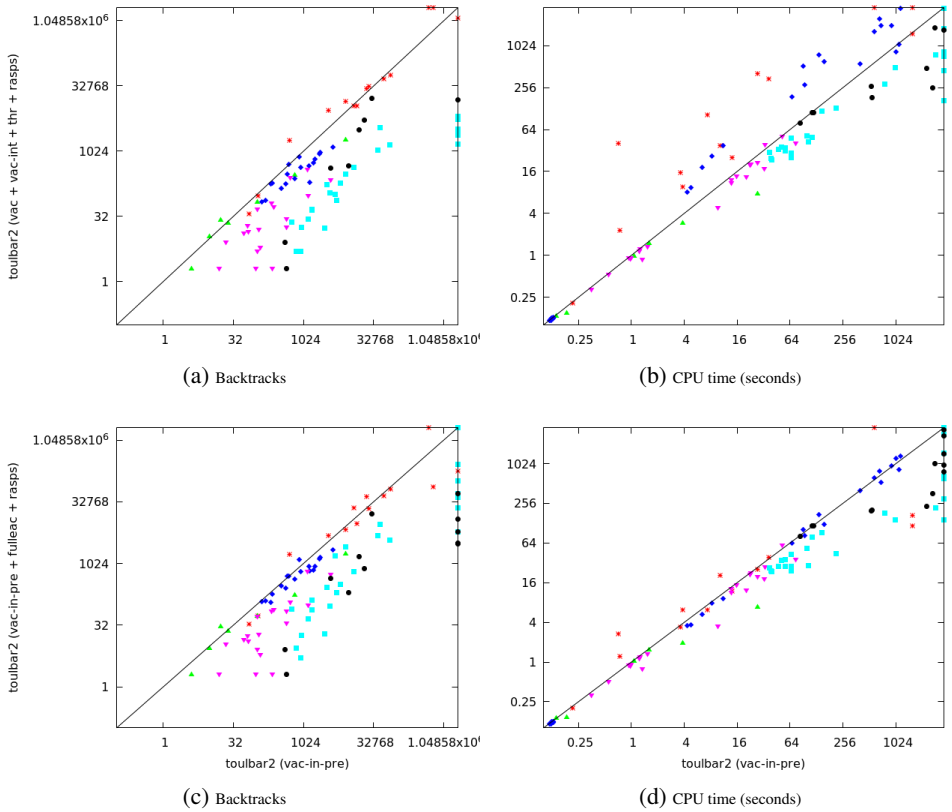
Lorsque VAC est utilisé uniquement dans le prétraitement, l'utilisation de RASPS en plus améliore considérablement les temps d'exécution, sauf pour Warehouse et certains CELAR (voir figure 6.10). Si nous ajoutons VAC-intégralité et RASPS lorsque nous utilisons VAC pendant la recherche, nous parvenons à surpasser VAC dans le prétraitement pour toutes les familles sauf CELAR et Warehouse, où la surcharge de VAC est trop élevée (voir Figure 6.11a et 6.11b).

De plus, si nous comparons les méthodes utilisant VAC dans le prétraitement



(a) Backtracks (b) CPU time (seconds)

Figure 6.10: Comparaison avec et sans RASPS pour VAC en prétraitement et EDAC pendant la recherche.



(a) Backtracks (b) CPU time (seconds)

(c) Backtracks (d) CPU time (seconds)

Figure 6.11: Comparaison avec la ToulBar2 par défaut. CELAR: ✳, CPD: ●, ProteinDesign: ▲, ProteinFolding: ▼, Warehouse: ◆, Worms: ■.

uniquement, alors l'exploitation de notre heuristique de branchement plus simple

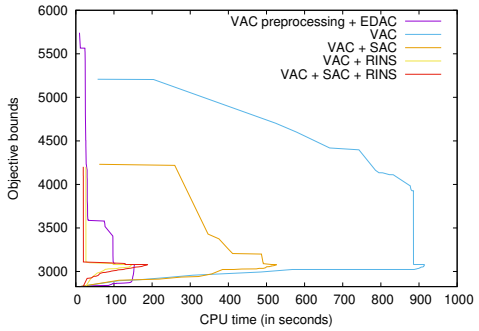


Figure 6.12: GraphiqueCELAIRE11 instance.

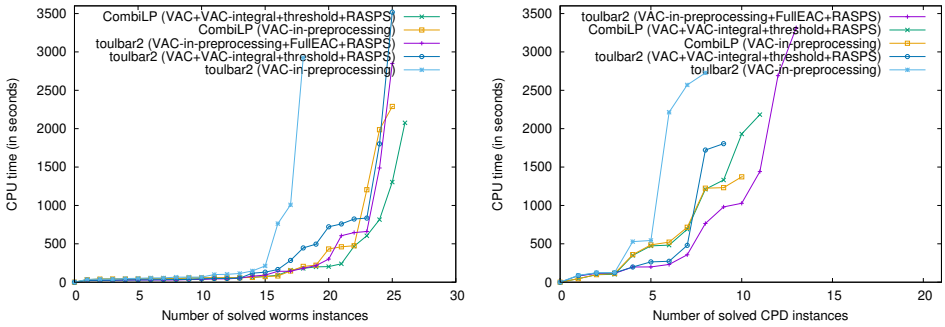
Full EAC et RASPS est encore plus performante dans la plupart des cas, étant aussi bonne que ToulBar2 par défaut sur les instances de l'entrepôt (55 instances résolues en moyenne en 128 secondes) et comparable sur CELAR (notre approche a résolu le graphe13 et scen06 un ordre de grandeur plus vite, mais n'a pas pu résoudre le graphe11 par rapport à VAC par défaut dans le prétraitement, voir Figure 6.11c, 6.11d).

Voir dans la Figure 6.12, les profils anytime sur l'instance CELAR graph11 [Cabon *et al.* 1999]. Pour ColorSeg, InPainting, ObjectSeg, Coloring et ProteinFolding, l'utilisation de RINS dans le prétraitement en plus de VAC donne également de meilleurs temps CPU par rapport à VAC seul. VAC-intégralité entraîne une diminution significative des temps CPU lorsqu'il est utilisé avec VAC pendant la recherche, pour EHI, QCP, DBN, ImageAlignment et ProteinFolding.

Ensuite, nous comparons ToulBar2 et CombiLP (qui utilise le même ToulBar2 comme solveur ILP interne) avec différentes techniques de bornes inférieures, montrant les avantages d'exploiter VAC-intégralité ou les extensions Full EAC et RASPS.

Dans la Figure 6.13a, nous voyons un graphe cactus[5] pour le benchmark Worms où il y a 30 instances. Nous avons résolu ces instances avec différentes combinaisons de solveurs et d'heuristiques avec une limite de temps CPU de 1 heure. Il a été rapporté que CombiLP a résolu 25 de ces instances dans [Haller *et al.* 2018] en 1 heure de temps CPU. Ici, nous comparons CombiLP avec les paramètres utilisés dans [Haller *et al.* 2018] (VAC dans le prétraitement et EDAC pendant la recherche), ainsi que notre version de ToulBar2 qui y est branchée. En plus de cela, nous avons une version autonome de ToulBar2, soit avec VAC dans le prétraitement et EDAC pendant la recherche, avec ou sans EAC complet, ou avec des options de branchement conscient de l'abcès, et RASPS. ToulBar2 seul peut aller jusqu'à 25 instances. Cependant, en branchant notre version de ToulBar2 dans CombiLP, nous parvenons à résoudre 26 de ces instances, ce qui fait 1 de plus que [Haller *et al.* 2018]. Un autre détail important est que, bien qu'il soit coûteux d'utiliser VAC tout au long de l'arbre de recherche, il devient meilleur avec [Haller *et al.* 2018] et

[5]Il montre sur l'axe x le nombre d'instances résolues pour une limite de temps donnée en axe y.



(a) Les instances de Worms.

(b) Les instances de CPD.

Figure 6.13: Comparaison avec COMBILP sur les instances Worms et CPD.

RASPS. Malgré tout, elle est légèrement dominée par la CAE complète. Pour ce benchmark, ces options entraînent une diminution significative de la taille de l'arbre de recherche qui compense le coût de l'application de VAC.

Cette heuristique plus simple a obtenu des résultats encore meilleurs sur le benchmark CPD (Figure 6.13b). Notre heuristique EAC complète avec RASPS a obtenu les meilleurs résultats, en résolvant 13 instances, par rapport à VAC-intégralité et RASPS qui en résolvent 9, et seulement 8 par défaut ToulBar2. COMBILP utilisant VAC pendant la recherche avec VAC-intégralité et RASPS a résolu 11 instances, au lieu de 10 sans ces options et VAC en prétraitement.

Dans [Ouali *et al.* 2017], ToulBar2 avec EDAC n'a résolu que 10 instances CPD parmi 21 dans une limite de temps CPU de 4 jours.

6.7 Conclusion

Nous avons réexaminé la propriété de cohérence d'arc stricte qui a été récemment utilisée dans un solveur de relaxation itérative. Nous avons identifié des propriétés qui facilitent son utilisation dans un solveur de type branch-and-bound et en particulier en conjonction avec l'algorithme VAC. Cette propriété nous permet d'intégrer des informations sur la relaxation que VAC calcule pour les utiliser dans des heuristiques. Nous avons présenté trois nouvelles heuristiques qui exploitent cette information, deux pour le branchement et l'autre pour trouver des bornes supérieures de bonne qualité. Dans une évaluation expérimentale, ces heuristiques ont montré de grandes performances dans certaines familles d'instances, améliorant l'état de l'art précédent. VAC-intégralité identifie une seule affectation partielle satisfaisable à coût nul dans un CSP particulier $Bool(P)$ du problème original P. D'autres techniques CSP telles que la substituabilité de voisinage [Freuder 1991] pourraient être utilisées pour détecter de plus grands sous-problèmes traitables. Le sous-problème intégral peut également être considéré comme une classe particulièrement facile à traiter, où chaque variable a une seule valeur. Par conséquent, une autre direction possible est de détecter les sous-problèmes qui sont traitables

pour des raisons plus sophistiquées.

CHAPTER 7

Conclusion

Réalisations

Dans cette thèse, nous avons contribué à l'état de l'art en matière d'apprentissage de la structure des BN, à savoir le problème d'apprentissage de la structure des réseaux bayésiens (BNSL), et de réponse aux requêtes d'explication la plus probable (MPE) et de minimisation sur les réseaux bayésiens (BN) et les réseaux à fonction de coût (CFN), qui sont tous deux NP-durs.

L'objectif commun, et le plus important, de toutes les implémentations présentées dans ce travail a été d'obtenir un meilleur compromis entre la force d'inférence et la vitesse d'inférence, quel que soit le problème. Il est vrai que lorsque les mécanismes d'inférence sont légers, il devient plus facile d'explorer une pléthore de nœuds de recherche par seconde. Malheureusement, avec une inférence légère, on est également plus susceptible de ne prendre en compte que des informations limitées sur l'état actuel de la recherche. D'autre part, un mécanisme d'inférence fort fournit une perspective plus perspicace sur les conséquences des décisions prises pendant la recherche, ce qui peut effectivement conduire à des décisions plus efficaces par la suite. Cependant, dans la plupart des cas rencontrés dans les applications réelles, cette opération devient si coûteuse que, inévitablement, nous nous demandons si cela en vaut la peine. Tout ceci nous a motivés à rechercher un équilibre entre ces deux notions clés : la vitesse et la force.

Pour BNSL, les algorithmes existants ont opté soit pour une force maximale d'inférence, comme les algorithmes basés sur la programmation en nombres entiers (IP) et le branch-and-cut, soit pour une vitesse maximale d'inférence, comme les algorithmes basés sur la programmation par contraintes (CP). Nous nous sommes d'abord intéressés à la manière d'effectuer plus rapidement l'élagage extensif des valeurs. Un travail précédent [Hoffmann & van Beek 2013] a fourni un propagateur GAC pour la contrainte d'acyclicité globale avec une complexité temporelle de $O(n^3d^2)$. Étant donné les grandes tailles de domaine inhérentes aux problèmes BNSL, avoir un facteur quadratique de d est, pour le moins, catastrophique. Nous avons réussi à concevoir un propagateur GAC de complexité $O(n^3d)$, et l'avons implémenté dans notre solveur ELSA, qui est basé sur le solveur de pointe basé sur CP CPBayes v1.1 [Van Beek & Hoffmann 2015]. Les résultats présentés dans le chapitre 3 ont prouvé que le gain de temps grâce à GAC augmente avec la taille des problèmes.

En continuant avec BNSL, nous mettons ensuite l'accent sur la génération de bornes inférieures. Étant donné les fortes limites inférieures fournies par la

méthode basée sur IP GOBNILP [Cussens 2011, Barlett & Cussens 2013], nous avons cherché des moyens d'implémenter une idée similaire dans notre solveur basé sur CP afin d'effectuer une recherche plus efficace. Dans le chapitre 4, nous avons spécifié les propriétés d'une classe spécifique d'inégalités, appelées inégalités de cluster, qui conduisent à un algorithme qui effectue une inférence beaucoup plus forte que celle basée sur CP, beaucoup plus rapide que celle basée sur IP.

L'obstacle majeur des problèmes BNSL est la croissance exponentielle de la taille des domaines avec chaque variable supplémentaire. Ceci est inévitable dans une certaine mesure, il n'est donc que raisonnable de chercher des moyens de représenter les domaines de manière plus compacte. À cette fin, nous avons mis au point une implémentation d'arbre de décision binaire (BDT). Les BDT nous ont permis d'exécuter des requêtes sur les valeurs de domaine beaucoup plus rapidement, et ont conduit à une augmentation supplémentaire des performances.

Pour les requêtes de minimisation dans les réseaux de fonctions de coût (CFN), nous avons identifié une faiblesse dans l'utilisation des relaxations de programmation linéaire par une classe spécifique de solveurs, qui inclut le solveur open source primé ToulBar2 [Cooper *et al.* 2010]. Nous avons prouvé que cette faiblesse peut conduire à des décisions de branchement sous-optimales et montré comment détecter les ensembles maximaux de telles décisions, qui peuvent alors être évitées par le solveur. Cela a permis à ToulBar2 de s'attaquer à des problèmes qui n'étaient auparavant résolubles que par des algorithmes hybrides.

Perspectives

Une amélioration potentielle réside dans la façon dont nous résolvons $LP_{\mathcal{W}}$, la LP introduite dans le chapitre 4. Les algorithmes existants en temps quasi linéaire pour résoudre les LP positives [Allen-Zhu & Orecchia 2015], ou un autre solveur adapté aux contraintes clausales que nous avons pourraient valoir la peine d'être essayés. En outre, nous pourrions essayer de nous rapprocher encore plus de l'algorithme VAC, bien qu'il y ait la limitation qu'actuellement l'algorithme VAC n'est pas généralisé pour les contraintes clausales. Au cas où nous déciderions d'essayer cela, la question de savoir si les arbres de décision seraient utiles est intéressante.

Une autre direction est certainement une implémentation plus sophistiquée des BDT. L'heuristique de gain d'information, en particulier pour certains ensembles de données comme nous l'avons vu au chapitre 5, peut s'avérer trop coûteuse pour être employée tout au long de la construction d'un BDT. Dans ce cas, il est envisageable de désactiver l'heuristique de gain d'information à une certaine profondeur, ou de ne pas l'utiliser du tout pour certaines variables.

Une autre direction est l'amélioration de la base de données des modèles. Actuellement, le calcul d'une borne inférieure prend beaucoup de temps en prétraitement. Nous pouvons le calculer de façon paresseuse. Lorsque nous passerons à cela, nous pourrons peut-être utiliser des sous-graphes plus grands.

Il y a aussi la question de savoir si le calcul des scores des ensembles de parents

pendant la recherche serait utile ou non. Une grande partie du coût de calcul n'est même pas présente dans le temps d'exécution de ELSA, puisque les scores sont déjà calculés avant même que nous exécutions ELSA. Cette opération peut prendre beaucoup de temps, même plus de 24 heures dans certains cas. Si nous l'intégrons à ELSA, cela ralentira effectivement le solveur, mais cela pourrait améliorer le temps global nécessaire pour calculer les scores et trouver la structure optimale.

Une autre direction de recherche est probablement inattendue pour le lecteur qui a parcouru tous les chapitres jusqu'ici, car elle n'a jamais été mentionnée : l'étude des méthodes basées sur les IP développées pour le célèbre problème du voyageur de commerce (TSP). La formulation ILP de BNSL sans les contraintes de cluster (CC) est fondamentalement le problème d'affectation (Assignment Problem (AP)). De même, le TSP sans les contraintes d'élimination des sous-tours (SECs) est également équivalent au AP. Le nombre de CCs et de SECs étant exponentiel dans le nombre de variables, ainsi que le fait que la relaxation LP du AP est très forte, le branch-and-cut est un choix populaire pour les implémentations basées sur IP à la fois pour BNSL et TSP. Un autre point commun entre les CC et les SEC est leur sémantique. Les CC éliminent *tous* les cycles alors que les SEC éliminent *tous sauf un* : les cycles qui couvrent l'ensemble des variables. Ainsi, d'une certaine manière, les SEC sont un sous-ensemble des CC, et toute méthode développée pour traiter les SEC peut être pertinente pour les CC.

Bibliography

[Achterberg 2009] T Achterberg. *SCIP: solving constraint integer programs*. Mathematical Programming Computation, vol. 1, no. 1, pages 1–41, 2009.

[Ağralı *et al.* 2017] Semra Ağralı, Z Caner Taşkın and A Tamer Ünal. *Employee scheduling in service industries with flexible employee availability and demand*. Omega, vol. 66, pages 159–169, 2017.

[Akers 1978] Sheldon B. Akers. *Binary decision diagrams*. IEEE Transactions on computers, vol. 27, no. 06, pages 509–516, 1978.

[Allen-Zhu & Orecchia 2015] Zeyuan Allen-Zhu and Lorenzo Orecchia. *Nearly-Linear Time Positive LP Solver with Faster Convergence Rate*. In Proc. of the Forty-Seventh Annual ACM Symposium on Theory of Computing, STOC'15, page 229–236, New York, NY, USA, 2015.

[Allouche *et al.* 2013] David Allouche, Christine Cierco-Ayrolles, Simon De Givry, Gérald Guillermin, Brigitte Mangin, Thomas Schiex, Jimmy Vandel and Matthieu Vignes. *A panel of learning methods for the reconstruction of gene regulatory networks in a systems genetics context*. In Gene Network Inference, pages 9–31. Springer, 2013.

[Allouche *et al.* 2014] 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.

[Allouche *et al.* 2015] D Allouche, S de Givry, G Katsirelos, T Schiex and M Zytnicki. *Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP*. In Proc. of CP-15, pages 12–28, Cork, Ireland, 2015.

[Avellaneda 2019] Florent Avellaneda. *Learning Optimal Decision Trees from Large Datasets*. 2019.

[Bacchus *et al.* 2002] F Bacchus, X Chen, P van Beek and T Walsh. *Binary vs. non-binary constraints*. Artificial Intelligence, vol. 140, no. 1/2, pages 1–37, 2002.

[Backofen *et al.* 1999] Rolf Backofen, Sebastian Will and Erich Bornberg-Bauer. *Application of constraint programming techniques for structure prediction of lattice proteins with extended alphabets.* Bioinformatics (Oxford, England), vol. 15, no. 3, pages 234–242, 1999.

[Bahar *et al.* 1997] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo and Fabio Somenzi. *Algebric decision diagrams and their applications*. Formal methods in system design, vol. 10, no. 2, pages 171–206, 1997.

[Baksh *et al.* 2018] Al-Amin Baksh, Rouzbeh Abbassi, Vikram Garaniya and Faisal Khan. *Marine transportation risk assessment using Bayesian Network: Application to Arctic waters*. Ocean Engineering, vol. 159, pages 422–436, 2018.

[Baptiste *et al.* 2001] Philippe Baptiste, Claude Le Pape and Wim Nuijten. Constraint-based scheduling: applying constraint programming to scheduling problems, volume 39. Springer Science & Business Media, 2001.

[Barahona & Krippahl 2008] Pedro Barahona and Ludwig Krippahl. *Constraint programming in structural bioinformatics*. Constraints, vol. 13, no. 1-2, pages 3–20, 2008.

[Barlett & Cussens 2013] M. Barlett and J. Cussens. *Advances in Bayesian Network Learning using Integer Programming*. In UAI'13, pages 182–191, 2013.

[Bartlett & Cussens 2017] Mark Bartlett and James Cussens. *Integer linear programming for the Bayesian network structure learning problem*. Artificial Intelligence, vol. 244, pages 258–271, 2017.

[Beck *et al.* 2011] J Christopher Beck, TK Feng and Jean-Paul Watson. *Combining constraint programming and local search for job-shop scheduling*. INFORMS Journal on Computing, vol. 23, no. 1, pages 1–14, 2011.

[Behjati & Beigy 2020] Shahab Behjati and Hamid Beigy. *Improved K2 algorithm for Bayesian network structure learning*. Engineering Applications of Artificial Intelligence, vol. 91, page 103617, 2020.

[Berg *et al.* 2014] Jeremias Berg, Matti Järvisalo and Brandon Malone. *Learning optimal bounded treewidth Bayesian networks via maximum satisfiability*. In Artificial Intelligence and Statistics, pages 86–95. PMLR, 2014.

[Bergman *et al.* 2014a] David Bergman, Andre A Cire, Willem-Jan van Hoeve and John N Hooker. *Optimization bounds from binary decision diagrams*. INFORMS Journal on Computing, vol. 26, no. 2, pages 253–268, 2014.

[Bergman *et al.* 2014b] David Bergman, Andre A Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat and Willem-Jan van Hoeve. *Parallel combinatorial optimization with decision diagrams*. In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pages 351–367. Springer, 2014.

[Bergman *et al.* 2016] David Bergman, Andre A Cire, Willem-Jan Van Hoeve and John N Hooker. *Discrete optimization with decision diagrams*. INFORMS Journal on Computing, vol. 28, no. 1, pages 47–66, 2016.

[Bessiere & Régin 1996] Christian Bessiere and Jean-Charles Régin. *MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems*. In International Conference on Principles and Practice of Constraint Programming, pages 61–75. Springer, 1996.

[Bessiere *et al.* 2009] Christian Bessiere, Emmanuel Hebrard and Barry O'Sullivan. *Minimising decision tree size as combinatorial optimisation*. In International Conference on Principles and Practice of Constraint Programming, pages 173–187. Springer, 2009.

[Borning 1981] Alan Borning. *The programming language aspects of ThingLab, a constraint-oriented simulation laboratory*. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 3, no. 4, pages 353–387, 1981.

[Boussemart *et al.* 2004] F Boussemart, F Hemery, C Lecoutre and L Sais. *Boosting Systematic Search by Weighting Constraints*. In Proc. of ECAI-04, pages 146–150, Valencia, Spain, 2004.

[Buntine 1991] Wray Buntine. *Theory refinement on Bayesian networks*. In Proc. of UAI, pages 52–60. Elsevier, 1991.

[Cabon *et al.* 1999] B Cabon, S de Givry, L Lobjois, T Schiex and JP Warners. *Radio Link Frequency Assignment*. Constraints, vol. 4, no. 1, pages 79–89, 1999.

[Campos & Ji 2011] Cassio P de Campos and Qiang Ji. *Efficient structure learning of Bayesian networks using constraints*. Journal of Machine Learning Research, vol. 12, no. Mar, pages 663–689, 2011.

[Cardie 1993] Claire Cardie. *Using decision trees to improve case-based learning*. In Proceedings of the tenth international conference on machine learning, pages 25–32, 1993.

[Chickering *et al.* 2004] D. Chickering, D. Heckerman and C. Meek. *Large-Sample Learning of Bayesian Networks is NP-Hard*. Journal of Machine Learning Research, vol. 5, pages 1287–1330, 2004.

[Chickering 1995] David Maxwell Chickering. *Learning Bayesian Networks is NP-Complete*. In Proc. of Fifth Int. Workshop on Artificial Intelligence and Statistics (AISTATS), pages 121–130, Key West, Florida, USA, 1995.

[Cofiño *et al.* 2002] Antonio S Cofiño, Rafael Cano, Carmen Sordo and José M Gutiérrez. *Bayesian networks for probabilistic weather prediction*. In

Proceedings of the 15th European Conference on Artificial Intelligence, pages 695–699, 2002.

[Cooper & Schiex 2004] M Cooper and T Schiex. *Arc consistency for soft constraints*. Artificial Intelligence, vol. 154, no. 1-2, pages 199–227, 2004.

[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.

[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.

[Cussens *et al.* 2017] James Cussens, Matti Järvisalo, Janne H Korhonen and Mark Bartlett. *Bayesian network structure learning with integer programming: Polytopes, facets and complexity*. Journal of Artificial Intelligence Research, vol. 58, pages 185–229, 2017.

[Cussens 2008] J. Cussens. *Bayesian network learning by compiling to weighted MAX-SAT*. In UAI'08, pages 105–112, 2008.

[Cussens 2011] James Cussens. *Bayesian network learning with cutting planes*. In Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011), pages 153–160. AUAI Press, 2011.

[Dai *et al.* 2020] Jingguo Dai, Jia Ren and Wencai Du. *Decomposition-based Bayesian network structure learning algorithm using local topology information*. Knowledge-Based Systems, vol. 195, page 105602, 2020.

[Danna *et al.* 2005] E Danna, E Rothberg and C Le Pape. *Exploring relaxation induced neighborhoods to improve MIP solutions*. Mathematical Programming, vol. 102, no. 1, pages 71–90, 2005.

[Davies & Bacchus 2011] J Davies and F Bacchus. *Solving MAXSAT by Solving a Sequence of Simpler SAT Instances*. In Proc. of CP-11, pages 225–239, Perugia, Italy, 2011.

[De Backer *et al.* 2000] Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby and Patrick Prosser. *Solving vehicle routing problems using constraint programming and metaheuristics*. Journal of Heuristics, vol. 6, no. 4, pages 501–523, 2000.

[de Campos & Ji 2010] Cassio Polpo de Campos and Qiang Ji. *Properties of Bayesian Dirichlet Scores to Learn Bayesian Network Structures*. In Proc. of AAAI-00, Atlanta, Georgia, USA, 2010.

[de Campos *et al.* 2018] Cassio P de Campos, Mauro Scanagatta, Giorgio Corani and Marco Zaffalon. *Entropy-based pruning for learning Bayesian networks using BIC*. Artificial Intelligence, vol. 260, pages 42–50, 2018.

[de Givry *et al.* 2005] S de Givry, M Zytnicki, F Heras and J Larrosa. *Existential arc consistency: getting closer to full arc consistency in weighted CSPs*. In Proc. of IJCAI-05, pages 84–89, Edinburgh, Scotland, 2005.

[Demirer *et al.* 2006] Riza Demirer, Ronald R Mau and Catherine Shenoy. *Bayesian networks: a decision tool to improve portfolio risk analysis*. Journal of applied finance, vol. 16, no. 2, page 106, 2006.

[Demirovic *et al.* 2018] Emir Demirovic, Geoffrey Chu and Peter 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.

[Downey & Fellows 2013] R G. Downey and M R. Fellows. Fundamentals of parameterized complexity. Texts in Computer Science. Springer, 2013.

[Elaidi *et al.* 2018] Halima Elaidi, Younes Elhaddar, Zahra Benabbou and Hassan Abbar. *An idea of a clustering algorithm using support vector machines based on binary decision tree*. In 2018 International Conference on Intelligent Systems and Computer Vision (ISCV), pages 1–5. IEEE, 2018.

[Fan & Yuan 2015] Xiannian Fan and Changhe Yuan. *An Improved Lower Bound for Bayesian Network Structure Learning*. In Proc. of AAAI-15, Austin, Texas, 2015.

[Fan *et al.* 2014] Xiannian Fan, Changhe Yuan and Brandon M Malone. *Tightening Bounds for Bayesian Network Structure Learning.* In AAAI, volume 4, pages 2439–2445, 2014.

[Felner *et al.* 2004] Ariel Felner, Richard E Korf and Sarit Hanan. *Additive pattern database heuristics*. Journal of Artificial Intelligence Research, vol. 22, pages 279–318, 2004.

[Freuder 1978] Eugene C Freuder. *Synthesizing constraint expressions*. Communications of the ACM, vol. 21, no. 11, pages 958–966, 1978.

[Freuder 1982] Eugene C Freuder. *A sufficient condition for backtrack-free search*. Journal of the ACM (JACM), vol. 29, no. 1, pages 24–32, 1982.

[Freuder 1991] Eugene C Freuder. *Eliminating interchangeable values in constraint satisfaction problems*. In AAAI, volume 91, pages 227–233, 1991.

[Geman & Graffigne 1986] Stuart Geman and Christine Graffigne. *Markov random field image models and their applications to computer vision*. In Proceedings

of the international congress of mathematicians, volume 1, page 2. Berkeley, CA, 1986.

[Gemela 2001] Jozef Gemela. *Financial analysis using Bayesian networks*. Applied Stochastic Models in Business and Industry, vol. 17, no. 1, pages 57–67, 2001.

[Gent *et al.* 1996] Ian P Gent, Ewan MacIntyre, Patrick Presser, Barbara M Smith and Toby Walsh. *An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem*. In International Conference on Principles and Practice of Constraint Programming, pages 179–193. Springer, 1996.

[Gervet & Van Hentenryck 2006] Carmen Gervet and Pascal Van Hentenryck. *Length-lex ordering for set csps*. In AAAI, pages 48–53, 2006.

[Gervet 1997] Carmen Gervet. *Interval propagation to reason about sets: Definition and implementation of a practical language*. Constraints, vol. 1, no. 3, pages 191–244, 1997.

[Gravier *et al.* 1999] Guillaume Gravier, Marc Sigelle and Gerard Chollet. *Markov random field modeling for speech recognition*. Aust. J. Intell. Inform. Process. Syst, vol. 5, no. 4, pages 245–252, 1999.

[Gravier *et al.* 2000] Guillaume Gravier, Marc Sigelle and Gérard Chollet. *A markov random field model for automatic speech recognition*. In Proceedings 15th International Conference on Pattern Recognition. ICPR-2000, volume 3, pages 254–257. IEEE, 2000.

[Guimarans *et al.* 2011] Daniel Guimarans, Rosa Herrero, Daniel Riera, Angel A Juan and Juan José Ramos. *Combining probabilistic algorithms, constraint programming and lagrangian relaxation to solve the vehicle routing problem*. Annals of Mathematics and Artificial Intelligence, vol. 62, no. 3, pages 299–315, 2011.

[Güngör *et al.* 2018] Murat Güngör, Ali Tamer Ünal and Z Caner Taşkın. *A parallel machine lot-sizing and scheduling problem with a secondary resource and cumulative demand*. International Journal of Production Research, vol. 56, no. 9, pages 3344–3357, 2018.

[Haller *et al.* 2018] S Haller, P Swoboda and B Savchynskyy. *Exact MAP-Inference by Confining Combinatorial Search With LP Relaxation*. In Proc. of AAAI-18, pages 6581–6588, New Orleans, Louisiana, USA, 2018.

[Haralick & Elliott 1980] Robert M Haralick and Gordon L Elliott. *Increasing tree search efficiency for constraint satisfaction problems*. Artificial intelligence, vol. 14, no. 3, pages 263–313, 1980.

[Heckerman *et al.* 1995] David Heckerman, Dan Geiger and David M Chickering. *Learning Bayesian networks: The combination of knowledge and statistical data*. Machine learning, vol. 20, no. 3, pages 197–243, 1995.

[Heras & Larrosa 2006] Federico Heras and Javier Larrosa. *New inference rules for efficient Max-SAT solving*. In AAAI, pages 68–73, 2006.

[Hoffmann & van Beek 2013] Hella-Franziska Hoffmann and Peter van Beek. *A Global Acyclicity Constraint for Bayesian Network Structure Learning*. CP Doctoral Program, page 91, 2013.

[Hurley *et al.* 2016] B Hurley, B O'Sullivan, D Allouche, G Katsirelos, T Schiex, M Zytnicki and S de Givry. *Multi-Language Evaluation of Exact Solvers in Graphical Model Discrete Optimization*. Constraints, vol. 21, no. 3, pages 413–434, 2016.

[Jaakkola *et al.* 2010] Tommi Jaakkola, David Sontag, Amir Globerson and Marina Meila. *Learning Bayesian network structure using LP relaxations*. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pages 358–365. JMLR Workshop and Conference Proceedings, 2010.

[Jensen & Arnspang 1999] Kristoffer Jensen and Jens Arnspang. *Binary decision tree classification of musical sounds*. In ICMC, 1999.

[Johansson & Falkman 2008] Fredrik Johansson and Goran Falkman. *A Bayesian network approach to threat evaluation with application to an air defense scenario*. In 2008 11th International conference on information fusion, pages 1–7. IEEE, 2008.

[Junker 2004] Ulrich Junker. *QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems*. In Proceedings of the 19th national conference on Artifical intelligence, pages 167–172, 2004.

[Kainmueller *et al.* 2014] D Kainmueller, F Jug, C Rother and G Myers. *Active Graph Matching for Automatic Joint Segmentation and Annotation of C. elegans*. In Medical Image Computing and Computer-Assisted Intervention, pages 81–88, Boston, USA, 2014.

[Kennett *et al.* 2001] Russell J Kennett, Kevin B Korb and Ann E Nicholson. *Seabreeze prediction using Bayesian networks*. In Pacific-Asia Conference on Knowledge Discovery and Data Mining, pages 148–153. Springer, 2001.

[Koller & Friedman 2009] Daphne Koller and Nir Friedman. Probabilistic graphical models: principles and techniques. MIT press, 2009.

[Lam & Bacchus 1994] Wai Lam and Fahiem Bacchus. *Using New Data to Refine a Bayesian Network*. In Proc. of UAI, pages 383–390, 1994.

[Larrosa & Dechter 2000] Javier Larrosa and Rina Dechter. *On the dual representation of non-binary semiring-based CSPs*. In CP'2000 workshop on soft constraints. Citeseer, 2000.

[Larrosa & Dechter 2003] Javier Larrosa and Rina Dechter. *Boosting search with variable elimination in constraint optimization and constraint satisfaction problems*. Constraints, vol. 8, no. 3, pages 303–326, 2003.

[Laurent & Rivest 1976] Hyafil Laurent and Ronald L Rivest. *Constructing optimal binary decision trees is NP-complete*. Information processing letters, vol. 5, no. 1, pages 15–17, 1976.

[Lauriere 1978] Jena-Lonis Lauriere. *A language and a program for stating and solving combinatorial problems*. Artificial intelligence, vol. 10, no. 1, pages 29–127, 1978.

[Lecoutre *et al.* 2009] C Lecoutre, L Sais, S Tabary and V Vidal. *Reasoning from last conflict(s) in constraint programming*. Artificial Intelligence, vol. 173, no. 18, pages 1592–1614, 2009.

[Lee & Lee 2006] Chang-Ju Lee and Kun Jai Lee. *Application of Bayesian network to the probabilistic risk assessment of nuclear waste disposal*. Reliability Engineering & System Safety, vol. 91, no. 5, pages 515–532, 2006.

[Lee & van Beek 2017a] Colin Lee and Peter van Beek. *An experimental analysis of anytime algorithms for Bayesian network structure learning*. In Advanced Methodologies for Bayesian Networks, pages 69–80, 2017.

[Lee & van Beek 2017b] Colin Lee and Peter van Beek. *Metaheuristics for score-and-search Bayesian network structure learning*. In Canadian Conference on Artificial Intelligence, pages 129–141. Springer, 2017.

[Lee 1959] Chang-Yeong Lee. *Representation of switching circuits by binary-decision programs*. The Bell System Technical Journal, vol. 38, no. 4, pages 985–999, 1959.

[Li 1994] Stan Z Li. *Markov random field models in computer vision*. In European conference on computer vision, pages 361–370. Springer, 1994.

[Li 2009] Stan Z Li. Markov random field modeling in image analysis. Springer Science & Business Media, 2009.

[Liberatore 2000] Paolo Liberatore. *On the complexity of choosing the branching literal in DPLL*. Artificial intelligence, vol. 116, no. 1-2, pages 315–326, 2000.

[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.

[Liu *et al.* 2016] Fei Liu, Shao-Wu Zhang, Wei-Feng Guo, Ze-Gang Wei and Luonan Chen. *Inference of gene regulatory network based on local Bayesian networks*. PLoS computational biology, vol. 12, no. 8, page e1005024, 2016.

[Liu *et al.* 2017] Hui Liu, Shuigeng Zhou, Wai Lam and Jihong Guan. *A new hybrid method for learning bayesian networks: Separation and reunion*. Knowledge-Based Systems, vol. 121, pages 185–197, 2017.

[López-Ortiz *et al.* 2003] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp and Peter Van Beek. *A fast and simple algorithm for bounds consistency of the alldifferent constraint*. In IJCAI, volume 3, pages 245–250, 2003.

[Lovász & Plummer 2009] László Lovász and Michael D Plummer. Matching theory, volume 367. American Mathematical Soc., 2009.

[Luo *et al.* 2005] Jiebo Luo, Andreas E Savakis and Amit Singhal. *A Bayesian network-based framework for semantic image understanding*. Pattern recognition, vol. 38, no. 6, pages 919–934, 2005.

[Malone *et al.* 2011a] Brandon Malone, Changhe Yuan and Eric Hansen. *Memory-efficient dynamic programming for learning optimal Bayesian networks*. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 25, 2011.

[Malone *et al.* 2011b] Brandon Malone, Changhe Yuan, Eric A Hansen and Susan Bridges. *Improving the scalability of optimal Bayesian network learning with external-memory frontier breadth-first branch and bound search*. In Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, pages 479–488, 2011.

[Marques-Silva & Mencía 2020] João Marques-Silva and Carlos Mencía. *Reasoning About Inconsistent Formulas*. In Christian Bessiere, editor, Proc. of IJCAI-2020, pages 4899–4906, 2020.

[Minato 1993] Shin-ichi Minato. *Zero-suppressed BDDs for set manipulation in combinatorial problems*. In Proceedings of the 30th International Design Automation Conference, pages 272–277, 1993.

[Minton *et al.* 1992] Steven Minton, Mark D Johnston, Andrew B Philips and Philip Laird. *Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems*. Artificial intelligence, vol. 58, no. 1-3, pages 161–205, 1992.

[Montanari 1974] Ugo Montanari. *Networks of constraints: Fundamental properties and applications to picture processing*. Information sciences, vol. 7, pages 95–132, 1974.

[Morgado *et al.* 2013] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes and João Marques-Silva. *Iterative and core-guided MaxSAT solving: A survey and assessment*. Constraints An Int. J., vol. 18, no. 4, pages 478–534, 2013.

[Morgado *et al.* 2014] A Morgado, A Ignatiev and J Marques-Silva. *MSCG: Robust Core-Guided MaxSAT Solving*. JSAT, vol. 9, pages 129–134, 2014.

[Murty 1983] Katta G Murty. Linear programming. Springer, 1983.

[Nedevschi *et al.* 2006] Sergiu Nedevschi, Jaspal S Sandhu, Joyojeet Pal, Rodrigo Fonseca and Kentaro Toyama. *Bayesian networks: an exploratory tool for understanding ICT adoption*. In 2006 International Conference on Information and Communication Technologies and Development, pages 277–284. IEEE, 2006.

[Ouali *et al.* 2017] A Ouali, D Allouche, S de Givry, S Loudni, Y Lebbah, F Eckhardt and L Loukil. *Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization*. In Proc. of UAI-17, pages 550–559, Sydney, Australia, 2017.

[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, page 103194, 2020.

[Padberg & Rinaldi 1987] Manfred Padberg and Giovanni Rinaldi. *Optimization of a 532-city symmetric traveling salesman problem by branch and cut*. Operations Research Letters, vol. 6, no. 1, pages 1–7, 1987.

[Pearl 1988] Judea Pearl. Probabilistic reasoning in intelligent systems: networks of plausible inference. Elsevier, 1988.

[Refalo 2004] Philippe Refalo. *Impact-based search strategies for constraint programming*. In International Conference on Principles and Practice of Constraint Programming, pages 557–571. Springer, 2004.

[Régin 1994] Jean-Charles Régin. *A filtering algorithm for constraints of difference in CSPs*. In AAAI, volume 94, pages 362–367, 1994.

[Rice & Kulhari 2008] Michael Rice and Sanjay Kulhari. *A survey of static variable ordering heuristics for efficient BDD/MDD construction*. University of California, Tech. Rep, page 130, 2008.

[Rossi *et al.* 2006] Francesca Rossi, Peter Van Beek and Toby Walsh. Handbook of constraint programming. Elsevier, 2006.

[Savchynskyy *et al.* 2013] B Savchynskyy, JH Kappes, P Swoboda and C Schnörr. *Global MAP-optimality by shrinking the combinatorial search area with convex relaxation*. In Proc. of NIPS-13, pages 1950–1958, Lake Tahoe, Nevada, USA, 2013.

[Scanagatta *et al.* 2015] Mauro Scanagatta, Cassio P de Campos, Giorgio Corani and Marco Zaffalon. *Learning Bayesian networks with thousands of variables*. Proc. of NeurIPS, vol. 28, pages 1864–1872, 2015.

[Schoenemann *et al.* 2014] Thomas Schoenemann, Vladimir Kolmogorov, S Nowozin, P Gehler, J Jancsary and C Lampert. *Generalized sequential tree-reweighted message passing*. Advanced Structured Prediction, vol. 75, 2014.

[Schrijver 1998] Alexander Schrijver. Theory of linear and integer programming. John Wiley & Sons, 1998.

[Schwarz 1978] Gideon Schwarz. *Estimating the dimension of a model*. The Annals of Statistics, vol. 6, no. 2, pages 461–464, 1978.

[Scutari *et al.* 2019] Marco Scutari, Catharina Elisabeth Graafland and José Manuel Gutiérrez. *Who learns better Bayesian network structures: Accuracy and speed of structure learning algorithms*. International Journal of Approximate Reasoning, vol. 115, pages 235–253, 2019.

[Shannon 1938] Claude E Shannon. *A symbolic analysis of relay and switching circuits*. Electrical Engineering, vol. 57, no. 12, pages 713–723, 1938.

[Shaw 1998] Paul Shaw. *Using constraint programming and local search methods to solve vehicle routing problems*. In International conference on principles and practice of constraint programming, pages 417–431. Springer, 1998.

[Sierra *et al.* 2018] Leonardo A Sierra, Víctor Yepes, Tatiana García-Segura and Eugenio Pellicer. *Bayesian network method for decision-making about the social sustainability of infrastructure projects*. Journal of Cleaner Production, vol. 176, pages 521–534, 2018.

[Silander & Myllymäki 2006] Tomi Silander and Petri Myllymäki. *A simple approach for finding the globally optimal Bayesian network structure*. In Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence, pages 445–452, 2006.

[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.

[Stassopoulou *et al.* 1998] Athena Stassopoulou, Maria Petrou and Josef Kittler. *Application of a Bayesian network in a GIS based decision making system*. International Journal of Geographical Information Science, vol. 12, no. 1, pages 23–46, 1998.

[Sticha *et al.* 2006] Paul J Sticha, Dennis M Buede and Richard L Rees. *Bayesian model of the effect of personality in predicting decisionmaker behavior*. In Proceedings of the 22nd conference on uncertainty in artificial intelligence. Citeseer, 2006.

[Suk *et al.* 2008] Heung-Il Suk, Bong-Kee Sin and Seong-Whan Lee. *Recognizing hand gestures using dynamic bayesian network*. In 2008 8th IEEE International Conference on Automatic Face & Gesture Recognition, pages 1–6. IEEE, 2008.

[Sỳkora *et al.* 2018] Miroslav Sỳkora, Jana Markova and Dimitris Diamantidis. *Bayesian network application for the risk assessment of existing energy production units*. Reliability Engineering & System Safety, vol. 169, pages 312–320, 2018.

[Taşkın *et al.* 2015] Z Caner Taşkın, Semra Ağralı, A Tamer Ünal, Vahdet Belada and Filiz Gökten-Yılmaz. *Mathematical programming-based sales and operations planning at vestel Electronics*. Interfaces, vol. 45, no. 4, pages 325–340, 2015.

[Trilling *et al.* 2006] Lorraine Trilling, Alain Guinet and Dominiue Le Magny. *Nurse scheduling using integer linear programming and constraint programming*. IFAC Proceedings Volumes, vol. 39, no. 3, pages 671–676, 2006.

[Trucco *et al.* 2008] Paolo Trucco, Enrico Cagno, Fabrizio Ruggeri and Ottavio Grande. *A Bayesian Belief Network modelling of organisational factors in risk analysis: A case study in maritime transportation*. Reliability Engineering & System Safety, vol. 93, no. 6, pages 845–856, 2008.

[Van Beek & Chen 1999] Peter Van Beek and Xinguang Chen. *CPlan: A constraint programming approach to planning*. In AAAI/IAAI, pages 585–590, 1999.

[Van Beek & Hoffmann 2015] Peter Van Beek and Hella-Franziska Hoffmann. *Machine learning of Bayesian networks using constraint programming*. In International Conference on Principles and Practice of Constraint Programming, pages 429–445. Springer, 2015.

[Walsh 2000] Toby Walsh. *SAT vs CSP*. In Proc. of the Sixth International Conference on Principles and Practice of Constraint Programming, pages 441–456, 2000.

[Wasyluk *et al.* 2001] Hanna Wasyluk, A Onisko and MJ Druzdzel. *Support of diagnosis of liver disorders based on a causal Bayesian network model*. Medical Science Monitor, vol. 7, no. 1; SUPP, pages 327–332, 2001.

[Werner 2007] Tomas Werner. *A linear programming approach to max-sum problem: A review*. IEEE transactions on pattern analysis and machine intelligence, vol. 29, no. 7, pages 1165–1179, 2007.

[Wolsey 2020] Laurence A Wolsey. Integer programming. John Wiley & Sons, 2020.

[Xing *et al.* 2017] Linlin Xing, Maozu Guo, Xiaoyan Liu, Chunyu Wang, Lei Wang and Yin Zhang. *An improved Bayesian network method for reconstructing gene regulatory network based on candidate auto selection*. BMC genomics, vol. 18, no. 9, pages 17–30, 2017.

[Yuan & Malone 2013] Changhe Yuan and Brandon Malone. *Learning Optimal Bayesian Networks: A Shortest Path Perspective*. J. of Artificial Intelligence Research, vol. 48, pages 23–65, 2013.

[Yuille & Kersten 2006] Alan Yuille and Daniel Kersten. *Vision as Bayesian inference: analysis by synthesis?* Trends in cognitive sciences, vol. 10, no. 7, pages 301–308, 2006.

[Zhang & Ji 2011] Lei Zhang and Qiang Ji. *A Bayesian network model for automatic and interactive image segmentation*. IEEE Transactions on Image Processing, vol. 20, no. 9, pages 2582–2593, 2011.

Title: Exact Methods for Bayesian Network Structure Learning and Cost Function Networks

Abstract: Discrete Graphical Models (GMs) represent joint functions over large sets of discrete variables as a combination of smaller functions. There exist several instantiations of GMs, including directed probabilistic GMs like Bayesian Networks (BNs) and undirected deterministic models like Cost Function Networks (CFNs). Queries like Most Probable Explanation (MPE) on BNs and its equivalent on CFNs, which is cost minimisation, are NP-hard, but there exist robust solving techniques which have found a wide range of applications in fields such as bioinformatics, image processing, and risk analysis.

In this thesis, we make contributions to the state of the art in learning the structure of BNs, namely the Bayesian Network Structure Learning problem (BNSL), and answering MPE and minimisation queries on BNs and CFNs.

For BNSL, we discover a new point in the design space of search algorithms, which achieves a different trade-off between inference strength and speed of inference. Existing algorithms for it opt for either maximal strength of inference, like the algorithms based on Integer Programming (IP) and branch-and-cut, or maximal speed of inference, like the algorithms based on Constraint Programming (CP). We specify properties of a specific class of inequalities, called cluster inequalities, which lead to an algorithm that performs much stronger inference than that based on CP, much faster than that based on IP. We combine this with novel ideas for stronger propagation and more compact domain representations to achieve state-of-the-art performance in the open source solver ELSA (Exact Learning of bayesian network Structure using Acyclicity reasoning).

For CFNs, we identify a weakness in the use of linear programming relaxations by a specific class of solvers, which includes the award-winning open source ToulBar2 solver. We prove that this weakness can lead to suboptimal branching decisions and show how to detect maximal sets of such decisions, which can then be avoided by the solver. This allows ToulBar2 to tackle problems previously solvable only by hybrid algorithms.

Keywords: combinatorial optimisation, weighted constraint network, local consistency, bayesian network

---

Titre: Méthodes Exactes pour l'Apprentissage de la Structure d'un Réseau Bayésien et les Réseaux de Fonction de Coût

Résumé: Les modèles graphiques discrets représentent des fonctions jointes sur de grands ensembles de variables en tant qu'une combinaison de fonctions plus petites. Il existe plusieurs instanciations de modèles graphiques, notamment des modèles probabilistes et dirigés comme les réseaux Bayésiens, ou des modèles déterministes et non-dirigés comme les réseaux de fonctions de coûts. Des requêtes comme trouver l'explication la plus probable (MPE) sur un réseau Bayésiens, et son équivalent, trouver une solution de coût minimum sur un réseau de fonctions de coût, sont toutes les deux des tâches d'optimisation combinatoire NP-difficiles. Il existe cependant des techniques de résolution robustes qui ont une large gamme de domaines d'applications, notamment les réseaux de régulation de gènes, l'analyse de risques et le traitement des images.

Dans ce travail, nous contribuons à l'état de l'art de l'apprentissage de la structure des réseaux Bayésiens (BNSL), et répondons à des requêtes de MPE et de minimisation des coûts sur les réseaux Bayésiens et les réseaux de fonctions de coût.

Pour le BNSL, nous découvrons un nouveau point dans l'espace de conception des algorithmes de recherche qui atteint un compromis différent entre la qualité et la vitesse de l'inférence. Les algorithmes existants optent soit pour la qualité maximale de l'inférence en utilisant la programmation linéaire en nombres entiers (PLNE) et la séparation et évaluation, soit pour la vitesse de l'inférence en utilisant la programmation par contraintes (PPC). Nous définissons des propriétés d'une classe spéciale d'inégalités, qui sont appelées «les inégalités de cluster» et qui mènent à un algorithme avec une qualité d'inférence beaucoup plus puissante que celle basée sur la PPC, et beaucoup plus rapide que celle basée sur la PLNE. Nous combinons cet algorithme avec des idées originales pour une propagation renforcée ainsi qu'une représentation de domaines plus compacte, afin d'obtenir des performances dépassant l'état de l'art dans le solveur open source ELSA (Exact Learning of bayesian network Structure using Acyclicity reasoning).

Pour les réseaux de fonctions de coût, nous identifions une faiblesse dans l'utilisation de la relaxation continue dans une classe spécifique de solveurs, y compris le solveur primé «ToulBar2». Nous prouvons que cette faiblesse peut entraîner des décisions de branchement sous-optimales et montrons comment détecter un ensemble maximal de telles décisions qui peuvent ensuite être évitées par le solveur. Cela permet à ToulBar2 de résoudre des problèmes qui étaient auparavant solvables uniquement par des algorithmes hybrides.

Mots-clés: optimisation combinatoire, réseau de contraintes pondérées, cohérences locales, réseau bayésien