Louvain School of Engineering

Ph.D. Thesis

Hélène Verhaeghe

The extensional constraint

Jury Members:
Prof. Yves Deville
Prof. Claude-Guy Quimper
Prof. Jean-Charles Régin
Prof. Peter Van Roy

Advisors:
Prof. Pierre Schaus
Prof. Christophe Lecoutre

2021

Abstract

Extensional constraints are crucial in Constraint Programming. They represent allowed combinations of values for a subset of variables (the scope of the constraint) using extensional representation forms such as tables (lists of tuples of constraint solutions) or `MDD`*s* (layered acyclic directed graphs where each path represents a constraint solution). Such extensional forms allow the modelization of virtually any kind of constraints.

This type of constraint is among the first ones available in constraint solvers. A lot of progress has been made since the original design of the first propagator of table constraints: advanced use of supports, simple tabular reduction, bitwise computations, reseting opperations, table compression, and `MDD`*s*. The most recent algorithm prior to this thesis is `Compact-Table`. It advantageously uses a data structure called reversible sparse bitsets to speed up the computations.

The work in this thesis initiates with `Compact-Table`. The goal is to extend it to handle other kinds of extensional representation. The first addressed representation is about compressed tables, i.e. tables containing tuples wich do not only contain single values but also simple unary ($*$, $\neq v$, $\leq v$, $\geq v$, $\in S$, $\notin S$) or binary ($= x + v$, $\neq x + v$, $\leq x + v$, $\geq x + v$) restrictions. One such compressed tuple allows representing several classical ones. This led to the $\texttt{CT}^*$ and $\texttt{CT}^{bs}$ algorithms, handling respectively short and basic smart tables. The second addressed issue concerns negative tables, i.e. tables listing forbidden combinations of values. This results in the $\texttt{CT}_{neg}$ and $\texttt{CT}^*_{neg}$ algorithms, handling respectively negative and negative short tables. The third and last adaptation addresses diagram structures, i.e. graphs such as `MDD`*s* or other layered graphical structure. This led to the `CD` and $\texttt{CD}^{bs}$ algorithms, handling respectively diagrams and basic smart diagrams.

Being able to use such diversity of representation helps to counter-

balance the main drawback of classical table representations, which is their potentially exponential growth in size. Compressed tables, negative tables, and diagrams help reduce the memory consumption (storage size) required to store an equivalent representation.

Acknowledgements

Firstly, I would like to thank my advisors, Prof. Pierre Schaus and Prof. Christophe Lecoutre, for their guidance during my thesis. They help me grow into the researcher I'm now, guiding me, teaching me, allowing me to research some of my own ideas, giving me some great opportunities,... I am thankful to Prof. Yves Deville and Prof. Siegfried Nijssen, the members of my thesis committee. I would also like to thank all the jury members, Prof. Yves Deville, Prof. Peter Van Roy, Prof. Jean-Charles Régin, and Prof. Claude-Guy Quimper, for their constructive insights, which helped improve my thesis.

I am also grateful for all the researchers I have had the pleasure to discuss with at conferences or/and work with on papers. I would also thank the CP community for the great conference I had the pleasure to attend.

I am extremely thankful to my friends and colleagues at the INGI department for the great environment in which I had the chance to work. More precisely, I would like to thank Vanessa M. and the secretary team for all the logistic help they are bringing to everyone; John A., Gaël A. and Ratheil H., who shared my office during these years; Guillaume D., with whom I had had so much interesting whiteboard talks; the members of the AiA group, for all the great time passed in conferences and all the great games of Perudo; and, Charles T., Fabien D., Gorby K., Mathieu J., and all the others for all the discussions during breaks, card games, RPGs, anime nights, board games and other various social activities.

Furthermore, many thanks to an understanding family and friends that supported me in this effort. I also thank my parents for giving me the opportunities and experiences that have made me who I am.

Contents

Chapter 1

Introduction

It is nice to know that the computer understands the problem. But I would like to understand it too.

- Eugene Wigner

1.1 Introduction

Since a long time ago, mankind has sought efficiency in every task, from the invention of the wheel to the sending off rockets to space. The definition of efficiency depends on the context but includes a wide range of objectives such as decreasing the time taken by some actions, decreasing the quantity of some raw material used, increasing the profits,... generally while satisfying a set of constraints.

Optimality is defined as the most efficient way to do some tasks. Intuitively, people have sought optimality leading to the earliest definitions of greedy algorithms. However, scientists did not wait for the invention of computers to solve some optimality problems. In the 17^{th} century, Newton and Raphson designed the Newton–Raphson method, which aims at finding the optimums of functions. During the 18^{th} century, Lagrange invented the relaxation method of Lagrangian multiplicators. However, applying such methods to complex problems was not yet possible since every computation had to be done by hand.

Since the popularization of computers, and the non-stopping improvement of the hardware, automatized optimization has become more accessible to anyone. These factors enter into the successive improvements of optimization algorithms. All these improvements allowed to tackle more complex problems with more and more variables and constraints; for example, the birth in 1947 of the simplex algorithm, which aims to solve linear optimization problems.

Constraint Programming (CP) is a way to solve combinatorial optimization problems (problems dealing with finite domains) in an automated way. Typically, such problems are modeled using variables and constraints. The model is then fed to a CP solver in order to exhibit solutions.

This thesis is about one type of constraint available in CP solvers: extensional constraints.

Let us define an example of such constraints using online product configurators. Such software provides the user a list of attributes and values for each of them. For example, the configurator of a laptop seller (Fig. 1.1) will display attributes such as the size of the screen, the size, and type of the internal disk, the language of the keyboard,... Values are available for each attribute, such as 17', 15' or 13' for the screen size or 512 Gb, 1 Tb or 2 Tb for the disk size. The configurator also displays all the possible items the user can buy, in our example, each available computer. Each item corresponds to an attribution of one value to each characteristic. E.g., the iPear 500 has a screen of 13', with 512 Gb SSD, the NVision 516 graphic card,... The user can interact with the configurator by selecting some values for some attributes. For each attribute, the user can select a single value or a subset of the initially available values. The configurator then reduces the available options by removing the items not valid anymore regarding the attributes' remaining values. The software also removes some values from other attributes when they

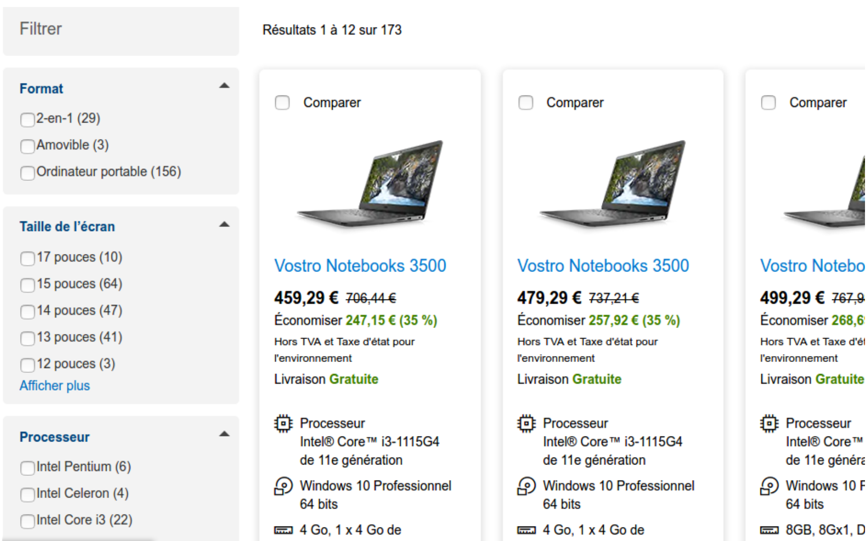


Figure 1.1: Screenshot of the online product configurator of Dell (`www.dell.com`).

are no longer part of a possible combination. For example, if a user selects 17' as screen size, the configurator removes the ones with other screen sizes but may also remove the 128 Gb SSD option, for example, as no computers with 17' are equipped with such a disk.

Formally, attributes correspond to the variables involved in the constraints. Each variable has some values possible, corresponding to each of the values of the related attributes. The set of possible items is called the table. Each item is called a tuple of the table. The action of removing items after selecting a subset of valid values is called updating the table. The action of removing some values because an item is not in the table anymore is called filtering. This type of constraint is called a table constraint, which is a kind of extensional constraint.

On a more global view, extensional constraints are among the oldest and most generic families of constraints in the Constraint Programming paradigm. It links some variables to an explicit definition of the solution of the constraint. The two most known representations are the table and the multi-valued decision diagram (abbreviated `MDD`). In the table constraint version, the constraint solutions are listed in a simple table, each row corresponding to one solution. In the `MDD` constraint version, solutions are represented as a multi-valued decision diagram where each path corresponds to a solution.

However, in many cases, extensional constraints can have many entries in the table (its oldest version). Some other constructs were designed to reduce the size of the input. Such constructs are the addition of unary and even binary compression elements as values of the tables (for example, $\langle * \rangle$, $\langle \neq v \rangle$,...), using complementary tables (i.e. the negative tables), using diagrams.

A new algorithm, called Compact-Table, was introduced in 2015 [DHL$^+$16]. This table-oriented propagator for extensional constraints uses bitwise operations to speed up the propagation. Since this new algorithm only helps the regular positive table, it was decided to adapt it to some of the extensional constraint variations.

The work done can be summarized by the schema in Fig. 1.2. All of the work achieved in this thesis starts from the Compact-Table algorithm (`CT`). It uses bitwise operations through a particular data structure called the reversible sparse bitset. This structure helps speeding up the computation in comparison with previous table constraint algorithms. This algorithm was extended following three orthogonal directions. The first one extends the algorithm to handle tables using compression elements such as $\langle * \rangle$, $\langle \leq v \rangle$,... leading to the $\texttt{CT}^*$ and $\texttt{CT}^{bs}$ algorithms. The second one extends the algorithm to handle negative tables (i.e. complement tables), leading to the $\texttt{CT}_{neg}$ and $\texttt{CT}^*_{neg}$ algorithms. The last one extends the algorithm to handle another representation of extensional constraint, the diagram (the most known type of diagram being the Multi-Valued Decision Diagram, i.e. the `MDD`) leading to the `CD` and $\texttt{CD}^{bs}$ algorithms.

This thesis is organized into four parts. The first part explains the state of the art. It is composed of three chapters. The first one (Chap. 2) introduces the Constraint Programming paradigm's bases. The second (Chap. 3) presents the history of extensional constraints. The last one (Chap. 4) details data structures widely used in this thesis. The second part explains in detail the two input structures possible in the extensional constraint, namely the table (Chap. 5) and the diagram (Chap. 6). The last part details the various propagators designed regarding the three main axes, with one chapter per ax. First, the positive table with the addition of compression (Chap. 7), then the negative tables (Chap. 8) and finally the diagrams (Chap. 9).

1.2 Contributions

The work in this thesis led to the publication of several papers in various conferences. The contributions are:

– A first conference paper at AAAI17 on the extension of Compact-Table to short tables, negative tables, and negative short tables.

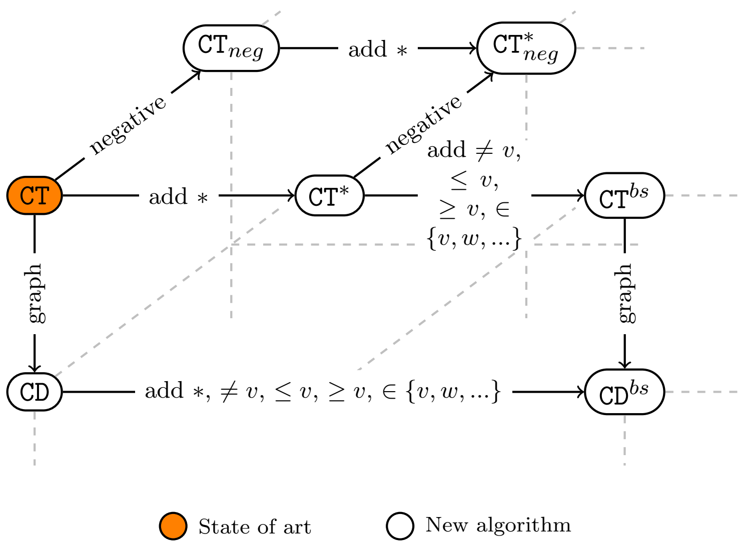


Figure 1.2: Links between the various algorithms developped during this thesis.

A summary of this paper was also accepted at JFPC17.

  - Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-table to negative and short tables. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence and the Twenty-Ninth Innovative Applications of Artificial Intelligence Conference*, volume 5, 2017
  - Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extension de compact-table aux tables négatives et concises. In *Treizièmes journées Francophones de Programmation par Contraintes (JFPC17)*, 2017

- A second conference paper at CP2017 on the extension of Compact-Table to basic smart tables. A summary of this paper was also accepted at JFPC18.

  - Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extending compact-table to basic smart tables. In *International Conference on Principles and Practice of Constraint Programming*, pages 297–307. Springer, 2017
  - Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extension de compact-table aux tables simplement intelligentes. In *Quatorzièmes journées Francophones de Programmation par Contraintes (JFPC18)*, 2018

- A third conference paper at IJCAI18 on Compact-Diagram, the adaptation of Compact-Table to MDD (and layered graph in general). A summary of this paper was also accepted at JFPC19.

  - Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bitsets. 2018
  - Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-diagram propagateur efficace pour la contrainte (s)MDD. In *Quinzièmes journées Francophones de Programmation par Contraintes (JFPC19)*, 2019

- A fourth conference paper at CPAIOR19 on Compact-Diagram to basic smart MVDs. A summary of this paper was also accepted at JFPC19.

  - Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-diagram to basic smart multi-valued variable diagrams. 2019

– Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extension de compact-diagram aux smart MVD. In *Quinzièmes journées Francophones de Programmation par Contraintes (JFPC19)*, 2019

The contributions also include the open-source implementation of the algorithms described in the papers in OscaR [Tea].

Besides, but not directly related to this thesis's work, a fifth paper was submitted to CP2019 on learning optimal decision trees using CP. This paper was accepted to the journal fast track of the conference. A two-page summary of this paper was also accepted at BENELEARN19. We also were invited to present a 4-page extended abstract to the sister conference track at IJCAI20. A summary of this paper was also accepted at JFPC21.

– Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. In *The 25th International Conference on Principles and Practice of Constraint Programming (CP2019)*, 2019

– Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. In Katrien Beuls, Bart Bogaerts, Gianluca Bontempi, Pierre Geurts, Nick Harley, Bertrand Lebichot, Tom Lenaerts, Gilles Louppe, and Paul Van Eecke, editors, *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019*, volume 2491 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019

– Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming (extended abstract). In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4765–4769. ijcai.org, 2020

– Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. pages 1–25. Springer, 2020

– Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Apprentissage d'arbres de décision

optimaux grâce à la programmation par contraintes. In *Seizièmes journées Francophones de Programmation par Contraintes (JFPC21)*, 2021

Part I

Background

Chapter 2

The Constraint Programming Paradigm

Computer science inverts the normal. In normal science, you're given a world, and your job is to find out the rules. In computer science, you give the computer the rules, and it creates the world.

- Alan Kay

2.1 Introduction

This chapter gives an introduction to the main concepts behind the Constraint Programming (CP) paradigm.

CP is a declarative way to solve combinatorial optimization problems. The user's job is to describe what should be a solution and not how to find it. The solver is let to decide how to solve the problem. Due to that, it is often considered close to the holy grail of solving problems [Fre96].

This paradigm has also proven its use over other techniques in several domains such as scheduling [RP97, BLPN12, Lab03, LM12, LRSV18] and data mining [DRGN10, Gun15, SAG17]. Lately, a growing interest has appeared for the use of Constraint Programming in order to solve machine learning problems [DRGN10, ANS20, BOP20, VNP$^+$20a, CMR$^+$20].

More can be learned on CP by reading [Apt03], [RVBW06] or [Lau18].

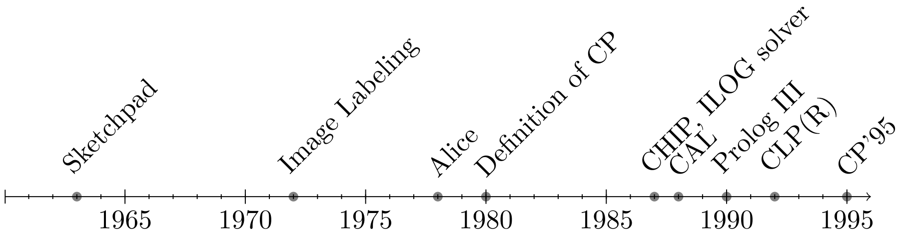


Figure 2.1: Chronology of the big milestones leading to Constraint Programming.

2.2 A Brief History

The Constraint Programming paradigm finds its early roots in the sixties where Sketchpad [Sut64], the ancestor of modern computer-aided design programs, was designed by Sutherland during his Ph.D. thesis. This program is considered as one of the earliest constraint systems. The reasoning is based on a relaxation method, starting from a given assignment of values to variables, constraints are then used to adapt these values to be respected.

Waltz [Wal72] was the first to use a domain reduction method in his image labeling software in 1972. His software is the first to have variables, domains, and constraints to eliminate the domains' values.

A bit later, in 1978, ALICE [Lau78] is introduced by Lauriere. He defined it as *"A language and a program for stating and solving combinatorial problems"*. This language is the first to introduce the AllDifferent constraint [vH01].

In 1980, Steel obtained his Ph.D. thesis with his dissertation called *"The definition and implementation of a computer programming language based on constraints"* [SJ80], defining formally for the first time what is Constraint Programming.

During the eighties and the nineties, several researches around the world were made on Constraint Programming and several frameworks and languages appeared: in Japan, CAL [ASS+88], GDCC [THS+92] and cuProlog [Tsu92], in France, Prolog III [Col90], in Europe, CHIP [DSVH87] and in Australia, CLP(R) [JMSY92].

As Constraint Programming begins to solve practical problems such as scheduling problems, commercial usage begins to appear in the nineties with commercial systems such as Charme, CHIP V4, and ILOG solver.

In the meantime, in 1995, the first edition of the International Conference on Principles and Practice of Constraint Programming was held in France (Fig.2.2).

2.3 What is CP?

Constraint Programming is a paradigm which solves combinatorial problems such as *constraint satisfaction problem* (CSP) (Def. 2.1) and *constraint optimization problem* (COP) (Def. 2.2).

Definition 2.1. ***Constraint Satisfaction Problem*** *(conventionnal definition as given p.16 of [RVBW06])*
A Constraint Satisfaction Problem (CSP) is a triple $\mathcal{P} = \langle X, D, C \rangle$ *where* X *is an* n*-tuple of variables* $X = \langle x_1, x_2, \ldots, x_n \rangle$*,* D *is a corresponding* n*-tuple of domains* $D = \langle D_1, D_2, \ldots, D_n \rangle$ *such that* $x_i \in D_i$ *and* C *is a* e*-tuple of constraint* $C = \langle C_1, C_2, \ldots, C_e \rangle$*. A constraint* C_j *is a pair* $\langle R_{S_j}, S_j \rangle$ *where* R_{S_j} *is a relation on the variables in* $S_i = scope(C_i)$*. In other words,* R_i *is a subset of the Cartesian product of the domains of the variables in* S_i*.*

Definition 2.2. ***Constraint Optimization Problem***
A Constraint Optimization Problem (COP) is a CSP with a cost function over the variables that must be minimized or maximized.

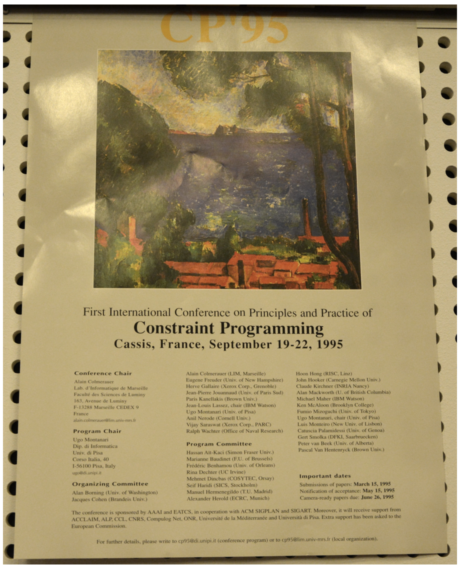


Figure 2.2: Poster of the first edition of the International Conference on Principles and Practice of Constraint Programming (CP'95, Cassis, France).

We often represent the variables and constraints as a *constraint network* (CN). Each variable x is represented by a node in this network and each constraint c is represented by an arc (or hyperarc) between $\texttt{scp}(c)$ nodes.

To solve such a problem with a CP solver, one must first model the problem using variables, domains, and constraints. Secondly, a search has to be chosen to find solutions. The CP paradigm is therefore often summarized by the following equation:

$$CP = MODEL + SEARCH$$

2.3.1 Modeling a Problem

A *model* is a high-level formulation of a problem. It formulates a problem in terms of *variables*, *domains*, and *constraints* involving some of the variables. A *domain* is a finite set of possible values associated to each variable. A *constraint* is a restriction of the associations of values allowed for the variables involved. The model is a declarative expression of the solutions to the problem. Several formulations can exist for the same problem. A solution for a problem consists in assigning each variable with a value from its domains, while respecting the constraints.

Let us take the example of the n-queens problem. Given a chessboard of size n by n, how should n queens be placed such that none of them can attack each other. As a reminder, two queens can attack each other if they are on the same line, column, or diagonal.

A *first* model consists of having one variable by square of the board (x_{ij}, with $i, j \in \{1, 2, \dots, n\}$); each of them having as domain $\{0, 1\}$, 0 meaning the square is empty, 1 meaning it contains a queen. For each line, at most 1 queen can be placed on it. This is modeled by a constraint *atMost*, on each variable forming a line, ensuring the maximum number of occurrences of the value 1 is 1. The same is done for each of the columns and diagonals.

A *second* model is built with n variables, one by column (x_i, with $i \in \{1, 2, \dots, n\}$); each of them represents one queen's position (i.e. which line is it on) that belongs to this column. The domain of these variables is $\{1, 2, \dots, n\}$. An *AllDifferent* constraint is added between all the variables to ensure a given line number can be assigned to a single queen. For the diagonals, the following arithmetic constraints ensure there are no two queens on the same diagonal.

$$|x_j - x_i| \neq j - i \qquad \forall i, j \in \{1, 2, \dots, n\}, i < j$$

The efficiency of a model depends on various factors, such as the number of variables and constraints (each additional variable may increase the depth of the search tree), the constraints used (some constraints are more efficient than others), if the model is used in isolation or is part of a bigger problem.

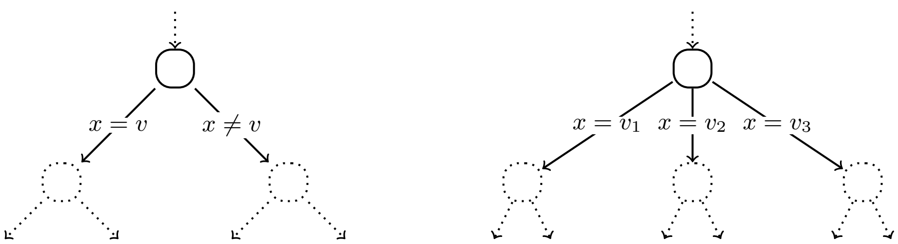
2.3.2 Searching for a Solution

The second part of the paradigm is about the search. The search works as a depth-first exploration of the search space. This is done by developing a search tree (i.e. a tree representation of the search space). The shape of the search tree is also given by the user. Generic ones can be used. A more complex search tree based on known heuristics can also be used depending on what the user knows about its problem.

The search tree is composed of nodes and leaves. Leaves contain a state where every variable has been assigned to one value. The nodes contain a decision. Two decisions are mainly used: the *binary* (Fig. 2.3a) and the *non-binary* (Fig. 2.3b). The *binary* node selects a variable and a value from its domain. In the first branch, the value is assigned to the variable. In the second one, the value is removed from the domain. The *non-binary* node selects a variable. It has one branch per value in the domain. Each of these branches assigns the value to the variable.

Some search trees defined as *static* are fully known from the beginning. Some are dynamic, and their building depends on the evolution of the domains during the search. An example of a static search tree is the lexical one. The variables are ordered at the beginning. At a given depth i, the i^{th} variable of the sequence is selected and a *non-binary* node is done. An example of a dynamic search tree is the first fail one. The variable with the smallest current remaining domain is selected, and a *binary* node is done using one of the remaining values from the domain.

Generally, a smaller search tree (its size is the number of nodes and leaves) allows a more efficient search since fewer nodes have to be ex-



(a) *binary* search node (b) *non-binary* search node

Figure 2.3: Example of the different search nodes.

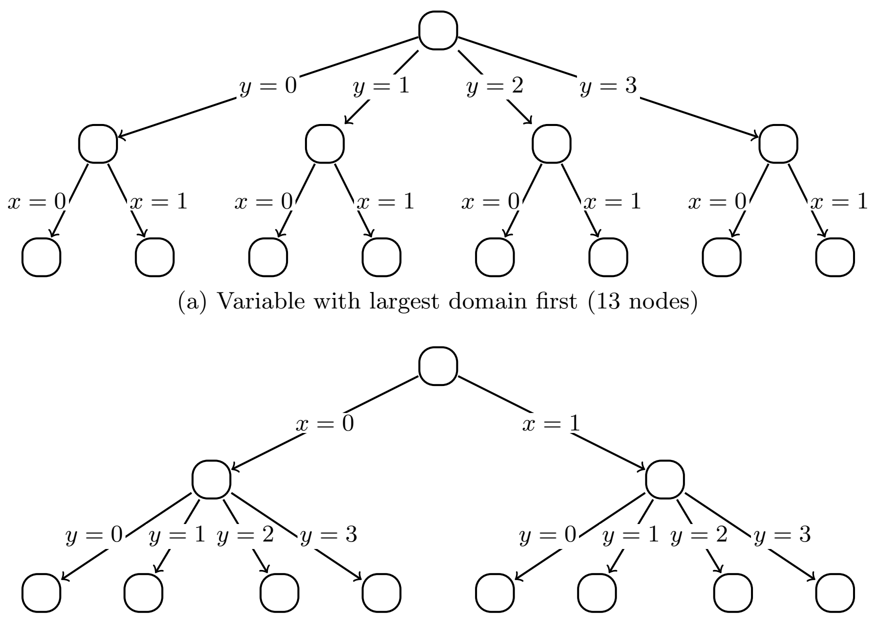
plored. Figure 2.4 shows how starting with variables with smaller domains generally helps reduce the size of the search tree. The first subtree (Fig. 2.4b) has 13 nodes/leaves. The second (Fig. 2.4a) has 11 nodes/leaves.

2.4 Components of a CP Solver

The solver works with five main components: the variables, the constraints propagators, the fix-point algorithm, the search algorithm, and the state restoration mechanism.

2.4.1 The Variables

Each variable has an associated domain. The solver can ask them which values remain, whether they are bound, whether their domain still contains at least one value, to remove a value from their domains, which values were removed from the domain since last propagation,...



(a) Variable with largest domain first (13 nodes)

(b) Variable with smallest domain first (11 nodes)

Figure 2.4: Example demonstrating the impact on the number of node of different searches ($\texttt{dom}(x) = \{0, 1\}$ and $\texttt{dom}(y) = \{0, 1, 2, 3\}$).

2.4.2 The Constraints Propagators

The constraints contain the algorithms responsible for filtering out impossible remaining values out of the domains based on their current values. These algorithms are called propagators. Each constraint may have several propagators of different efficiency.

One can measure the efficiency of a propagator using the propagation strength. Two of the most commonly used strengths are the bound consistency (Def. 2.3) and the generalized arc consistency (Def. 2.4). A greater strength often implies a more complex and time-consuming algorithm. However, it can also reduce the search's size and, thus, the number of times the propagator is called. It is thus not straightforward to choose the right propagator.

Definition 2.3. ***Bound Consistency***
Several bound consistency exist [CHLS06]. The two most use when dealing with integer domains are the bound(D) consistency *and* bound(Z) consistency*.*

To achieve bound(D) consistency *(BC(D)), also called range consistency, for each variable involved in the constraint, for every value of its domains, a solution should be possible by selecting an integer for each other variables which is between the lower and upper bound (this integer may not be part of the domain).*

To achieve bound(Z) consistency *(BC(Z)), for each variable involved in the constraint, for the lower and upper bounds of its domains, a solution should be possible by selecting an integer for each other variables which is between the lower and upper bound (this integer may not be part of the domain).*

Definition 2.4. ***Generalized Arc Consistency***
A constraint achieves generalized arc consistency (GAC) if, after its propagation, all the remaining values of each variable involved in the constraint are part of possible solutions of the constraint. This means that, for each variable x involved in the constraint, for each value v of the domain of x, there should exist the possibility of a valid assignment of all the variables, with x assigned v such that the constraint is respected.

Here is a simple example to show the difference between BC and GAC. Given the variable X, Y and Z and their respective domains $\texttt{dom}(X) = \{0, 4\}$, $\texttt{dom}(Y) = \{0, 1, 2\}$ and $\texttt{dom}(Z) = \{0, 1, 2, 3, 4, 5, 6, 7\}$, and the constraint $X + Y = Z$. A BC propagator looks at each variable's bound and determines that 7 cannot be part of the domain of Z since no combination of values within the range of the bounds of the domains of X and Y give 7. However, as 6 can result from $2 + 4$, the propagator stops there, and only 7 is removed here. The full analysis of the BC propagator is summarized in Fig. 2.5a. A GAC propagator looks for each value of each variable and verifies that a solution is possible. In addition to the check for the bounds (Fig. 2.5a), it also checks other values inside the domains (Fig. 2.5b). In this case, the inner domain value 3 is removed from the domain of Z in addition to the removal of 7.

2.4.3 The Fix-Point Algorithm

The fix-point algorithm's goal is to move the solver state to a new stable state or a failure state. A stable state can be defined as a state where none of the constraints can remove any value anymore. A failure state is a state where at least one variable does not have any value in its domain. The fix-point starts by creating a pool of constraints waiting to be called. In some solvers this pool is implemented using a priority queue. This pool is initialized with the constraints impacted (i.e. the constraints with some modified variables in their scope) by the decision

Tested	Support	Result
$X = 0$	$Y = 0$ & $Z = 0$	LB of X consistent
$X = 4$	$Y = 2$ & $Z = 6$	UB of X consistent
$Y = 0$	$X = 0$ & $Z = 0$	LB of Y consistent
$Y = 2$	$X = 4$ & $Z = 6$	UB of Y consistent
$Z = 0$	$X = 0$ & $Y = 0$	LB of Z consistent
$Z = 7$	?	UB of Z unconsistent
$Z = 6$	$X = 4$ & $Y = 2$	UB of $Z = 6$ consistent

(a) Bound Concistancy Analysis

Tested	Support	Result
$Y = 1$	$X = 0$ & $Z = 1$	$Y = 1$ consistent
$Z = 1$	$X = 0$ & $Y = 1$	$Z = 1$ consistent
$Z = 2$	$X = 0$ & $Y = 2$	$Z = 2$ consistent
$Z = 3$	?	$Z = 3$ unconsistent
$Z = 4$	$X = 4$ & $Y = 0$	$Z = 4$ consistent
$Z = 5$	$X = 4$ & $Y = 1$	$Z = 5$ consistent

(b) Rest of the GAC Analysis

Figure 2.5: Analysis of the filtering performed by BC(Z) and GAC propagators on $X + Y = X$.

of the node inside which the fix-point is called. The pool is processed one constraint at a time until either the pool is empty (stable state) or a failure is detected (failure state). Each time a constraint filters out some values from a domain, the impacted constraints are added to the pool.

2.4.4 The Search Algorithm

The search algorithm is responsible for finding the solution(s) of the problem. It works generally as a depth-first exploration of the search space, starting with a node, thoroughly exploring the first of its children before exploring the second one, and so on. Each node represents a restriction of the initial problem with some values removed from the domains. Except for the root node, each node represents the sub-problem of its parents where an additional decision (a given reduction of the domains) has been applied.

When reaching a node, the search first applies the decision associated with it (typically, assigning or removing a given value from a variable domain). Then, it runs the fix-point algorithm. What the search does next depends on its result.

- If the state is stable and all variables are bound to values, then a solution is reached. The node has no children and is called a solution leaf. The user is notified, the search goes back and can continue.

- If the state is stable with remaining unbound variables, then it will continue the depth-first exploration of the tree and explore the children.

- If the state is failing, then no solutions can be found by continuing the exploration in this branch. The search goes back and continues to another branch. The node is called a dead leaf.

2.4.5 The State Restoration Mechanism

When the search has to go back to explore a new branch, it must also restore the state as it was in the parent node. This is called *backtracking* and is done by the state restoration mechanism. *Checkpoints saves* are made at the end of each explored nodes of the search tree, and a *restore* operation helps retrieve the state. The solver is said *copy*-based if the restorations are made by retrieving a stored copy of the wanted state. The solver is said *trail*-based if the restorations are made using the changes in the state since the last checkpoint.

2.5 Conclusion

In this thesis, we will focus on one important component of constraint solvers, which corresponds to the constraints propagators, and, more precisely, those for constraints defined in extensional forms. We call them extensional constraints as they capture using an explicit representation all allowed combinations of values (constraint's solutions). This category includes the table and the multi-valued decision diagram constraints (or `MDD` constraints). They are considered as universal since any other constraint can be expressed as an extension constraint.

The solver used for this thesis implementations and experiments is Oscar [Tea]. It is a trail-based backtracking open-source solver written in Scala. Thus, the following algorithms are defined using the trail state restoration mechanism, but they could easily be adapted to a copy-based solver (as already done for `CT` [IS18]).

Chapter 3

Extensional Constraints

People think that computer science is the art of geniuses, but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

- Donald Knuth

3.1 Introduction

The extensional constraints family is a family of global constraints. Their common point is that they explicitly contain all their solutions. Two well-known explicit representations of the set of solutions are:

– using a table: A table constraint C associates a set of variable `scp`(C), called the scope, to a set of tuples, called the table. Each of them is associating a value for each variables. Each tuple represents a solution to the constraint. Figure 3.1a shows an example of a simple table constraint.

– using a diagram, such as a multi-valued decision diagram (an MDD): An MDD constraint C associates a set of variable `scp`(C), called the scope, to a labeled, layered acyclic directed graph with decision nodes, called the MDD. Each of the layers of the diagram corresponds to a distinct variable from the scope. Each edge of a given level associated a value to the variable of the level. Each path within the diagram (from `ROOT` to `END`) represents a solution to the constraint. Figure 3.1b shows an example of a simple MDD constraint.

Extensional constraints are often considered as the most generic ones possible as they can represent any constraint whether a mathematical expression can easily represent them or not.

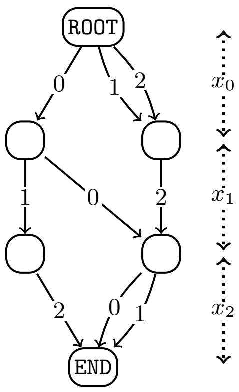
This chapter highlights the main concepts introduced over the years about the extensional constraint while looking at a quite exhaustive list of the propagators designed over time.

3.2 History of Extensional Constraints

The timeline Fig. 3.2 displays the various propagators for the extensional constraints over time. Each propagator brought new improvements: supports, reset, MDD, bitwise operations,...

	x_0	x_1	x_2
τ_0	0	0	0
τ_1	0	0	1
τ_2	0	1	2
τ_3	1	2	0
τ_4	1	2	1
τ_5	2	2	0
τ_6	2	2	1

(a) An example of a table constraint



(b) An example of an MDD constraint

Figure 3.1: Examples of extensional constraints.

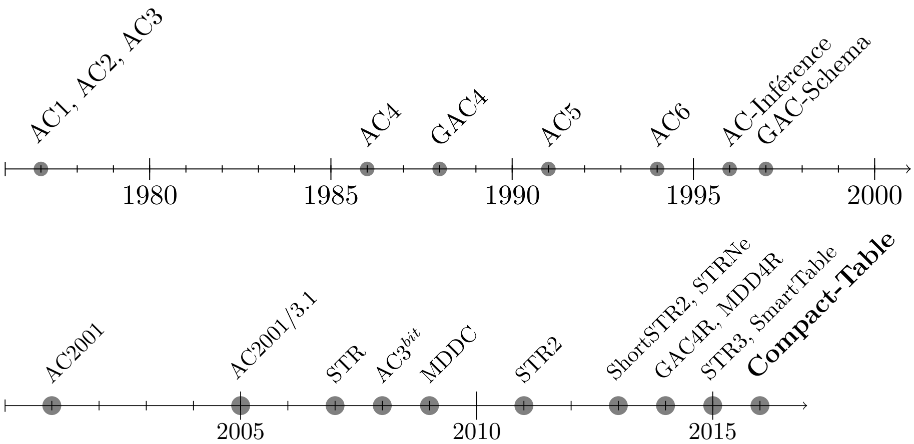


Figure 3.2: History of the work on extensionnal constraint.

3.2.1 Genesis of Constraints, Propagation, and Filtering

The history of extensional constraints goes way back in time. Starting in 1977 with the birth of the idea of arc-consistency (algorithms `AC1`, `AC2` and `AC3` [Mac77]). Arc-consistency is the generalized arc consistency (Def. 2.4) applied to a pair of variables. At the time, the scope of the considered constraints was restricted to two. The three algorithms are reasoning on the network of variable and constraint (each node represents a variable and each edge, a constraint between two variables).

`AC1`. `AC1` first checks for the unary constraints. Then it revises the whole set of binary constraints in search of values to be removed. If at least one value is removed, it revises the whole set again. It only stops when none of the revisions triggered any changes.

`AC2`. `AC1` can clearly waste time since not all binary constraints must be revised again at each time of the main loop. Only those affected by a change (i.e. those involving a modified variable) should be revised again. `AC2` improves this by collecting in a set the arcs to be revised again. This set is used in the next iteration.

`AC3`. `AC2` is somewhat inefficient in some cases. A constraint can be added to the constraints to be revised at the next loop while still in the set of constraints to be revised at this loop. In this case, there is no need to add it to the future set since it will be revised in the current loop. `AC3` is the result of this optimization.

The `revise` method is the genesis of any constraints (propagation and filtering), while the AC algorithms were the genesis of the fix-point algorithm.

3.2.2 `AC4` and the Support

A few years later, [MH86] revisited the AC algorithms, leading to the `AC4` version. This version introduces the concept of *support*. It states that, given a variable x and a value $a \in \texttt{dom}(x)$, as long as a admits a support from each of the variables linked to x in the network, a remains a possible value for x. A support for a in a variable y being a value $b \in \texttt{dom}(y)$ such as $x = a \wedge y = b$ is allowed.

To keep track of the supports, counters and sets of supports were introduced for each constraint and each value of its variables. These counters are decreased when a value is removed, and propagation is triggered to another pair variable value if a counter is down to 0.

AC5 [DVH], AC2001 [BR01] and AC2001/3.1 [BRYZ05] are other improvements of the algorithm. Another improved algorithm, AC6 [Bes94], manages to improve the space complexity of AC4 while keeping its optimal worst-case time complexity. AC-Inference [Rég95] is another developed algorithm which differs a bit from the other AC algorithms. Instead of systematically make the checks for support for each pair of variable-value, it uses the computations of some supports to already deduce some other support, removing the need to compute them.

3.2.3 GAC4

GAC4 [MM88] is the generalization of AC4 to more than two variables. It also uses a table as input. A support for (x, a) is thus a tuple τ for which the value associated to x is a. For each (x, a), the set of support is built. When the support set is empty, the value is removed from the domain of the variable.

3.2.4 AC3^{bit} and the First Bitwise Approaches

Using bitwise operations to speedup computations is not a new concept. Already in 1979, [McG79] used bitvectors to represent domains and set of supports. Ullmann [Ull76] did some optimizations based on bitwise operations. In 1996, Bliek [Bli96] demonstrated the potential of representing constraints using bit vectors to speed up the propagation. Also in 1996, in his thesis [Rég95], J-C Régin made use of a set representation, and more precisely bit vectors, to speedup the propagation of table constraint. He proposes an adaptation of AC-Inférence based on the use of bit vectors.

In 2008, another approach using bitwise operation is designed. In the AC3^{bit} [LV08], the domains are defined using bitsets. The supports are also stored using bitsets which allow to quickly compute the remaining values by intersecting the current domain with the supports using bitwise operations. The introduction of these bitsets allows a speedup compared to AC3

3.2.5 The Simple Tabular Reduction (STR) Family

Simple Tabular Reduction (or STR for short) was first introduced in 2007 [Ull07]. The idea behind this GAC algorithm is to modify the table of the constraint dynamically: whenever tuples became invalid, they are removed from the table. The tuples currently valid represent the current table set. At each step of the propagation, the validity of each tuple is tested. A subset of the current table, corresponding to the detected invalid tuples, is discarded. This current table is stored using a sparse set made reversible to allow the state to restore itself during backtracking. Filtering is achieved by looking at each tuple in the current table set and gathering each valid value for each unbound variable. Values not gathered are then removed from domains.

In 2011, STR2 [Lec11] was proposed as an optimized version of the initial STR algorithm. It brings two main optimizations.

First, during filtering, a value is supported as soon as a valid tuple using this value is identified. Consequently, if all values of the domain of a given unbound variable have supporting tuples, there is no need to continue searching for more supporting tuples regarding this variable. This is done by keeping a set, $\mathtt{S}^{\mathtt{sup}}$, of variable yet without support for each value. This set is initialized with the unbound variables ($\mathtt{S}^{\mathtt{sup}} = \{x \in scp : |dom(x)| > 1\}$). A set `gacValue[x]`, for each variable x, is used to maintain the set of values yet without support during the filtering process. This set is initialized with the remaining values in the domain ($\mathtt{gacValue[x]} = \mathtt{dom}(x)$). Filtering is thus done only on variables in $\mathtt{S}^{\mathtt{sup}}$ until they are removed when `gacValue[x]` becomes empty.

Secondly, checking the validity of a tuple does not require testing values for each variable. As every tuple is valid regarding a given variable at the end of the propagation at a given node, if the domain has not changed at the next propagation, all the tuples still have a valid value for this variable. Validity thus does not require testing the value corresponding to the variable. This is done by checking the validity only on the set of variable S^{val}, initialized at the beginning of the propagation, which contains the variable whose domain has changed since the last propagation ($\mathtt{S}^{\mathtt{val}} = \{x \in \mathtt{scp}() : |\Delta_x| > 0\}$, where $|\Delta_x| = |\mathtt{lastSize[x]}| - |\mathtt{dom}(x)|$). A backtracked array `lastSize[x]` is kept for each variable to know where some changes have happened since the last propagation.

STR3 [LLY15] was published in 2015. Its crucial element is a data structure removing unnecessary traversal of the table. It is complementary to STR2 as it works better when the average number of remaining tuples in the table during the search stays high but worse in the opposite case.

3.2.6 MDDC: Arrival of the MDD

In 2009, Cheng and Yap [CY10] exploited the connection between tables and Multi-Valued decision diagrams (MDD*s*) [Bry86]. In an MDD, each path from ROOT to END corresponds to a tuple in a table. For example, on MDD in Fig. 3.1b, each path corresponds to a tuple of table in Fig. 3.1a. In some cases, the MDD can be exponentially smaller than the corresponding table. This is the motivation behind `mddc`, the propagator based on an `MDD` representation.

In `mddc`, they use the `MDD` structure to find more efficiently supports for each remaining value. Similarly, as the table was reduced in the STR family of algorithms, `mddc` maintains a reduced version of the initial `MDD`.

`MDD`*s* are also proven usefull outside the scope of extensionnal constraints: in the context of domain storage [AHHT07, HvHH10], the use of limited width `MDD`*s* to model some constraints [BCvH14], the use of `MDD`*s* to solve optimization problems [BCvHH16], .

3.2.7 ShortSTR2, SmartTable,... : The Arrival of Compressed Tables

The arrival of `MDD`-based propagators highlighted the biggest problem of tables: their size. From this point, all kinds of compressed tables were invented, each aiming to reduce its size. Following the introduction of new tables, new propagators using these compressed tables directly were introduced.

Here is a non-exaustive list of such compressed tables:

– short tables (Fig. 3.3a), i.e. tables allowing the $*$ value representing any possible value. `shortSTR2` [JN13] is a propagator handling this kind of table.

– c-tuples (Fig. 3.3b), i.e. tables allowing in tuples the presence of subsets of values instead of single values. The c-tuples are based on the concept of Global Cut Seed [FM01]. `STR2-c` and `STR3-c` [XY13, KW07] are some propagators adapted to handle such tuples.

– tuple sequences [Rég11] (Fig. 3.3c), i.e. tuples defined by a Global Cut Seed, a minimum tuple and a maximum tuple (given an ordering). Each tuple represented by the GCS within the defined bounds is in the table. This kind of table allows an easy representation of the complementary table.

– smart tables (Fig. 3.3d), i.e. tables allowing in tuples restrictions that can be unary constraints or binary constraints such as $*$, $\neq v$, $\leq v$, $\in S$, $= x + v$, $\geq x + v$, representing several values from the domain or even small binary constraint within the table. These tables can be handled by the `smart table` algorithm [MDL15].

– sliced tables (Fig. 3.3e), i.e. tables represented as several cardinal products between a prefix and smaller tables. The `STR-slice` algorithm [GHLR14] handles such tables.

– segmented tables (Fig. 3.3f), i.e a generalization of sliced tables, containing tuples made of possibly several sub-tables, simple values and/or the universal value $*$. The `SegmentedConstraint` algorithm [ALM20] handles such tables.

– ...

Most of the algorithms for filtering such tables use the technique of tabular reduction.

	x	y	z
τ_1	1	$*$	4
τ_2	0	1	$*$
τ_3	$*$	2	3
τ_4	2	0	2
τ_5	2	$*$	1

(a) Short table

	x	y	z
τ_1	$\{0,2\}$	$\{1,2,3\}$	$\{1,2\}$
τ_2	$\{0\}$	$\{0,1\}$	$\{0\}$
τ_3	$\{1,2\}$	$\{0,3\}$	$\{2,3\}$
τ_4	$\{1,3\}$	$\{2,3\}$	$\{0,1\}$
τ_5	$\{2,3\}$	$\{1,2\}$	$\{0,3\}$

(b) Table containing c-tuples

	min	max	GCS
τ_1	$(0,0,0)$	$(0,1,2)$	$(\{0\},\{1,2\},\{0,2\})$
τ_2	$(1,0,1)$	$(1,1,1)$	$(\{0,1,2\},\{0,1,2\},\{0,1,2\})$
τ_3	$(1,1,0)$	$(2,0,3)$	$(\{0,1,2\},\{1,3\},\{0,2\})$
τ_4	$(0,2,0)$	$(2,0,0)$	$(\{0,2\},\{1,3\},\{0,1,2\})$
τ_5	$(2,0,0)$	$(2,1,2)$	$(\{2,3\},\{1,2\},\{0,1,2\})$

(c) Table containing tuple sequences

	x	y	z
τ_1	1	$= x+1$	4
τ_2	$\in \{0,2\}$	1	$*$
τ_3	$*$	≤ 2	3
τ_4	$\geq 2-y$	0	$\neq 2$
τ_5	2	$\neq z$	1

(d) Smart table

$\begin{array}{\|c\|}\hline x \\ \hline 0 \\ 1 \\ \hline\end{array} \otimes \begin{array}{\|c\|c\|}\hline y & z \\ \hline 0 & 1 \\ 1 & 2 \\ \hline\end{array}$
$\begin{array}{\|c\|c\|}\hline x & z \\ \hline 1 & 1 \\ 2 & 2 \\ \hline\end{array} \otimes \begin{array}{\|c\|}\hline y \\ \hline 0 \\ \hline\end{array}$

(e) Sliced table

	w	x	y	z
τ_1	$\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$		$*$	$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$
τ_2	$*$	$\begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix}$		$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$

(f) Segmented table

Figure 3.3: Examples of various compressed tables.

From this point, we distinct ground tuples (tuples containing only simple values) from compressed tuples (tuples containing a compressed representation of values).

3.2.8 STRNe: Introducing Negative Tables

A negative table (also called conflict table) contains tuples, but instead of representing solutions, they represent non-solutions of a constraint. The negative table is the complement of a positive table in the universe of all the tuples possible given the domains.

`STRNe` [LLGL13], also based on the STR algorithm family handles such tables.

3.2.9 GAC4R & MDD4R: Interest of Reseting

Until now, the algorithms based themselves on the values removed since the last propagation. They updated their state incrementally at each step. However, in [PR14], they show that in some cases, when too many values are removed at once during the propagation, it could be beneficial to rebuild the state from the current domain of the variables. The concept of reset was introduced.

3.2.10 Compact-Table: The bitwise Computation Revolution

Compact-Table, the latest table constraint propagator, on which this thesis's work is mainly based, was presented at CP2016 [DHL+16]. It uses the concept of bitwise computation between bitsets (Sec. 4.2.2) to reduce the computation time drastically.

As for the previous algorithms, the propagation of Compact-Table is composed of two main phases. First, the update phase, whose goal is to update the remaining table's representation (here the bitset `currtable`). This phase can be model by the mathematical invariant inv. 3.1. Second, the filtering phase, which finds which values have to be removed from the unbound variables' domains. Again, this could be formulated as a matematical invariant inv. 3.2. The respect of these two invariants by `CT` makes it a GAC algorithm (Prop. 3.1). The pseudo-code of the algorithm can be found in Algo. 1.

Invariant 3.1 (Current table update). *Given the notations: T^0, the initial table (i.e. before any propagation occurs), T^c, the reduced table at a given current state c of the propagation, and, $\texttt{dom}^c(x)$, the domain of x at the current state c. A tuple τ belongs to the current table T^c if and* *only if it was a tuple of the initial table and all its values still belongs to*

Algorithm 1: The Compact-Table algorithm

```
Method updateTable()                                   // Invariant 3.1
    foreach variable x ∈ S^val do
        mask ← 0^64
        if |Δ_x| < |dom^c(x)| then                     // Classical update
            foreach value a ∈ Δ_x do
                mask ← mask | supports[x, a]
            mask ←~ mask
        else                                           // Reset update
            foreach value a ∈ dom^c(x) do
                mask ← mask | supports[x, a]
        currtable ← currtable & mask

Method filterDomains()
    foreach variable x ∈ S^sup do
        foreach value a ∈ dom^c(x) do
            intersection ← currtable & supports[x, a]
            if intersection = 0^64 then                // Invariant 3.2
                dom^c(x) ← dom^c(x) \ {a}
                currtable ← currtable & ~ supports[x, a]

Method enforceGAC()
    S^val ← {x ∈ scp : lastSizes[x] ≠ |dom^c(x)|}
    S^sup ← {x ∈ scp : |dom^c(x)| > 1}
    updateTable()
    count ← nb1s(currtable)  // nb1s detailed in Algo. 2
    if count = ∏_{x∈scp} |dom^c(x)| then               // Invariant 3.3
        return ⊤                           // desactivation of the cst
    if count = 0 then                                  // Invariant 3.4
        return ⊥                                // backtrack triggered
    filterDomains()
    foreach variable x ∈ S^val do
        lastSizes[x] ← |dom^c(x)|
```

the respective current domains of the associated variables from `scp`*.*

$$\Big(\tau \in T^0 \wedge \forall x \in \texttt{scp},\ \tau[x] \in \texttt{dom}^c(x)\Big) \Leftrightarrow \Big(\tau \in T^c\Big)$$

Invariant 3.2 (**Domain filtering**)**.** *Given any variable $x \in \texttt{scp}$, each value v in $\texttt{dom}^c(x)$ should appear in at least one tuple $\tau \in T^c$.*

$$\forall x \in \texttt{scp},\ \forall v \in \texttt{dom}^c(x),\ \exists \tau \in T^c,\ \tau[x] = v$$

Proposition 3.1. *A positive table constraint enforces GAC if inv. 3.1 and inv. 3.2 hold.*

Proof. By means of inv. 3.1, the set of valid tuples is maintained. Invariant 3.2 detects when a given value (x, a) can be removed if necessary. □

In addition, two other invariants (inv. 3.3 and inv. 3.4), direct consequences from the two initial one, can be considered to speed up the process. The first one describes the case when any assignment is a solution, and the second describes the case when there are no solutions anymore. Detecting these cases earlier in the computations may help reduce the total computation time.

Invariant 3.3 (**Entailement**)**.** *A positive table constraint is entailed if and only if the table contains all the possible tuple w.r.t. the domains of the variables.on.*

$$\Big(\big|\{\tau : \tau \in T^c\}\big| = \prod_{x \in \texttt{scp}} \big|\texttt{dom}(x)\big|\Big) \Leftrightarrow \top$$

Invariant 3.4 (**Emptiness**)**.** *A positive table constraint is falsified if and only if it is empty.*

$$\Big(T^c = \emptyset\Big) \Leftrightarrow \bot$$

The next subsections introduce the `CT` algorithm, start with the data structures used, then the update phase, followed by the filtering phase, and finish with the algorithm as a whole.

Algorithm 2: The `nb1s` method

Method `nb1s`**(bs:Bitset)**
    count $\leftarrow 0$
    foreach $i \in 1..bs.length$ **do**
        count $\leftarrow$ count +
         java.lang.Long.bitCount(bs.words[i])
    return count

3.2.10.1 Data Structure

The main data structure, called `currtable`, is a Reversible Sparse Bitset (Chap. 4). Its purpose is to represent the tuples from T^c, i.e. the tuples from the initial table T^0, which are still valid. Its formal definition is given at Def. 3.2.

Definition 3.2. *`currtable` (as used in CT)*
`currtable` is a reversible sparse bitset. It associates one bit to each of the tuples of a given table T^0. At a given time, `currtable` represents a given T^c, subset of T^0, valid regarding the domains' values at that time. Given any $\tau \in T^0$,

$$\texttt{currtable}\langle\tau\rangle = \begin{cases} 1 & \textit{iff } \tau \in T^c \\ 0 & \textit{iff } \tau \notin T^c \end{cases}$$

Figure 3.4 shows an example of `currtable`.

Also, some immutable bitsets called `supports` are precomputed at the setup of the constraint. They are meant to ease the computations and avoid computing several times the same thing during the propagation. The same bit position as in `currtable` is used for each of the tuples of the table. Its formal definition is given at Def. 3.3.

τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
0	1	0	0	1	0	0

(a) `currtable` (assuming $\texttt{dom}(x)_0 = \{0,1\}$, $\texttt{dom}(x)_1 = \{0,2\}$ and $\texttt{dom}(x)_2 = \{1,2\}$)

	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
supports$[x_0, 0]$	1	1	1	0	0	0	0
supports$[x_0, 1]$	0	0	0	1	1	0	0
supports$[x_0, 2]$	0	0	0	0	0	1	1
supports$[x_1, 0]$	1	1	0	0	0	0	0
supports$[x_1, 1]$	0	0	1	0	0	0	0
supports$[x_1, 2]$	0	0	0	1	1	1	1
supports$[x_2, 0]$	1	0	0	1	0	1	0
supports$[x_2, 1]$	0	1	0	0	1	0	1
supports$[x_2, 2]$	0	0	1	0	0	0	0

(b) All `supports` for each pair of variable-value

Figure 3.4: An example of `currtable` and `supports`, corresponding to the table at Fig. 3.1a.

Definition 3.3. *supports (as used in CT)*
Given a ground tuple τ, for a given variable x, $\forall v \in \texttt{dom}(x)$,

$$\texttt{supports}[x, v]\langle\tau\rangle = \begin{cases} 1 & \textit{iff } \tau[x] = v \\ 0 & \textit{iff } \tau[x] \neq v \end{cases}$$

This formula defines a given $\texttt{supports}[x, v]$ for a variable x and a variable $v \in \texttt{dom}(x)$ to be the bitset containing the tuples supporting the value v from $\texttt{dom}(x)$.
Figure 3.4 shows an example of **`supports`**.

3.2.10.2 The Update Phase

As in GAC4R, in `CT`, the update (Algo. 1 line line 1) w.r.t. a variable can be executed in two ways: the classical way and the reset way.

In the classical way (Algo. 1 line 4), the update is done regarding the removed values since the last propagation from the domain of a variable x (called the delta of the variable[1], Δ_x).

The `supports` corresponding to the values in the Δ_x are unified into a temporary variable `mask` (bitwise AND operation between the `supports`). This union corresponds to a set of tuples no more valid w.r.t. the newest removed values from x. This `mask` is then removed from `currtable`, updating the representation of the table w.r.t. the variable x (bitwise AND operation between `currtable` and the bitwise NOT operation on the `mask`).

In the reset way (Algo. 1 line 8), the update is done regarding the remaining values in the domain of the variable x. The `supports` corresponding to the values in $dom(x)$ are unified into a temporary variable `mask` (bitwise or operation between the `supports`). This union corresponds to a set of tuple potentially valid. The `mask` is then intersected with `currtable`, updating the representation of the table w.r.t. the variable x (bitwise AND operation between `currtable` and `mask`).

During the update, `currtable` has to be updated w.r.t. all the variables modified since last propagation. For each of the modified variables, the choice is made between the classical update and the reset update. As the complexity of the classical update is $\mathcal{O}(|\Delta_x|)$ and the complexity of the reset update is $\mathcal{O}(|\texttt{dom}(x)|)$, the choice is made by comparing the size of Δ_x and $\texttt{dom}(x)$.

[1]In [lCdSMSSL13], a sparse-set domain implementation for obtaining Δ_x without overhead is described

Complexity. The worst-case time complexity of the update phase is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{S}^{\mathtt{val}}} \big(\min(|\Delta_x|, |\mathtt{dom}^c(x)|)\big) \left\lceil \frac{|T^0|}{w} \right\rceil\Big)$$

where w is the number of bits into a word (i.e, for java Long type, $w = 64$). The worst-case space complexity of the update is

$$\mathcal{O}(1)$$

as it does not use any space not already preallocated.

3.2.10.3 The Filtering Phase

The filtering tries each of the values from the unbound variables' domains. The set $\mathtt{S}^{\mathtt{sup}}$ contains the unbound variables. The goal is to identify those who can lead to inconsistencies. To do so, the filtering invariant (inv. 3.2) is applied.

The idea is to check if for each value v for an unbound variable x there exists remaining tuples supporting v to x. This is done by verifying the value of the intersection between `currtable` and `supports`$[x, v]$ (bitwise AND operation between the `currtable` and `supports`$[x, v]$). An empty intersection means no remaining tuples associate v to x. Therefore v can be removed from `dom`(x).

Complexity. The worst-case time complexity of the filtering phase is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{S}^{\mathtt{sup}}} (|\mathtt{dom}^c(x)|) \left\lceil \frac{|T^0|}{w} \right\rceil\Big)$$

where w is the number of bits into a word (i.e, for java Long type, $w = 64$). The worst-case space complexity of the filtering is

$$\mathcal{O}(1)$$

as it uses only a fixed number of temporary variables and preallocated variables.

3.2.10.4 GAC and Complexity

`enforceGAC`() (Algo. 1 line 19) is the entry point of the propagator. It first updates the table (using inv. 3.1), then tests the entailment (inv. 3.3) and the emptiness (inv. 3.4) property and finally filters the values from the domains (using inv. 3.2).

Proposition 3.4. *Algorithm 1 applied to a positive table constraint C enforces GAC.*

Proof. By means of Method `updateTable`() and statement at Algo. 1 line 18, we maintain the set of conflicts on C in `currtable`. At line 16, we can detect if no more support exists for a given value (x, a), and delete it if necessary. $\square$

Complexity. The worst-case time complexity is

$$\mathcal{O}\Big(\underbrace{\sum_{x \in \mathtt{S}^{\mathtt{val}}} \big(\min(|\Delta_x|, |\mathtt{dom}^c(x)|)\big) \left\lceil \frac{|T^0|}{w} \right\rceil}_{update} + \underbrace{\left\lceil \frac{|T^0|}{w} \right\rceil}_{invariants 3.3 \& 3.4} + \underbrace{\sum_{x \in \mathtt{S}^{\mathtt{sup}}} (|\mathtt{dom}^c(x)|) \left\lceil \frac{|T^0|}{w} \right\rceil k}_{filtering}\Big)$$

Since $|\mathtt{scp}| \geq |\mathtt{S}^{\mathtt{sup}}|$ and $|\mathtt{scp}| \geq |\mathtt{S}^{\mathtt{val}}|$, this can be globally reduced to

$$\mathcal{O}\Big(|\mathtt{scp}|\ d^c \left\lceil \frac{|T^0|}{w} \right\rceil\Big)$$

where $d^c = \max_{x \in \mathtt{S}^{\mathtt{sup}} \bigcup \mathtt{S}^{\mathtt{val}}} \{|\mathtt{dom}^c(x)|\}$ is the size of the largest of the current unbound variable domain at last propagation and w is the number of bits in a word (i.e, for Java Long type, $w = 64$).
The worst-case space complexity is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{scp}} |\mathtt{dom}^0(x)| \left\lceil \frac{|T^0|}{w} \right\rceil\Big)$$

which can be globally reduced to

$$\mathcal{O}\Big(|\mathtt{scp}|\ d^0 \left\lceil \frac{|T^0|}{w} \right\rceil\Big)$$

where $d^0 = \max_{x \in \mathtt{scp}} \{|\mathtt{dom}^0(x)|\}$ is the size of the largest initial domain and w is the number of bits in a word.

3.3 Conclusion

This chapter retraced the history of the numerous developements concerning extensional constraints, which is one of the oldest form of constraints in the Constraint Programming paradigm. Successively proposed algorithms have brought the mechanisms that are present in the last state-of-the-art table propagator: namely, efficient data structure for storing supports, simple tabular reduction, reseting operations, and the use of bitsets. The latest algorithm in this evolution is `Compact-Table` (`CT`), which combines all these elements.

This chapter also retraced the other forms of extensional constraints in the literature: namely, several forms of compressed tables, negative tables, and `MDD`*s*. These forms help to counterbalance one of the weaknesses of the tables, i.e. they can grow on exponentially.

This thesis adapts `Compact-table`, the last state-of-the-art table propagator, to other forms of extensional constraints. First, to short, basic smart and smart tables, leading to the $\texttt{CT}^*$ and $\texttt{CT}^{bs}$ extensions. Second, to negative tables, leading to the $\texttt{CT}_{neg}$ and $\texttt{CT}^*_{neg}$ extensions. Finally, to `MDD`*s* and diagrams in general, leading to the `CD` and $\texttt{CD}^{bs}$ extensions.

Chapter 4

About Sets and Reversible Structures

One curious thing about growing up is that you don't only move forward in time; you move backwards as well, as pieces of your parents' and grandparents' lives come to you.

- Philip Pullman

4.1 Introduction

As the previous chapter mentioned, the last improvement in table constraints is mainly due to the use of a reversible sparse bitset. This chapter aims at explaining this data structure and other types of sets used in this thesis: the reversible sparse sets and the bitsets. The differences between these set implementations are explained here. This chapter also describes how the backtracking mechanism used in trail-based solver works.

4.2 Sets

The formal definition of the set the following Def. 4.1.

Definition 4.1. ***Set***
A set S is an unsorted collection of items, each present at most once in the set. These items belong to the universe of items $\mathcal{U}$. For example, $S_1 = \{i_2, i_5, i_6\}$ is a set containing three items, named i_2, i_5 and i_6. The empty set is often represented by $\{\}$ or $\emptyset$.

In our context, a set implementation has two orthogonal features:

– dense or sparse implementation

– array or bitset implementation

These features are explained in the next sections.

Besides, to work with the trail-based solver's backtracking mechanism, the required implementation has been made reversible, i.e. they can automatically revert to a previous state.

Each implementation has its strengths and weaknesses. Which set to use is motivated by which operations will be the most used. For example, an algorithm iterating many times on a set will prefer a sparse implementation, an algorithm using many union and intersection operations will favor a bitset implementation,...

4.2.1 Dense versus Sparse Implementation

A *dense* set (Fig. 4.1a) is traditionally represented by a collection of Boolean. Each of these corresponds to an item in the space of the set. The Boolean associated with item i is set to **`true`** (✔) if the item is in the set, **`false`** (✗) if not. Checking the presence of a given item is then easy. Union, intersection, and complement are also rather trivial operations to perform. However, iterating over the set's items requires iterating over all the possible items in the universe, whether the set is almost empty or full. A dense set representation is more often used when a large portion of the universe belongs to the set.

A *sparse* set (Fig. 4.1b) is represented by a collection of integers. An additional integer states the number of items present in the set. The collection decreases of size when some items are removed and increases when some are added. To check whether an item belongs to the set it is required to iterate over all the items. Union, intersection, and complement operations are often more complicated due to the inefficient way of checking whether an item belongs to the set. On the other hand, iterating on the set's items is relatively easy, and the complexity

universe	0	1	2	3	4	5
in the set?	✗	✔	✗	✔	✔	✗

(a) Dense set

In the set

1	3	4	$\perp$	$\perp$	$\perp$

↑

size $= 3$

(b) Sparse set

Figure 4.1: Example of dense and sparse representation of a set containing values $\{1, 3, 4\}$ from a universe $\{0, 1, 2, 3, 4, 5\}$.

depends precisely on the number of items in the set. Such a set is often implemented using a single array and an integer i. The integer is responsible for storing the set's size, and items in the set are stored in the first i^{th} slots in the array. A sparse set representation is more often used when a small portion of the universe belongs to the set.

Both views have their strengths and weaknesses. A middle ground also exists by implementing the set using two arrays and an integer i. The first array and the integer work the same as in the sparse implementation. The second array contains, for each item, where the item is stored in the first array. This allows an easy check of whether an item belongs to the set or not by comparing the index to the set's size and an easy iteration over the items in the set. The drawback here resides in the increased memory usage and increased operations when adding or removing an item.

4.2.2 Array versus Bitset Implementation

A *array-based* representation (Fig. 4.2a) uses arrays to store either the Boolean corresponding to an item or the item itself. They allow the processing of items one by one. Their drawback is their storage size. There is at least a byte (smallest unit processed by a CPU) for each item dedicated to it. In practice, it is more than a byte. An array-based implementation can be dense or sparse, as explained in the previous section.

A *bitset-based* representation (Fig. 4.2b) uses a collection of bits to model the set. Each of these bits is associated with one item potentially available in the set. The bit associated with item i is set to 1 if the item is in the set, 0 otherwise. Using longs, this implementation allows for stacking 64 items in a single word unit that a CPU can simultaneously

universe	0	1	2	3	4	5
in the set?	✗	✔	✗	✔	✔	✗

(a) Array representation

	b_0	b_1	b_2	b_3	b_4	b_5
bits	0	1	0	1	1	0
words	2			6		

(b) Bitset representation

Figure 4.2: Example of array-based and bitset-base representation of a set containing values $\{1, 3, 4\}$ from a universe $\{0, 1, 2, 3, 4, 5\}$ (assuming words composed of 3 bits).

process. If the set contains more than 64 items, an array of long is used, each storing 64 items. This allows efficient union, intersection, and complement operations through bitwise operations. However, checking a single item and iterating on each of them now requires more complex operations. A bitset-based representation can also be either dense or sparse. A dense bitset representation is achieved generally using an array of longs with each of the bits of the longs associated with an item. A sparse bitset representation is achieved using two arrays and an integer. The first one, an array of longs keeps the items. The second one, an array of integers, contains the indexes of words from the first item where there is at least one item present, i.e. non-empty words. Indexes are stored at the beginning of the array, and the integer tracks the number of non-empty indexes.

4.3 Reversibles Data Structures

A reversible data structure is a structure able to restore itself to a previous state with the help of a context data structure (used in trail-based solvers). The context is responsible for storing the changes, creating and maintaining the save state points, and triggering the restorations.

Algorithm 3 displays the pseudo-code of the Context class. It is composed of a first stack containing the restorations, a second stack containing the save points, and a timestamp. The restorations encapsulate the operations required to restore a reversible data structure to a previous state. The save points describe the states in which the program can return. The timestamp is used by the reversible to know whether a new restoration is required.

Algorithm 4 contains the abstract class used for the restorations. Each restoration models one object and one state. In its restore method, it contains all the operations required to restore the object to the state saved.

At some critical point, the trail-based solver uses the context to save and restore the state. Restoration can be done only once for each save. The reversible data structure uses the context to know when they need to create a restoration. The context stores all the restorations in order to apply them all when restoring to a previous state.

There are two means of creating a reversible data structure.

- First, by creating it from scratch. Algorithm 5 shows the class's pseudo-code of a simple reversible integer structure. Each reversible is linked to a given context.

– Second, by composition of other already existing reversible data structure. For example, a reversible array of integers can be built by using an array of reversible integers.

4.4 Used Implementations

Three types of sets are used in this thesis. They are similar to those already introduced in the `CT` algorithm.

– the reversible sparse set: Algorithm 7 gives the pseudo-code of the

Algorithm 3: Pseudo-code of the Context class

```
class Context
    private Stack<Restoration> storage
    private Stack<Integer> savepoints
    private Integer timeStamp
    Constructor Context(x : ReversibleInt, v : integer)
        storage = new Stack<Restoration>
        savepoints = new Stack<Integer>
        timeStamp = 0
    Method addRestoration(Restoration r)
        storage.push(r)
    Method save()
        savepoints.push(storage.size())
        timestamp += 1
    Method backtrack()
        point = savepoints.pop()
        while ¬ (storage.size() = point) do
            storage.pop().restore()
        timestamp += 1
    Method getTimeStamp()
        return timestamp
```

Algorithm 4: Pseudo-code of the Restoration abstract class

```
abstract class Restoration
    Method restore()
        ...        // restoration operations, to implement
```

ReversibleSparseSet class. They are used, in the propagators, to store the set of remaining unbound variables. Figure 4.3 shows an example of the evolution of the internal representation during some steps of execution.

– the reversible sparse bitset: Algorithm 8 gives the pseudo-code of the ReversibleSparseBitSet class. They are used to store

Algorithm 5: Specification of a simple reversible integer

```
class ReversibleInt
    private c : context
    private timeStamp : integer
    private value : integer
    Constructor ReversibleInt(c : context, v : integer)
        context ← c
        value ← v
        timeStamp ← 0
    Method getValue()
        return value
    Method setValue(newvalue : integer)
        contextTimeStamp ← context.getTimeStamp()
        if timeStamp ≠ contextTimeStamp then
            timeStamp ← contextTimeStamp
            context.addRestoration(new RestoreInt(this,value))
        value = newvalue
    Method restore(v : integer)
        value = v
```

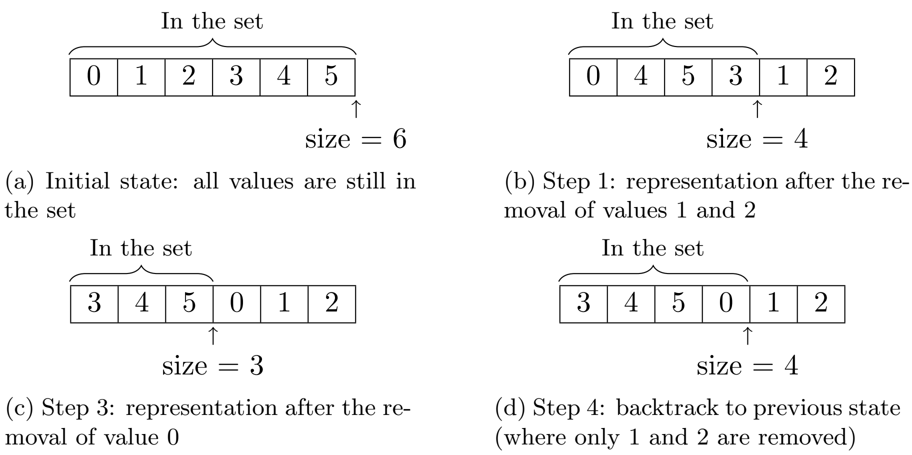
Algorithm 6: Specification of a modification data structure

```
class RestoreInt implement Restoration
    object : ReversibleInt
    value : integer
    Constructor RestoreInt(x : ReversibleInt, v : integer)
        object = x
        value = v
    Method restore()
        object.restore(value)
```

the current table's representation (contains the set of tuples still valid at some point) or the current diagram (contains the set of edges still valid at some point). The reversible sparse bitset is modified using immutable bitsets. To reduce the overhead of the reversible nature of the data structure, the union and intersection are performed using a temporary variable. Figure 4.4 shows an example of the evolution of the internal representation during some steps of execution.

– the bitset: Algorithm 9 and Algo. 10 give the pseudo-code of the `BitSet` class. They are used to store pre-computed sets. Those are then intersected or unioned with reversible sparse bitsets. As we are not modifying them in our algorithms, the class's pseudo-code only contains a way to create the object and a method used to access one given word of the bitset.

As it can be seen in the pseudo-codes, for the reversible sparse set part of each implementation, the values (either the simple values in the reversible sparse set either the index of the nonempty words in the reversible sparse bitset) of the universe are moved around in the array used to represent the set. In this implementation, the backtrack guarantees the restitution of values inside the set. However, the internal state restored may not be identical (the order of the values inside the array may vary). This can lead to a different order of the values while iterating on the structure.



(a) Initial state: all values are still in the set

(b) Step 1: representation after the removal of values 1 and 2

(c) Step 3: representation after the removal of value 0

(d) Step 4: backtrack to previous state (where only 1 and 2 are removed)

Figure 4.3: Example of the use of a reversible sparse set in the universe $\{0, 1, 2, 3, 4, 5\}$.

4.5 Conclusion

Each of the set implementations has strengths and weaknesses. The use of one instead of another is justified by which operations are needed the most. Dense sets are more used when there is a need to easily check whether one value is still in the set. Sparse sets are more used when there is a need for a traversal of all elements individually but when unions, intersections, and checks for a given value are not so much used. Bitsets are used when intersection and union are critical operations. However, verifying one individual bit to check for one value is less straightforward. Sparse bitsets allow easy intersection and union but also allow quick verification of the emptiness.

The use of reversibility allows the structure to easily revert to its previous state when using a trail-based solver. Making reversible an existing data structure in an efficient way is thus crucial to the success of propagation algorithms. In this thesis, the used data structures are the reversible sparse sets (Algo. 7), the reversible sparse bitsets (Algo. 8)

$$\overbrace{\begin{array}{|c|c|}\hline 0 & 1 \\ \hline\end{array}}^{\text{Nonempty words}}$$

$$\uparrow \text{ size} = 2$$

	b_0	b_1	b_2	b_3	b_4	b_5
bits	1	1	1	1	1	1
words	7			7		

(a) Initial state: all values are still in the set

$$\overbrace{\begin{array}{|c|c|}\hline 0 & 1 \\ \hline\end{array}}^{\text{Nonempty words}}$$

$$\uparrow \text{ size} = 2$$

	b_0	b_1	b_2	b_3	b_4	b_5
bits	1	0	0	1	1	1
words	4			7		

(b) Step 1: representation after the removal of values 1 and 2

$$\overbrace{\begin{array}{|c|}\hline 1 \\ \hline\end{array}}^{\text{Nonempty words}}\begin{array}{|c|}\hline 0 \\ \hline\end{array}$$

$$\uparrow \text{ size} = 1$$

	b_0	b_1	b_2	b_3	b_4	b_5
bits	0	0	0	1	1	1
words	0			7		

(c) Step 3: representation after the removal of value 0

$$\overbrace{\begin{array}{|c|c|}\hline 1 & 0 \\ \hline\end{array}}^{\text{Nonempty words}}$$

$$\uparrow \text{ size} = 2$$

	b_0	b_1	b_2	b_3	b_4	b_5
bits	1	0	0	1	1	1
words	4			7		

(d) Step 4: backtrack to previous state (where only 1 and 2 are removed)

Figure 4.4: Example of the use of a reversible sparse bitset in the universe $\{0, 1, 2, 3, 4, 5\}$ (assuming words composed of 3 bits).

and the bitsets (Algo. 9).

Algorithm 7: Pseudo-code of the reversible sparse set class

```
class ReversibleSparseSet
    set : Array[integer]
    size : ReversibleInt
    iterIdx : integer
    isRemoved : Boolean
    /* Construct a ReversibleSparseSet object
       representing a set with U = {0, 1, ..., n − 1} and
       initially containing all the items.  This
       implementation allows only to remove item from
       the set.  A previous state of the set may be
       retrieved by the context.                          */
    Constructor ReversibleSparseSet(c:context,n :
     integer)
        set ← new Array of size n where set[i] = i
        size ← new ReversibleInt(c,n)
        iterIdx ← size
        rem ← false
    /* Return true if the set is empty, false otherwise
                                                          */
    Method isEmpty()
        return size.getValue() == 0
    /* Remove the item currently at index i (assuming
       0 ≤ i < size) of the set.                          */
    Method startIter()
        iterIdx ← size
        rem ← false
    /* Remove the item currently at index i (assuming
       0 ≤ i < size) of the set.                          */
    Method hasnext()
        return iterIdx > 0
    /* Remove the item currently at index i (assuming
       0 ≤ i < size) of the set.                          */
    Method next()
        if iterIdx > 0 then
            iterIdx ← iterIdx -1
            rem ← false
    /* Remove the item currently at index iterIdx
       (assuming 0 ≤ iterIdx < size) of the set.          */
    Method removeCurrent()
        if ¬rem then
            size ← size-1
            temp ← set[iterIdx]
            set[iterIdx] ← set[size]
            set[size] ← temp
            rem ← true
```

Algorithm 8: Pseudo-code of the reversible sparse bitset class

```
class ReversibleSparseBitset
    words : Array[ReversibleLong]
    indexes : ReversibleSparseSet
    temp : Bitset
    /* Return true if the set is empty, false otherwise */
    Method isEmpty()
        return indexes.isEmpty()
    /* Clear the temporary bitset used for computation.
       Clear only the usefull words, i.e. those which
       are not empty yet in the set. */
    Method clearCollect()
        indexes.startIter
        while indexes.hasNext() do
            temp.emptyWord(indexes.next())
    /* Add the bitset to the temporary bitset. Only
       apply to the usefull words, i.e. those which
       are not empty yet in the set. */
    Method unionCollect(bs: Bitset)
        indexes.startIter
        while indexes.hasNext() do
            temp.unionWord(bs,indexes.next());
    /* Intersect the bitset with the temporary bitset.
       Only apply to the usefull words, i.e. those
       which are not empty yet in the set. */
    Method intersectCollect(bs: Bitset)
        indexes.startIter
        while indexes.hasNext() do
            temp.intersectWord(bs,indexes.next());
    /* Remove the bitset (equivalent to the
       intersection with the complement) with the
       temporary bitset. Only apply to the usefull
       words, i.e. those which are not empty yet in
       the set. */
    Method removeFromCollect(bs: Bitset)
        indexes.startIter
        while indexes.hasNext() do
            temp.removeWord(bs,indexes.next());
```

Algorithm 9: Pseudo-code of the bitset class (part 1)

```
class Bitset
    words : Array[long]
    nWord : integer
    /* Construct a Bitset object representing a set
       with S = {0, 1, ..., n − 1} and the values contained
       in iter as initial values                              */
    Constructor Bitset(n : integer, iter: Iterator)
        nWord ← ⌊(n+63)/64⌋
        words ← new Array of size nWord
        foreach item in iter do
            wordID ← ⌊item/64⌋
            bitID ← item − 64 ∗ wordID
            words[wordID] ← words[wordID] | 2^bitID
    /* Return the value of the word at index i
       (assuming 0 ≤ i < nWord)                               */
    Method getWord(i : integer)
        return words[i]
    /* Empty the set                                          */
    Method emptySet()
        foreach i ∈ [0; size.getValue()[ do
            emptyWord(i)
    /* Empty only the word at index i (assuming
       0 ≤ i < nWord)                                         */
    Method emptyWord(index:integer)
        words[index] ← 0L
    /* The bitset is modified to correspond to the
       union between its initial value and bs                 */
    Method union(bs:Bitset)
        foreach i ∈ [0; size.getValue()[ do
            unionWord(bs,i)
    /* The word at index i (assuming 0 ≤ i < nWord) of
       the bitset is modified to correspond to the
       union between its initial value and the word at
       index i of bs. The bitwise AND operation (&) is
       used to perform the operation.                         */
    Method unionWord(bs:Bitset,index:integer)
        words[i] ← words[i] & bs.getWord(i);
```

Algorithm 10: Pseudo-code of the bitset class (part 2)

```
class Bitset
    /* The bitset is modified to correspond to the
       intersection between its initial value and bs
       */
    Method intersect(bs:Bitset)
        foreach i ∈ [0; size.getValue()[ do
            intersectWord(bs,i)
    /* The word at index i (assuming 0 ≤ i < nWord) of
       the bitset is modified to correspond to the
       intersection between its initial value and the
       word at index i of bs.  The bitwise OR operation
       (|) is used to perform the operation.          */
    Method intersectWord(bs:Bitset,index:integer)
        words[i] ← words[i] | bs.getWord(i);
    /* The bitset is modified to correspond to the
       result of removing each item in bs from the
       initial value of the bitset                    */
    Method remove(bs:Bitset)
        foreach i ∈ [0; size.getValue()[ do
            intersectWord(bs,i)
    /* The word at index i (assuming 0 ≤ i < nWord) of
       the bitset is modified to correspond to the
       removal of the word at index i of bs from the
       initial value of the bitset.  The bitwise AND
       operation (&) and the bitwise NOT (∼) are used
       to perform the operation.                      */
    Method removeWord(bs:Bitset,index:integer)
        words[i] ← words[i] & ∼bs.getWord(i);
```

Part II

Structures

Chapter 5

Tables for Constraints

If I designed a computer with 200 chips, I tried to design it with 150. And then I would try to design it with 100. I just tried to find every trick I could in life to design things real tiny.

- Steve Wozniak

5.1 Introduction

A table (Def. 5.1) is a generic term to define a collection of tuples of values. The semantic of a table depends on the adjective attributed to it. A table can be positive or negative, ground, short, basic smart or smart,...

The table is the input of one of the variants of the extensional constraint. This chapter first defines the different kinds of tables possible. Then the link between tables and CNF/DNF is explained. Finally, the compression problem is discussed.

5.2 Definitions

Let us first define the generic table at Def. 5.1.

Definition 5.1. ***Tables, tuples and domains***
A domain $\mathcal{D}$ *is a collection, finite or infinite, continuous or not, of values of the same type (integer, double, char,...). The* domain of a table *is the ordered sequence of r domains $\mathbb{D} = (\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_r)$, with r called the* arity *of the table.*
A tuple is a sequence of values $(v_1, v_2, ..., v_k)$. A tuple belongs to $\mathbb{D}$ iff $k = r$ and $v_i \in \mathcal{D}_i$ $(\forall 1 \leq i \leq r)$. The table *is a set of tuples belonging*

to its domain.
The universe table $\mathbb{U}$ *is a table containing all possible tuples allowed by its domain, i.e. the cardinal product of each of the elements of the domain of the table* $\mathcal{D}_1 \times \mathcal{D}_2 \times \ldots \times \mathcal{D}_r$*. The size of the universe table is* $\prod_{i=1}^{r} |\mathcal{D}_i|$*. This size is finite only if each* $\mathcal{D}_i$ *composing* $\mathbb{D}$ *is finite.*
Figure 5.1 displays examples of different tables.

The domain of a table is often represented using a tuple of variables. In this case the domain of each variable is used to define the range of allowed values for each colum. For example, given the table in Fig. 5.1a and given the variable x_1, x_2, x_3, and x_4 and their associated domains, $\texttt{dom}(x_1) = \{a, b\}$, $\texttt{dom}(x_2) = \{a, b, c, d\}$, $\texttt{dom}(x_3) = \{a, b, c, d, e\}$, and $\texttt{dom}(x_4) = \{a, b, c, d, e\}$, the domain of the table can be defined by the variable tuple (x_1, x_2, x_3, x_4). Using the variable notation, the first value of τ_1 can be written $\tau_1[x_1]$.

5.2.1 Positive and Negative Tables

From a semantic point of view, the list of tuples can relate to two semantic of tables. First, when the table is defined as a *positive* table, the tuples listed belong to the table described. Second, when the table is defined as a *negative* table (also called *conflict* table), the tuples listed are the forbidden instantiations of the variables. The set of forbidden tuples is the complementary of the set of accepting tuples in the universe table $\mathbb{U}$.

Given a positive table P and a negative table N, P and N are sementically equivalent iff $P \cap N = \emptyset$ and $P \cup N = \mathbb{U} = \mathcal{D}_1 \times \mathcal{D}_2 \times \ldots \times \mathcal{D}_N$. Said otherwise, using set theory and given the universe table $\mathbb{U}$, N is the complementary set of P.

τ_1	a	b	c	c
τ_2	a	c	d	c
τ_3	b	a	e	a
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(a) 1^{st} example

τ_1	red	1	5, 2
τ_2	green	1	3, 5
τ_3	red	2	6, 2
$\vdots$	$\vdots$	$\vdots$	$\vdots$

(b) 2^{nd} example

τ_1	1	2	2
τ_2	2	3	3
τ_3	3	5	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$

(c) 3^{rd} example

Figure 5.1: Several examples of generic Table. Their domains are, respectivelly, $\mathbb{D}_{Fig.\ 5.1a} = (\{a, b\}, \{a, b, c, d\}, \{a, b, c, d, e\}, \{a, b, c, d, e\})$, $\mathbb{D}_{Fig.\ 5.1b} = (\{green, red, yellow, ...\}, \{0, 1, 2, 3, ...\}, [0.0, 10.0])$ and $\mathbb{D}_{Fig.\ 5.1c} = (\{0, 1, 2, 3\}, \{0, 1, 2, 3\}, \{0, 1, 2, 3\})$.

5.2.2 Compressed Tables

A table can easily become huge. Indeed, for a domain of table composed of N domains of size M, the size of $\mathbb{U}$, the corresponding universe table, is M^N. This corresponds to an upper bound on the maximum number of tuples in any table with this given domain.

As seen in Chap. 3, several forms of compressed tables have been proposed (smart tables, sliced tables,...). We decided to focus on the smart table family of compression, where the table still consists of a set of tuples.

Definition 5.2 defines the formal concepts of ground tables and ground tuples. The addition of a universal value allowed to define a first level of compression: the short table (Def. 5.3). The addition of other unary relations leads to the basic smart table (Def. 5.4). Finally, using binary relation defines the smart table (Def. 5.5). All the elements are described Def. 5.6.

Definition 5.2. ***Ground Table and Ground Tuple***
A ground table is a table composed of ground tuples. A ground tuple is a tuple only composed of single value elements $\langle = v\rangle$. Often, the shortcut of writing only v is used. Figure 5.2a shows an example of ground table.

Definition 5.3. ***Short Table and Short Tuple***
A short table is a table composed of short tuples. A short tuple is a tuple composed of single values $\langle = v\rangle$ and/or universal values $\langle * \rangle$ elements. By definition, any ground table is thus a short table too. Figure 5.2b shows an example of short table.

τ_1	1	2	4
τ_2	0	1	0
τ_3	1	2	3
τ_4	2	0	2
τ_5	2	3	1

(a) Ground table

τ_1	1	$*$	4
τ_2	0	1	$*$
τ_3	$*$	2	3
τ_4	2	0	2
τ_5	2	$*$	1

(b) Short table

τ_1	1	$*$	4
τ_2	$\in \{0,2\}$	1	$*$
τ_3	$*$	≤ 2	3
τ_4	≥ 2	0	$\neq 2$
τ_5	2	2	1

(c) Basic smart table

τ_1	1	$= x+1$	4
τ_2	$\in \{0,2\}$	1	$*$
τ_3	$*$	≤ 2	3
τ_4	$\geq 2-y$	0	$\neq 2$
τ_5	2	$\neq z$	1

(d) Smart table

Figure 5.2: Example of all types of compressed tables

Definition 5.4. ***Basic Smart Table and Basic Smart Tuple***
A basic smart table is a table composed of basic smart tuples. A basic smart tuple is a tuple composed of basic smart elements which are tuple restrictions which corresponds to unary constraints: single values $\langle = v \rangle$*, universal values* $\langle * \rangle$*, exclusion values* $\langle \neq v \rangle$*, upper bound values* $\langle \leq v \rangle$ $\langle < v \rangle$*, lower bound values* $\langle \geq v \rangle$ $\langle > v \rangle$ *and/or sets values* $\langle \in S \rangle$ $\langle \notin S \rangle$*. By definition, any short table is thus a basic smart table too. Figure 5.2c shows an example of basic smart table.*

Definition 5.5. ***Smart Table and Smart Tuple***
A smart table is a table composed of smart tuples. A smart tuple is a tuple composed of smart elements and/or basic smart elements. Smart elements are tuple restrictions xhich corresponds to binary constraints: $\langle = v \rangle$, $\langle * \rangle$, $\langle \neq v \rangle$, $\langle \leq v \rangle$, $\langle < v \rangle$, $\langle \geq v \rangle$, $\langle > v \rangle$, $\langle \in S \rangle$*, or* $\langle \notin S \rangle$*. By definition, any basic smart table is thus a smart table too. Figure 5.2d shows an example of smart table.*

Definition 5.6. ***Compression Elements (Basic Smart and Smart)***
Basic smart elements are unary expressions of the following forms:

– $\langle = v \rangle$*: the single value, representing only the value v (for simplicity, this one is often writen simply v in tables)*

– $\langle * \rangle$*: the universal value, representing any value*

– $\langle \neq v \rangle$*: the exclusion, representing any value except value v*

– $\langle \leq v \rangle$ *(resp.* $\langle < v \rangle$*): representing any value lower or equal (resp. strictly lower) to value v*

– $\langle \geq v \rangle$ *(resp.* $\langle > v \rangle$*): representing any value heigher or equal (resp. strictly higher) to value v*

– $\langle \in S \rangle$ *(resp.* $\langle \notin S \rangle$*): representing any value contained (resp. not contained) in the set of value S*

Smart elements are binary expressions of the following forms:

– $\langle = x + v \rangle$

– $\langle \neq x + v \rangle$

– $\langle \leq x + v \rangle$ *(resp.* $\langle < x + v \rangle$*)*

– $\langle \geq x + v \rangle$ *(resp.* $\langle > x + v \rangle$*)*

The $\langle \leq v \rangle$*,* $\langle \geq v \rangle$*,* $\langle \leq x + v \rangle$*, and* $\langle \geq x + v \rangle$ *make only sense if an ordering is defined on the values contained in the domains.*

5.3 CNF and DNF are Tables

Interestingly, there is a strong link between the DNF/CNF duality and the positive/negative table duality. Indeed, any DNF (Def. 5.7) can be written as a positive short table while any CNF (Def. 5.8) can be written as a negative short table.

Definition 5.7. ***DNF***
A DNF (i.e Disjunctive Normal From*) is a canonical form of Boolean formula. It consists of a disjunction (OR) of several conjunctions (AND) of literals.*

$$(X_{11} \land X_{12} \land \ldots \land X_{1k_1}) \lor (X_{21} \land \ldots \land X_{2k_2}) \lor \ldots \lor (X_{l1} \land \ldots \land X_{lk_l})$$

Definition 5.8. ***CNF***
A CNF (i.e. Conjunctive Normal From*) is a canonical form of Boolean formula. It consists of a conjunction (AND) of several disjunctions (OR) of literals.*

$$(X_{11} \lor X_{12} \lor \ldots \lor X_{1k_1}) \land (X_{21} \lor \ldots \lor X_{2k_2}) \land \ldots \land (X_{l1} \lor \ldots \lor X_{lk_l})$$

DNF as a positive short table. One can view a positive table as a collection of conjunctive clauses (i.e. a conjunction of literals), each corresponding to one of the tuples. At least one of these clauses (i.e. this corresponds to the OR part) should be satisfied to satisfy the table. To satisfy a clause, all its literal should be satisfied (i.e. this corresponds to the AND part). The DNF formula Eq. (5.1) can be transformed into the positive short table in Fig. 5.3.

$$(x_1 \land x_2 \land \neg x_4) \lor (x_2 \land \neg x_3 \land \neg x_5) \lor (\neg x_1 \land \neg x_2 \land x_5) \tag{5.1}$$

The transformation is the following. Each clause corresponds to one tuple of the table, and each literal used corresponds to a column. For each literal present in the clause, the value **`true`** is set in the corresponding column if the literal is not negated, **`false`** otherwise. For each literal not present in the clause, the universal value is used. Each ground tuple allowed by this table corresponds to a possible solution of the DFA.

CNF as a negative short table. For the CNF, the transformation is a bit different. The negative table is viewed as a collection of disjunctive clauses (i.e. a disjunction of literals), each corresponding to one of the tuples. All the clauses (i.e. this corresponds to the AND part) should be satisfied to satisfy the table. To satisfy a clause, at least one of the literal should be satisfied (i.e. this corresponds to the OR part). The CNF formula Eq. (5.2) can be transformed into the negative short table in Fig. 5.4.

$$(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_2 \vee x_5) \tag{5.2}$$

The transformation is the following. Each clause corresponds to one tuple of the table, and each literal used corresponds to a column. For each literal present in the clause, the value **`false`** is set in the corresponding column if the literal is not negated, **`true`** otherwise. For each literal not present in the clause, the universal value is used. Each tuple represents the only combination of the values of the literal, not satisfying the corresponding clause.

These similarities between DNFs/CNFs and tables help us to provide some NP-completeness proof for some of the problems encountered.

Complexity It is trivial to see that the complexity of generating the positive (resp. negative) short table corresponding to a DNF (resp. CNF) is $\mathcal{O}(td)$ where t is the number of clauses (which also correspond to the number of tuples in the table) and d is the number of different literals present in the clauses.

	x_1	x_2	x_3	x_4	x_5
τ_1	$true$	$true$	$*$	$false$	$*$
τ_2	$*$	$true$	$false$	$*$	$false$
τ_3	$false$	$false$	$*$	$*$	$true$

Figure 5.3: Result of the transformation of the DNF formula (Eq. (5.1)) to a positive short table

	x_1	x_2	x_3	x_4	x_5
τ_1	$false$	$false$	$*$	$true$	$*$
τ_2	$*$	$false$	$true$	$*$	$true$
τ_3	$true$	$true$	$*$	$*$	$false$

Figure 5.4: Result of the transformation of the CNF formula (Eq. (5.2)) to a negative short table

5.4 The Compression Problem

The compression problem consists of finding a more compact (ideally a smaller) table (short table, basic smart table,...), which corresponds to a given ground table (or an already partially compressed table). We can first show that the minimization of a Boolean short table is NP-complete (Prop. 5.9). The minimization of a basic smart table is NP-hard at least as difficult.

Proposition 5.9. *Finding the smallest Boolean short table is NP-complete.*

Proof. Equivalence to the minimization of a DNF formula can be shown.

Polynomial reduction of the minimization of a DNF formula to the compression problem. The problem of minimizing the size of a DNF formula [KS08] can be reduced to the compression problem. As previously shown, a DNF can be written polynomial time as a Boolean short table. The minimal DNF formula corresponds to the minimal Boolean short table.

Polynomial reduction of the compression problem to the minimization of a DNF formula. Any Boolean short table is also equivalent to a DNF formula and can be transformed into one in a polynomial time. The minimal Boolean short table corresponds to the minimal DNF formula. □

The first thing noticed is that some tables cannot be compressed at all. Then some algorithms, which aim at solving the problem, are defined.

5.4.1 Incompressibility of some Tables

Finding an equivalent smallest basic smart table (i.e. a compressed table with fewer tuples) is not always possible. Figure 5.5 shows multiple possible decomposition of a given tuple into less compressed tables. As shown by the figure, intuitively, a tuple with k basic smart elements is composed of tuples with $k-1$ basic smart elements that have $arity - 1$ values in common. These tuples with $k-1$ basic smart elements are formed by tuples with $k-2$ elements,... up to ground tuples.

This means, to be able to compress, we need at least several tuples with values in common. To formalize our propositions, we first need two definitions. The first one, Def. 5.10, is an adaptation of the Hamming distance (metric to measure the difference between two binary numbers) to tuples. The second one, Def. 5.11, is about trivial element of compression. This leads to Prop. 5.12 defining a criterium of incompressibility.

Definition 5.10. *Hamming Distance Between two Tuples*
The Hamming distance between two tuples is the number of positions where values differ. Ex: the Hamming distance between $(1, 1, 0)$ *and* $(0, 1, 1)$ *is* 2*, between* $(0, 0, 0)$ *and* $(0, 1, 0)$ *is* 1

Definition 5.11. *Trivial Compression*
A trivial compression is when a compression element only represents one value, i.e. $*$ *used with a domain having a size of* 1*,* $\neq v$ *used with a domain having only two values and* v *being one of them,* $\leq v$ *(resp.* $\geq v$*) used with* v *the smallest (resp. biggest) value of the domain,* $\in S$ *used with the size of* S *being equal to one,...*

Proposition 5.12. *Suppose the Hamming distance between each pair of tuples contained in a ground table (without duplicated tuples) is each time higher or equal to* 2*. In that case, there is no compression possible using the basic smart compressions (excepts trivial ones).*

Proof. This can be proven recursively. Given a tuple τ with only one non-trivial basic smart compression element K (representing at least two values v_1 and v_2), we can easily see that this tuple represents at least two tuples where K was respectively replaced by v_1 and v_2, other values being the same. The Hamming distance between these two tuples is

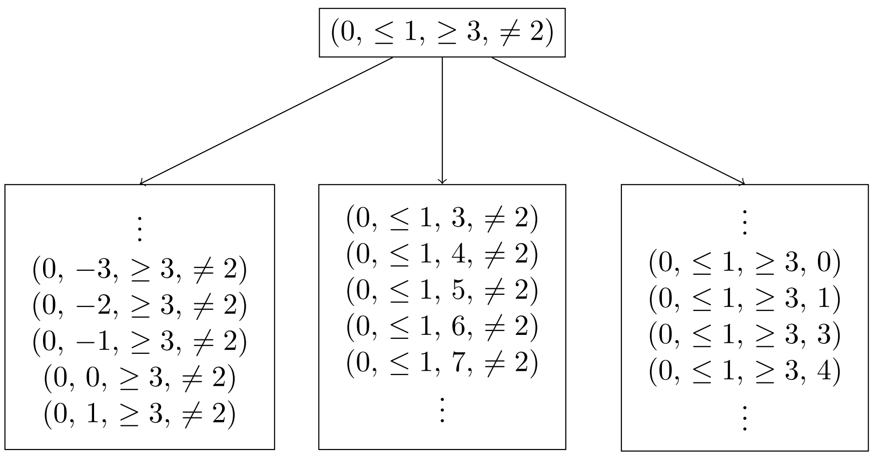


Figure 5.5: Example of a 4-tuple with 3 basic smart elements being developped into sets of 4-tuples with 2 basic smart elements sharing 3 common values (3 developpements possible).

1. Recursively, we can show that a tuple with 2 nontrivial compression represents at least two tuples with one compression element,... a tuple with n nontrivial compression represents at least two tuples with $n-1$ compression elements. $\square$

A second proposition can be directly derived from it:

Proposition 5.13. *Given a tuple τ from a given table, if for each other tuples τ_i of the table, the Hamming distance between τ and τ_i is equal or greater than two, τ cannot be used with another tuple to generate a compression.*

These propositions may help identify the number of incompressible tuples, thus reducing the compression algorithm's input to tuples potentially compressible with others.

5.4.2 Compression Algorithms

As the problem is difficult, using an exact algorithm is untractable. Instead, greedy algorithms are preferred. However, the solution obtained may not be optimal.

The next subsection describes three original algorithms. The first one, based on data-mining techniques, generates short tables. The second one is greedy and generate basic smart tables. The last one is purely theoretical and has not been implemented. It solves the exact problem.

These algorithms only work for the compression of tables in basic smart table. To compress tables into smart tables see [LCKLD].

5.4.2.1 Mining Short Tables

Compressing a table into a short table can be related to the frequent itemset mining problem [Bor12] (Def. 5.14).

Definition 5.14. ***Frequent Itemset Mining Problem***
Given a predefined set of items (universe), a transaction is a subset of this universe. Given a database of transactions (i.e. a set of transactions) and a given threshold t, the frequent itemset problem consists of finding all the patterns (also a subset of the universe) included in at least t transactions of the database.

A table can be viewed as a database: each tuple corresponding to a transaction and each value associated with a position corresponding to an item. A frequent itemset mining algorithm (such as `coversize`

[SAG17]) searches for the frequent itemsets, i.e. the short tuple candidates. A candidate represents a short tuple if the set of all the transactions containing it corresponds to the set of all valid tuples with the given assigned values (i.e. the values corresponding to the candidate itemset). This is valid if the number of occurrences corresponds to the product of the domain size of the variables uninvolved in the itemset.

Algorithm 11 shows the pseudo-code to create the corresponding short table. It relies on some auxillary fonctions: `mapToTransactions` and `mapToShortTuple` which take care of the mapping of pair (x, v) into corresponding items, and `frequentItemsetMining` which can be any frequent itemset mining algorithms ([SAG17] for example).

Figure 5.6 illustrates the process. First, the table (Fig. 5.6a) is mapped, using a given mapping function (Fig. 5.6b), into the database (Fig. 5.6c). From the database, the frequent itemsets (Fig. 5.6d) are extracted. The itemset are then evaluated. Itemset i_0 have an occurrence of 4, which corresponds to the threshold. The corresponding short tuple $(0, *, *)$ is thus added to the short table. Itemsets i_2 and i_5 does not meet the threshold. Itemsets i_0, i_2 and i_0, i_3 meet the threshold but are already subsumed by the tuple $(0, *, *)$. Itemset i_2, i_5 meet the threshold and is not entirely covered already. The short tuple $(*, 0, 1)$ is thus added. At the end all ground tuples are covered by the short tuples which lead to the result table (Fig. 5.6e).

Algorithm 11: Pseudo-code of the mining of short table

Method **`compressIntoShortTable`(T:table,scp:scope)**
    database ← `mapToTransactions`(T)
    freqItemset ← `frequentItemsetMining`(database,threshold)
    shortTable ← ∅
    foreach *(itemset,nOccurence) from freqItemset* **do**
        candidate ← `mapToShortTuple`(itemset)
        if $nOccurence = \prod_{x \in scp; x \notin candidate} |dom(x)|$ **then**
            if *candidate not already represented by another tuple* **then**
                shortTable ← shortTable ∪ {*candidate*}
    foreach τ *from T* **do**
        if $\tau \notin$ *shortTable* **then** `// ∉ is seen here as not covered by any short tuple`
            shortTable ← shortTable ∪ {τ}

5.4.2.2 Greedy Compression of Basic smart tables

We introduce a heuristic compression algorithm to generate a basic smart table from a given (ordinary) table. It focuses on column constraints of the form $\leq v$ and $\geq v$. Other forms can be obtained by post-processing: i) expressions $\leq dom(x)$.`max` or $\geq dom(x)$.`min` can be replaced by $*$, and ii) two tuples that are identical except on a column where we have respectively $\leq v-1$ and $\geq v+1$ can be merged by simply using $\neq v$. Expressions $\in S$ and $\notin S$ were not considered in this heuristic to avoid costly set operations.

The compression algorithm proceeds in r steps, r being the arity of the table. The algorithm handles two tables at each step: the c-table (compressed table) and the r-table (residual table). The union of these two tables is always equivalent to the initial table. At step i, each tuple

	x	y	z
τ_1	0	0	0
τ_2	0	0	1
τ_3	0	1	0
τ_4	0	1	1
τ_5	1	0	1

(a) Table

(Var,Val)	item
$(x,0)$	i_0
$(x,1)$	i_1
$(y,0)$	i_2
$(y,1)$	i_3
$(z,0)$	i_4
$(z,1)$	i_5

(b) Mapping

Transaction	Items
t_1	i_0, i_2, i_4
t_2	i_0, i_2, i_5
t_3	i_0, i_3, i_4
t_4	i_0, i_3, i_5
t_5	i_1, i_2, i_5

(c) Database

Itemset	nOccurence
i_0	4
i_2	3
i_5	3
i_0, i_2	2
i_0, i_3	2
i_2, i_5	2
i_3	2
i_4	2

(d) Frequent itemsets

	x	y	z
τ_a	0	$*$	$*$
τ_b	$*$	0	1

(e) Result table

Figure 5.6: An example of the mining of short tables.

of the c-table has exactly i column constraints of the form $\leq v$ or $\geq v$. When $i = 0$, the c-table is the initial table, and the r-table is empty. After step r, the resulting table of the algorithm is the union of the c-table and the r-table. The computation at a given step is the following. Several abstract tuples are generated from the tuples in the c-table, used to introduce new tuples with one more column constraint of the form $\leq v$ or $\geq v$. The new tuples that cover at least two tuples in the c-table are gathered in a new c-table used in the next step. The uncovered tuples in the c-table are added to the r-table.

More formally, at a given step, we define an *abstract tuple* as a tuple taken from the current c-table with one of its literal value $x = a$ replaced by the symbol '?'. At step i, there are thus $(r - i) \cdot t_c$ possible abstract tuples, with t_c the size of c-table. An abstract tuple can be matched against so-called strictly compatible (resp. compatible) tuples. A basic smart tuple τ is strictly compatible (resp. compatible) with an abstract tuple ρ iff for each $1 \leq j \leq r$, the form of $\tau[j]$ is strictly compatible (resp. compatible) with the form of $\rho[j]$. Compatibility of forms is intuitive: a value v is compatible with the same value v and also with '?', the form $\leq v$ (resp. $\geq v$) is compatible with $\leq w$ (resp. $\geq w$) provided that $w \geq v$ (resp. $w \leq v$). Strict compatibility requires compatibility and $w = v$.

We denote by S_c^ρ (resp., S_{sc}^ρ) the sets of tuples from the current c-table that are compatible (resp., strictly compatible) with ρ, an abstract tuple. Note that the computation of these two sets can be done in $O(r.t_c)$ and that we have $S_{sc}^\rho \subset S_c^\rho$. Given $S_c^\rho = \{\tau_1, \ldots, \tau_k\}$, we denote by V^ρ set of values $\{\tau_1[j], \ldots, \tau_k[j]\}$ where j is the column index of ? in ρ. If, given the domain of x_j, a subset of V^ρ can be represented by $x_j \leq v$ (or $x_j \geq v$), then a new basic smart tuple ρ' is generated, where ρ' is the tuple ρ with ? replaced by $\leq v$ (or $\geq v$). The corresponding tuples in S_{sc} can be removed as the new smart tuple covers them. However, the tuples only present in S_c cannot be removed. In practice, a new basic smart tuple is only introduced if it ensures a reduction of the table (i.e. at least two tuples can be removed). As t_c is $O(t)$, the total complexity of the compression algorithm is $O(r^3t^2)$.

Example. Let us consider the abstract tuple $\rho = (1, ?, \leq 1)$. In the following set of basic smart tuples $\{\tau_1 = (1, 0, \leq 1), \tau_2 = (1, 1, \leq 2), \tau_3 = (1, 2, \leq 1)\}$, the tuples τ_1 and τ_3 are strictly compatible with ρ, the tuple τ_2 is only compatible with ρ. The new smart tuple $(1, \leq 2, \leq 1)$ is then generated, allowing us to remove both τ_1 and τ_3. The tuple τ_2 is necessary to generate this new tuple, but cannot be removed from the table.

Results We have studied the compression of the tables that are present in the instances of the benchmark. The benchmark used is derived from the instances available on the XCSP3 website [BLP16] restricted to tables constraints only. The set of all the tables from these instances forms the benchmark. The compression ratio is defined as $\frac{t'}{t}$, where t and t' respectively denote the numbers of tuples in the initial and compressed tables. Using the algorithm described above, we obtain the results displayed in Fig. 5.7. As expected, dense tables (i.e. tables with a high number of tuples compared to the Cartesian product of domains) lead to good compression. This can be observed in particular with the series PigeonsPlus that contains dense instances (making them highly compressible) and the series Renault containing instances with a wide range of tables (many of them being well compressed). On the other hand, the series Kakuro or Nonogram contains very sparse tables that cannot be compressed.

5.4.2.3 Exact Method for Basic Smart Table

Because we only deal with finite domains, an exact method can be derived from the problem of finding optimal submatrices (Def. 5.15) and more precisely from the maximum weighted submatrix coverage problem (Def. 5.16). However, this method is NP-Complete and unusable in practice. It is only presented for the theoretical beauty of the formulation.

Definition 5.15. ***Maximal-Sum Submatrix Problem***
Given a matrix $\mathcal{M} \in \mathbb{R}^{m \times n}$. Let $R = \{1, \ldots, m\}$ and $C = \{1, \ldots, n\}$ be index sets for rows and for columns, respectively. The maximal-sum

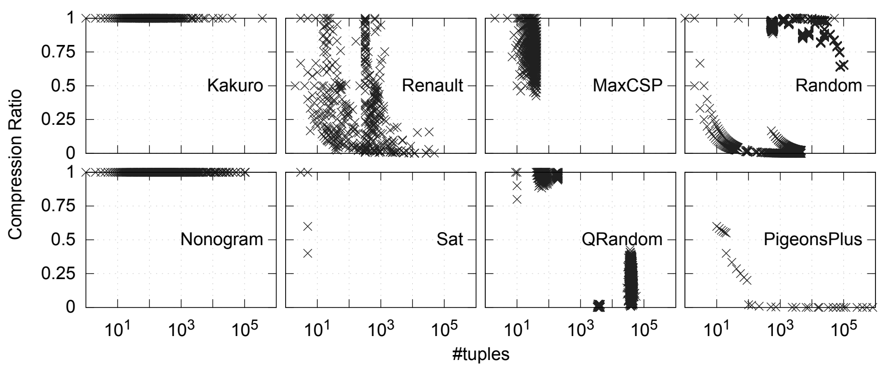


Figure 5.7: Compression ratio of all the table, classed by family of instances.

submatrix is the submatrix (I^*, J^*), with $I^* \subseteq R$ and $J^* \subseteq C$, such that:

$$(I^*, J^*) = \arg\max_{I,J} \sum_{i \in I, j \in J} \mathcal{M}_{i,j}$$

Definition 5.16. ***The Maximum Weighted Submatrix Coverage Problem***
Given a matrix $\mathcal{M} \in \mathbb{R}^{m \times n}$ and a parameter K, the maximum weighted submatrix coverage problem is to select a set S^* of submatrices (R_k, C_k) with $k = 1, \ldots, K$ such that the sum of the elements covered is maximum.

The formulation relies on the possibility to formulate any table of arity r as a Boolean r-dimension matrix. Each dimension of the matrix is associated with one of the sub-domains of the table. Each column is thus associated with one of the associated sub-domain's finite values in each dimension of the matrix. Following these mappings, each cell of the r-dimension matrix corresponds to one of the tuples in the corresponding universe table $\mathbb{U}$. Creating a weighted matrix from this binary matrix is easy. To reach the table's optimal and exact compression, one must ensure no tuple not in the table arises as a ground tuple of a compressed element. To ensure that, a cost of $-\infty$ is set for each tuple which is not present.

Example Here is an example of the mapping for a table of arity 2. The mapping leads thus to a simple 2-dimensions matrix. Assuming two variables, x and y, with domains $\texttt{dom}(x) = \{0, 1, 2, 3, 4\}$ and $\texttt{dom}(y) = \{0, 1, 2, 3, 4\}$, and a table (Fig. 5.8a) linking x and y. It is possible to consider each tuple as a point in a 2D-Space. As the domains are discrete, this 2D-Space can be represented as a matrix (Fig. 5.8b). The next step is the creation of the weight matrix (Fig. 5.8c). To each cell with a tuple, the cost of 1 is associated. To the other cells, without a tuple, a cost of $-\infty$ is associated.

The next part of solving the problem as a maximum submatrixes problem is to define what kind of matrixes we are looking for. The compression problem can be reduced to finding the smallest K with the value of the objective equal to the number of tuples.

Finally, when we have a solution, we can retrieve the basic smart tuples from the solution as each submatrix with a positive weight corresponds to a compressed tuple. The resulting basic smart element depends on the columns selected of a given dimension.

- $\langle = i \rangle$ s used if only one column of a given dimension is part of the submatrix

- $\langle * \rangle$ is used if all the columns of a given dimension are part of the submatrix

- $\langle \neq v \rangle$ is used if all but one of the columns is selected

- $\langle \leq v \rangle$ (resp. $\langle \geq v \rangle$) is used when the only consecutive columns from first to v (resp. v to last)

- $\langle \in S \rangle$ is used in any other cases, regrouping the selected columns

Finding the minimum number of submatrix with a positive total cost leads to the optimal basic smart table.

Example The solution of our example table (Fig. 5.8a) is then a mapping from the submatrix to compressed tuples. For the example, an optimal solution is with 5 submatrix:

- Submatrix $S_x = \{0\}$ $S_y = \{0, 1, 2, 3, 4\}$ (Fig. 5.9a) corresponding to $(\in \{0\}, \in \{0, 1, 2, 3, 4\})$, refined in $(0, *)$

- Submatrix $S_x = \{0, 2\}$ $S_y = \{0\}$ (Fig. 5.9b) corresponding to $(\in \{0, 2\}, \in \{0\})$, refined in $(\in \{0, 2\}, 0)$

	x	y
τ_0	0	0
τ_1	0	1
τ_2	0	2
τ_3	0	3
τ_4	0	4
τ_5	1	3
τ_6	2	0
τ_7	3	2
τ_8	3	3
τ_9	4	1
τ_{10}	4	4

(a) Example table

	0	1	2	3	4
0	τ_0	τ_1	τ_2	τ_3	τ_4
1				τ_5	
2	τ_6				
3			τ_7	τ_8	
4		τ_9			τ_{10}

(b) Space representation of the table

	0	1	2	3	4
0	1	1	1	1	1
1	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$
2	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	1	1	1
4	$-\infty$	1	$-\infty$	$-\infty$	1

(c) Corresponding weight matrix

Figure 5.8: Illustration on how to map a table into a matrix.

– Submatrix $S_x = \{0,4\}$ $S_y = \{1,4\}$ (Fig. 5.9c) corresponding to $(\in \{0,4\}, \in \{1,4\})$, which can not be refined using more precise unary restrictions

– Submatrix $S_x = \{0,3\}$ $S_y = \{2,3,4\}$ (Fig. 5.9d) corresponding to $(\in \{0,3\}, \in \{2,3,4\})$, refined in $(\in \{0,3\}, \geq 2)$

– Submatrix $S_x = \{0,1\}$ $S_y = \{3\}$ (Fig. 5.9e) corresponding to $(\in \{0,1\}, \in \{3\})$, refined in $(\leq 1, 3)$

The generalization to arities higher than two is made by working with the submatrix problem with tensors (matrix of higher dimensions).

However, even if this method leads to an optimal result, it is intractable in practice.

	0	1	2	3	4
0	1	1	1	1	1
1	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$
2	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	1	1	1
4	$-\infty$	1	$-\infty$	$-\infty$	1

(a) Tuple $(0, *)$

	0	1	2	3	4
0	1	1	1	1	1
1	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$
2	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	1	1	1
4	$-\infty$	1	$-\infty$	$-\infty$	1

(b) Tuple $(\in \{0,2\}, 0)$

	0	1	2	3	4
0	1	1	1	1	1
1	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$
2	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	1	1	1
4	$-\infty$	1	$-\infty$	$-\infty$	1

(c) Tuple $(\in \{0,4\}, \in \{1,4\})$

	0	1	2	3	4
0	1	1	1	1	1
1	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$
2	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	1	1	1
4	$-\infty$	1	$-\infty$	$-\infty$	1

(d) Tuple $(\in \{0,3\}, \geq 2)$

	0	1	2	3	4
0	1	1	1	1	1
1	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$
2	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	1	1	1
4	$-\infty$	1	$-\infty$	$-\infty$	1

(e) Tuple $(\leq 1, 3)$

Figure 5.9: Submatrices composing the solution of the example problem.

5.5 Conclusion

This chapter introduced the various kinds of tables used in this thesis and the differences between them. The primary dichotomy in table semantics lies between positive and negative tables. Tuples contained in positive tables represent allowed instantiations of variables, while tuples in negative tables represent forbidden instantiations.

Interestingly, the size of tables can be reduced by compressing them using unary or binary constraints used as values inside the tuples. These tables are called smart tables. Unfortunately, compressing ground tables into basic smart tables is a complex problem. This is why greedy approaches are the only ones that can be used in practice for now. However, compression is not always possible, as discussed in this chapter. The greedy compression algorithm was published as part of the [VLDS17] paper.

Chapter 6

Diagrams for Constraints

It's always good to take an orthogonal view of something. It develops ideas.

- Ken Thompson

6.1 Introduction

This chapter presents all variations of (decision) diagrams (Def. 6.1) used in this thesis.

Definition 6.1. ***Diagrams, Nodes, Arcs, Paths, `ROOT` and `END`***
A diagram *is a layered oriented acyclic graph. It is described by a pair (Ω,Θ) where Ω is the set of nodes forming the diagram and Θ is the set of arcs.*

As in any acyclic graph, each arc $\varepsilon \in \Theta$ is described by specifying its source node, called the tail ($t(\varepsilon) \in \Omega$), and its target node, called the head ($h(\varepsilon) \in \Omega$). The tuple notation $\varepsilon = (t(\varepsilon), h(\varepsilon))$ is also used.

In diagrams, arcs and nodes are organized in layers. There are N layers of arcs (numbered from 0 to $N-1$) and $N+1$ layers of nodes (numbered from 0 to N). N is called the arity *of the diagram. Each node $n \in \Omega$ is assigned to exactly one node layer $L_n(n)$. Each arc $\varepsilon \in \Theta$ is assigned to exactly one arc layer $L_a(\varepsilon)$. Each arc belonging to the arc layer l initiates in node layer l and reaches node layer $l+1$. The node layers 0 and N contain both one unique node. The first is called the `ROOT`, and the second is called the `END`.*

Figure 6.1 gives an example of a diagram.

There are several subclasses of diagrams. Each of them depending on how the arcs are labeled (Def. 6.2).

Definition 6.2. ***Labeled Diagrams, Domains, Binary and Multi-valued***
As for tables (Def. 5.1), we can define the domain of a diagram. The domain of a diagram *is the ordered sequence of N domains $\mathbb{D} = (\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{N-1})$, when N is the* arity *of the diagram. Each of the domains of the sequence is associated with one of the arc layers of the diagram. A labeled diagram is a diagram where each arc is associated with a label. The label of an arc $l(\varepsilon)$ belongs to the domain of the associated arc layer $l(\varepsilon) \in \mathcal{D}_{L_a(\varepsilon)}$. The same label cannot be used for two arcs sharing the same tail and the same head. A triplet notation is often used to describe labeled arcs: $\varepsilon = (t(\varepsilon), l(\varepsilon), h(\varepsilon))$.*

When the domains contain only two values, the diagram is said to be binary*. Otherwise, it is called multi-valued.*

Each of these subclasses can be associated with an equivalent table. The tuples associated to each path (Def. 6.3) inside it can represent a table.

Definition 6.3. ***Path in a Diagram***
A path *in a diagram is an alternate sequence of nodes and arcs. It starts at the `ROOT` and ends at the `END`. Each arc ε in the sequence is preceded by its source $t(\varepsilon)$ and followed by its destination $h(\varepsilon)$. Figure 6.1 shows, using bold arrows, an example of a path. This diagram contains a total of eleven different paths.*

In labeled diagrams, we can extract the sequence of the arcs' labels along the path. This sequence of labels is called the tuple associated with the path.

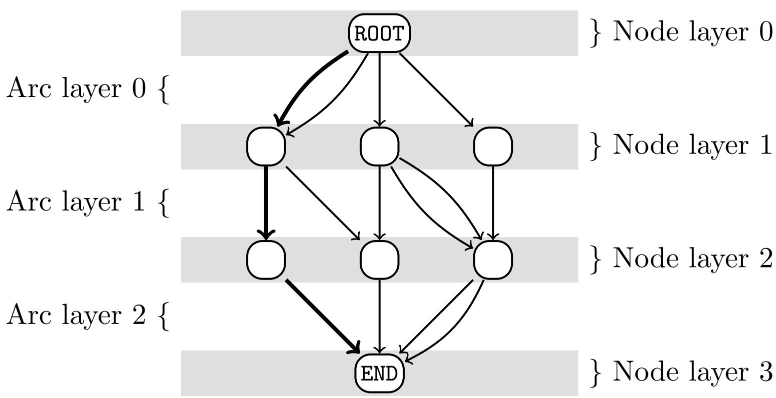


Figure 6.1: An example of diagram of arity 3 with its arc and node layers highlighted. The bold arcs represent an example of path within the diagram.

The following sections introduce the `MVD`, `MDD`, and `sMDD` subclasses and the concept of determinism of nodes. Then, we introduce the $bs-$`MVD`, $bs-$`MDD`, and $bs-$`sMDD` extentions, which result from the handling of basic smart elements (Def. 5.6) in the diagram. Further, some properties linking tables and diagrams are given. Finally, some experiments compare the various types of diagrams.

6.2 Ground Diagrams

Ground diagrams are the simplest form of labeled diagram (Def. 6.4).

Definition 6.4. ***Ground diagrams***
A ground diagram *is a labeled diagram where each label represents only a single value* $\langle = v \rangle$*.*

The notion of determinism (Def. 6.5) about the set of incoming/outgoing arcs of a node is introduced to simplify the definition of subclasses of ground diagrams.

Definition 6.5. ***In-d, out-d, in-nd and out-nd Node***
A node is in-d *(in-deterministic) if and only if at most one incoming arc by available label is allowed. Otherwise, it is said to be* in-nd *(in-non-deterministic).*
A node is out-d *(out-deterministic if and only if at most one outgoing arc by available label is allowed. Otherwise, it is said to be* out-nd *(out-non-deterministic).*
Out-d nodes are also commonly called decision *nodes. By definition each in-d (resp. out-d) node can also be said to be in-nd (resp. out-nd).*
Figure 6.2 gives an example of these possible combinations.

Using this definition, one can define three types of ground diagrams: the `MVD` already known but not so much used, the `MDD`, already well known and well used, and the `sMDD`, new in-between structure.

6.2.1 Multi-Valued Variable Diagrams (MVDs)

The formal definition of the multi-valued variable diagrams is the following Def. 6.6. The multi-valued variable diagrams used in this thesis corresponds to the ordered `MVD` as defined by [AFNP14].

Definition 6.6. ***Multi-Valued Variable Diagram***
A multi-valued variable diagram, also called `MVD`, is a ground diagram where any node is in-nd and out-nd. Figure 6.3 gives an illustration.

However, by construction, some nodes are always in-d or out-d (Prop. 6.7).

Proposition 6.7. *`MVD`s always have their nodes of level* 1 *(resp.* $N-1$*) in-d (resp. out-d).*

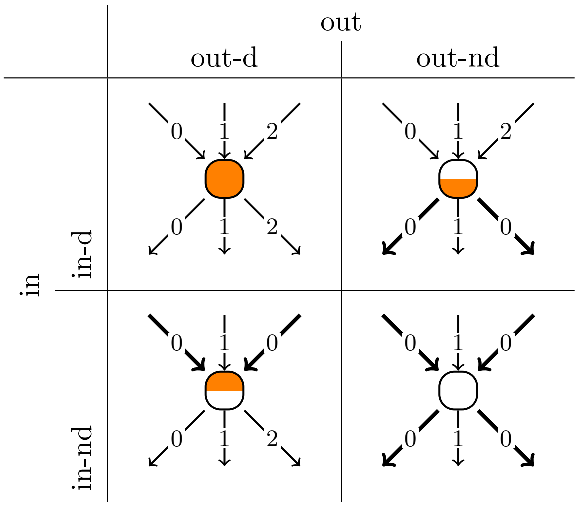


Figure 6.2: Example of nodes with in-d, out-d, in-nd and out-nd. Non-determinism is highlighted by bold arcs.

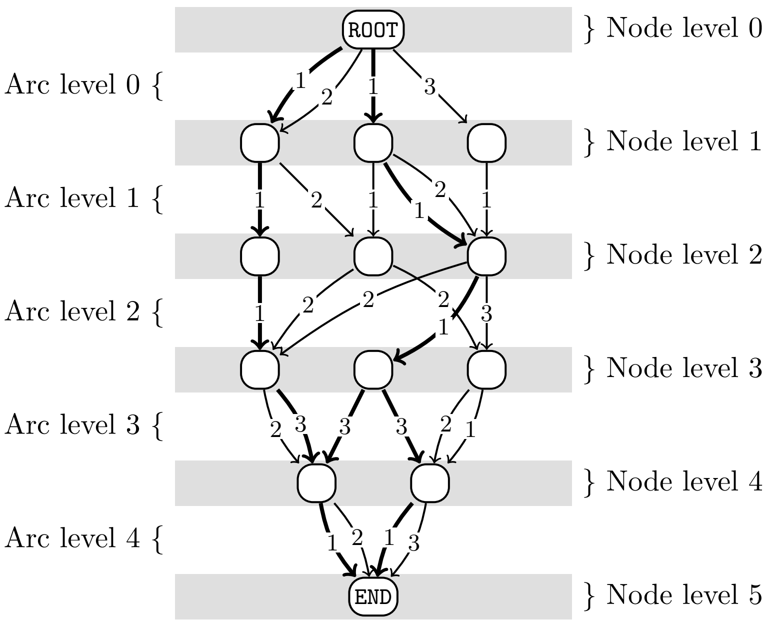


Figure 6.3: An `MVD`.

Proof. This is the combination of two facts. First, there is only one node at level 0 (resp. N), i.e. the `ROOT` (resp. `END`). Secondly, no two arcs can share the same source, destination, and label. Assuming two arcs sharing the same label targeting (resp. sourced from) a given node of level 1 (resp. $N-1$), as their source (resp. target) lies in level 0 (resp. N), their source (resp. target) is, by the first fact, `ROOT` (resp. `END`). They would therefore share their source, target, and label, which contradicts the second fact. This prooves that nodes at level 1 (resp. $N-1$) cannot have more than one arc by label entering (resp. exiting) them, making them in-d (resp. out-d). □

A ground table and an `MVD` with the same arity are equivalent if for each tuple of the ground table, there exists a path in the `MVD` associated to this tuple and if for each path of the `MVD`, the associated tuple exists in the ground table. Figure 6.4 displays the table corresponding to the `MVD` in Fig. 6.3.

Remark. Due to the in-nd property of `MVD`*s*, several paths can be associated with the same tuple. For example, the highlighted tuple in Fig. 6.4 corresponds to the three highlighted paths in Fig. 6.3.

x_0	x_1	x_2	x_3	x_4	#
1	1	1	2	1	
1	1	1	2	2	
1	**1**	**1**	**3**	**1**	(×3)
1	1	1	3	2	(×2)
1	1	1	3	3	
1	1	2	1	1	
1	1	2	1	3	
1	1	2	2	1	(×3)
1	1	2	2	2	(×2)
1	1	2	2	3	
1	1	2	3	1	(×2)
1	1	2	3	2	(×2)
1	1	3	1	1	
1	1	3	1	3	
1	1	3	2	1	
1	1	3	2	3	
1	2	1	3	1	(×2)
1	2	1	3	2	

x_1	x_2	x_3	x_4	x_5	#
1	2	1	3	3	
1	2	2	1	1	
1	2	2	1	3	
1	2	2	2	1	(×3)
1	2	2	2	2	(×2)
1	2	2	2	3	
1	2	2	3	1	(×2)
1	2	2	3	2	(×2)
1	2	3	1	1	
1	2	3	1	3	
1	2	3	2	1	
1	2	3	2	3	
2	1	1	2	1	
2	1	1	2	2	
2	1	1	3	1	
2	1	1	3	2	
2	2	2	1	1	
2	2	2	1	3	

x_1	x_2	x_3	x_4	x_5	#
2	2	2	2	1	(×2)
2	2	2	2	2	
2	2	2	2	3	
2	2	2	3	1	
2	2	2	3	2	
3	1	1	3	1	(×2)
3	1	1	3	2	
3	1	1	3	3	
3	1	2	2	1	
3	1	2	2	2	
3	1	2	3	1	
3	1	2	3	2	
3	1	3	1	1	
3	1	3	1	3	
3	1	3	2	1	
3	1	3	2	3	

Figure 6.4: The equivalent ground table to Fig. 6.3 (assuming arc level i is associated to variable x_i). Last column indicates the number of associated paths in the `MVD` if there is more than one.

6.2.2 Multi-Valued Decision Diagrams (`MDD`*s*)

A well-known subclass of ground diagrams is the `MDD` (Def. 6.8). The multi-valued decision diagrams used in this thesis corresponds to the ordered `MDD` as defined by [AFNP14] which are a generalization of the ordered `BDD` as defined by [DM02a]. The definition of the `BDD` goes back to the 80' [Bry86] where they were first used (in their non-ordered version) to describe Boolean formulas.

Definition 6.8. ***Multi-Valued Decision Diagram***
A multi-valued decision diagram is a multi-valued variable diagram where all nodes are in-nd and out-d. Figure 6.5 gives an example of an `MDD`. A binary decision diagram (`BDD`) is the binary version of the `MDD` (Fig. 6.6f).

An `MDD` and a ground table are two different representations of the same set of tuples. However, the out-d nature of the nodes of the `MDD` makes it impossible to have more than one path corresponding to each of the tuples (Prop. 6.10).

Definition 6.9. ***Path Uniqueness***
A given diagram is said to have the path uniqueness property if there is at most one path that could be associated with it for any given tuple. In

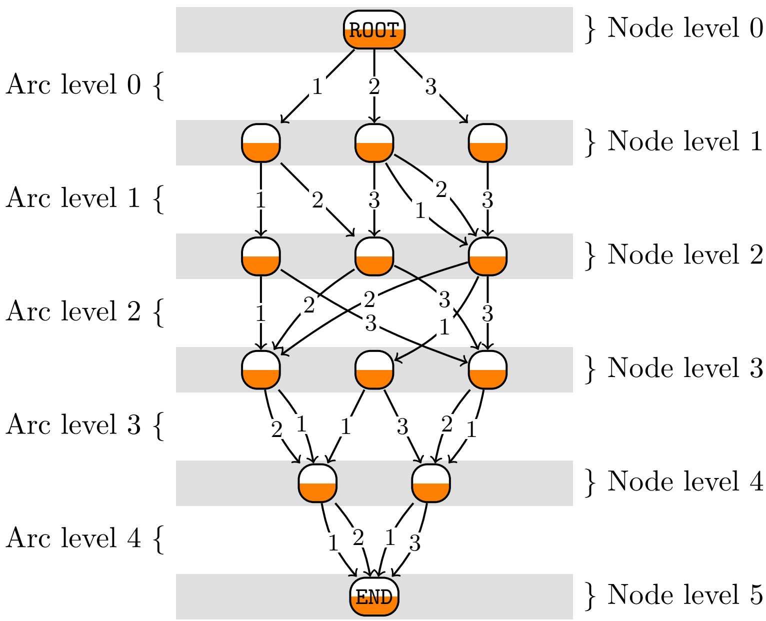


Figure 6.5: An `MDD`.

other words, a given diagram is said to have the path uniqueness property if the size of the corresponding ground table is always the number of paths in the diagram.

Proposition 6.10. *Path Uniqueness of* MDDs
Any MDD *has the path uniqueness property.*

Proof. Given a node and a label, the out-d property states that at most one node of the next layer can be reached. Applying this from `ROOT` to `END` leads to at most one path possible for a given tuple. The size of the equivalent ground table corresponds thus to the number of paths in the `MDD`. □

`pReduce`: Transforming a Table into an `MDD`

Reduction algorithms for generating decision diagrams from tables have been proposed in the literature. A first algorithm based on a breadth-first bottom-up exploration was proposed in [Bry86] for `BDD`*s*, and a second algorithm, using a dictionary and called `mddify`, was proposed in [CY08, CY10] for `MDD`*s*. More recently, `pReduce` [PR15] has been shown to admit a better worst-case time complexity than `mddify`.

Figure 6.6 illustrates the creation of an `MDD` in the spirit of `pReduce`. For clarity and simplicity, the process is illustrated starting from a binary table, thus leading to a `BDD`. In the illustrations, dashed and plain arcs stand for labels with values 0 and 1, respectively. A generalization of the process to an `MDD` is straightforward.

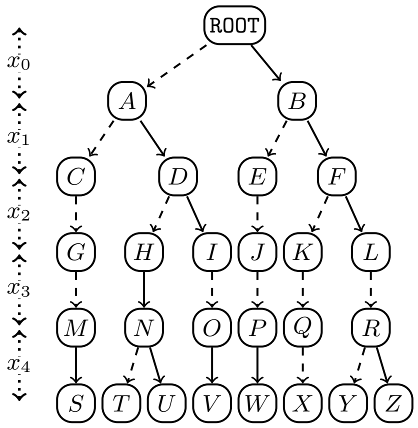
Initially, a table (Fig. 6.6a) and an ordering of the associated variables (x_0 to x_4, in a lexicographic order) are required. The tuples of these tables are sorted using a lexical ordering on the values following the given variable ordering (here, the table is already sorted). Then, the trie [GJMN07] (i.e. prefix tree) corresponding to this table is created by grouping the tuples with common prefixes. Figure 6.6b represents the table with the merged prefixes and Fig. 6.6c represents the resulting trie. Sorting the table was introduced in `pReduce` to reduce the complexity during the creation of the trie. A (non-reduced) `MDD` can be easily derived from this trie (Fig. 6.6d) by merging all the leaves of the trie (nodes S to Z) into one single `END` leaf. The `MDD` is, then, reduced by successively merging nodes when possible, from bottom to top. Merging is done by finding nodes having similar sets of outgoing arcs. Two sets of outgoing arcs are similar if they have the same cardinality, and for each arc in one set, there is an arc in the other set with the same label (value) and the same head node. In our example, one can observe that nodes M, O, and P have only one outgoing arc, each one labeled with 1 and

x_0	x_1	x_2	x_3	x_4
0	0	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	1
1	0	0	0	1
1	1	0	0	0
1	1	1	0	0
1	1	1	0	1

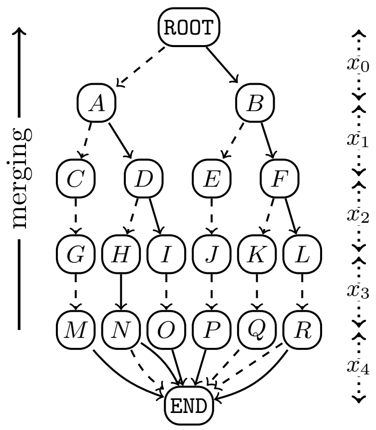
(a) Sorted table

<table>
<tr><th>x_0</th><th>x_1</th><th>x_2</th><th>x_3</th><th>x_4</th></tr>
<tr><td rowspan="4">0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td rowspan="3">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td>0</td></tr>
<tr><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td rowspan="4">1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td>0</td></tr>
<tr><td>1</td></tr>
</table>

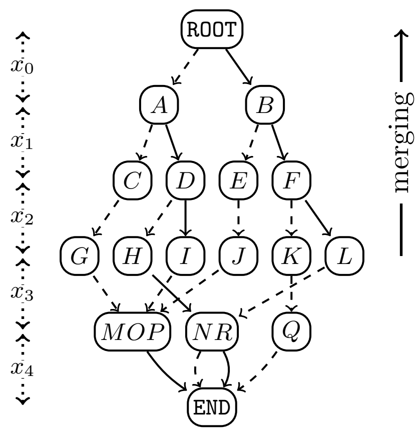
(b) Table with merged prefixes



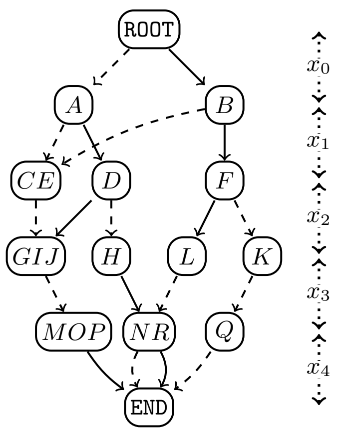
(c) Trie



(d) MDD from the Trie (level of x_5 has been reduced)



(e) MDD during reduction (level of x_4 has been reduced)



(f) Reduced MDD

Figure 6.6: Process of reducing a Table into an `MDD`.

reaching `END`. Hence, these nodes can be merged (leading to node MOP in Fig. 6.6e). Then, G, I, and J have now both only one outgoing arc, each one labeled with 0 and pointing to MOP. They can therefore be merged into GIJ. The `MDD` resulting from this iterative merging process is shown in Fig. 6.6f.

Complexity. The complexity of creating the trie and the `MDD` from the trie is

$$\mathcal{O}(m(t+d))$$

As stated in [PR15], the time and space complexities of `pReduce` are

$$\mathcal{O}(n+m+d)$$

where n is the number of nodes in the diagram, m is its arity and d is the size of the domain.

6.2.3 Semi Multi-Valued Decision Diagrams (sMDD*s*)

Definition 6.11. ***Semi Multi-Valued Decision Diagram***
A semi multi-valued decision diagram is a diagram where the nodes of levels in $[0;\lfloor\frac{r}{2}\rfloor[$ are in-nd and out-d and the nodes of levels in $]\lfloor\frac{r}{2}\rfloor+1;r]$ are in-d and out-nd. The nodes at levels $\lfloor\frac{r}{2}\rfloor$ and $\lfloor\frac{r}{2}\rfloor+1$ are in-nd and out-nd.
Figure 6.7 gives an example of an sMDD. A semi binary decision diagram (sBDD) is the binary version of the sMDD (Fig. 6.8h).

One interest of `sMDD`*s* over `MDD`*s* is the potential reduction of the number of nodes. Assuming an uniform variable domain size equal to d, the number of nodes in the initial trie is $\mathcal{O}(d^r)$ for the `MDD` while it is $\mathcal{O}(d^{r/2})$ for the `sMDD`. The gain can thus be very substantial, although merging renders precise predictions challenging to make.

Proposition 6.12. ***Path Uniqueness of** sMDDs*
Any sMDD has the path uniqueness property.

Proof. There is a unique path in the top of the diagram representing each prefix. There is also a unique path in the bottom of the diagram representing each suffix. This is derived from the demonstration of Prop. 6.10. Thus, a given tuple will always be represented by a path composed of the prefix partial path and the suffix partial path, both linked by the complementary edge. Two different paths for the same tuple would require two complementary edges between the end of the prefix partial path and the start of the suffix partial path. Nevertheless, this can only arise if the edges are labeled with two different values (since two edges between the same pair of nodes cannot be labeled with the same value). Hence, a contradiction since these two paths would represent two different tuples. □

`sReduce`: Transforming a Table into an `sMDD`

The transformation algorithm, called `sReduce`, results of an addaptation of `pReduce`. Figure 6.8 illustrates the creation of an `sMDD` in the spirit of `sReduce`. For clarity and simplicity, the process is illustrated starting from a binary table, thus leading to a `sBDD`. In the illustrations, dashed and plain arcs stand for labels with values 0 and 1, respectively. A generalization of the process to an `sMDD` is straightforward.

The algorithm is composed of five main steps. It starts with a table (Fig. 6.6a) and an ordering of the associated variables (x_0 to x_4, in a lexicographic order).

First, the initial table is split in two main parts:

– the p-table (table for the prefixes) corresponding to a restriction of the table to its first $\lfloor \frac{r}{2} \rfloor$ columns (or variables),

– the s-table (table for the suffixes) corresponding to a restriction of the table to its last $r - \lfloor \frac{r}{2} \rfloor - 1$ columns (or variables),

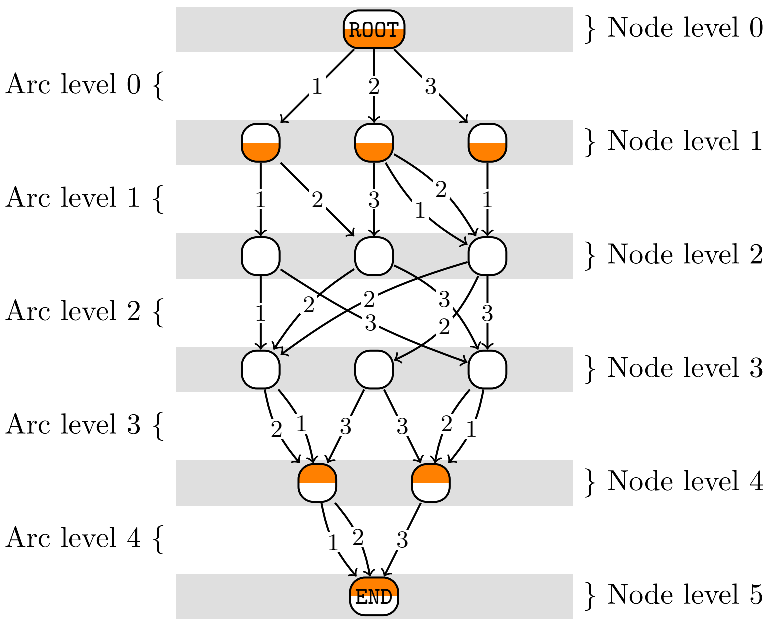


Figure 6.7: An `sMDD`.

At this point, note that all variables, except one, are involved in one of these two partial tables. On our example with $r = 5$, we obtain a p-table with 2 columns (corresponding to x_0 and x_1) and an s-table with 2 columns (corresponding to x_3 and x_4). The missing column (for variable x_2) will be considered in a later stage.

Second, duplicates are removed from the p-table (resp. s-table), which is then sorted using a lexicographic (resp. colexicographic[1]) order. After these steps, we obtain the p-table and the s-table shown in Fig. 6.8a

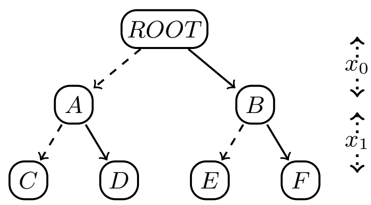
[1] Ordering is done by reading numbers from right to left.

x_0	x_1
0	0
0	1
1	0
1	1

(a) p-table

<table><tr><td rowspan="2">0</td><td>0</td></tr><tr><td>1</td></tr><tr><td rowspan="2">1</td><td>0</td></tr><tr><td>1</td></tr></table>

(b) p-table with merged prefixes



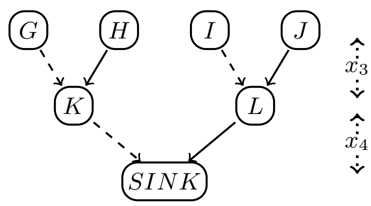
(c) p-trie

x_3	x_4
0	0
1	0
0	1
1	1

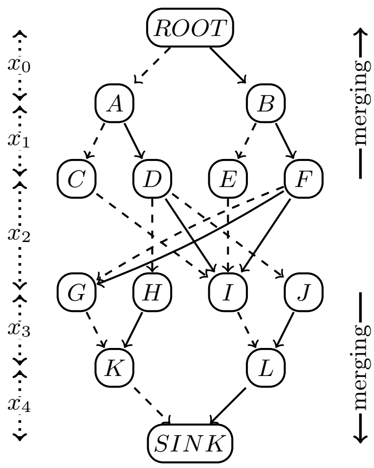
(d) s-table

<table><tr><td>0</td><td rowspan="2">0</td></tr><tr><td>1</td></tr><tr><td>0</td><td rowspan="2">1</td></tr><tr><td>1</td></tr></table>

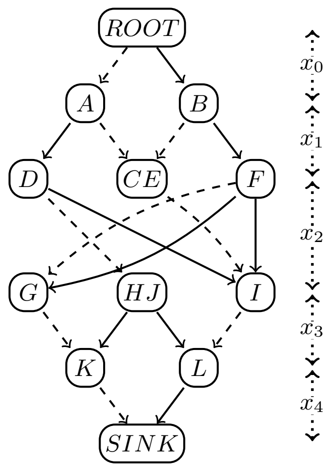
(e) s-table with merged sufixes



(f) s-trie



(g) sMDD



(h) Reduced sMDD

Figure 6.8: Reducing a Table into an `sMDD`.

and Fig. 6.8d.

Third, we build some equivalent tables sharing prefixes and suffixes. Equivalent trie (for the p-table) and inverted trie (for the s-table) are naturally derived from them (we call them p-trie and s-trie). Notably, the columns' order is preserved, and we start with a special root node for the p-tree, whereas we finish with a special `END` node for the s-tree. An illustration is given by Fig. 6.8b, Fig. 6.8c, Fig. 6.8e and Fig. 6.8f.

Fourth, for each tuple τ in the initial table, an arc is built. It links the node in the p-trie corresponding to the end of the prefix of τ and the node in the s-trie corresponding to the start of the suffix of τ. This arc is labeled with the intermediate variable value, which was involved neither in the p-table nor in the s-table. We obtain a new diagram, depicted in Fig. 6.8g, where arcs have been added for x_2.

Fifth, reduction, as in *pReduce*, is performed twice. On the one hand, from bottom to top, merging can be conducted by starting from the nodes that were leaves in the p-trie. For merging, the algorithm searches for similarities between sets of outgoing arcs. As an illustration, let us consider nodes C and E in Fig. 6.8g. These two nodes have both one outgoing arc with the same label 0 and the same head: therefore, they can be merged (node CE in Fig. 6.8h). On the other hand, from top to bottom, merging can be conducted by starting from the nodes with no parent in the s-trie. For merging, the algorithm searches now for similarities between sets of incoming arcs. As an illustration, observe how nodes H and J in Fig. 6.8g can be merged (node HJ in Fig. 6.8h). The graph obtained after complete reduction is depicted in Fig. 6.8h.

Proposition 6.13. *The graph obtained after executing `sReduce` on any specified table is an `sMDD`.*

Proof. Before executing merging operations, the diagram (at the end of step 4) is an `sMDD`, by construction. Merging conducted in the first pass (bottom-up) preserves out-determinism of any node at a level $< \lfloor \frac{r}{2} \rfloor$, while merging conducted in the second pass (top-down) preserves in-determinism of any node at a level $> \lfloor \frac{r}{2} \rfloor + 1$. $\square$

Complexity. Note that the complexity of `sReduce` is basically the same as `pReduce` as operations are essentially the same (sorting and merging).

6.2.4 `pReduce` versus `sReduce`

We first compared `sReduce` with `pReduce`. Similar execution times were observed for `sReduce` and `pReduce`. Concerning the size of the diagrams, Fig. 6.9 shows two performance profiles [DM02b] that allow us to compare the number of nodes and arcs globally in the generated `MDD`*s* and `sMDD`*s* for all the tables involved in our benchmark (around 230,000 tables of arity greater than or equal to 3). A performance profile is a cumulative distribution of the speedup performance of an algorithm $s \in S$ compared to other algorithms of S over a set I of instances: $\rho_s(\tau) = \frac{1}{|I|} \times |\{i \in I : r_{i,\,s} \leq \tau\}|$ where the performance ratio is defined as $r_{i,\,s} = \frac{t_{i,\,s}}{\min\{t_{i,\,s}|s \in S\}}$ with $t_{i,\,s}$ the time obtained with algorithm $s \in S$ on instance $i \in I$. A ratio $r_{i,\,s} = 1$ thus means that s is the fastest on instance i.

As we predicted, the number of nodes is significantly reduced in the generated sMDDs (more than a factor 8 for at least 70% of the tables), while the number of arcs tends to be slightly higher.

6.3 Basic Smart Diagrams

A way to compress even more the diagrams is to use the compression elements (Def. 5.6) as labels. We call such diagrams basic smart diagrams (Def. 6.14).

Definition 6.14. ***Basic Smart Diagram***
A basic smart diagram (basic smart `MVD` ($bs-$`MVD`), basic smart `MDD` ($bs-$`MDD`), basic smart `sMDD` ($bs-$`sMDD`),...) is a diagram where the arcs are labeled with basic smart elements (Def. 5.6), i.e. unary expressions of the form $\langle = v\rangle$, $\langle * \rangle$, $\langle \neq v\rangle$, $\langle \leq v\rangle$, $\langle < v\rangle$, $\langle \geq v\rangle$, $\langle > v\rangle$, $\langle \in S\rangle$ or $\langle \notin S\rangle$. An example of a basic smart `MVD` ($bs-$`MVD`) is given by Fig. 6.10.

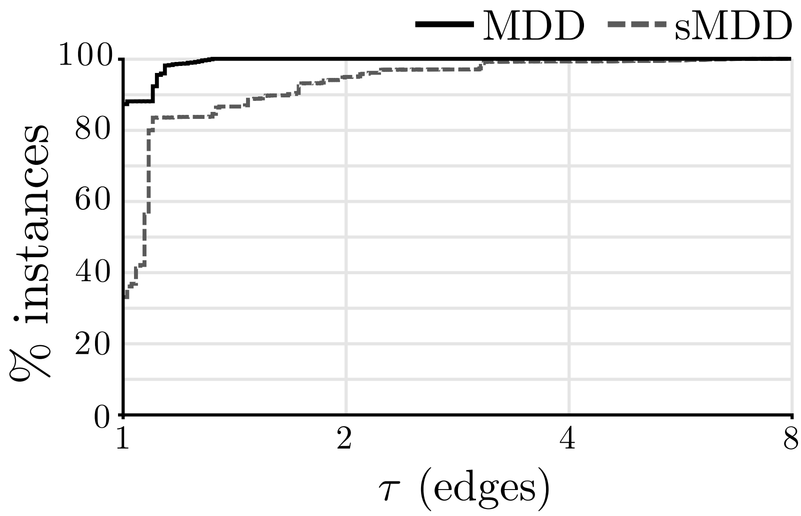
There are two ways of generating a basic smart diagram from a table (Fig. 6.11). Both involves two steps. The first consists of transforming the table into a basic smart table, then transforming it into the basic smart diagram. The second required the table to be transformed into a diagram and then into the basic smart diagram. The proprieties of the resulting graph depend on the transformation path taken.

6.3.1 From Basic Smart Table to Basic Smart `MVD`

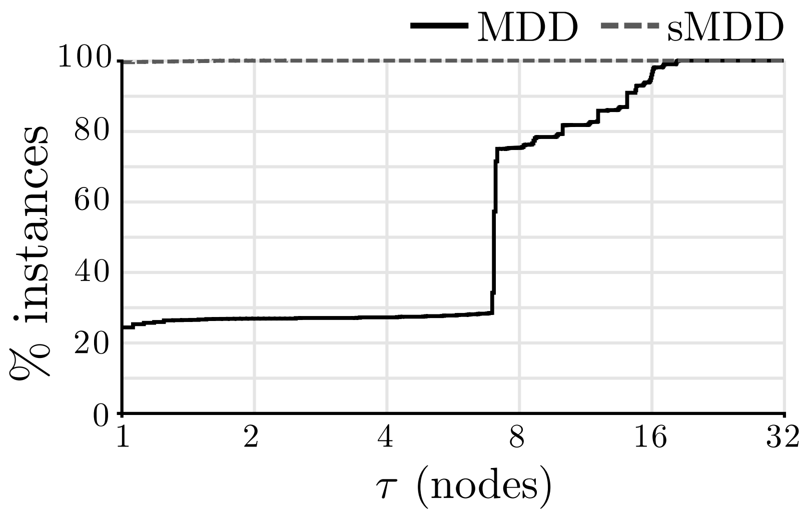
To create a $bs-$`MVD` from a $bs-$`table`, one can easily adapt a known procedure to construct diagrams, such as `pReduce` (see Sec. 6.2.2 for more details), or `sReduce` (see Sec. 6.2.3 for more details). The adaptation is of `pReduce` (resp. `sReduce`) is called `pReduce`$_{bs}$ (resp. `sReduce`$_{bs}$).

Even if `pReduce` (resp. `sReduce`) creates `MDD` (resp. `sMDD`), `pReduce`$_{bs}$ does not generate $bs-$`MDD` (resp. $bs-$`sMDD`). In fact, it always represents $bs-$`MVD`.

However, the property of path uniqueness may be conserved on one condition. As a $bs-$`table` may contain overlapping tuples (i.e. two tuples representing the same ground tuple), several paths may represent the same ground tuple. The presence of overlapping tuples in the $bs-$`table` would lead to overlapping paths in resulting diagrams. Each ground tuple common to the overlapping tuples is represented by all the



(a) Comparison of the number of edges



(b) Comparison of the number of nodes

Figure 6.9: Comparing the size of the generated `MDD`*s* and `sMDD`*s*.

corresponding paths. The resulting diagram does not have the unique path property in that case. If the initial $bs-$`table` does not contain any overlapping tuple then, no ground tuples are represented by more than one tuple at the same point, and therefore not represented by more than one path. In this case, the unique path property applied.

`pReduce`$_{bs}$: Transforming a $bs-$`table` into a $bs-$`MDD`

The four steps of the procedure `pReduce` are the following. First, the tuples of the table are sorted using lexicographic ordering. Second, the corresponding trie (i.e. prefix tree) is created by sharing common prefixes among the tuples. Third, a diagram is derived from the trie by merging all the trie leaves to form the `END` node. Finally, the diagram is reduced by merging, in a bottom-up way, each pair of nodes having

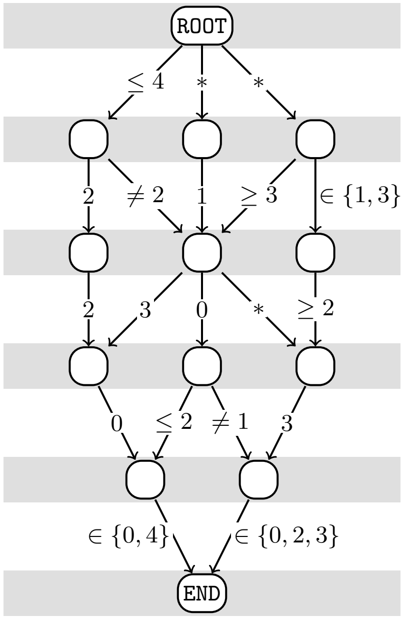


Figure 6.10: A basic smart `MVD`.

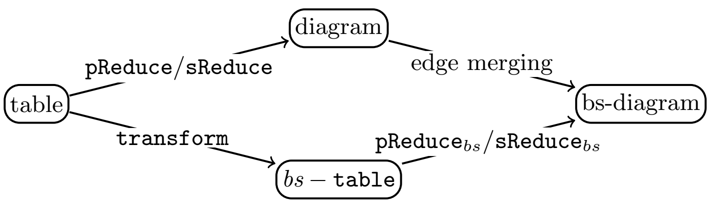


Figure 6.11: Ways to turn a table into a bs-diagram.

similar sets of outgoing arcs. Two sets of outgoing arcs are similar if they have the same cardinality, and for each arc in one set, there is an arc in the other set with the same label (value) and the same head.

Actually, for adapting it to $bs-$ `table`, we need to impose a total order on expressions (unary constraints) involved in basic smart tuples. For example, we can associate a pair of integers with each expression (unary constraint). The first element of the pair represents the type (operator) of the expression, and the second element the operand involved in the expression. Figure 6.12 illustrates the naturally derived lexicographic order. Using such order requires a simple hypothesis: two different used compression elements can represent the same subset of values from the domain. For example, $\leq dom.max$ and $*$ represent the same element. Therefore, for benefiting from a defined ordering, each occurrence of $\leq dom.max$ should be replaced by $*$.

Figure 6.13 illustrates through an example the four steps of `pReduce`$_{bs}$: going from a sorted $bs-$ `table` (Fig. 6.13a) to a trie (Fig. 6.13b), then into an `MVD` (Fig. 6.13c) and finally into a reduced `MVD` (Fig. 6.13d), where the highlighted node is the result of merging two nodes with similar outgoing sets of arcs). This example shows that `pReduce`$_{bs}$ does not necessarily generate a $bs-$ `MDD`, because some nodes are not out-d, possibly leading to several paths for the same tuple as it is the case for $(1, 1, 1)$.

`sReduce`$_{bs}$: Transforming a $bs-$ `table` into a $bs-$ `MDD`

Using the same ordering, a similar adaptation is possible for `sReduce`, the procedure that generates `sMDD`s, leading to `sReduce`$_{bs}$.

Expression	Representation
$= v$	$(0, v)$
$\neq v$	$(1, v)$
$*$	$(2, 0)$
$\leq v$	$(3, v)$
$\geq v$	$(4, v)$
$\in S$	$(5, \sum_{i \in S} 2^i)$
$\notin S$	$(6, \sum_{i \notin S} 2^i)$

Figure 6.12: Lexicographic Order on Expressions.

6.3.2 From Diagram to Basic Smart Diagram

Generating a basic smart diagram from a ground diagram is straightforward. At each layer i, we process every group G of (at least two) arcs sharing the same tail and head nodes. Specifically, we can compare $V = \{l(\omega) : \omega \in G\}$ with $\texttt{dom}(x_i)$, and consequently apply some rules (given in order of priority) for merging some arcs of G:

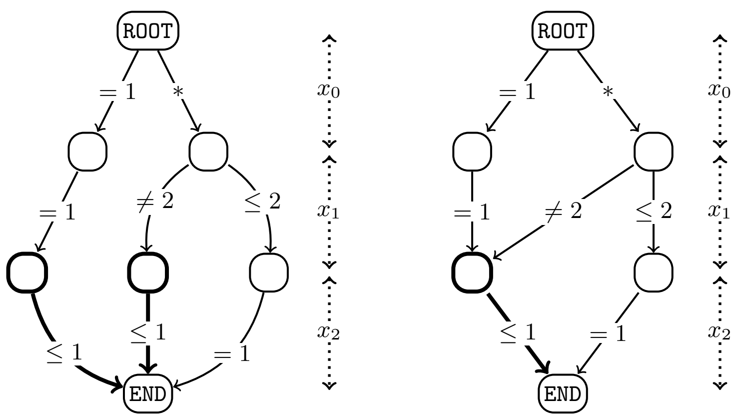
1. if $V = \texttt{dom}(x_i)$, then G is replaced by a unique arc labeled with $\langle * \rangle$
2. else if $\exists v \in \texttt{dom}(x_i)$ s.t. $V \cup \{v\} = \texttt{dom}(x_i)$, then G is replaced by a unique arc labeled with $\langle \neq v \rangle$
3. else
   (a) if m, defined as $\max\{v : \{v' \in \texttt{dom}(x_i) : v' \leq v\} \subseteq G\}$ is not equal to $\texttt{dom}(x_i).min$, then $G^m = \{\omega \in G : l(\omega) \leq m\}$ is replaced by a unique arc labeled with $\langle \leq m \rangle$. Otherwise, $G^m = \emptyset$.
   (b) if M, defined as $\min\{v : \{v' \in \texttt{dom}(x_i) : v' \geq v\} \subseteq G \setminus G^m\}$, is not equal to $\texttt{dom}(x_i).max$, then $G^M = \{\omega \in G \setminus G^m :$

x_1	x_2	x_3
$= 1$	$= 1$	≤ 1
$*$	$\neq 2$	≤ 1
$*$	≤ 2	$= 1$

(a) Sorted Table

x_1	x_2	x_3
$= 1$	$= 1$	≤ 1
$*$	$\neq 2$	≤ 1
	≤ 2	$= 1$

(b) Trie



(c) Trivial MVD (d) Reduced MVD

Figure 6.13: Turning a $bs - \texttt{table}$ into a $bs - \texttt{MVD}$ using $\texttt{pReduce}_{bs}$.

$l(\omega) \geq M\}$ is replaced by a unique arc labeled with $\langle \geq M \rangle$. Otherwise, $G^M = \emptyset$.

(c) Finally, given $G' = G \backslash G^m \backslash G^M$, if $|G'| > 1$ then G' is replaced by a unique arc labeled with $\langle \in S \rangle$ where $S = \{l(\omega) : \omega \in G'\}$ if $|S| \leq |\mathtt{dom}(x_i) \setminus G|$, otherwise G' is replaced by a unique arc labeled with $\langle \in S' \rangle$ where $S' = \{l(\omega) : \omega \in \mathtt{dom}(x_i) \setminus G\}$.

Figure 6.14 illustrates these merging rules. The variable of interest x_i has a domain (initially) composed of 10 values, and white cells represent the values that are present in G.

Note that our merging procedure keeps at most three arcs between any two nodes. An example is given in Fig. 6.15.

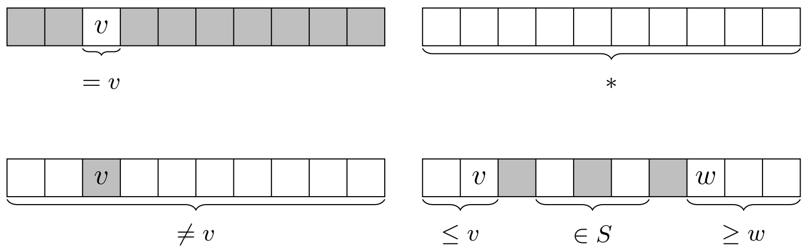
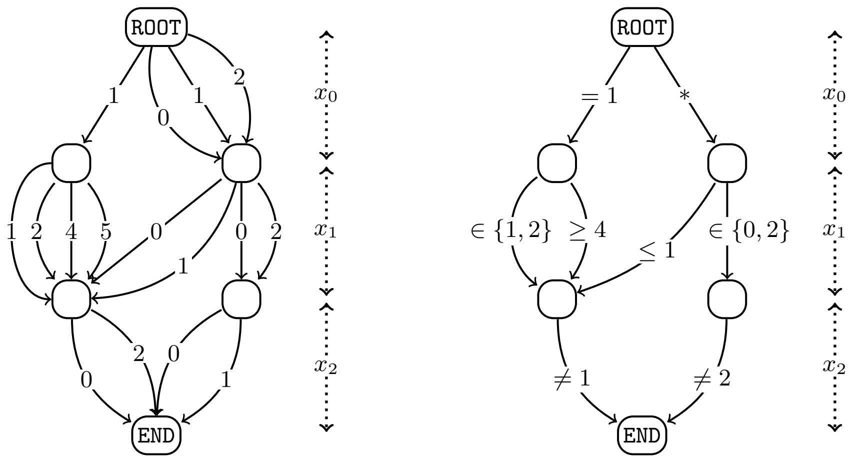


Figure 6.14: Illustration of the possible merging rules (on a domain of size 10).



(a) An MVD

(b) An equivalent $bs - \mathtt{MVD}$

Figure 6.15: Transforming an MVD into an equivalent $bs - \mathtt{MVD}$. The domains of the variables are $\mathtt{dom}(x_0) = \{0, 1, 2\}$, $\mathtt{dom}(x_1) = \{0, 1, 2, 3, 4, 5\}$ and $\mathtt{dom}(x_2) = \{0, 1, 2\}$.

6.3.3 Comparison of the Different Transformations

Depending on the main data structure (table or diagram) and possible transformation, we use different names to describe the benchmark suite:

- β_t: the initial benchmark. It is a set of roughly $4,000$ instances only containing (positive) table constraints and available on the XCSP3 website [BLP16].

- β_{bst}: instances of β_t have been transformed into instances where $bs-\texttt{table}$ replace (ordinary) tables. The compression used is the one presented in Sec. 5.4.2.2.

- β_{mdd}: instances of β_t have been transformed into instances where MDDs replace (ordinary) tables. The algorithm pReduce [PR15] was used.

- β_{bsmvd}: instances of β_{bst} have been transformed into instances where $bs-\texttt{MVD}s$ replace $bs-\texttt{table}$. The algorithm pReduce$_{bs}$ was used.

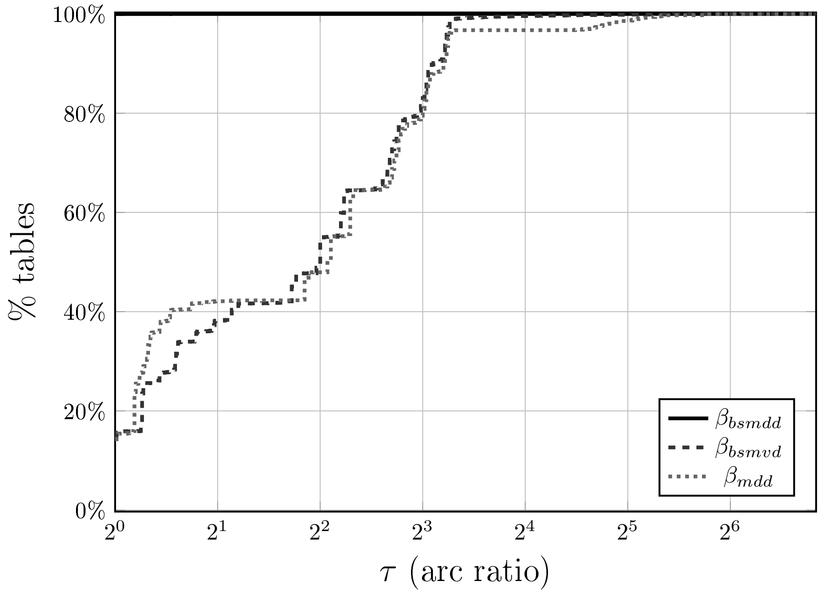
- β_{bsmdd}: instances of β_{mdd} have been transformed into instances where $bs-\texttt{MDD}s$ replace $\texttt{MDD}s$.

To start, we consider the results depicted in Fig. 6.16. The three benchmarks involving MVDs are β_{mdd}, β_{bsmvd} and β_{bsmdd}. In terms of compression, the clear winner is β_{bsmdd} with substantially fewer arcs than in the diagrams generated by the two other approaches. Let us recall that this approach consists of two main steps: 1) creating a graph and 2) merging arcs greedily. The alternative approach β_{bsmvd} that creates first a $bs-\texttt{table}$, and then converts it into a $bs-\texttt{MVD}$ is worse both in terms of the number of nodes and the number of arcs, even when compared to a standard generation of MDDs (β_{mdd}). One explanation is that, despite starting from smaller tables, there is less chance to merge nodes due to the proliferation of constraint labels in the compressed tables.

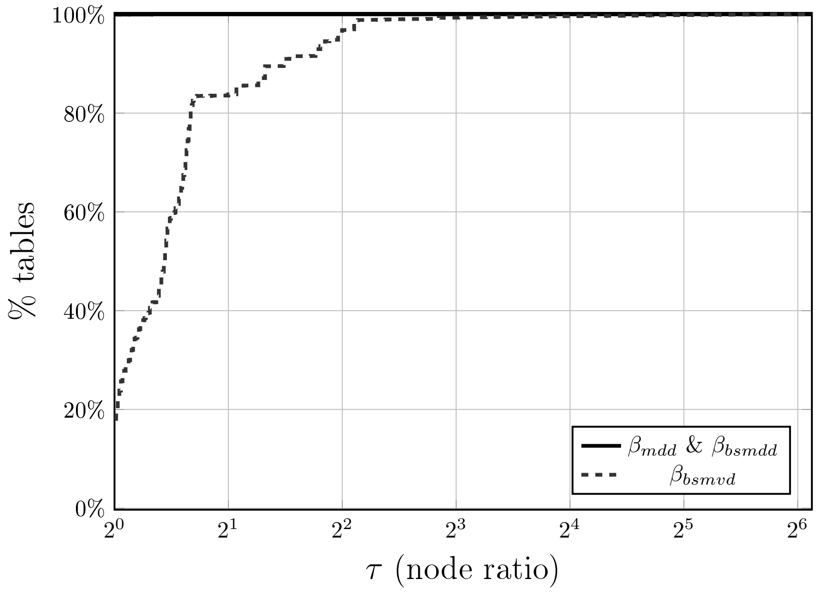
6.4 Incompressibility of some Diagrams

As for tables, some properties can be expressed about the compression of the diagrams that have the path uniqueness property. The first one (Prop. 6.15) links the incompressibility of diagrams to the incompressibility of tables.

Proposition 6.15. *A ground diagram with the path uniqueness property*



(a) Comparison of the number of edges



(b) Comparison of the number of nodes

Figure 6.16: Performance profile comparing the structure of the graphs from β_{bsmdd}, β_{bsmvd} and β_{mdd}.

has at most one arc between every two pairs of nodes if and only if the corresponding ground table has no non-trivial possible compression.

Proof. **At most one arc $\Rightarrow$ no non-trivial compression.**
Consider the simplest table with one tuple with one non-trivial compression element (for example, the table with the tuple $(1, *, 1)$. Every diagram with path uniqueness property representing it will result in a diagram with only one path, with a non-trivial compression element, meaning several edges in the corresponding ground diagram. Hence the contradiction.
No non-trivial compression $\Rightarrow$ at most one arc.
Assume the corresponding diagram could have a non-trivial compression on one arc, i.e. has at least two arcs with the same source and destination. As each path through the graph corresponds to a tuple, the path(s) going through that non-trivially compressed arc would correspond to a valid tuple containing a compression. Hence the contradiction. □

The second proposition (Prop. 6.17) is more an intuition. It expresses a possible metric that could be used to predict a diagram's wideness based on its associated table. This metric (Def. 6.16) is again based on the Hamming distance.

Definition 6.16. ***Midle Hamming Distance between two Tuples***
The midle Hamming distance between two tuples is the size of the tuple minus the sum of the sizes of the common prefix and suffix. Ex: the tail Hamming distance between $(1, 1, 0)$ *and* $(0, 1, 0)$ *is* 1*, between* $(0, 0, 0)$ *and* $(0, 1, 0)$ *is* 1*.*

Proposition 6.17. *Given a table, the wideness of a corresponding diagram with the uniqueness property is linked to the middle Hamming distances between the table's tuples. The higher the distance, the fewer merges can happen during the diagram's construction, resulting in a wider diagram.*

Proof. Given a two-tuple table and a corresponding diagram with two paths, these paths can share arcs only when sharing the same prefix or suffix. This means that the higher the prefix/suffix Hamming distance is between two tuples, the higher the number of arcs and nodes in the diagram. Given a larger table, the higher the distribution of the pairwise tail Hamming distance is, the wider the diagram should be expected to be. □

6.5 Conclusion

MDD*s* have already proven to be a critical compression tool to help reducing the memory space used by an extensional representation. Our work on diagrams shows that even a bit of non-determinism can even achieve a greater reduction, especially concerning the number of nodes required for the representation. Being able to build an MVD should help further reducing the size of the representation. The benefit of basic smart elements lies in the reduction of the number of arcs.

The work on sMDD*s* was published as part of the [VLS18] paper. The work on basic smart MDD*s* was published as part of the [VLS19a] paper.

Part III

Propagation Algorithms

Chapter 7

Filtering Positive Smart Table Constraints

A gravitational wave is a very slight stretching in one dimension. If there's a gravitational wave traveling towards you, you get a stretch in the dimension that's perpendicular to the direction it's moving. And then perpendicular to that first stretch, you have a compression along the other dimension.

- Rainer Weiss

7.1 Introduction

Our work on positive table constraints focuses on adapting the Compact-Table (Sec. 3.2.10) propagation algorithm to handle basic smart tables (Def. 5.4) without decompressing them. This is done incrementally, by first handling short tables (Def. 5.3) and then basic smart tables (Def. 5.4).

As already seen in Chap. 3, the `CT` algorithm follows some invariants (inv. 7.1, which states which tuples belong to T^c, and inv. 7.2, which states which values belong to $\texttt{dom}^c(x)$) guaranteeing the correctness of propagation. Respecting these invariant leads to a GAC propagator as stated by Prop. 7.1. Two additional invariants (inv. 7.3, which states when any assignement is a solution, and inv. 7.4, which states when there is no solution) are derived from inv. 7.2. They do not change the propagation strength. However, as inv. 7.4 is quicker than checking inv. 7.2 for each variable, adding it may speed up the propagation. Adding inv. 7.3 may help detecting earlier whether the constraint may be deactivated (i.e. when the constraint is always valid).

Invariant 7.1 (**Current Table Update - Ground Tables**). *Given the notations: T^0, the initial table (i.e. before any propagation occurs), T^c, the reduced table at a given current state c of propagation, and, $\mathtt{dom}^c(x)$, the domain of x at the current state c. A ground tuple τ belongs to the current table T^c if and only if it is a ground tuple of the initial table and all its values still belong to the respective current domains of the associated variables from* `scp`.

$$\Big(\tau \in T^0 \wedge \forall x \in \mathtt{scp},\ \tau[x] \in \mathtt{dom}^c(x)\Big) \Leftrightarrow \Big(\tau \in T^c\Big)$$

Invariant 7.2 (**Domain Filtering - Ground Tables**). *Given any variable $x \in \mathtt{scp}$, each value v in $\mathtt{dom}^c(x)$ should appear in at least one of the ground tuple $\tau \in T^c$.*

$$\forall x \in \mathtt{scp},\ \forall v \in \mathtt{dom}^c(x),\ \exists \tau \in T^c,\ \tau[x] = v$$

Proposition 7.1 (**GAC Filtering - Ground Tables**). *A positive table constraint enforces GAC if inv. 7.1 and inv. 7.2 hold.*

Proof. By means of inv. 7.1, the set of valid tuples is maintained. Invariant 7.2 detects when a given value (x, a) can be removed if necessary. □

Invariant 7.3 (**Entailement - Ground Tables**). *A positive table constraint is entailed if and only if the table contains all the possible tuple w.r.t. the domains of the variables.*

$$\Big(\big|\{\tau : \tau \in T^c\}\big| = \prod_{x \in \mathtt{scp}} \big|\mathtt{dom}(x)\big|\Big) \Leftrightarrow \top$$

Invariant 7.4 (**Emptiness - Ground Tables**). *A positive table constraint is falsify if and only if it is empty.*

$$\Big(T^c = \emptyset\Big) \Leftrightarrow \bot$$

To handle basic smart elements, when filtering tables, the first step is to update the required invariants to this new situation. The first element to keep in mind is that the table may contain both basic smart tuples and ground tuples. As each element of each table may represent several values, we consider them as sets of values (instead of single values). The element $\langle = v\rangle$ corresponds thus to the set $\{v\}$, the element $\langle \neq v\rangle$ corresponds to $\mathtt{dom}(x) \setminus \{v\}$,... This eases the update of the invariants.

The updated invariants (inv. 7.5 and inv. 7.6) allow proving again a GAC filtering (Prop. 7.2) holds. From them we can also derive two invariants (inv. 7.7 and inv. 7.8) which define conditions where the constraint is either always **`true`** or always **`false`**.

Invariant 7.5 (**Current Table Update - Basic Smart Tables**). *Given the notations: T^0, the initial table (i.e. before any propagation occurs), T^c, the reduced table at a given current state c of the propagation, and, $\mathtt{dom}^c(x)$, the domain of x at the current state c. A tuple τ belongs to the current table T^c if and only if it was a tuple of the initial table and all its values still belongs to the respective current domains of the associated variables from* `scp`.

$$\Big(\tau \in T^0 \wedge \forall x \in \mathtt{scp},\ \tau[x] \cap \mathtt{dom}^c(x) \neq \emptyset\Big) \Leftrightarrow \Big(\tau \in T^c\Big)$$

Invariant 7.6 (**Domain Filtering - Basic Smart Tables**). *Given any variable $x \in \mathtt{scp}$, each value v in $\mathtt{dom}^c(x)$ should be included in at least one of the tuple $\tau \in T^c$.*

$$\forall x \in \mathtt{scp},\ \forall v \in \mathtt{dom}^c(x),\ \exists \tau \in T^c,\ v \in \tau[x]$$

Proposition 7.2 (**GAC Filtering - Basic Smart Tables**). *A positive table constraint enforces GAC if inv. 7.5 and inv. 7.6 hold.*

Proof. By means of inv. 7.5, the set of valid tuples is maintained. Invariant 7.6 detects when a given value (x, a) can be removed. ☐

Invariant 7.7 (**Entailement - Basic Smart Tables**). *A positive table containing all the possible ground tuple allows each of them to be a solution.*

$$\Big(\big|\{\tau : \exists \tau \in T^c, \mathtt{s.t.}\ \tau \subseteq \tau\}\big| = \prod_{x \in \mathtt{scp}} \big|\mathtt{dom}(x)\big|\Big) \Leftrightarrow \top$$

Invariant 7.8 (**Emptiness - Basic Smart Tables**). *An empty positive table does not have a solution.*

$$\Big(T^c = \emptyset\Big) \Leftrightarrow \bot$$

The adaptation of the `CT` algorithm to handle basic smart elements is done incrementally. To preserve GAC, the two main invariants (inv. 7.5 and inv. 7.6) should stay valid.

Finally, let us recall two assumptions. First, regarding the inputs of the table constraint, we assumed a finite integer domain for each variable, containing values 0, 1, ..., $size - 1$. We can easily adapt the algorithm to other finite domain. Secondly, we assume the variables are responsible for triggering a backtrack when their domains become empty.

7.2 Adaptations to Compact-Table

First, `CT` is extended to handle short tables (by adding the $\langle * \rangle$). Then, the other smart elements are added one by one: $\langle \neq v \rangle$, followed by the duo $\langle \leq v \rangle / \langle \geq v \rangle$ and finally the $\langle \in S \rangle$. Finally, a way to solve smart table is given, using a simple mapping to a basic smart table.

7.2.1 `CT`*: Handling Short Tables

The first step is to extend Compact-Table to handle Short Tables (Def. 5.3). Short Tables contain two types of Basic Smart Elements: $\langle = v \rangle$ and $\langle * \rangle$.

As we now consider sets of values intead of standalone values insides the tuples, a small update is done to the initial definition of the `supports` (Def. 3.3). This new definition (Def. 7.3) does not change the semantic of the `supports`.

Definition 7.3. ***supports* (as Used in *CT** and *CT*^{bs})**
Given a tuple τ, for a given variable x, $\forall v \in$ dom(x),

$$\texttt{supports}[x, v]\langle \tau \rangle = \begin{cases} 1 & \textit{iff } v \in \tau[x] \\ 0 & \textit{iff } v \notin \tau[x] \end{cases}$$

In other words, the bitset $\texttt{supports}[x, v]$*, for a variable x and a value $v \in$ dom(x), represents the tuples supporting the value v from dom(x).*

The intuition behind the modification is that each tuple containing $\langle * \rangle$ associated for given variable x remains valid regarding x as long as `dom`(x) is not empty. In addition, let us assume an empty domain variable triggers inconsistency by itself. Using this assumption, the job of the table propagation algorithm is to keep valid these tuples whatever the modification of `dom`(x) is.

The modification of the propagator consists of a duplication of the precomputed bitsets. The second set is called `supports`* and is defined formally in Def. 7.4.

Definition 7.4. ***supports** (as Used in *CT**)**
Given a short tuple τ, for a given variable x, $\forall v \in$ dom(x),

$$\texttt{supports}^*[x, v] = \begin{cases} 1 & \textit{iff } \tau[x] \textit{ is } \langle = v \rangle \\ 0 & \textit{iff } \tau[x] \in \{\langle = w \rangle, \langle * \rangle\} \textit{ with } w \neq v \end{cases}$$

This formula defines a given $\texttt{supports}^*[x, v]$ *for a variable x and a value $v \in$ dom(x) to be the bitset containing the tuples supporting exclusively one value (v) from x. Figure 7.1, for tuples τ_1 and τ_2, displays an example of* `supports` *and* `supports`* *for a given short table.*

This new series of supports* bitsets is used in the classical update where currtable is updated using the removed values (Δ). Removing a value v from a domain should only remove the tuples supporting exclusively v. Thus, tuples supporting several values, such as the ones containing $\langle * \rangle$, can't be invalidated during the classical update. This is avoided using supports* instead of supports when doing this update. Invariant 7.5 is thus followed. Algorithm 12 shows the modification done to the initial CT algorithm (Algo. 1) to handle $\langle * \rangle$. This version is called CT*.

For the filtering part, no modification is needed since it already guarantees the validity of inv. 7.6.

Finally, one can notice that inv. 7.7 is never checked in the algorithm. Due to potentially overlapping tuples, the complexity required to compute the number of ground tuples outgrows the benefits of the invariant. As the invariant is not mendatory to achieve GAC, it was decided not to include this verification in the algorithm.

Complexity. The complexities (both time and spacial) remain exactly the same as CT. Only the space used is impacted since there is twice the number of precomputed bitsets (supports and supports*).

7.2.2 Handling the $\langle \neq v \rangle$

The next basic smart element added is $\langle \neq v \rangle$. In fact, no modification to Algo. 12 is required. Only the definition of supports* requires a slight update.

	x	y	z
τ_1	1	$*$	3
τ_2	$*$	2	1
τ_3	$\neq 2$	$\neq 1$	2
τ_4	$*$	$\neq 2$	$*$

(a) Table

	supports				supports*			
	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4
$(x,1)$	1	1	1	1	1	0	0	0
$(x,2)$	0	1	0	1	0	0	0	0
$(x,3)$	0	1	1	1	0	0	0	0
$(y,1)$	1	0	0	1	0	0	0	0
$(y,2)$	1	1	1	0	0	0	0	0
$(y,3)$	1	0	1	1	0	1	0	0
			...					

(b) Bitsets

Figure 7.1: Illustration of the bitsets supports and supports* (dark grey highlight the bits for $\langle * \rangle$ and light grey, the bits for $\langle \neq v \rangle$).

Definition 7.5. ***supports**** *as Used to Handle* $\langle = v\rangle$*,* $\langle * \rangle$ *and* $\langle \neq v\rangle$
Given a basic smart tuple τ *containing only the smart element* $\langle = v\rangle$*,* $\langle * \rangle$ *and* $\langle \neq v\rangle$*, for a given variable* x*,* $\forall v \in \texttt{dom}(x)$*,*

$$\texttt{supports}^*[x, v] = \begin{cases} 1 & \textit{iff } \tau[x] \textit{ is } \langle = v\rangle \\ 0 & \textit{iff } \tau[x] \in \{\langle = w\rangle, \langle * \rangle, \langle \neq u\rangle\} \textit{ with } w \neq v \end{cases}$$

The intuitive definition is the same as previously: $\texttt{supports}^*[x, v]$ *for a variable* x *and a value* $v \in \texttt{dom}(x)$ *is the bitset containing the tuples supporting exclusively one value (v) from* x*. Figure 7.1 shows an exam-*

Algorithm 12: The $\texttt{CT}^*$ algorithm

```
Method updateTable()                              // Invariant 7.5
    foreach variable x ∈ S^val do
        mask ← 0^64
        if |Δ_x| < |dom^c(x)| then                // Classical update
            foreach value a ∈ Δ_x do
                mask ← mask | supports*[x, a]
            mask ←∼ mask
        else                                      // Reset update
            foreach value a ∈ dom^c(x) do
                mask ← mask | supports[x, a]
        currtable ← currtable & mask

Method filterDomains()           // Same filtering as CT
    (Algo. 1), Invariant 7.5
    foreach variable x ∈ S^sup do
        foreach value a ∈ dom^c(x) do
            intersection ← currtable & supports[x, a]
            if intersection = 0^64 then
                dom^c(x) ← dom^c(x) \ {a}
                currtable ← currtable & ∼ supports[x, a]

Method enforceGAC()
    updateTable()
    count ← nb1s(currtable)
    if count = 0 then                             // Invariant 7.8
        return ⊥                         // backtrack triggered
    filterDomains()
```

ple of both structure, `supports` and `supports`*, for a table containing $\langle = v\rangle$, $\langle *\rangle$ and $\langle \neq v\rangle$ elements.

Given this new definition of `supports`*, Algo. 12 still enforces GAC for tuples containing $\langle = v\rangle$, $\langle *\rangle$ and $\langle \neq v\rangle$.

Let us demonstrate the correctness for a tuple with $\langle \neq v\rangle$ associated to x when at least one value of $\texttt{dom}(x)$ is removed. There are three different cases to consider:

- $\texttt{dom}(x) = \emptyset$: as hypothesized, an empty domain successfully triggers itself a backtrack.

- If $|\Delta_x| < |\texttt{dom}^c(x)|$, a classical update is used : As $|\Delta_x|$ is as least one (since we run the udate only on variables in $\texttt{S}^{\texttt{val}}$, i.e. variables with modified domains), `dom` (x) contains at least two values. In this case, the tuple is always a support for the given variable and as the corresponding bits in `supports`* are set to 0 by construction, the tuple is successfully not removed from `currtable`.

- If $|\Delta_x| \geq |\texttt{dom}^c(x)|$, a reset update is used : Reconstructing the `currtable` from `supports` is always correct.

This confirms that `CT`* does not need adaptations to handle $\langle \neq v\rangle$, adapting the way we build `supports`* automatically is only mandatory to manage this case.

7.2.3 Handling $\langle \geq v\rangle$ and $\langle \leq v\rangle$

First, note that it is sufficient to focus on expressions of the form $\langle \geq v\rangle$ and $\langle \leq v\rangle$. This is possible since $\langle > v\rangle$ and $\langle < v\rangle$ are equivalent to $\langle \geq v+1\rangle$ and $\langle \leq v-1\rangle$ with the assumption on the domains. We first introduce two additional arrays of bitsets: `supportsMin` (Def. 7.6) for $\langle \leq v\rangle$ and `supportsMax` (Def. 7.7) for $\langle \geq v\rangle$.

Definition 7.6. *`supportsMin`*
Given a smart tuple τ, for a given variable x, $\forall v \in \texttt{dom}(x)$,

$$\texttt{supportsMin}[x,v] = \begin{cases} 0 & \textit{iff } \{w\colon w \in \tau[x] \texttt{ and } w \geq v\} = \emptyset \\ 1 & \textit{iff } \{w\colon w \in \tau[x] \texttt{ and } w \geq v\} \neq \emptyset \end{cases}$$

Figure 7.2 displays an example of `supportsMin` for a given basic smart table.

Definition 7.7. *supportsMax*
Given a smart tuple τ, for a given variable x, $\forall v \in$ dom(x),

$$\texttt{supportsMax}[x,v] = \begin{cases} 0 & \textit{iff } \{w\colon w \in \tau[x] \textbf{ and } w \leq v\} = \emptyset \\ 1 & \textit{iff } \{w\colon w \in \tau[x] \textbf{ and } w \leq v\} \neq \emptyset \end{cases}$$

Figure 7.2 displays an example of `supportsMax` *for a given basic smart table.*

The definition of `supports`* require again a slight adjustment (Def. 7.8). Its semantics is however unchanged: only *explicit* supports of (x, a) are considered.

Definition 7.8. ***supports* as used to handle $\langle = v\rangle$, $\langle *\rangle$, $\langle \neq v\rangle$, $\langle \leq v\rangle$ and $\langle \geq v\rangle$***
Given a basic smart tuple τ containing only the smart elements $\langle = v\rangle$, $\langle *\rangle$, $\langle \neq v\rangle$, $\langle \leq v\rangle$ and $\langle \geq v\rangle$, for a given variable x, $\forall v \in$ dom(x),

$$\texttt{supports}^*[x,v] = \begin{cases} 1 & \textit{iff } \tau[x] \textit{ is } \langle = v\rangle \\ 0 & \textit{iff } \tau[x] \in \{\langle = w\rangle, \langle *\rangle, \langle \neq u\rangle, \langle \leq u\rangle, \langle \geq u\rangle\} \textit{ with } w \neq v \end{cases}$$

The intuitive definition is the same as previously: `supports`$^*[x,v]$ *for a variable x and a value $v \in$ dom(x) is the bitset containing the tuples supporting exclusively one value (v) from x. Figure 7.2 shows an example both structure of* `supports` *and* `supports`* *for the given basic smart table.*

Starting from the `CT`* algorithm, only the lines of the classical update requires some changes (Algo. 12 line 4). The classical update is replaced by the lines at Algo. 13. Note that `min` (resp. `max`) denotes the smallest (resp. largest) value of `dom` (x), whereas `minChanged`() (resp. `maxChanged`()) is a method that returns true when `min` (resp. `max`) has changed since the last call of the algorithm. Line 1 is slightly modified to compensate the overhead induced by the two operations. Because lines 5-8 handle all the values that are respectively less than and greater than `min` and `max`, we only consider at line 3 the values $a \in \Delta_x$ such that `dom`(x).`min` $< a <$ `dom`(x).`max`.

Correctness is shown for $\langle \leq v\rangle$, considering all cases at column x for tuple τ. The case $|\texttt{dom}(x)| = 0$ is as trivial as in the last section. For the case of the reset-based update, as `supports` precisely depicts the acceptance of values by tuples, this is necessarily correct. Finally in the incremental update (Algo. 13), due to the constructions of the bitsets, i.e. the bit for τ in `supports`* is always set to 0 (resp. 1), updating depends only on `supportsMin` (resp. `supportsMax`). By definition of

$\langle \leq v \rangle$, if dom(x).min is lower than v, there is at least one value in the domain which is meaning still supported. By construction, the bit for τ in supportsMin[x,dom (x).min] is 1, keeping τ in currtable when used in the new update. If dom(x).min $> v$, by construction, the bit for τ is 0 in supportsMin[x,dom (x).min], leading to a removal of τ.

	x	y	z
τ_1	$\neq 1$	$*$	3
τ_2	3	≤ 2	$\neq 1$
τ_3	< 3	2	$\neq 2$
τ_4	> 2	≥ 2	$*$

(a) Table

	supports				supports*				supportsMin				supportsMax			
	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4
$(x,1)$	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0
$(x,2)$	1	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0
$(x,3)$	1	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1
$(y,1)$	1	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0
$(y,2)$	1	1	1	1	0	0	1	0	1	1	1	1	1	1	1	1
$(y,3)$	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1
								...								

(b) Bitsets

Figure 7.2: Illustration of the bitsets supports, supports*, supportsMin and supportsMax (light grey highlight the bits for $\langle \leq v \rangle$ and dark grey, the bits for $\langle \geq v \rangle$).

Algorithm 13: The classical update handling $\langle =, \rangle$ $\langle \neq v \rangle$, $\langle * \rangle$, $\langle \leq v \rangle$ and $\langle \geq v \rangle$

if $|\Delta_x| + 2 < |$*dom*$^c(x)|$ **then** // Classical update
  foreach *value* $a \in \Delta_x$ *such that* *dom*(x).*min* $< a <$ *dom*(x).*max* **do**
    mask ← mask | supports*[x, a]
  mask ← ∼ mask
  if *dom (x).minChanged()* **then**
    mask ← mask & supportsMin[x, dom(x).min]
  if *dom (x).maxChanged()* **then**
    mask ← mask & supportsMax[x, dom(x).max]

7.2.4 Handling $\langle \in S \rangle$

There is no easy way to handle expressions of the form $\langle \in S \rangle$ (or $\langle \notin S \rangle$) using incremental updates (on bitsets). To work, an incremental update would require counters to keep track of the number of remaining values from each set still in the domain. When such counter drops to zero, the corresponding tuple would be removed from `currtable`. Unfortunately, such a method is conflicting with the bitset philosophy. The sets would be handled separately, diminishing the use of bitsets. A solution could be to gather the same sets in the same words and assigning them a shared counter. In practice, the situation is much improbable to arise. Because of the combinatorial explosion of the number of possible sets, the probability of having the same set multiple times in the same table decreases while the size of the domain increase. Moreover, if we have sets concerning two different variables, gathering the bit associated with similar sets on one variable may scatter the bit related to similar sets on the second one.

We propose to systematically execute reset-based update as done in [WXYL16] for passing from `STRbit` to `STRbit-C` (`STRbit` is an algorithm based on the `STR` family of propagator which also use bitsets to speedup the computations, `STRbit-C` is its extension, handling c-tuples, i.e. tuples with $\langle \in S \rangle$ elements). More precisely, as soon as a variable is involved in an expression of the form $\langle \in S \rangle$ (or $\langle \notin S \rangle$) in one of the tuples of the basic smart table, a reset-based update is forced.

7.2.5 The $\texttt{CT}^{bs}$ Algorithm

The $\texttt{CT}^{bs}$ algorithm is defined by Algo. 14. It uses `supports` (Def. 7.3), `supports`* (Def. 7.8), `supportsMin` (Def. 7.6) and `supportsMax` (Def. 7.7). It classified the variables into two complementary sets: The variables forwich a set $\langle \in S \rangle$ has been used ($\texttt{S}^{\langle \in S \rangle}$) and those forwich not ($\texttt{scp} \setminus \texttt{S}^{\langle \in S \rangle}$). These two categories are used during the update phase. This update is carried with reset only for the first categories of variable and with incremental or reset, depending on the delta, for the second. The filtering is then done on the remaining values for the unbound variables using the `supports`.

These steps guarantee a GAC propagation since the two invariants inv. 7.5 and inv. 7.6 are made valid by the modifications.

One can notice that inv. 7.7 is never checked in the algorithms (as already in $\texttt{CT}^*$). The issue about counting the number of unique ground tuples being still there.

Complexity. Both complexities remains the same as CT. To recall, the worst-case time complexity is

$$\mathcal{O}\Big(|\mathtt{scp}|\ d^c\ \Big\lceil\frac{|T^0|}{w}\Big\rceil\Big)$$

Algorithm 14: $\mathtt{CT}^{bs}$

```
Method updateTable()                              // Invariant 7.1
    foreach variable x ∈ S^val do
        mask ← 0^64
        if |Δ_x| + 2 < |dom^c(x)| ∧ x ∉ S^⟨∈S⟩ then    // Classical update
            foreach value a ∈ Δ_x such that dom(x).min < a < dom(x).max do
                mask ← mask | supports*[x, a]
            mask ← ~ mask
            if dom (x).minChanged() then
                mask ← mask & supportsMin[x, dom(x).min]
            if dom (x).maxChanged() then
                mask ← mask & supportsMax[x, dom(x).max]
        else                                          // Reset update
            foreach value a ∈ dom^c(x) do
                mask ← mask | supports[x, a]
        currtable ← currtable & mask

Method filterDomains()            // Same filtering as CT (Algo. 1), Invariant 7.2
    foreach variable x ∈ S^sup do
        foreach value a ∈ dom^c(x) do
            intersection ← currtable & supports[x, a]
            if intersection = 0^64 then
                dom^c(x) ← dom^c(x) \ {a}
                currtable ← currtable & ~supports[x, a]

Method enforceGAC()
    updateTable()
    count ← nb1s(currtable)
    if count = 0 then                                 // Invariant 7.4
        return ⊥                                      // backtrack triggered
    filterDomains()
```

where $d^c = \max_{x \in S^{\text{sup}} \bigcup S^{\text{val}}} \{|\texttt{dom}^c(x)|\}$ is the size of the largest of the current domain of the variables unbound at last propagation and w is the number of bits into a word (i.e, for Java Long type, $w = 64$).
And the worst-case space complexity is

$$\mathcal{O}\Big(|\texttt{scp}|\ d^0 \left\lceil \frac{|T^0|}{w} \right\rceil\Big)$$

where $d^0 = \max_{x \in \texttt{scp}} \{|\texttt{dom}^0(x)|\}$ is the size of the largest initial domain and w is the number of bits into a word (i.e for Java Long type, $w = 64$).

7.2.6 Handling Full Smart Elements

Handling the binary constraint into the table leads to lots of challenges. Two tuples with a given same smart element at the same position may not lead to the same propagation on all the domains. A smart element used in two different tuples may not lead to the same propagation. A table with the scope (w, x, y, z) containing the tuples $\tau_1 = (0, *, = x, = y)$ and $\tau_2 = (1, *, = x, *)$ illustrates this issue. If 1 is removed from the domain of x, both τ_1 and τ_2, by the smart element $= x$, agrees on removing 1 from the domain of y. The domain of z is unchanged since every possible value is still possible in at least one of the tuples (in this case only allowed by τ_2). However, if now 1 is removed from the domain of w, then 1 is not possible anymore for z as this was the only tuple allowing this value anymore. This example shows how treating the identical binary element the same way could endanger the GAC property of the algorithm.

Using a mapping together with the introduction of some additional variables, a smart table can be transformed into a basic smart table. The transformation consists of first adding a new variable for each pair of interacting variables. The value of this variable will be the difference between their two values. This is achieved using a simple mathematical constraint. In our example two new variables should be added: $aux_{(x,y)}$ for the pair (x, y) and $aux_{(y,z)}$ for the pair (y, z). And the two constraints added are $aux_{(x,y)} = X - Y$ and $aux_{(y,z)} = y - z$. Secondly, a new table is created based on the initial one and the addition of the new variable. For each tuple without any smart element, the corresponding new tuple is just the old one extended by $\langle * \rangle$ for the new inserted variables. When a smart element is present, it is replaced when a $\langle * \rangle$ and the corresponding new variable gets a different value. Figure 7.3 shows all the different cases. Figure 7.3a shows an initial smart table using all the available smart elements. The character $\_$ is used to represent any basic smart element. Figure 7.3b shows the resulting table after the mapping. The use of this basic smart constraint, in addition to the new variable $aux_{(x,y)}$ and the additional constraint $aux_{(x,y)} = x - y$ is equivalent to the initial smart table.

This transformation works even when there is a cycle between the smart elements. There is a cycle between the smart elements of a tuple if there is a cycle in the constraint network (CN) corresponding to the tuple. The vertices of the CN are the tuples' variables and there is an edge between two vertices if there is one smart element linking the two variables. When there is a cycle, some values of the additional variable may result in the intersection of two values. For example, given the tuple $(\leq y, \neq x)$, the first smart element would associate the value ≤ 0 to $aux_{(x,y)}$ while the second would associate the value $\neq 0$. The value used in the mapping should be the intersection between the two, i.e. < 0.

As $\texttt{CT}^{bs}$ is GAC, the propagation over this mapped table is GAC. If the auxillary constraints ara also GAC, the combination of the mapped table and the additional constraints together are able to achieve GAC when a stable state is reached (after potentially several iterations switching between the constraints).

Complexity. The table resulting of the mapping still has t tuples but may have in the worst case $r + \frac{r(r-1)}{2}$ variables, with t the number of tuples contained in the initial smart table and r its arity. The worst-case time complexity of the propagation of $\texttt{CT}^{bs}$ using the metric of the initial

	x	y
τ_1	_	$= x$
τ_2	_	$= x + v$
τ_3	_	$\neq x + v$
τ_4	_	$\leq x + v$
τ_5	_	$\geq x + v$
τ_6	$= y$	_
τ_7	$= y + v$	_
τ_8	$\neq y + v$	_
τ_9	$\leq y + v$	_
τ_{10}	$\geq y + v$	_

(a) Initial smart table

	x	y	$aux_{(x,y)}$
τ_1	_	$*$	0
τ_2	_	$*$	$-v$
τ_3	_	$*$	$\neq -v$
τ_4	_	$*$	$\leq -v$
τ_5	_	$*$	$\geq -v$
τ_6	$*$	_	0
τ_7	$*$	_	v
τ_8	$*$	_	$\neq v$
τ_9	$*$	_	$\leq v$
τ_{10}	$*$	_	$\geq v$

(b) Correponding basic smart table

Figure 7.3: Illustration of mapping smart tables into basic smart tables.

table is thus

$$\mathcal{O}\Big(r^2\ d\ \Big\lceil \frac{t}{w} \Big\rceil\Big)$$

where d is the size of the largest of the current domain of the variables unbound at last propagation and w is the number of bits into a word (i.e, for Java Long type, $w = 64$). And the worst-case space complexity is

$$\mathcal{O}\Big(r^2\ d\ \Big\lceil \frac{t}{w} \Big\rceil\Big)$$

where d is the size of the largest initial domain and w is the number of bits into a word (i.e for Java Long type, $w = 64$).

7.3 Integer Intervals

One could think about an additional interval basic smart element to add compression to the tables.

An integer interval $\{a..b\}$ can be represented as the conjunction of a $\langle \geq a \rangle$ and a $\langle \leq b \rangle$. Given the current definition of the various supporting bitsets, they would be filled the following way:

- the associated bit in `supports`$[x, v]$ would be equal to 1 for each value $v \in \{a..b\}$, 0 otherwise
- the associated bit in `supports`$^*[x, v]$ would be equal to 0 for each value v
- the associated bit in `supportsMin`$[x, v]$ would be equal to 1 for each value $v \leq b$, 0 otherwise
- the associated bit in `supportsMax`$[x, v]$ would be equal to 1 for each value $v \geq a$, 0 otherwise

However this does not keep the GAC property of the algorithm in all cases. The algorithm is still GAC if the domain stays complete (all integer values from the minimum to the maximum value of the domain belongs to it) and become weaker if some values are missing. This loss of consistency level only arises if the incremental update is used.

Figure 7.4 shows an example of a case where the algorithm is not GAC. Given the table in Fig. 7.4a, if values 2, 3 and 4 are removed from the domains of x, then, tuple τ_2 should be removed from `currtable`. If the incremental update is used, the empty mask is first unified to the `supports`* of the removed values (the orange bitsets in Fig. 7.4b). As they are all empty, the `mask` is still empty after this step. Its complementary is thus a full bitset. It is then intersected with the `supportsMin` and `supportsMax` of the current minimum and maximum of the domain (the blue bitsets). The mask is full at the end and its intersection with `currtable` would not remove any tuples which contradicts with the fact that τ_2 should be removed in a GAC propagator. However, using the reset update, the `supports` of the remaining values are used (the green bitsets). The union of these let to a bitset containing only τ_1 and τ_3, which is GAC.

7.4 Enforcing Bound Consistency with CT

A simple bound(D) consistency (Def. 2.3) version of Compact-Table can be created by using only the `supportsMin` and `supportsMax` supports (as defined Def. 7.6 and Def. 7.7) during the update when the domain contains more than one value left.

This is given by Algo. 15.

7.5 Results

7.5.1 Experiments Results with CT*

The series used contains 600 randomly generated instances, each with 20 variables whose domain sizes range from 5 to 7, and 40 random

	x	y
τ_1	1	1
τ_2	$\{2..4\}$	1
τ_3	5	1

(a) Table

	supports			supports*			supportsMin			supportsMax		
	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3
$(x, 1)$	1	0	0	1	0	0	1	1	1	1	0	0
$(x, 2)$	0	1	0	0	0	0	0	1	1	1	1	0
$(x, 3)$	0	1	0	0	0	0	0	1	1	1	1	0
$(x, 4)$	0	1	0	0	0	0	0	1	1	1	1	0
$(x, 5)$	0	0	1	0	0	1	0	0	1	1	1	1

(b) Supports

Figure 7.4: Illustration on how to handle intervalls in tables.

positive short table constraints of arities 6 or 7, each table having a tightness (proportion of tuples from the universe table which are present) comprised between 0.5% and 2% and a proportion of short tuples equal to 1%, 5%, 10% and 20%.

Figure 7.5 shows the results obtained on these positive short tables, mainly comparing `CT`* and `ShortSTR2` [JN13]. Clearly, `CT`* outperforms `ShortSTR2` that is at least 7 times slower than `CT`* for 50% of the instances. We have also tested `CT` and `STR2` [Lec11] on these instances after converting short tables into ordinary tuples. Here, we can can observe that `CT`* is twice faster than `CT` on 20% of the instances, while saving memory space.

One can also notice that `CT` is slightly better than `CT`* on 12% of the instances. This can arise with low compression instances. In such cases, the compression does not reduce much the number of words in the bitsets. But, the repartition of the tuples among the words may differ, leading to a different decrease of the active words set in the bitsets, leading to a slight difference in the performances.

7.5.2 Experiments Results with $\texttt{CT}^{bs}$

To assess the efficiency of $\texttt{CT}^{bs}$, notably the interest of using the different forms of expressions, tables have been compressed using our algorithm in three different related ways: (1) compression with $\langle \leq v\rangle$ and $\langle \geq v\rangle$, (2) compression with $\langle \leq v\rangle$ and $\langle \geq v\rangle$ followed by a post-processing to detect $\langle * \rangle$ and $\langle \neq v\rangle$ and (3) compression with $\langle \leq v\rangle$ and $\langle \geq v\rangle$ followed by a transformation into set restrictions (e.g., $\leq v$ is written

Algorithm 15: Bound consistent version for `CT`

```
Method updateTable()
    foreach variable x ∈ S^val do
        if |dom(x)| == 1 then
            currtable ← currtable & supports[x, x.value]
        else
            mask ← ~ 0^64
            if dom (x).minChanged() then
                mask ← mask & supportsMin[x, dom(x).min]
            if dom (x).maxChanged() then
                mask ← mask & supportsMax[x, dom(x).max]
            currtable ← currtable & mask
```

as $\{i : i \leq v\}$). The initial set of instance consists of the instances containing only table constraints in the XCSP3 repository [BLP16]. The compression algorithm used on the initial set is the greedy compression decribed in Sec. 5.4.2.2.

Figure 7.6 shows the performance profile [DM02b] for $\texttt{CT}^{bs}$ with these three related compression approaches and also for standard CT on uncompressed tables. A point (x, y) on the plot indicates the percentage of instances that can be solved within a time-limit that is at most x times the time taken by the best algorithm. The performance profile was based only on instances showing enough compression (rate ≤ 0.9) and requiring at least 2 seconds of solving time. With a timeout set to 10 min, only 60 instances matched out these criteria out of the $4,000$ tested instances.

Obtained results show that simple compression (1) brings a slight speedup compared to CT. Notice, however, that the computation time for an instance was reduced up to a factor of 7. Because post-processing (2) brought less than 3% of additional compression, it is not surprising that $\texttt{CT}^{bs}$ with approaches (1) and (2) are close.

As expected, handling tables with set restrictions only, approach (3), induces an overhead as no incremental updates can be performed. The overhead is however limited (at most a factor two). The computation time taken by Method `updateDomain()` in Algo. 1 is not much reduced when using basic smart tables (mainly, because of the residue caching described in [DHL+16]). This explains why the observed speed-ups are

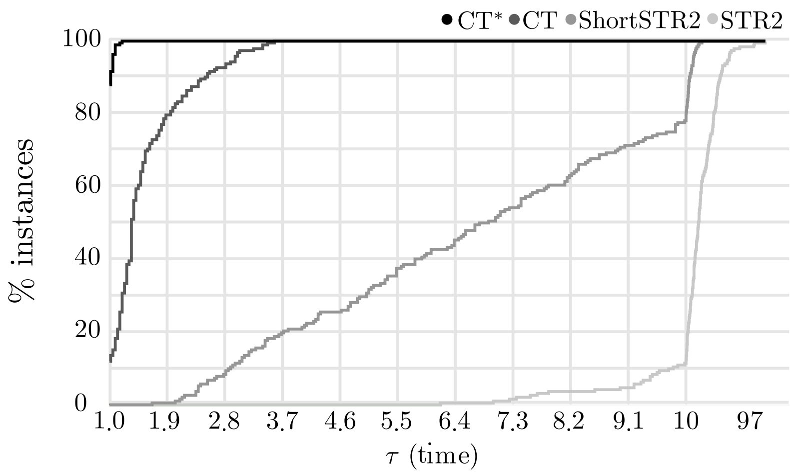


Figure 7.5: Performance profile comparing $\texttt{CT}^*$ with algorithms `CT`, `STR2` and `ShortSTR2`.

not proportional to the compression ratios.

7.6 Conclusion

This chapter presented the extension of CT to handle short, basic smart, and smart tables. The resulting algorithms are CT* and $\texttt{CT}^{bs}$. Their structure is very similar to the one of CT, using an update and a filtering phase. In addition, several new bitsets have been introduced to improve with the update phase.

Efficient handling of such compressed tables has several benefits. In some problems where it is relevant to generate tables, such as using a technique such as auto-tabling [DBC+17], users could now generate compressed tables directly. Compression can also be applied to store tables in order to reduce their size. The handling of such tables also helps handling bigger equivalent ground tables.

The CT* extension was published as part of the [VLS17a] paper. The $\texttt{CT}^{bs}$ extension was published as part of the [VLDS17] paper.

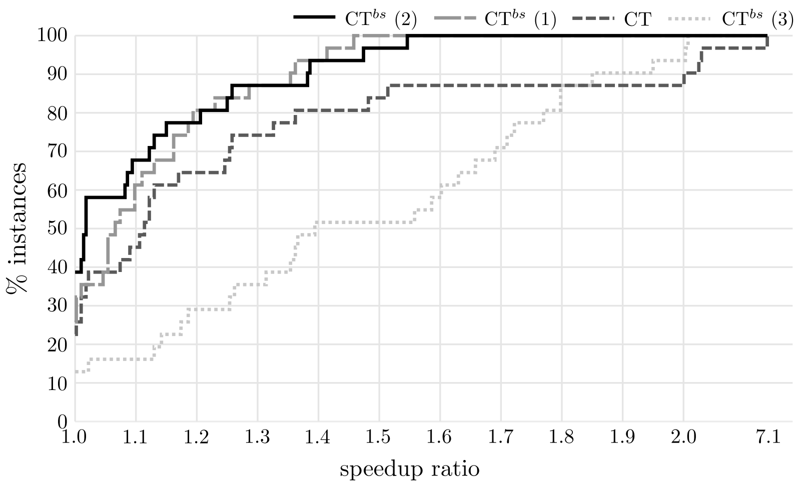


Figure 7.6: Performance profile comparing $\texttt{CT}^{bs}$ (on the same benchmark with different level of compression) with algorithm CT.

Chapter 8

Filtering Negative Smart Table Constraints

I think of feedback as constructive, not positive or negative. You choose to do what you want with it.

- Denise Morrison

8.1 Introduction

This chapter deals with the propagation of negative tables (Sec. 5.2.1). A negative table constraint takes as input a negative table T of arity r (called the initial table T^0) and a sequence X of r variables (called the scope $\texttt{scp}$). Each tuple of the table corresponds to an forbidden instantiation of the variables (i.e. the constraint should fail if $\exists \tau \in T, \forall x \in \texttt{scp}, x = \tau[x]$).

The propagation of negative tables follows the invariants inv. 8.1, which states which tuples belong to T^c and inv. 8.2, which states which values belong to $\texttt{dom}^c$. Respecting these invariant leads to a GAC propagator as stated by Prop. 8.1. Two additional invariants (inv. 8.3, which states when there is no solution, and inv. 8.4, which states when any assignement is a solution) are derived from inv. 8.2. They do not change the propagation strength. However, as checking inv. 8.3 is cheaper than checking inv. 8.2 for each variable, exploiting it may speed up the propagation. Adding inv. 8.4 may help detect earlier if the constraint may be deactivated (always valid).

Invariant 8.1 (Current table update). *Given the notations: T^0, the initial table (i.e. before any propagation occurs), T^c, the reduced table at a given current state c of the propagation, and, $\texttt{dom}^c(x)$, the domain* *of x at the current state c. A ground tuple τ belongs to the current table T^c if and only if it was a ground tuple of the initial table and all its values still belongs to the respective current domains of the associated variables from* `scp`*.*

$$\Big(\tau \in T^0 \wedge \forall x \in \texttt{scp},\ \tau[x] \in \texttt{dom}^c(x)\Big) \Leftrightarrow \Big(\tau \in T^c\Big)$$

Remark: This invariant is the same as inv. 7.1 for the positive table propagator.

Invariant 8.2 (**Domain filtering**)**.** *Given any variable $x \in \texttt{scp}$, a value v is in $\texttt{dom}^c(x)$ if there is a valid tuple satisfing $\tau[x] = v$ that does not belong to T^c.*

$$\forall x \in \texttt{scp},\ \forall v \in \texttt{dom}^c(x),\ \big|\{\tau \in T^c :\ \tau[x] = v\}\big| < \prod_{y \in \texttt{scp}: y \neq x} \big|\texttt{dom}^c(y)\big|$$

Proposition 8.1. *A negative table constraint enforces GAC if inv. 8.1 and inv. 8.2 hold.*

Proof. By means of inv. 8.1, the set of conflicting tuples is maintained. Invariant 8.2 detects when a given value (x, a) can be removed. □

Invariant 8.3 (**Entailement**)**.** *A negative table containing all current valid tuples does not have a solution.*

$$\Big(\big|\{\tau : \tau \in T^c\}\big| = \prod_{x \in \texttt{scp}} \big|\texttt{dom}(x)\big|\Big) \Leftrightarrow \bot$$

Remark: This invariant has the opposite effect as inv. 7.3 for the positive table propagator.

Invariant 8.4 (**Emptiness**)**.** *An empty negative table allows any possible instantiation.*

$$\Big(T^c = \emptyset\Big) \Leftrightarrow \top$$

Remark: This invariant has the opposite effect as inv. 7.4 for the positive table propagator.

In the following sections, the $\texttt{CT}_{neg}$ propagator, `Compact-Table` for negative table, is explained. Then, a first extension to $\texttt{CT}_{neg}$, called $\texttt{CT}^*_{neg}$, dealing with negative short tables, is presented. The chapter also emphasizes some difficulties behind the propagation of compressed negative tables using bitsets.

8.2 $\texttt{CT}_{neg}$: $\texttt{CT}$ for Negative Tables

The $\texttt{CT}_{neg}$ algorithm is an adaptation of the $\texttt{CT}$ algorithm to negative tables. The propagator is similar to the one of $\texttt{CT}$, i.e. there is an update phase followed by a filtering phase. The data structures used and their computations are also the same. However, as the context is different, as the tuples are *conflicts* (i.e. forbidden instantiations).

The pseudo-code of $\texttt{CT}_{neg}$ is given by Algo. 16. It requires the function `nb1s()` (Algo. 2) which allows to count the total number of bits set to 1 in a bitset by executing an optimized bitwise statement such as `java.lang.Long.bitCount` [War13].

The following subsection explains how the update is performed. Then, the filtering process is described. Finally, the complete algorithm is detailed.

8.2.1 The Update Phase

The update phase (Algo. 16 line 1) is exactly the same as for $\texttt{CT}$. This is a direct consequence of sharing the same invariant about the update of the current table (inv. 8.1) and the same data structures. More details about the code and the complexity can be found in Sec. 3.2.10.2.

8.2.2 The Filtering Phase

The filtering phase (Algo. 16 line 12), however, differs from $\texttt{CT}$.

When filtering, we try each of the values v from the domains of the unbound variables x from $\texttt{scp}$. The goal is to identify those that can lead to inconsistencies. To do so, the filtering invariant (rule 8.2) is used.

The idea is to count, for each pair (x, v), with $x \in \texttt{S}^{\texttt{sup}}$ and $v \in \texttt{dom}(x)$, how many valid tuples satisfying $\tau[x] = v$ are in the table T^c. This count is then compared to the total number of valid tuples satisfying $\tau[x] = v$. When these two numbers are equal, by inv. 8.2, v may be removed from $\texttt{dom}(x)$.

Optimization of the computation of the threshold (Algo. 16 line 16). By definition of $\texttt{S}^{\texttt{sup}}$, we know that only the variables within $\texttt{S}^{\texttt{sup}}$ have $|\texttt{dom}| > 1$, therefore

$$\prod_{y \in \texttt{scp}: y \neq x} |\texttt{dom}^c(y)| = \prod_{y \in \texttt{S}^{\texttt{sup}}: y \neq x} |\texttt{dom}^c(y)|$$

Optimization of the update of `currtable` within the filtering (Algo. 16 line 19). Updating `currtable` at each modification implies two things. First, as it is a reversible structure, this operation may take more time to check if a partial save should be done. Second, com-

Algorithm 16: $\mathtt{CT}_{neg}$

1 **Method** **`updateTable()`** `// Same update as CT (Algo. 1), inv. 8.1`
2   **foreach** *variable* $x \in S^{val}$ **do**
3   `mask` $\leftarrow 0^{64}$
4   **if** $|\Delta_x| < |\mathit{dom}^c(x)|$ **then** `// Classical update`
5   **foreach** *value* $a \in \Delta_x$ **do**
6   `mask` $\leftarrow$ `mask` $|$ `supports`$[x, a]$
7   `mask` $\leftarrow \sim$ `mask`
8   **else** `// Reset update`
9   **foreach** *value* $a \in \mathit{dom}^c(x)$ **do**
10   `mask` $\leftarrow$ `mask` $|$ `supports`$[x, a]$
11   `currtable` $\leftarrow$ `currtable` $\&$ `mask`

12 **Method** **`filterDomains()`**
13   **foreach** *variable* $x \in S^{sup}$ **do**
14   **foreach** *value* $a \in \mathit{dom}^c(x)$ **do**
15   `intersection` $\leftarrow$ `currtable` $\&$ `supports`$[x, a]$
16   `threshold` $\leftarrow \prod_{y \in \mathtt{scp}: y \neq x} |\mathtt{dom}^c(y)|$
17   **if** *nb1s(intersection)* $=$ *threshold* **then** `// Invariant 8.2`
18   `dom`$^c(x) \leftarrow$ `dom`$^c(x) \setminus \{a\}$
19   `currtable` $\leftarrow$ `currtable` $\&$ $\sim$`supports`$[x, a]$

20 **Method** **`enforceGAC()`**
21   updateTable()
22   count $\leftarrow$ nb1s(currtable)
23   **if** *count* $= 0$ **then** `// Invariant 8.4`
24   **return** $\top$ `// desactivation of the cst`
25   **if** *count* $= \prod_{x \in scp} |\mathit{dom}(x)|$ **then** `// Invariant 8.3`
26   **return** $\perp$ `// backtrack triggered`
27   filterDomains()

puting the threshold must be done for each variable. To avoid this, we must compute this product initially. The threshold is thus obtained by dividing it by the current domain size. Removed tuples are collected in `mask` and `currtable` is modified only once at the end.

These two optimisations lead to the second version of the `filterDomains`() method Algo. 17.

Complexity. The worst-case time complexity of the filtering phase of $\mathtt{CT}_{neg}$ is

$$\mathcal{O}\Big(\Big\lceil\frac{|T^0|}{w}\Big\rceil k \sum_{x\in \mathtt{S^{sup}}} \big|\mathtt{dom}^c(x)\big|\Big)$$

where w is the number of bits into a word (i.e, for java Long type, $w = 64$) and k is the complexity of the bitcount operation used in the `nb1s` method (Algo. 2 line 4). $k = \log(w)$ when using `Long.bitCount` (in Java) or can even be $k = 1$ on some architectures. The worst-case space complexity of the filtering phase of $\mathtt{CT}_{neg}$ is

$$\mathcal{O}(1)$$

as it uses only a fixed number of temporary variables and preallocated variables.

Algorithm 17: Optimized version of the filtering phase of $\mathtt{CT}_{neg}$

Method **`filterDomains()`**
- $\mathtt{initthreshold} \leftarrow \prod_{y\in \mathtt{S^{sup}}} \big|\mathtt{dom}^c(y)\big|$
- $\mathtt{mask} \leftarrow 0^{64}$
- **foreach** *variable* $x \in S^{sup}$ **do**
  - **foreach** *value* $a \in \mathtt{dom}^c(x)$ **do**
    - $\mathtt{intersection} \leftarrow \mathtt{currtable}\ \&\ \mathtt{supports}[x, a]$
    - $\mathtt{threshold} \leftarrow \frac{\mathtt{initthreshold}}{\left|\mathtt{dom}^c(x)\right|}$
    - **if** *nb1s(intersection) = threshold* **then**
      - // Invariant 8.2
      - $\mathtt{dom}^c(x) \leftarrow \mathtt{dom}^c(x) \setminus \{a\}$
      - $\mathtt{mask} \leftarrow \mathtt{mask}\ |\ \mathtt{supports}[x, a]$
- $\mathtt{currtable} \leftarrow \mathtt{currtable}\ \&\ {\sim}\mathtt{mask}$

8.2.3 GAC and Complexity

`enforceGAC`() (Algo. 16 line 20) is the entry point of the propagator. It first updates the table (using inv. 8.1), then it tests emptiness (inv. 8.4) and entailment (inv. 8.3) and finally filters the arc-unconsistent values from all domains (using inv. 8.2).

Proposition 8.2. *Algorithm 16 with the optimized filtering phase of Algo. 17, applied to a negative table constraint enforces GAC.*

Proof. By means of Method `updateTable`() and statement at Algo. 16 line 19, we maintain the set of conflicts in `currtable`. This respects inv. 8.1. At line 17, we can detect if no more support exists for a given value (x, a), and delete it if necessary. This respects inv. 8.2. By Prop. 8.1, the algorithm is GAC. □

Complexity. The worst-case time complexity is

$$\mathcal{O}\Big(\underbrace{\left\lceil\frac{|T^0|}{w}\right\rceil \sum_{x\in \mathtt{S}^{\mathtt{val}}} \min\big(|\Delta_x|, |\mathtt{dom}^c(x)|\big)}_{update} + \underbrace{\left\lceil\frac{|T^0|}{w}\right\rceil k}_{inv.\ 8.3\ \&\ inv.\ 8.4} + \underbrace{\left\lceil\frac{|T^0|}{w}\right\rceil k \sum_{x\in \mathtt{S}^{\mathtt{sup}}} \big|\mathtt{dom}^c(x)\big|}_{filtering}\Big)$$

Since $|\mathtt{scp}| \geq |\mathtt{S}^{\mathtt{sup}}|$ and $|\mathtt{scp}| \geq |\mathtt{S}^{\mathtt{val}}|$, this can be globally reduced to

$$\mathcal{O}\Big(|\mathtt{scp}|\ d\ \left\lceil\frac{|T^0|}{w}\right\rceil k\Big)$$

where d is the size of the largest of the current domains, w is the number of bits into a word (i.e, for Java Long type, $w = 64$) and k is the complexity of the bitcount operation used in the `nb1s` method (i.e. for Java API method `java.lang.Long.bitCount`, $k = \log(w)$). This corresponds to the complexity of `CT` times k. The worst-case space complexity is

$$\mathcal{O}\Big(\left\lceil\frac{|T^0|}{w}\right\rceil \sum_{x\in \mathtt{scp}} |\mathtt{dom}^0(x)|\Big)$$

which can be simplified to

$$\mathcal{O}\Big(|\mathtt{scp}|\ d\ \left\lceil\frac{|T^0|}{w}\right\rceil\Big)$$

where d is the size of the largest of the initial domains. This corresponds to the size taken by the precomputed structures.

8.3 $\texttt{CT}^*_{neg}$: Handling Negative Short Table

First of all, let us notice that it is not trivial to apply inv. 8.2 and inv. 8.3 on any negative short table. This is due to the possibility of having the same ground tuple represented by several short tuples at the same time. In this thesis, this phenomenon will be called *overlapping.* Actually, finding a solution in a negative short table with overlapping tuples (and in extension any compressed negative table with overlapping tuples) is NP-complete (as proved in the next subsection).

This is why we focus on negative short tables without overlapping. In this case, $\texttt{CT}^*_{neg}$ can be seen as the evolution of both $\texttt{CT}_{neg}$ and $\texttt{CT}^*$. As negative short tables have the same structures as positive short tables, their update is carried out in the same way as $\texttt{CT}^*$, using the additional precomputed bitset $\texttt{supports}^*$. As the table is negative, there is a need to count the remaining tuples related to a given value for a given variable. The filtering is therefore inspired by $\texttt{CT}_{neg}$.

The following subsections first detail the demonstration of the NP-completeness of the problem with overlapping tuples. Then, the update and the filtering of the algorithm tackling the problem without overlap is explained. Finally, the complete algorithm handling negative short tables (Algo. 18 and Algo. 19) is explained.

8.3.1 NP-Completeness of the Problem with Overlapping Tuples

Proposition 8.3. *The problem of finding a solution in negative short table with overlapping is NP-Complete.*

Proof. We can show this by reducing the well-known NP-complete 3-Sat problem (determining the satisfiability of a Boolean formula in conjunctive normal form where each clause is limited to at most three literals) to a negative short table through a polynomial algorithm and by proving that verifying a solution is possible with a polynomial algorithm [Kar72].

Polynomial reduction of 3-Sat to negative short table. Let us take the following example of a 3-Sat instance:

$$(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_2 \vee x_5) \tag{8.1}$$

This instance, containing 3 clauses and involving 5 literals, is false if either one of the clauses is false.

The 3-Sat problem corresponds to a CNF (i.e. conjunctive normal form) formula. As already seen in Sec. 5.3, a CNF is equivalent to a negative Boolean table. Let us recall the transformation. The first

Algorithm 18: $\mathtt{CT}^*_{neg}$

```
Method updateTable() // Same update as CT* (Algo. 12), Invariant 8.1
    foreach variable x ∈ S^val do
        mask ← 0^64
        if |Δ_x| < |dom^c(x)| then                 // Classical update
            foreach value a ∈ Δ_x do
                mask ← mask | supports*[x, a]
            mask ← ~ mask
        else                                        // Reset update
            foreach value a ∈ dom^c(x) do
                mask ← mask | supports[x, a]
        currtable ← currtable & mask
Method filterDomains()
    initthreshold ← ∏_{y∈S^sup} |dom^c(y)|
    mask ← 0^64
    foreach variable x ∈ S^sup do
        foreach value a ∈ dom^c(x) do
            intersection ← currtable & supports[x, a]
            threshold ← initthreshold / |dom^c(x)|
            if nb1s*(intersection,weights) = threshold then
             // Invariant 8.2
                dom^c(x) ← dom^c(x) \ {a}
                mask ← mask | supports[x, a]
    currtable ← currtable & ~mask
Method enforceGAC()
    updateTable()
    count ← nb1s*(currtable,weights)
    if count = 0 then                               // Invariant 8.4
        return ⊤                      // desactivation of the cst
    if count = ∏_{x∈scp} |dom(x)| then              // Invariant 8.3
        return ⊥                            // backtrack triggered
    filterDomains()
```

clause $(x_1 \vee x_2 \vee \neg x_4)$ is false for each instantiation where x_1 and x_2 are **false** and x_4 is **true**. The instantiations allowing this to happen can be represented by the single short tuple

$$(\texttt{false}, \texttt{false}, *, \texttt{true}, *)$$

where x_1 and x_2 are **false**, x_4 is **true** and the other variables can take any values (represented by the universal value $*$).

If the same reasoning is applied to all the clauses, we obtain a short table of forbidden instantiation, i.e. a negative short table. Figure 8.1 contains the negative table corresponding to the 3-Sat instance Eq. (8.1).

The process of creating a tuple for each clause is in $\mathcal{O}(r)$, r being the number of literals (i.e. arity of the resulting table). The total complexity of the transformation of the 3-Sat into a table is thus $\mathcal{O}(rt)$, r being the number of literals (i.e. arity of the resulting table) and t being the number of clauses (i.e. the tuples in the resulting table). The reduction is polynomial.

Verification of a solution in polynomial time To verify if a tuple is a solution of a negative short table, we need to verify it is not included in any of the tuples. To do so, we need to compare each of the values of the solution to the values of the tuples, this is done in $\mathcal{O}(r)$, with r the arity of the table. And as we have to do it for possibly all tuples, the

Algorithm 19: The `nb1s`* method

```
Method nb1s*(bs:Bitset)
    count ← 0
    foreach i ∈ 1..bs.length do
        count ← count + nbSubsumedTuples(i) ×
         java.lang.Long.bitCount(bs.words[i])
    return count
```

	x_1	x_2	x_3	x_4	x_5
τ_1	$false$	$false$	$*$	$true$	$*$
τ_2	$*$	$false$	$true$	$*$	$true$
τ_3	$true$	$true$	$*$	$*$	$false$

Figure 8.1: The negative short table corresponding the the 3-Sat problem Eq. (8.1).

total complexity is $\mathcal{O}(rt)$, r being the arity and t being the number of tuples. The verification is polynomial. $\square$

8.3.2 The Update Phase

The update phase (Algo. 18 line 1) is exactly the same as for $\texttt{CT}^*$. This is a direct consequence of sharing the exact same invariant about the update of the current table (inv. 8.1). More details about the code and the complexity can be found in Sec. 7.2.1.

8.3.3 The Filtering Phase

As in $\texttt{CT}_{neg}$, the idea of the filtering phase (Algo. 18 line 12) requires to count, for each pair (x, v), with $x \in \texttt{S}^{\texttt{sup}}$ and $v \in \texttt{dom}(x)$, how many ground tuples satisfying $\tau[x] = v$ are represented by the table T^c. This count determines whether we are to keep or remove v from $\texttt{dom}(x)$.

Unfortunately, here, each tuple may represent several ground tuples simultaneously. However, counting the number of ground tuples does not require to decompose the tuple. For each tuple, the number of ground tuples it represents depends on the number of universal value used and where there are used. The number of ground tuples represented by a short tuple is the cardinal product of the size of the domains for which the universal values are used. For example, in the table in Fig. 8.2a, τ_1 represents $|\texttt{dom}(y)|$ ground tuples and τ_4 represents $|\texttt{dom}(x)||\texttt{dom}(y)|$. One can notice that two tuples with the same number of $*$ at the same positions represent the same number of ground tuples. We call them $*$-*similar* (Def. 8.4).

Definition 8.4. $*$-*similarity*
Two (ordinary or short) tuples are $*$-similar *iff they contain the same number of* $*$ *and at the same positions.*

For each tuple, the count of the ground tuples satisfying $\tau[x] = v$ is the cardinal product of the size of the domains of the variables wich are not x and which for the universal values are used. If the condition $\tau[x] = v$ or $\tau[x] = *$ is not met, the number of ground tuples represented is 0. For example, in Fig. 8.2a, if (x, c) is the pair of interest (i.e. we want to count how many ground tuples satisfy $\tau[x] = c$), τ_1 in the table in Fig. 8.2a represents $|\texttt{dom}(y)|$ ground tuples, τ_4 represents $\frac{|\texttt{dom}(x)||\texttt{dom}(y)|}{|\texttt{dom}(x)|} = |\texttt{dom}(y)|$ and τ_2 represents 0.

One difficulty is to count (efficiently) the number of tuples subsumed by short tuples. In order to speedup the counting operation, the idea is to group the tuples such that each computer word of the current table only refers to $*$-similar tuples. To make things clear, let us consider the negative short table depicted in Fig. 8.2a. It contains 5 tuples, and one can observe the $*$-similarity of τ_2 with τ_3 (since they are both ordinary tuples), and of τ_1 with τ_5. We then split this table of 5 tuples into three groups. Importantly, in order to have only $*$-similar tuples in each computer word (important property for counting, as seen later), we propose a very simple procedure that consists in padding entries for each incomplete word with dummy tuples (i.e. tuples only containing a special value $\perp$ that is not present in the initial domains of the variables) until the word is complete. Assuming computer 4-bits words, on our example, we obtain 3 words as shown in Fig. 8.2b. The restructured bitset `currtable` is shown in Fig. 8.2c; note the presence of bits initially set to 0 to discard dummy tuples. Of course, we need to take dummy tuples into account when building the `supports` and `supports`* structures in order to keep all bitwise operation sound.

Once the bitset `currtable` has been restructured, counting can be advantageously achieved for a given computer word in conjunction with bitwise operations. Indeed, the number of ground tuples subsumed by any short tuple referred to in a given word of `currtable` is necessarily the same due to the $*$-similarities. For example, assuming that $\texttt{dom}(y) = \{a, b, c\}$, τ_1 and τ_5, referred to in the second word of `currtable`, subsume exactly 3 ordinary tuples each. For simplicity, in what follows, we consider that $\texttt{nbSubsumedTuples}(i)$ indicates the number of ordinary tuples subsumed by any (short) tuple referred to in the ith word of `currtable`. On our example, $\texttt{nbSubsumedTuples}(2)$ returns 3. With this auxiliary function, which can benefit from a cache in practice, counting is now performed by Function `nb1s`* (Algo. 19).

Complexity. The worst-case time complexity of the filtering phase of $\texttt{CT}^*_{neg}$ is

$$\mathcal{O}\Big(\Big\lceil\frac{|T^0|}{w}\Big\rceil k \sum_{x \in \mathsf{S}^{\mathsf{sup}}} \big|\texttt{dom}^c(x)\big|\Big)$$

where w is the number of bits into a word (i.e, for java Long type, $w = 64$) and k is the complexity of the bitcount operation used in the `nb1s`* method. $k = \log(w)$ when using `Long.bitCount` (in Java) or can even be $k = 1$ on some architectures. The worst-case space complexity of the filtering is

$$\mathcal{O}(1)$$

as it uses only a fixed number of temporary variable and preallocated variables.

8.3.4 GAC and Complexity

`enforceGAC()` (Algo. 18 line 23) is the entry point of the propagator. It first updates the table (inv. 8.1), then tests the entailment (inv. 8.3) and the emptiness (inv. 8.4) property and finally filters the values from the domains (inv. 8.2).

Proposition 8.5. *Algorithm 18, applied to a negative short table (without overlaps) constraint c enforces GAC.*

Proof. Using `supports`* to update `currtable` allows the algorithm to respect inv. 8.1. inv. 8.2 is respected by the filtering. By Prop. 8.1, the algorithm is GAC. □

	x	y	z
τ_1	c	$*$	a
τ_2	a	b	c
τ_3	b	c	b
τ_4	$*$	$*$	d
τ_5	b	$*$	a

(a) A negative short table

	x	y	z
τ_2	a	b	c
τ_3	b	c	b
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
τ_1	c	$*$	a
τ_5	b	$*$	a
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
τ_4	$*$	$*$	d
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$

(b) Tuples of the table grouped by $*$-similarity on 4-bit words

τ_2	τ_3			τ_1	τ_5			τ_4			
1	1	0	0	1	1	0	0	1	0	0	0

(c) Restructured Bitset `currTable`

Figure 8.2: Restructuration of negative short tables to a negative short table.

Complexity. The worst-case time complexity is

$$\mathcal{O}\Big(\underbrace{\left\lceil\frac{|T^{0\prime}|}{w}\right\rceil \sum_{x\in \mathtt{S}^{\mathtt{val}}} \min\left(|\Delta_x|, |\mathtt{dom}^c(x)|\right) +}_{update} \underbrace{\left\lceil\frac{|T^{0\prime}|}{w}\right\rceil k}_{inv.\ 8.3\ \&\ inv.\ 8.4} + \underbrace{\left\lceil\frac{|T^{0\prime}|}{w}\right\rceil k \sum_{x\in \mathtt{S}^{\mathtt{sup}}} \left|\mathtt{dom}^c(x)\right|}_{filtering}\Big)$$

where $T^{0\prime}$ is T^0 with the dummy tuples added. Since $|\mathtt{scp}| \geq |\mathtt{S}^{\mathtt{sup}}|$ and $|\mathtt{scp}| \geq |\mathtt{S}^{\mathtt{val}}|$ this can be globally reduced to

$$\mathcal{O}\Big(|\mathtt{scp}|\ d\ \left\lceil\frac{|T^{0\prime}|}{w}\right\rceil k\Big)$$

where d is the size of the largest of the current domains, w is the number of bits into a word (i.e, for Java Long type, $w = 64$) and k is the complexity of the bitcount operation used in the `nb1s` method (i.e. for Java API method `java.lang.Long.bitCount`, $k = \log(w)$). This corresponds to the complexity of `CT` times k. The worst-case space complexity is

$$\mathcal{O}\Big(\left\lceil\frac{|T^{0\prime}|}{w}\right\rceil \sum_{x\in \mathtt{scp}} |\mathtt{dom}^0(x)|\Big)$$

which can be simplified to

$$\mathcal{O}\Big(|\mathtt{scp}|\ d\ \left\lceil\frac{|T^{0\prime}|}{w}\right\rceil\Big)$$

where d is the size of the largest of the initial domains. This corresponds to the size taken by the precomputed structures.

8.4 Results

The series we used contains 600 instances, each with 20 variables whose domain sizes range from 5 to 7, and 40 random negative short table constraints of arities 6 or 7, each table having a tightness comprised between 0.5% and 2% and a proportion of short tuples equal to 1%, 5%, 10% and 20%. Figure 8.3 shows that $\mathtt{CT}_{neg}$ and $\mathtt{CT}^*_{neg}$ are slightly outperformed (at most 1.4 and 1.6 times slower, respectively) by `STRNe` [LLGL13], which is an adaptation of `STR2` for negative tables; for $\mathtt{CT}_{neg}$ and `STRNe`, note that short tables had to be converted into ordinary tables.

The second series we used does not involve short tables and contains 45 (more difficult) instances, each with 10 variables whose domain size is 5, and 40 random negative table constraints of arity 6, each table having a tightness of 10%, 20%, ..., 90%. Figure 8.4 shows the results we obtained with $\texttt{CT}_{neg}$ and STRNe. We also plot the curve for $\texttt{CT}^*_{neg}$ even if only ordinary tables are present, so as to observe the overhead introduced by the handling of the *-similarity groups. Clearly, $\texttt{CT}_{neg}$ outperforms STRNe that requires at least 3 times more time for around

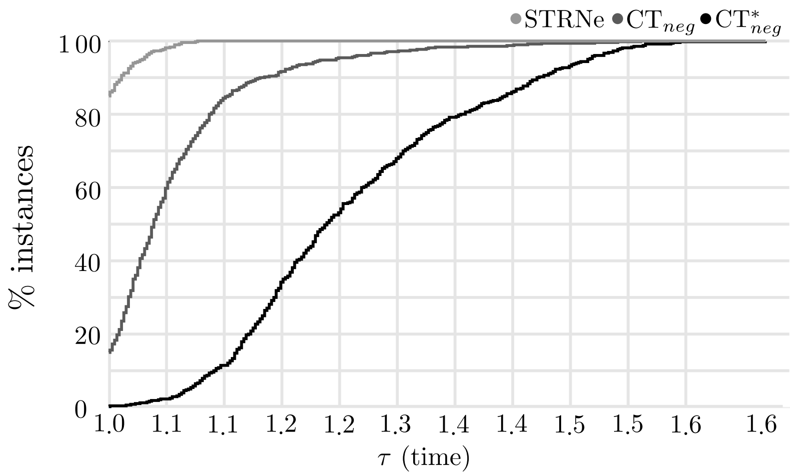


Figure 8.3: Results on Negative Short Tables – Small Domains.

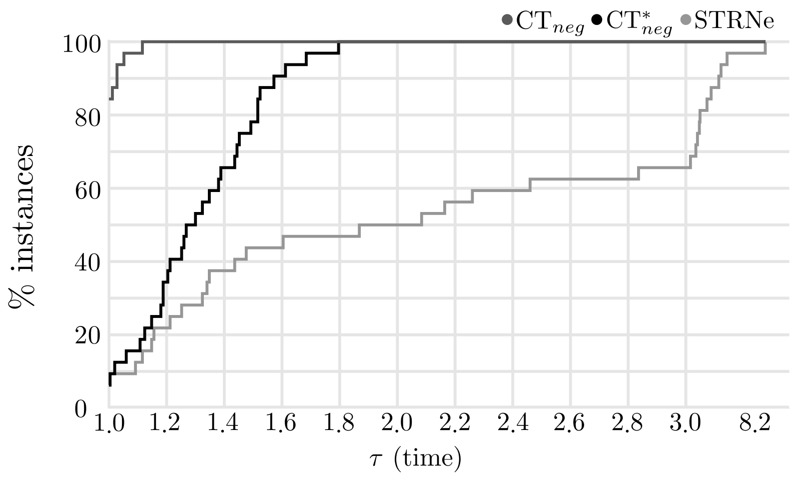


Figure 8.4: Results on Negative Tables.

half of the hardest instances. Unlike the previous series that only contains satisfiable instances, about half of the instances of this series are unsatisfiable, making $\texttt{CT}_{neg}$ more suitable in general when the outcome of the problem is not known in advance.

The third series contains 100 instances, each with 3 variables whose domain size is 100, and 40 random negative short table constraints of arity 3, each table having a tightness ranging from 0.5% to 2% and a proportion of short tuples equal to 5%, 10% and 20% (with no overlapping between short tuples). Here, we want to emphasize that $\texttt{CT}^*_{neg}$ can be very efficient, compared to STRNe, when the domain sizes and the number of short tuples are very large. This is visible in Fig. 8.5. Roughly speaking, $\texttt{CT}^*_{neg}$ is about 10 times speedier on average.

8.5 Conclusion

In this chapter, we presented $\texttt{CT}_{neg}$ and $\texttt{CT}^*_{neg}$, two extensions of CT. These are two propagators able to propagate a negative table (i.e. conflict table) representing the forbidden instantiations.

The adaptations required for $\texttt{CT}_{neg}$ are pretty straightforward. However, handling negative short table raises issues due to the NP-complete nature of the problem. Polynomial handling of negative short tables is only reachable with a non-overlapping hypothesis on the table's structure. This led to the design of the $\texttt{CT}^*_{neg}$ algorithm.

Handling negative (basic) smart tables using bitwise operation would

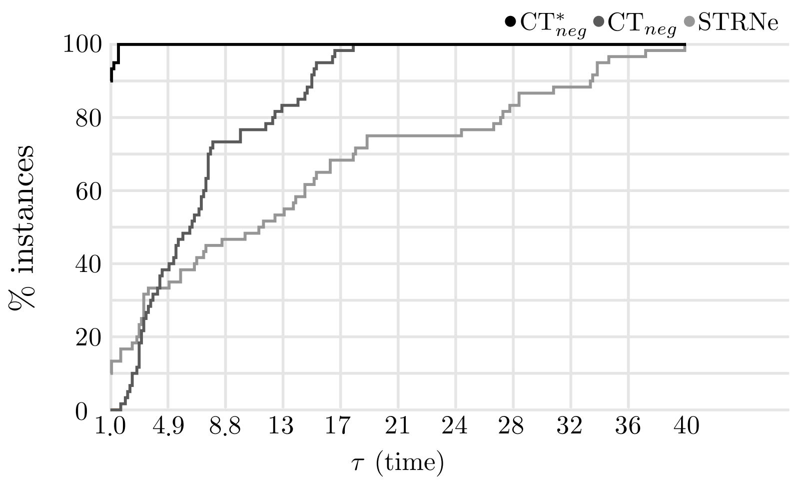


Figure 8.5: Results on Negative Short Tables – Large Domains.

be even more challenging. First, the hypothesis of non-overlapping would still be in application. Secondly, it would still be necessary to compute the number of ground tuples represented by each of the compressed tuples. To use bitsets efficiently, this would require the words to contain, again, only similar tuples. The use of any (basic) smart elements increases the number of combinations possible of these elements. This increases significantly the number of additional words required to construct a bitset with only similar tuples by words. This may lead to a situation where $\frac{|T^{0\prime}|}{w}$ approaches the value of $|T^0|$, removing any benefits of using bitsets.

The $\texttt{CT}_{neg}$ and $\texttt{CT}^*_{neg}$ extensions were published as part of the [VLS17a] paper.

CHAPTER 9

Filtering Basic Smart Diagram Constraints

There's a difference between being ignorant and being stupid... For me, an ignorant person is someone who makes the wrong decision or a bad choice because he or she does not have the proper facts. If you give that person the facts and the proper information you have alleviated that ignorance, and they make the right decision.

- Daryl Davis

9.1 Introduction

This chapter describes some algorithms based on bitwise operation for a diagram-based version of the extensional constraint type. The diagram we focus on is the Multi-Valued Variable Diagram (`MVD`). The `MVD` constraint takes as input an `MVD` (Ω,Θ) of arity r (called the initial diagram (Ω^0, Θ^0)) and a sequence X of r variables (called the scope `scp`)[1]. Each path of the `MVD` corresponds to an instantiation of the variables.

Due to strong links existing between `MVD`*s* and tables, one can say that the propagation of `MVD`*s* follows the invariants inv. 9.1, which states which paths (and thus which edges) belong to (Ω^c, Θ^c), and inv. 9.2, which states which values belong to the current domain $\texttt{dom}^c$. These invariants are directly derived from the ones underlying the propagation of `CT`. As a result, respecting these invariants leads to a GAC propagator

[1]The domain of the `MVD` used should be included in the sequence of the domains (i.e. given $\mathbb{D} = (\mathcal{D}_1, \dots, \mathcal{D}_r)$, the domain of (Ω, Θ), $\forall i \in [1; r], \mathcal{D}_i \subseteq \texttt{dom}(X[i])$). If it is not the case initially, the input diagram should be restricted to the path valid w.r.t. the domains of the variables

as stated by Prop. 9.1. Again, two additional invariants (inv. 9.3, which states when any instantiation is a solution, and inv. 9.4, which states when there is no solution) can be added. In practice, inv. 9.3 is not tested by our algorithms as the bitwise representation makes it too difficult to count the number of paths.

Invariant 9.1 (**Current Diagram Update**). *Given the notations: (Ω^0, Θ^0), the initial diagram (i.e. before any propagation occurs), (Ω^c, Θ^c), the reduced diagram at a given current state c of the propagation, and, $\texttt{dom}^c(x)$, the domain of x at the current state c. A ground arc ζ belongs to the current diagram (Ω^c, Θ^c) if and only if it is a ground arc of the initial diagram, its label still belongs to the current domain of the associated variable from $\texttt{scp}$ and its tail and head nodes still belong to the diagram.*

$$\Big(\zeta \in (\Omega^0, \Theta^0) \wedge l(\zeta) \in \texttt{dom}^c(L_a(\zeta)) \wedge h(\zeta), t(\zeta) \in (\Omega^c, \Theta^c)\Big) \Leftrightarrow \Big(\zeta \in (\Omega^c, \Theta^c)\Big)$$

A node belongs to the current diagram (Ω^c, Θ^c) if and only if it is a node of the initial diagram and if it exists at least one arc entering and one arc exiting the node.

$$\Big(n \in (\Omega^0, \Theta^0) \wedge \exists \zeta_i, \zeta_j \in (\Omega^c, \Theta^c), h(\zeta_i) = t(\zeta_j) = n\Big) \Leftrightarrow \Big(n \in (\Omega^c, \Theta^c)\Big)$$

The combination of these two subrules defines how to update the diagram.

Invariant 9.2 (**Domain Filtering**). *Given any variable $x \in \texttt{scp}$, each value v in $\texttt{dom}^c(x)$ should appear as label of at least one of the arcs $\zeta \in \Theta^c[x]$.*

$$\forall x \in \texttt{scp},\ \forall v \in \texttt{dom}^c(x),\ \exists \zeta \in \Theta^0[x],\ l(\zeta) = v$$

Proposition 9.1. *A diagram constraint enforces GAC if and only if inv. 9.1 and inv. 9.2 hold.*

Proof. By means of inv. 9.1, the set of valid paths is maintained. Invariant 9.2 detects when a given value (x, a) can be removed. ☐

Invariant 9.3 (**Entailement**). *A diagram in entailed if and only if it contains all the paths representing all the possible combinations of the values of each domain. For diagrams with the path uniqueness property (Prop. 6.9) such as MDDs, this is verifiable using the number of paths.*

$$\Big(\big|\{\tau : path(\tau) \in (\Omega^c, \Theta^c)\}\big| = \prod_{x \in \texttt{scp}} \big|\texttt{dom}(x)\big|\Big) \Leftrightarrow \top$$

Invariant 9.4 (Emptiness). *An empty diagram does not have any solution.*

$$\Big((\Omega^c, \Theta^c) = (\emptyset, \emptyset)\Big) \Leftrightarrow \bot$$

This chapter details first `CD`, the adaptation of `CT` to diagrams. Then, its extension, named $\texttt{CD}^{bs}$, aiming at handling basic smart diagrams, is presented. Finally, results on those two algorithms are given.

9.2 `Compact-Diagram`

`Compact-Diagram` borrows some principles from both `CT` [DHL+16] and `MDD4R` [PR14]. As for the previous algorithms, the propagation of `CD` is divided into the two usual steps. First, the update phase, whose goal is to update the representation of the remaining diagram (here the bitsets `currdiagram` (Def. 9.2)). Then, the filtering phase that finds which values have to be removed from the domains of the unbound variables. The pseudo-code of the algorithm can be found in Algo. 20 and in Algo. 21.

This section first describes how to construct the bitset structures `currdiagram` and its supports, then explains the two parts of the algorithm.

9.2.1 Data Structures

As in `CT`, the current state is maintained using reversible sparse bitsets. `currdiagram`$[x]$ (Def. 9.2) represents the valid arcs of the level associated to variable x. Each of the arcs is associated to a bit from the bitset. The arc is valid iff the bit is set to 1.

Definition 9.2. ***`currdiagram` (as Used in `CD`)***
`currdiagram` is a collection of reversible sparse bitsets. Given an initial diagram (Ω^0, Θ^0), for each variable x associated to a given layer of the diagram, `currdiagram`$[x]$ associates one bit to each of the arcs of $\Theta^0[x]$ (i.e. the arc level associated to x). At a given time c, `currdiagram` represents the arcs of a given (Ω^c, Θ^c), subset of (Ω^0, Θ^0), valid regarding the values of the domains at that time. For each variables x, given any arc $\zeta \in \Theta^0[x]$,

$$\texttt{currdiagram}[x]\langle\zeta\rangle = \begin{cases} 1 & \textit{iff } \zeta \in \Theta^c[x] \\ 0 & \textit{iff } \zeta \notin \Theta^c[x] \end{cases}$$

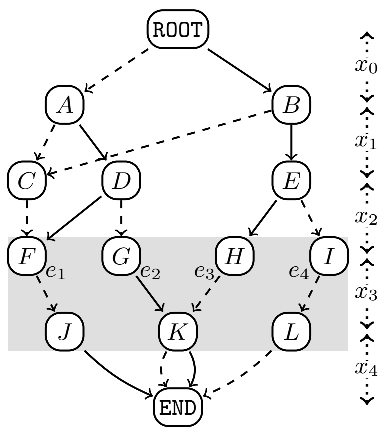
Figure 9.1 shows an example of `currdiagram`.

Algorithm 20: `Compact-Diagram` (part 1)

```
Method enforceGAC()
    S^val ← {x ∈ scp : lastSizes[x] ≠ |dom(x)|}
    S^sup ← {x ∈ scp : |dom(x)| > 1}
    updateDiagram()
    filterDomains()
    foreach variable x ∈ S^val do
        lastSizes[x] ← |dom(x)|

Method updateDiagram()
    foreach variable x ∈ scp do
        currdiagram[x].clearMask()
    updateMasks()
    propagateDown(x_1, false)
    propagateUp(x_r, false)

Method filterDomains()
    foreach variable x ∈ S^sup do
        foreach value a ∈ dom(x) do
            if currdiagram[x] & supports[x, a] = 0^64 then
                dom(x) ← dom(x) \ {a}
```



(a) Example `MDD`

	e_1	e_2	e_3	e_4
`currdiagram`$[x_3]$	1	1	1	1
`supports`$[x_3, 0]$	1	0	1	1
`supports`$[x_3, 1]$	0	1	0	0
`arcsT`$[F, x_3]$	1	0	0	0
`arcsT`$[G, x_3]$	0	1	0	0
`arcsT`$[H, x_3]$	0	0	1	0
`arcsT`$[I, x_3]$	0	0	0	1
`arcsH`$[x_3, J]$	1	0	0	0
`arcsH`$[x_3, K]$	0	1	1	0
`arcsH`$[x_3, L]$	0	0	0	1

(b) Bitsets

Figure 9.1: An example of `currdiagram`, `supports`, `arcsT` and `arcsH` for a given `MDD`.

To ease computations, at each level we find three types of pre-computed (i.e. computed at the begining) immutable bitsets. First, `supports`$[x, a]$ (Def. 9.3) indicates for each arc on the variable x whether or not the value a is initially supported by this arc. Second, `arcsH`$[x, i]$ (resp. `arcsT`$[i, x]$) (Def. 9.4) indicates for each arc on x whether node i is the head (resp. tail) of the arcs.

Algorithm 21: `Compact-Diagram` (part 2)

```
Method updateMasks()
    foreach variable x ∈ S^val do
        if |Δ_x| < |dom(x)| then                       // Incremental update
            foreach value a ∈ Δ_x do
                currdiagram[x].addToMask(supports[x, a])
        else                                           // Reset-based update
            foreach value a ∈ dom(x) do
                currdiagram[x].addToMask(supports[x, a])
            currdiagram[x].reverseMask()
Method propagateDown(x_i, localChange)
    if x_i ∈ S^val or localChange then
        currdiagram[x_i].removeMask()
        if currdiagram[x_i].isEmpty() then
            return ⊥
        if x_i ≠ x_r then
            localChange ← false
            foreach node
             ν ∈ {ν : currdiagram[x_{i+1}]&arcsT[x_{i+1}, ν] ≠ 0} do
                if currdiagram[x_i] & arcsH[x_i, ν] = 0^64 then
                    currdiagram[x_{i+1}] .addToMask(arcsT[x_{i+1}, ν])
                    localChange ← true
            propagateDown(x_{i+1}, localChange)
    else if x_i ≠ x_r then
        propagateDown(x_{i+1}, false)
Method propagateUp(x_i, localChange)
    /* Similar to propagateDown with x_1 instead of x_r,
    x_{i-1} instead of x_{i+1} and inverted use of arcsT and
    arcsH.                                                    */
```

Definition 9.3. ***supports* (as Used in CD)**
The bitset $\texttt{supports}[x, v]$ *contains the arcs of* $\Theta^0[x]$*, the level associated to* x*, supporting the value* v *from* $\texttt{dom}(x)$*.*
Given a diagram (Ω^0, Θ^0)*, given a variable* x *and given an arc* $\zeta \in \Theta^0$*,* $\forall v \in \texttt{dom}(x)$*,*

$$\texttt{supports}[x, v]\langle\zeta\rangle = \begin{cases} 1 & \textit{iff } l(\zeta) = v \\ 0 & \textit{iff } l(\zeta) = v \end{cases}$$

Figure 9.1 shows an example of ***supports****.*

Definition 9.4. ***arcsH* and *arcsT* (as Used in CD)**
The bitset $\texttt{arcsH}[x, n]$ *contains the arcs of* $\Theta^0[x]$*, the level associated to* x*, for which the node* $n \in \Omega^0$ *is the head.*
Given a diagram (Ω^0, Θ^0)*, given a variable* x *and given an arc* $\zeta \in \Theta^0$*,* $\forall n \in \Omega^0$*,*

$$\texttt{arcsH}[x, n]\langle\zeta\rangle = \begin{cases} 1 & \textit{iff } head(\zeta) = n \\ 0 & \textit{iff } head(\zeta) \neq n \end{cases}$$

The bitset $\texttt{arcsT}[n, x]$ *contains the arcs of* $\Theta^0[x]$*, the level associated to* x*, for which the node* $n \in \Omega^0$ *is the tail.*
Given a diagram (Ω^0, Θ^0)*, given a variable* x *and given an arc* $\zeta \in \Theta^0$*,* $\forall n \in \Omega^0$*,*

$$\texttt{arcsT}[n, x]\langle\zeta\rangle = \begin{cases} 1 & \textit{iff } tail(\zeta) = n \\ 0 & \textit{iff } tail(\zeta) \neq n \end{cases}$$

Figure 9.1 shows an example of ***arcsH*** *and* ***arcsT****.*

9.2.2 The Update Phase

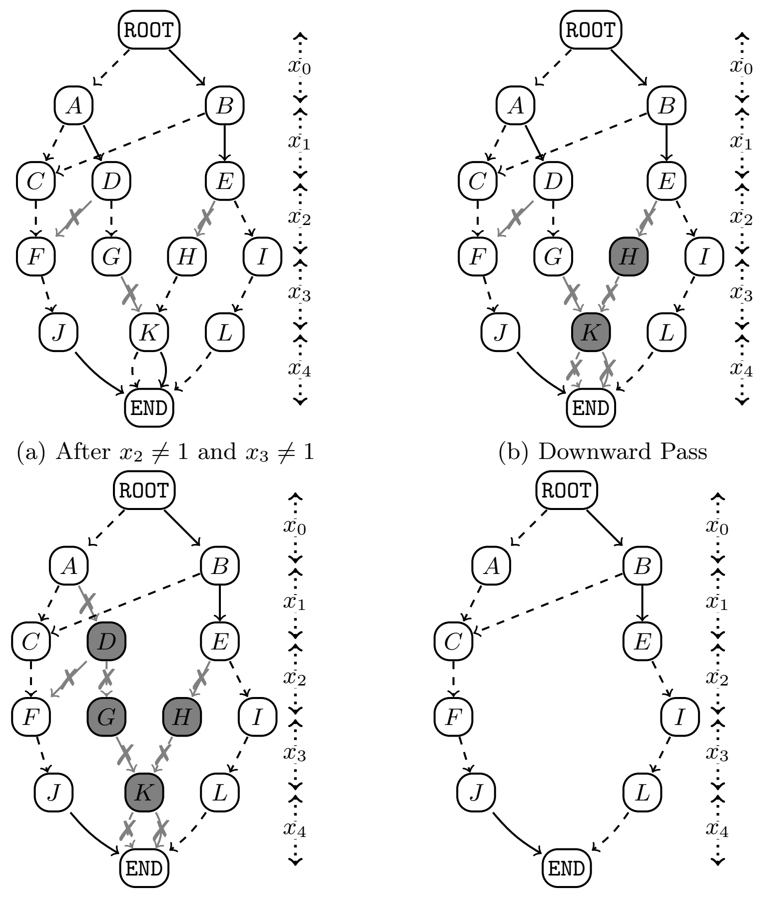
As in `MDD4R`, the goal of `updateDiagram`() (Algo. 20 line 8) is to remove the arcs that are no more part of a valid path. An arc can be:

- *directly* removed when the value of the label of the arc has been removed from the variable domain (since the previous call)

- *indirectly* removed when all paths involving the arc are no more valid.

Method `updateDiagram`() follows this observation: it identifies first the arcs that can be trivially removed before identifying those that can be untrivially removed. Figure 9.2 illustrates the whole updating process, considering the effect of having two deleted values on the `MDD` depicted in Fig. 9.1a. We shall refer to this illustration all along the description of this part of the algorithm.

In Method `updateDiagram`(), after reinitializing all masks associated with the variables in the scope of the constraint, all arcs that can be directly removed are handled by calling `updateMasks`() (Algo. 21 line 1). For each variable $x \in \mathtt{S}^{\mathtt{val}}$, `updateMasks`() operates on their associated masks. This method assumes an access to the set of values Δ_x removed from $\mathtt{dom}(x)$ since the last call to `enforceGAC`(). There are two ways of updating the masks (before updating `currdiagram` from these masks, later): either incrementally or from scratch after resetting as proposed in [PR14]. This is the strategy implemented in `updateMasks`(), by considering a reset-based computation when the size of the domain is smaller than the number of deleted values. In case of an incremental update (Algo. 21 line 3), the union of the arcs to be removed is collected by calling `addToMask`() for each structure `supports` corresponding to re-



(a) After $x_2 \neq 1$ and $x_3 \neq 1$

(b) Downward Pass

(c) Upward Pass

(d) New State of the MDD

Figure 9.2: Updating the `MDD` from Fig. 9.1a after $x_2 \neq 1 \land x_3 \neq 1$.

moved values, whereas in case of a reset-based update (Algo. 21 line 6), we perform the union of the arcs to be kept. To get masks ready to apply, we just need to reverse them when they have been built from present values. Unlike `CT`, the update of `currdiagram` from the computed masks is not done immediately. Figure 9.2a shows in gray the arcs that are added to the masks.

We need now to determine which arcs can be indirectly removed: this is achieved by calling the methods `propagateDown`() and `propagateUp`(), which, similarly to `MDD4R`, perform two passes on the diagram. During the downward (resp., upward) pass, each level is examined from the `ROOT` (resp., `END`) to the `END` (resp., `ROOT`)[2].

In Method `propagateDown`(), for a specified variable x_i, provided that some arcs on x_i have been removed (the presence of arcs directly removed are tested at line 11 of Algo. 20 with $x_i \in \mathtt{S}^{\mathtt{val}}$, and the presence of arcs indirectly removed are given by the Boolean variable `localChange`), we have to process (and propagate) them. To start, `currdiagram` is first updated (line 12 of Algo. 20), and if no more arcs on x_i remain, a backtrack is forced because there is necessarily a domain-wipe-out. If x_i is not the last variable in the scope of the constraint, we have to deal with x_{i+1}. Specifically, every node[3] ν that is the tail of a currently valid arc on x_{i+1} is tested: when there is no more valid arcs on x_i with ν as head, all arcs on x_{i+1} with ν as tail are then indirectly removed. In other words, if there is no more valid incoming arc for a node ν at level i, then all outgoing arcs of ν become invalid: this is implemented by the code at Algo. 20 line 19. Note that the search of supporting arcs is improved by the use of residues. This increases the odds of not testing too many words of `currdiagram`. Also, note how the variable `localChange` becomes true as soon as an arc is untrivially removed.

Figure 9.2b shows the behavior of downward propagation on our example. For the first two levels, nothing happens. However, at the level of x_2, we can see that all incoming arcs of the node H have been removed. Hence, the outgoing arcs of H are added to the mask associated with the next level, and removed when reaching this level. On the other hand, the node F has still one valid incoming arc. Figure 9.2c shows the result of upward propagation (after the downward one has been completed) and Fig. 9.2d shows the resulting current `MDD`.

Complexity. The worst-case time complexity of the update phase is the sum of the complexities of its two steps, i.e. the direct edge re-

[2]Actually, we can start propagation from the first and last unbound variables. For experiments, we used this code optimization.

[3]Those are maintained in practice in a reversible sparse-set as in [PR14].

moval (updateMasks) and the indirect edge removal (propagateDown and propagateUp).

The worst-case time complexity of updateMasks is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{S}^{\mathtt{val}}} \big(\min(|\Delta_x|, |\mathtt{dom}^c(x)|)\Big\lceil\frac{|\Theta^0[x]|}{w}\Big\rceil\big)\Big)$$

where w is the number of bits into a word (i.e, for java Long type, $w = 64$).

The worst-case time complexity of the propagateDown and propagateUp is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{scp}()} |\Omega^c[x]|\Big\lceil\frac{|\Theta^0[x]|}{w}\Big\rceil\Big)$$

The worst-case space complexity of the update phase is

$$\mathcal{O}(1)$$

as it uses only a fixed number of temporary variable and preallocated variables.

9.2.3 The Filtering Phase

The process of filtering domains is very similar to that described in CT. This is given by Method filterDomains() in Algo. 20 line 14. For each remaining unbound variable x in $\mathtt{S}^{\mathtt{sup}}$ and each value a in $\mathtt{dom}(x)$, the intersection between the valid arcs on x, currdiagram$[x]$, and the arcs labeled with value a, supports$[x, a]$, determines if a is still supported. An empty intersection means that a can be deleted. This is correct because all *remaining* arcs in currdiagram$[x]$ are necessarily part of a valid path in the graph thanks to the update.

Back to our example, remaining arcs as defined by currdiagram corresponds to the MDD depicted in Fig. 9.2d. Regarding x_4, currdiagram$[x_4]$ is 1001. Because supports$[x_4, 0]$ is 0101 and supports$[x_4, 1]$ is 1010, we can deduce (from bitwise intersections) that both values are still valid for x_4.

Complexity. The worst-case time complexity of the filtering phase is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{S}^{\mathtt{sup}}} (|\mathtt{dom}^c(x)|)\Big\lceil\frac{|\Theta^0[x]|}{w}\Big\rceil\Big)$$

where w is the number of bits into a word (i.e, for java Long type, $w = 64$). The worst-case space complexity of the filtering phase is

$$\mathcal{O}(1)$$

as it uses only a fixed number of temporary variables and preallocated variables.

9.2.4 GAC and Complexity

The main method in `CD` is `enforceGAC`(). After the initialization of the sets $\texttt{S}^{\texttt{val}}$ and $\texttt{S}^{\texttt{sup}}$, calling `updateDiagram`() allows us to update the diagram, and more specifically `currdiagram` to filter out (indices of) arcs that are no more valid. Once the graph is updated, it is possible to test whether each value has still a support, by calling `filterDomains`(). If ever a domain wipe-out (failure due to a domain becoming empty) occurs, an exception is thrown during the update of the graph (and so, this is not directly managed in this main method). At the end of `enforceGAC`(), `lastSizes` is updated in view of the next call.

Proposition 9.5. *The* `CD` *algorithm (Algorithm 20 and Algorithm 21) applied to a positive MVD constraint C enforces GAC.*

Proof. By means of Method `updateDiagram`(), we maintain the set of valid arcs in `currdiagram`. This respects inv. 9.1. Method `filterDomains`() allows to check if a given value (x, a) is still supported and delete it if not. This respects inv. 9.2. By Prop. 9.1, the algorithm is GAC. □

Complexity. The worst-case time complexity is

$$\underbrace{\mathcal{O}\Big(\sum_{x\in\texttt{S}^{\texttt{val}}}\big(\min(|\Delta_x|, |\texttt{dom}^c(x)|)\Big\lceil\frac{|\Theta^0[x]|}{w}\Big\rceil\big) + \sum_{x\in\texttt{scp}()}|\Omega^c[x]|\Big\lceil\frac{|\Theta^0[x]|}{w}\Big\rceil\Big)}_{update} +$$

$$\underbrace{\mathcal{O}\Big(\sum_{x\in\texttt{S}^{\texttt{sup}}}(|\texttt{dom}^c(x)|)\Big\lceil\frac{|\Theta^0[x]|}{w}\Big\rceil\Big)}_{filtering}$$

Since $|\texttt{scp}| \geq |\texttt{S}^{\texttt{sup}}|$ and $|\texttt{scp}| \geq |\texttt{S}^{\texttt{val}}|$ this can be globally reduced to

$$\mathcal{O}\Big(|\texttt{scp}|\ (d+n)\ \Big\lceil\frac{|A|}{w}\Big\rceil k\Big)$$

where d is the size of the largest current domain, n is the size of the current largest layer of nodes, A is the size of the initial largest layer of arcs, w is the number of bits into a word (i.e, for Java Long type, $w = 64$) and k is the complexity of the bitcount operation used in the nb1s method (i.e. for Java API method java.lang.Long.bitCount, $k = \log(w)$).

The worst-case space complexity is

$$\mathcal{O}\Big(\sum_{x \in \mathtt{scp}} (|\mathtt{dom}^0(x)| + |t(\Theta^0[x])| + |h(\Theta^0[x])|) \Big\lceil \frac{|\Theta^0[x]|}{w} \Big\rceil \Big)$$

which can be globally reduced to

$$\mathcal{O}\Big(|\mathtt{scp}|\ (d^0 + n^0)\ \Big\lceil \frac{|A|}{w} \Big\rceil \Big)$$

where $d^0 = \max_{x \in \mathtt{scp}} \{|\mathtt{dom}^0(x)|\}$ is the size of the largest initial domain, n^0 is the maximum number of node in one node layer and w is the number of bits into a word (i.e, for Java Long type, $w = 64$).

9.3 Compact-Diagram for Basic Smart Diagrams

CD and CT are quite similar in terms of design. Basically, both of them use bitsets called supports to respectively indentify the tuples or arcs that must be discarded. In the same spirit, we show how the ideas of $\mathtt{CT}^{bs}$ can be reused to adapt the method updateMask() of CD, leading to $\mathtt{CD}^{bs}$.

9.3.1 Simple Adaptation of $\mathtt{CT}^{bs}$

As in $\mathtt{CT}^{bs}$, in addition to bitsets supports, we introduce auxiliary bitsets:

- $\mathtt{supports}^*[x, a]$, the exclusive supports: for each arc for which the label of arc ω is exactly a ('$= a$'), the bit is set to 1,
- $\mathtt{supportsMin}[x, a]$, the lower bound supports: for each arc which would be still valid if the minimum of the domain was a, the bit is set to 1,
- $\mathtt{supportsMax}[x, a]$, the upper bound supports: for each arc which would be still valid if the maximum of the domain was a, the bit is set to 1.

Algorithm 22 displays the method updateMasks() for the simple version of $\mathtt{CD}^{bs}$. This is the simpliest adaptation for Compact-Diagram of the modifications made to pass from CT to $\mathtt{CT}^{bs}$. Resetting (and recomputing) is performed when the number of removed values (i.e. values in Δ_x) is too large by collecting the supports of every value in the current domain. As in $\texttt{CT}^{bs}$, resetting is also chosen if the layer contains $\langle \in S \rangle$ elements. Otherwise an incremental update is performed, using `supports`*, `supportsMin` and `supportsMax` in order to handle $\langle * \rangle$, $\langle \neq v \rangle$, $\langle \leq v \rangle$ and $\langle \geq v \rangle$.

Complexity. The time complexity of one call to `updateMasks`(), for a given variable x, is

$$\Theta\Big(d\Big\lceil \frac{|\Theta^0[x]|}{w} \Big\rceil\Big)$$

where d is $\min(|\Delta_x|, |\texttt{dom}(x)|)$ if $x \notin \texttt{S}^{\langle \in S \rangle}$ and $|\texttt{dom}(x)|$ if not.

9.3.2 Optimized Version of $\texttt{CD}^{bs}$

Contrary to $\texttt{CT}^{bs}$, where one bit in `currtable` is involved with every variable, in $\texttt{CD}^{bs}$, one bit only affects one layer of arcs and thus one variable. This allows us to introduce an optimized version that strongly relies on a partition of the arcs at each level i, defined as follows:

- $\texttt{C}^{\textsf{bas}}[x] = \{\vartheta \in \Theta[x] : op(l(\vartheta)) \in \{\langle = \rangle, \langle \neq \rangle, \langle * \rangle\}\}$,

---

Algorithm 22: Simple Version of $\texttt{CD}^{bs}$

Method **`updateMasks()`**
  foreach *variable* $x \in S^{val}\}$ **do**
    if $|\Delta_x| < |dom(x)| \wedge x \notin S^{\langle \in S \rangle}$ **then** `// Incremental update`
      foreach *value* $a \in \Delta_x$ **do**
        $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \texttt{supports}^*[x, a]$ `// bitwise OR`
      if $dom(x)$*.minChanged()* **then**
        $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \sim \texttt{supportsMin}[x, x.min]$
      if $dom(x)$*.maxChanged()* **then**
        $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \sim \texttt{supportsMax}[x, x.max]$
    else `// Reset-based update`
      foreach *value* $a \in dom(x)$ **do**
        $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \texttt{supports}[x, a]$ `// bitwise OR`
      $\texttt{mask}[x] \leftarrow \sim \texttt{mask}[x]$ `// bitwise NOT`

– $\mathtt{C}^{\mathtt{min}}[x] = \{\vartheta \in \Theta[x] : op(l(\vartheta)) \in \{\langle<\rangle, \langle\leq\rangle\}\}$,

– $\mathtt{C}^{\mathtt{max}}[x] = \{\vartheta \in \Theta[x] : op(l(\vartheta)) \in \{\langle>\rangle, \langle\geq\rangle\}\}$,

– $\mathtt{C}^{\mathtt{set}}[x] = \{\vartheta \in \Theta[x] : op(l(\vartheta)) \in \{\langle\in\rangle, \langle\notin\rangle\}\}$

The time complexity of Algo. 22 can be improved to reach $\Omega(\frac{|\Theta^0[x]|}{w})$ and $\mathcal{O}(d\frac{|\Theta^0[x]|}{w})$. For that, let us consider the hypothetical case of a variable with an operator in $\{\langle<\rangle, \langle\leq\rangle, \langle>\rangle, \langle\geq\rangle\}$ for each of its associated arc labels. In such a case, one can collect invalid arcs using lines 6 and 8 from Algo. 22, and there is no need to iterate over the sets $dom(x)$ or Δ_x. This favorable situation can be partially forced by sorting arcs in bitsets `supports` so that the bits in a computer word only represent arcs from a given category ($\mathtt{C}^{\mathtt{bas}}, \mathtt{C}^{\mathtt{set}}, \mathtt{C}^{\mathtt{min}}, \mathtt{C}^{\mathtt{max}}$). If each computer word is filled with (bits for) arcs belonging to the same category (dummy invalid arcs are used to complete a word if necessary), then only the required specific operations can be systematically applied to this word. This leads to Algo. 23 that iterates over the valid words and only applies the operations required by the category of the word (note that the category for the jth word is given by `currdiagram`$[x].category[j]$). Arcs from $\mathtt{C}^{\mathtt{bas}}$ are updated using `supports`* or `supports` (incremental or reset case). Arcs from $\mathtt{C}^{\mathtt{set}}$ are updated using `supports` in all cases. Arcs from $\mathtt{C}^{\mathtt{min}}$ and $\mathtt{C}^{\mathtt{max}}$ are updated using `supportsMin` and `supportsMin`, respectively. It appears that the categories $\mathtt{C}^{\mathtt{min}}$ and $\mathtt{C}^{\mathtt{max}}$ are particularly cheap to treat as they only imply one value.

An Interesting Observation. In Algo. 23, each valid word is associated with a (unique) category. From this fact, one can observe that `supportsMin` and `supportsMax` are useless.

Proof. For any variable x, and any word index j of `currdiagram`$[x]$, we have:

$$\mathtt{currdiagram}[x].category[j] = \mathtt{C}^{\mathtt{min}} \Rightarrow \mathtt{supportsMin}\ [\text{x,a}][j] = \mathtt{supports}[x,a][j]$$

Similarly,

$$\mathtt{currdiagram}[x].category[j] = \mathtt{C}^{\mathtt{max}} \Rightarrow \mathtt{supportsMax}\ [\text{x,a}][j] = \mathtt{supports}[x,a][j]$$

□

Proof. (sketch for $\texttt{C}^{\texttt{min}}$) By restricting the scope of the definitions of the bitsets to the word (index) j whose bits are exclusively associated with arcs from $\texttt{C}^{\texttt{min}}$, $\texttt{supports}[x, a][j]$ contains arcs represented by this word that accept the value a, i.e. arcs labeled by $\leq v$ with $v \geq a$, whereas $\texttt{supportsMin}[x, a][j]$ contains arcs for which $\exists b \in dom(x)$ accepted by the arcs such as $a \leq b$, i.e. arcs labeled by $\leq v$ with $v \geq a$. The two words end up to be equal: the exact same bits are set for both

Algorithm 23: Optimized Version of $\texttt{CD}^{bs}$

```
Method updateMasks()
    foreach variable x ∈ {x ∈ scp : |Δx| > 0} do
        foreach index j ∈ currdiagram[x].validWords do
            switch currdiagram[x].category[j] do
                case C^bas do
                    if |Δx| < |dom(x)| then          // Incremental update
                        foreach value a ∈ Δx do
                            mask[x][j] ← mask[x][j] | supports*[x, a][j]
                    else                             // Reset update
                        foreach value a ∈ dom(x) do
                            mask[x][j] ← mask[x][j] | supports[x, a][j]
                        mask[x][j] ← ~ mask[x][j]
                case C^set do
                    foreach value a ∈ dom(x) do
                        mask[x][j] ← mask[x][j] | supports[x, a][j]
                    mask[x][j] ← ~ mask[x][j]
                case C^min do
                    if dom(x).minChanged() then
                        mask[x][j] ← mask[x][j] | ~ supportsMin[x, x.min][j]
                case C^max do
                    if dom(x).maxChanged() then
                        mask[x][j] ← mask[x][j] | ~ supportsMax[x, x.max][j]
```

supports$[x, a][j]$ and supportsMin$[x, a][j]$. □

This observation is illustrated by Fig. 9.3. For any literal (x, a) and any word index j of category $\mathtt{C}^{\mathtt{min}}$ (resp., $\mathtt{C}^{\mathtt{max}}$), the word supportsMin$[x, v][j]$ (resp., supportsMax$[x, v][j]$) is equal to the word supports$[x, v][j]$. Therefore, we can simply use supports at lines 19 and 22. It means that the only required auxiliary bitset is supports* for words attached to $\mathtt{C}^{\mathtt{bas}}$.

Overall Complexity of the Propagator. Regarding the time complexity of the propagator (and not only the updateMasks() method), CD is $\mathcal{O}(max(n, d)r\frac{a}{w})$ where r is the arity of the constraint, d the greatest domain size, n (resp. a) the maximum number of nodes (resp. arcs) per level and w the size of computer words ($w = 64$ for Java long integer type). $\mathtt{CD}^{bs}$ keeps the same complexity. Regarding the space complexity, the maximum number of words of one bitset is $\lceil\frac{a}{w}\rceil + 3$. Per level, there is one currdiagram, d supports and supports* (its length is min 0 words, if $\mathtt{C}^{\mathtt{bas}} = \phi$ and $\lceil\frac{a}{w}\rceil$ max, if $|\mathtt{C}^{\mathtt{set}}| \leq w$, $|\mathtt{C}^{\mathtt{min}}| \leq w$ and $|\mathtt{C}^{\mathtt{max}}| \leq w$) and n arcsH and arcsT. The space complexity is thus $\mathcal{O}((d + n)r\frac{a}{w})$.

9.4 Results

The performances of CD and $\mathtt{CD}^{bs}$ have been evaluated. The benchmark used are the instances available on the XCSP3 website restricted to tables only. The results are compared using performance profiles.

9.4.1 Experiments Results with CD

To evaluate the performance of CD, two benchmarks were built from the instances from XCSP3. The first one results from the transformation of each table into an MDD using pReduce [PR15]. $\mathtt{CD}^{p}$ is the performance of CD on this benchmark. The second one results from the transformation of each table into an sMDD using sReduce (Sec. 6.2.3). $\mathtt{CD}^{s}$ is the performance of CD on this benchmark.

On Fig. 9.4, execution times of MDD4R, $\mathtt{CD}^{p}$ and $\mathtt{CD}^{s}$ are compared. Times are given for a complete exploration of the search space (i.e. to find all solutions), using each time the same variable and value ordering. Clearly, CD outperforms MDD4R, even when it is executed on "simple" MDD*s*. Using sMDD*s* just makes it more robust. For example, $\mathtt{CD}^{s}$, $\mathtt{CD}^{p}$ and MDD4R are at least twice slower than the best (virtual) algorithm on 5%, 20% and 35% of the instances, respectively. On Fig. 9.5, CT is additionally considered. In general, CT still outperforms decision diagram approaches, but the gap is reduced: 40% of the instances are solved by CD^s within a factor 2 compared to the time taken by CT, instead of 5% previously with MDD4R.

It is important to note that these global results do not tell the entire story. Indeed, when the compression is high, using decision diagrams remains the appropriate approach. For example, on the instance

	x
ω_0	$= 1$
ω_1	≤ 2
ω_2	≥ 1
ω_3	$\in \{1, 3\}$
ω_4	$\neq 1$

	x
ω_5	> 2
ω_6	$\notin \{0, 3\}$
ω_7	< 2
ω_8	$\neq 2$
ω_9	$*$

(a) Labels of Arcs

(Category)	word 0 $\texttt{C}^{\texttt{bas}}$				word 1 $\texttt{C}^{\texttt{set}}$				word 2 $\texttt{C}^{\texttt{min}}$				word 3 $\texttt{C}^{\texttt{max}}$			
	ω_0	ω_4	ω_8	ω_9	ω_3	ω_6			ω_1	ω_7			ω_2	ω_5		
$[x, 0]$	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0
$[x, 1]$	1	0	1	1	1	1	0	0	1	1	0	0	1	0	0	0
$[x, 2]$	0	1	0	1	0	1	0	0	1	0	0	0	1	0	0	0
$[x, 3]$	0	1	1	1	1	0	0	0	0	0	0	0	1	1	0	0
$[x, 4]$	0	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0

(b) Bitsets `supports` for literals on x

(Category) (From)	word 0 $\texttt{C}^{\texttt{bas}}$ supportsMin				word 1 $\texttt{C}^{\texttt{set}}$ no auxiliary				word 2 $\texttt{C}^{\texttt{min}}$ supportsMin				word 3 $\texttt{C}^{\texttt{max}}$ supportsMax			
	ω_0	ω_4	ω_8	ω_9	ω_3	ω_6			ω_1	ω_7			ω_2	ω_5		
$[x, 0]$	0	0	0	0	-	-	-	-	1	1	0	0	0	0	0	0
$[x, 1]$	1	0	0	0	-	-	-	-	1	1	0	0	1	0	0	0
$[x, 2]$	0	0	0	0	-	-	-	-	1	0	0	0	1	0	0	0
$[x, 3]$	0	0	0	0	-	-	-	-	0	0	0	0	1	1	0	0
$[x, 4]$	0	0	0	0	-	-	-	-	0	0	0	0	1	1	0	0

(c) Auxiliary bitsets for literals on x

Figure 9.3: Bitsets related to a variable x, assuming 10 associated arcs $\omega_0, \omega_1, \ldots$ in the $bs-\texttt{MVD}$. The size of computer words is assumed to be 4, for simplicity.

pigeonsPlus-11-06, the execution times of CT, MDD4R, $\texttt{CD}^p$ and $\texttt{CD}^s$ are respectively $T.O.(> 600s)$, $328s$, $128s$ and $126s$. This confirms the real interest of approaches based on decision diagrams.

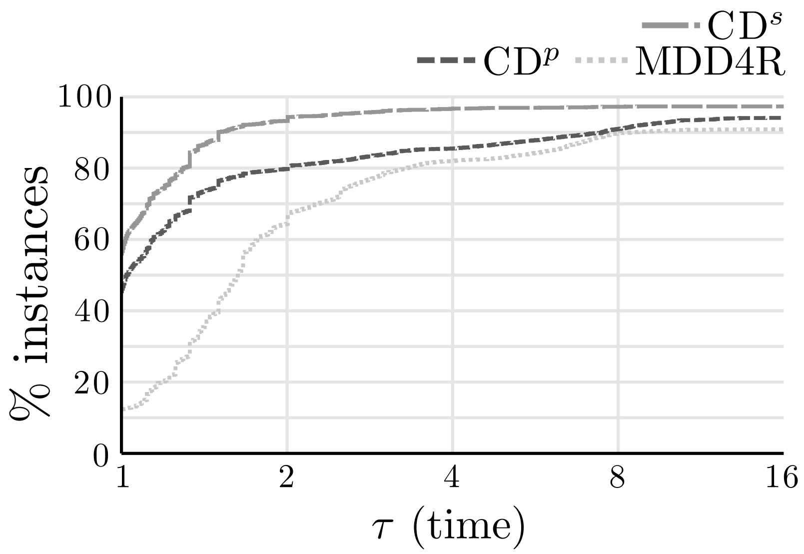


Figure 9.4: Comparing MDD4R, $\texttt{CD}^p$ and $\texttt{CD}^s$.

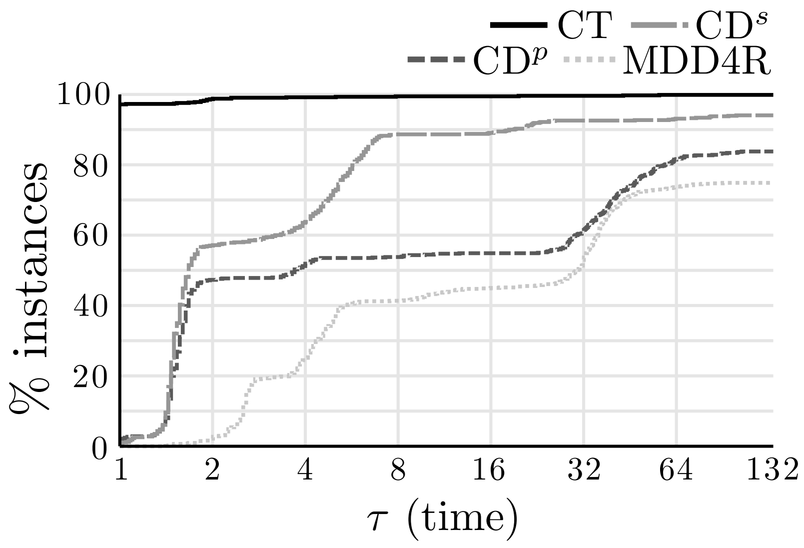


Figure 9.5: Comparing MDD4R, $\texttt{CD}^p$, $\texttt{CD}^s$ and CT.

9.4.2 Experiments Results with $\texttt{CD}^{bs}$

The benchmarks here are all derived from the initial benchmark from the XCSP3 website. They are defined as:

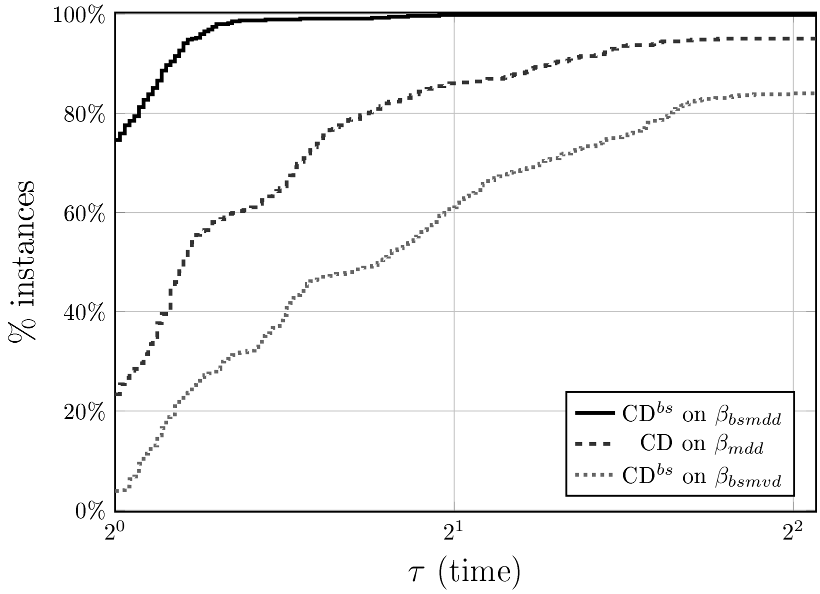


Figure 9.6: Comparing MDD4R, $\texttt{CD}^p$ and $\texttt{CD}^s$.

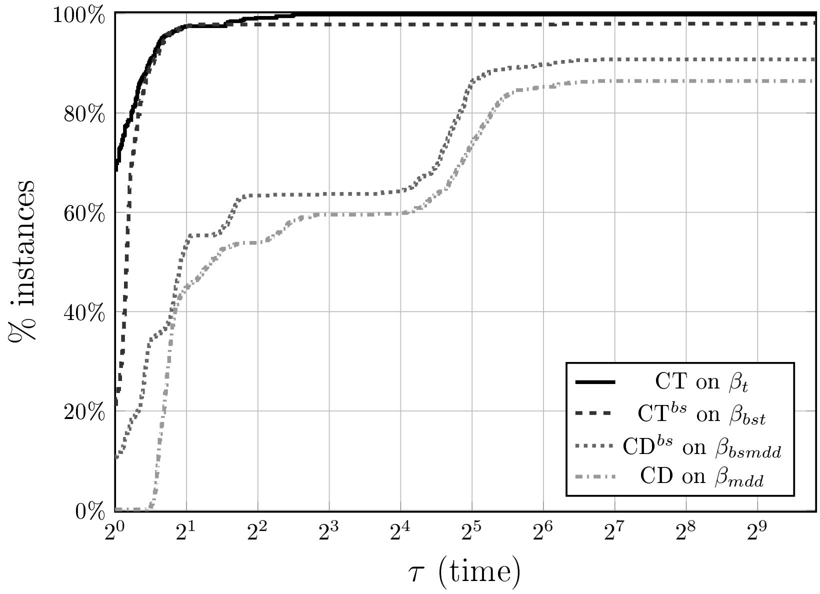


Figure 9.7: Comparing MDD4R, $\texttt{CD}^p$, $\texttt{CD}^s$ and CT.

– β_t: the initial benchmark. It is a set of roughly $4,000$ instances only containing (positive) table constraints, and available on the XCSP3 website [BLP16].

– β_{bst}: instances of β_t have been transformed into instances where $bs-\texttt{table}s$ replace (ordinary) tables. The compression used is the one presented in Sec. 5.4.2.2.

– β_{mdd}: instances of β_t have been transformed into instances where MDDs replace (ordinary) tables. The algorithm pReduce [PR15] was used.

– β_{bsmvd}: instances of β_{bst} have been transformed into instances where $bs-\texttt{MVD}s$ replace $bs-\texttt{table}s$. The algorithm pReduce$_{bs}$ was used.

– β_{bsmdd}: instances of β_{mdd} have been transformed into instances where $bs-\texttt{MDD}s$ replace $\texttt{MDD}s$.

Figure 9.6 shows the results of a comparison between CD and $\texttt{CD}^{bs}$. The filtering algorithm $\texttt{CD}^{bs}$, as it could be expected, obtains a larger speedup when applied on graphs with fewer nodes and arcs, i.e. on instances from β_{bsmdd}.

In particular, we can see that on the benchmark β_{bsmvd} (based on a compression into $bs-\texttt{table}s$, followed by a generation of $bs-\texttt{MVD}s$) $\texttt{CD}^{bs}$ performs worse than CD applied on β_{mdd} (standard $\texttt{MDD}s$). The reason is that graphs in β_{bsmvd} have generally a greater number of nodes than other equivalent graphs as shown before in Sec. 6.3.3. This follows the same conclusions regarding why CD was more efficient on $\texttt{sMDD}s$ (having fewer nodes than $\texttt{MDD}s$).

An interesting remark is that, contrarily to $\texttt{CT}^{bs}$, the presence of expressions '$\in S$' does not induce any overhead for $\texttt{CD}^{bs}$. Since the arcs involving expressions of the form '$\in S$' are gathered on the same bitwords, they don't prevent from doing an incremental update when considering the other categories of expressions, as it was the case for CT.

CT was shown to remain faster than CD despite the introduction of bitwise operations. We revisit the same experiment with the newly presented algorithm. Figure 9.7 compares four scenarios, including the use of CT: CT on β_t, $\texttt{CT}^{bs}$ on β_{bst}, CD on β_{mdd} and $\texttt{CD}^{bs}$ on β_{bsmdd}.

One can see that CT is still the best approach, followed by $\texttt{CT}^{bs}$. Nevertheless, as it can be observed in the figure, the gap is shrinking when using the new algorithm $\texttt{CD}^{bs}$. Also, there is now around 10% of the instances where $\texttt{CD}^{bs}$ is the fastest algorithm. A post analysis has shown that instances with larger domains are the most favorable for

$\mathtt{CD}^{bs}$. In such cases, we could observe for some tables a reduction by a factor of up to 8 on the number of arcs.

The main advantage of CD thus lies in the potential compactness of the diagrams, although this is really problem/constraint dependent. On the one side, some graphs, when expanded into tables, can't even fit in memory. On the other side, some constraints, like AllDifferent [Per17] are not well suited for an MDD representation because there is almost no compression. When CD can benefit from a large compression, it becomes faster.

For a fair comparison, the choice was made not to evaluate the new algorithm on a priori favorable problems, hence the benchmarks composed of problems that initially contain table constraints. Also the order of variables remained unchanged (order as described in the initial instances used). Optimizing this order may also have an impact on the size of the graphs [CGBR19].

In our opinion, having both CT and CD is useful: if, for a given constraint, a high compression (by an MDD or another diagram) is possible, CD should be used, otherwise CT is more suited. Also, the new algorithm should typically be used for solving combinatorial problems with complex constraints that can't even be represented in memory as simple tables. One good example of work in that direction is [RPR+16]. Another promising direction for applying this propagator is for solving combinatorial problems on Strings.

9.5 Conclusion

This chapter introduces the Compact-Diagram algorithm. Globally, it follows the same operational steps as CT. Contrary to tables, updating a diagram requires to propagate the removal of edges through the graph. This result in the addition of a two-way visitation of the diagram (from top to bottom and from bottom to top) to make the diagram consistent again. The $\mathtt{CD}^{bs}$ algorithm is inspired from the corresponding basic smart table propagator $\mathtt{CT}^{bs}$. However, the structural difference between tables and diagrams helps to make an improved adaptation. In $\mathtt{CT}^{bs}$, one bit is associated to each tuples, making the bit involved with all variables at once. In $\mathtt{CD}^{bs}$, this is not the case as each bit is associated to one arc and thus one variable. This allows an efficient sorting of the edges among the words of the bitset, allowing the most efficient update for each word.

The results on both algorithms show an improvement of the performance of the diagram-based propagator. However, compared to CT and $\mathtt{CT}^{bs}$ on equivalent tables, there is still a (now reduced) gap in performance.

One final thing to notice is that contrary to `MDD4R`, `CD` and `CD`^{bs} are not designed only for `MDD`s but for `MVD`s in general, i.e. any layered diagram and not only the ones constructed with decision nodes.

The `CD` algorithm was published as part of the [VLS18] paper (the algorithm is named `Compact-MDD` in this paper). The `CD`^{bs} extension was published as part of the [VLS19a] paper.

Part IV

Conclusion

Chapter 10

Conclusion

The last ever dolphin message was misinterpreted as a surprisingly sophisticated attempt to do a double-backwards-somersault through a hoop whilst whistling the 'Star Spangled Banner', but in fact the message was this: So long and thanks for all the fish.

- Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Conclusion

In this thesis, we have presented some developement that we have made concerning `Compact-Table`. This articulates upon two aspects of extensional constraints. First, some extensional representations were studied. Second, some propagators designed to handle these representations were explained.

Extentional representations

Two of the most used representations, tables, and `MDD`*s*, were studied on several aspects. The goal was to establish some of their limitations and try to enhance them in order to make them more efficient.

From the table point of view, we studied their compression into basic smart tables. These maintain the classical organization of tables using tuples with values. This is one of the aspects which makes the adaptation of `CT` possible. In addition, they add the possibility of using unary and binary constraints as values. Optimal compression is difficult to achieve. However, some greedy algorithms can achieve sufficiently good results. Tractable incompressibility of some tables has also been established.

From the diagram point of view, we studied how to create a new structure allowing more non-determinism, easy construction and usage (as for MDDs). This conducted us to introduce the sMDD data structure. The top half of the structure is like an MDD, composed of decision nodes. The bottom half of the structure is like an inverted MDD, composed of inverted decision nodes. The two middle node layers are composed of totally non-deterministic nodes. The main improvement brought by sMDDs is the high reduction in the number of nodes required. In addition, we studied the use of the same unary constraints as in basic smart tables in order to compress these diagrams into basic smart diagrams. The work on basic smart diagrams includes methods to create them from the equivalent table. The most efficient way to do it is by creating the corresponding MDD (or sMDD) and merging edges sharing the same tail and head.

Propagators

All extensions of CT share the same global structure as CT, using the improvements brought by the various algorithms through the history of extensional constraints. First, as CT, they keep track of the *reduction* of the representation in a structure representing the current representation using reversible sparse bitsets. Second, they use precomputed *bitsets* to store the *supports* used to speed up the computations. Third, their propagation is divided into two phases: the update phase and the filtering phase. The update phase, consisting of a combination of an *incremental* update and a *reset* update, proceeds with the reduction of the representation in order to update the current representation. The filtering phase proceeds with removing the values that are no longer supported by the current representation.

$\mathtt{CT}^*$ and $\mathtt{CT}^{bs}$ were the first designed types of extension. They target respectively short and basic smart tables. The modifications required the design of new additional precomputed bitsets: $\mathtt{supports}^*$ (mostly used to handle $\langle * \rangle$ and $\langle \neq v \rangle$), supportsMin (mostly used to handle $\langle \leq v \rangle$) and supportsMax (mostly used to handle $\langle \geq v \rangle$). In addition, a modification is required to the classical update in other to make it work. This results in efficient algorithms to handle these kinds of positive compressed tables.

$\mathtt{CT}_{neg}$ and $\mathtt{CT}^*_{neg}$ were the second designed types. They target respectively negative and negative short tables. Their design necessitated changing the way to verify the support of a value. Due to the conflicting nature of the tuples from the table, a value is still valid if it exists one possible instantiation (containing the value) not in the table. This required counting the supporting tuples and comparing the count to the total number of valid tuples possible. The $\mathtt{CT}_{neg}$ extensions require thus a modification to the filtering phase. This results in an efficient algorithm to handle negative tables. The GAC propagation on negative short tables is NP-complete. Adapting CT to this case is therefore complicated. We chose to set a hypothesis on tables to render the problem polynomial. This hypothesis imposes that there are no overlapping tuples in the table. This allows keeping a polynomial counting of the supporting tuple from the table. $\texttt{CT}^*_{neg}$ also uses the concept of $\texttt{supports}^*$ in its incremental update in order to handle the $\langle * \rangle$ contained in the table. The results on $\texttt{CT}^*_{neg}$ are more dependent on the structure of the table. To allow easy counting, some dummy tuples need to be added to the table in order to have words containing similar tuples, which can in some case impact the efficiency of the algorithm.

Finally, CD and $\texttt{CD}^{bs}$ were designed. One of the main modifications is the break of the main reversible sparse set into one for each layer of the diagram. The other essential modification comes from that removing an edge with an unsupported value is not enough. In addition, a two-way pass has to be performed on the diagram to make it consistent again. On MDD*s*, the results of CD allow an improvement compared to the state-of-the-art MDD propagator. However, it is still insufficient to close the gap between the performances of CT and the best MDD propagator. The use of basic smart MDD*s* with the propagator $\texttt{CD}^{bs}$ helps to close a bit more the gap.

Finally, the use of the new diagram representations base on more non-determinism, i.e. sMDD*s* and $bs-\texttt{sMDD}s$, due to their reduction of the number of nodes compared to the equivalent MDD*s* and $bs-\texttt{MDD}s$, leads to a better propagation than using MDD*s*. However, it still does not close the gap with the performances using the equivalent table representation.

Perspectives

This thesis has introduced several new propagators and had increased the variety of available extensional representations. However, the topic is far from being closed. There are at least three paths of further research that can be investigated.

Non-determinism in Diagrams

From the diagram point of view, this work has highlighted the usefulness of non-determinism with the new sMDD data structure. This structure only has non-deterministic nodes on two precise chosen layers. Evaluating the impact of the position of these two layers could lead to improvements in the structure. Also, studying ways to have more non-determinism inside diagrams in order to reduce even more their size could help speed up the resolution of some problems. Finally, finding procedures to generate such diagrams with non-determinism not limited to some specific layers could be crucial to help speed up the resolution of some problems.

Closing the Gap between Diagrams and Tables Propagators

A more extensive study of the non-determinism in diagrams is one way to close the gap. Another would be to improve even more the `CD` propagator. Even if the gap is not closed yet, it has already been acknowledged that some diagrams represent tables too big to be stored. This is the main reason why research on diagram propagators should not be abandoned even if the actual table propagators outperform the current diagrams propagators.

Direct Use of Compressed Tables and Non-Deterministic (Compressed) Diagrams

Now that efficient propagators are available for compressed tables, these modeling tools can be used more broadly. Adapting existing processes, such as auto-tabling [DBC+17] or automatic compilation of constraints into `MDD`*s* [HHOT08, dUGSS19], to generate automatically compressed table or non-deterministic (compressed) diagrams is also something that should be studied, now that enhanced propagators are available.

Bibliography

Bibliography

[AFNP14] Jérôme Amilhastre, Hélène Fargier, Alexandre Niveau, and Cédric Pralet. Compiling csps: A complexity map of (non-deterministic) multivalued decision diagrams. *International Journal on Artificial Intelligence Tools*, 23(04):1460015, 2014.

[AHHT07] H. Andersen, T. Hadzic, J. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of CP'07*, pages 118–132, 2007.

[ALM20] Gilles Audemard, Christophe Lecoutre, and Mehdi Maamar. Segmented tables: An efficient modeling tool for constraint reasoning. *Frontiers in Artificial Intelligence and Applications*, 325:315–322, 2020.

[ANS20] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Pydl8. 5: a library for learning optimal decision trees. In *International Joint Conference on Artificial Intelligence*, 2020.

[Apt03] Krzysztof Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[ASS+88] A Aiba, K Sakai, Y Sato, D Hawley, and R Hasegawa. Constraint logic programming language cal. *Proc. International Conference on Fifth Generation Computer Systems*, pages 263–276, 1988.

[BCvH14] D. Bergman, A. Ciré, and W. van Hoeve. MDD propagation for sequence constraints. *Journal of Artificial Intelligence Research*, 50:697–722, 2014.

[BCvHH16] D. Bergman, A. Ciré, W. van Hoeve, and J. Hooker. *Decision diagrams for optimization.* Springer, 2016.

[Bes94] Christian Bessiere. Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1):179–190, 1994.

[Bli96] Christian Bliek. *Wordwise algorithms and improved heuristics for solving hard constraint satisfaction problems.* Citeseer, 1996.

[BLP16] F. Boussemart, C. Lecoutre, and C. Piette. XCSP3: An integrated format for benchmarking combinatorial constrained problems. Technical Report arXiv:1611.03398, CoRR, 2016. Available from `http://www.xcsp.org`.

[BLPN12] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2012.

[BOP20] Behrouz Babaki, Bilel Omrani, and Gilles Pesant. Combinatorial search in cp-based iterated belief propagation. In *International Conference on Principles and Practice of Constraint Programming*, pages 21–36. Springer, 2020.

[Bor12] Christian Borgelt. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 2(6):437–456, 2012.

[BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, volume 1, pages 309–315, 2001.

[Bry86] Randal Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BRYZ05] Christian Bessière, Jean-Charles Régin, Roland HC Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[CGBR19] Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of AAAI'19*, 2019.

[CHLS06] Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J Stuckey. Finite domain bounds consistency revisited. In *Australasian Joint Conference on Artificial Intelligence*, pages 49–58. Springer, 2006.

[CMR+20] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. *arXiv preprint arXiv:2006.01610*, 2020.

[Col90] Alain Colmerauer. An introduction to prolog iii. *Communications of the ACM*, 33(7):69–90, 1990.

[CY08] Kenil Cheng and Roland Yap. Maintaining generalized arc consistency on ad-hoc r-ary constraints. In *Proceedings of CP'08*, pages 509–523, 2008.

[CY10] Kenil Cheng and Roland Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[DBC+17] Jip J. Dekker, Gustav Björdal, Mats Carlsson, Pierre Flener, and Jean-Noël Monette. Auto-tabling for subproblem presolving in minizinc. *Constraints An Int. J.*, 22(4):512–529, 2017.

[DHL+16] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.

[DM02a] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DM02b] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[DRGN10] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for data mining and machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 2010.

[DSVH87] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Extending equation solving and constraint handling in logic programming. In *Colloquium on Resolution of Equations in Algebraic Structures (CREAS), Texas*, 1987.

[dUGSS19] Diego de Uña, Graeme Gange, Peter Schachte, and Peter J Stuckey. Compiling cp subproblems to mdds and d-dnnfs. *Constraints*, 24(1):56–93, 2019.

[DVH] Y Deville and P Van Hentenryck. An efficient arc consistency algorithm for a class of csps. In *Proceedings of IJCAI-91*, pages 325–330.

[FM01] Filippo Focacci and Michaela Milano. Global cut framework for removing symmetries. In *International Conference on Principles and Practice of Constraint Programming*, pages 77–92. Springer, 2001.

[Fre96] Eugene Freuder. In pursuit of the holy grail. *ACM Computing Surveys (CSUR)*, 28(4es):63–es, 1996.

[GHLR14] Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel. Sliced table constraints: Combining compression and tabular reduction. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 120–135. Springer, 2014.

[GJMN07] Ian Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.

[Gun15] Tias Guns. Declarative pattern mining using constraint programming. *Constraints*, 20(4):492–493, 2015.

[HHOT08] T. Hadzic, J. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *Proceedings of CP'08*, pages 448–462, 2008.

[HvHH10] S. Hoda, W. van Hoeve, and J. Hooker. A systematic approach to MDD-Based constraint programming. In *Proceedings of CP'10*, pages 266–280, 2010.

[IS18] Linnea Ingmar and Christian Schulte. Making compact-table compact. In *International Conference on Principles and Practice of Constraint Programming*, pages 210–218. Springer, 2018.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J Stuckey, and Roland HC Yap. The clp (r) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, 1992.

[JN13] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[KS08] Subhash Khot and Rishi Saket. Hardness of minimizing and learning dnf expressions. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 231–240. IEEE, 2008.

[KW07] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.

[Lab03] Philippe Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.

[Lau78] JL Lauriere. Alice, a language for intelligent combinatorial exploration. *Artificial Intelligence*, 10:29–127, 1978.

[Lau18] Laurent Michel, Pierre Schaus, Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming, 2018. Available from `https://minicp.bitbucket.io`.

[lCdSMSSL13] V. le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *Proceeding of TRICS'13*, pages 1–10, 2013.

[LCKLD] Baudouin Le Charlier, Minh Thanh Khong, Christophe Lecoutre, and Yves Deville. Automatic synthesis of smart table constraints by abstraction of table constraints.

[Lec11] Christophe Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[LLGL13] Hongbo Li, Yanchun Liang, Jinsong Guo, and Zhanshan Li. Making simple tabular reductionworks on negative table constraints. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[LLY15] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland HC Yap. Str3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220:1–27, 2015.

[LM12] Michele Lombardi and Michela Milano. Optimal methods for resource allocation and scheduling: a cross-disciplinary survey. *Constraints*, 17(1):51–85, 2012.

[LRSV18] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

[LV08] Christophe Lecoutre and Julien Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters (CPL)*, 2:21–35, 2008.

[Mac77] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[McG79] James J McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229–250, 1979.

[MDL15] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 271–287. Springer, 2015.

[MH86] Roger Mohr and Thomas C Henderson. Arc and path consistency revisited. *Artificial intelligence*, 28(2):225–233, 1986.

[MM88] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of the 8th European conference on artificial intelligence*, pages 651–656, 1988.

[Per17] G. Perez. *Decision diagrams: constraints and algorithms*. PhD thesis, Université de Nice, 2017.

[PR14] Guillaume Perez and Jean-Charles Régin. Improving gac-4 for table and mdd constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 606–621. Springer, 2014.

[PR15] Guillaume Perez and Jean-Charles Régin. Efficient operations on MDDs for building constraint programming models. In *Proceedings of IJCAI'15*, pages 374–380, 2015.

[Rég95] JC Régin. Développement d'outils algorithmiques pour l'intelligence artificielle. *Applicationa la chimie organique. These de doctorat, Université des Sciences et Techniques du Languedoc, Montpellier*, 1995.

[Rég11] Jean-Charles Régin. Improving the expressiveness of table constraints. In *The 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011)*, 2011.

[RP97] Jean-Charles Régin and Jean-François Puget. A filtering algorithm for global sequencing constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 32–46. Springer, 1997.

[RPR+16] Pierre Roy, Guillaume Perez, Jean-Charles Régin, Alexandre Papadopoulos, François Pachet, and Marco Marchini. Enforcing structure on temporal sequences:

the allen constraint. In *International conference on principles and practice of constraint programming*, pages 786–801. Springer, 2016.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[SAG17] Pierre Schaus, John OR Aoga, and Tias Guns. Coversize: A global constraint for frequency-based itemset mining. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–546. Springer, 2017.

[SJ80] Guy L Steele Jr. The definition and implementation of a computer programming language based on constraints. 1980.

[Sut64] Ivan E Sutherland. Sketchpad a man-machine graphical communication system. *Transactions of the Society for Computer Simulation*, 2(5):R–3, 1964.

[Tea] OscaR Team. Oscar: Operational research in scala, 2012. *Available under the LGPL licence from https://bitbucket. org/oscarlib/oscar.*

[THS+92] Satoshi Terasaki, David J Hawley, Hiroyuki Sawada, Ken Satoh, Satoshi Menju, Taro Kawagishi, Noboru Iwayama, and Akira Aiba. Parallel constraint logic programming language gdcc and its parallel constraint solvers. *ICOT Technical Report*, 1992.

[Tsu92] Hiroshi Tsuda. cu-prolog for constraint-based grammar. In *Proc. Int. Conf. 5th Generation Computer Systems' 92*, pages 347–356, 1992.

[Ull76] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[Ull07] Julian R Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177(18):3639–3678, 2007.

[vH01] Willem-Jan van Hoeve. The alldifferent constraint: A survey. *arXiv preprint cs/0105015*, 2001.

[VLDS17] Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extending compact-table to basic smart tables. In *International Conference on Principles and Practice of Constraint Programming*, pages 297–307. Springer, 2017.

[VLDS18] Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extension de compact-table aux tables simplement intelligentes. In *Quatorzièmes journées Francophones de Programmation par Contraintes (JFPC18)*, 2018.

[VLS17a] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-table to negative and short tables. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence and the Twenty-Ninth Innovative Applications of Artificial Intelligence Conference*, volume 5, 2017.

[VLS17b] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extension de compact-table aux tables négatives et concises. In *Treizièmes journées Francophones de Programmation par Contraintes (JFPC17)*, 2017.

[VLS18] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bitsets. 2018.

[VLS19a] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-diagram to basic smart multi-valued variable diagrams. 2019.

[VLS19b] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-diagram propagateur efficace pour la contrainte (s)MDD. In *Quinzièmes journées Francophones de Programmation par Contraintes (JFPC19)*, 2019.

[VLS19c] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extension de compact-diagram aux smart MVD. In *Quinzièmes journées Francophones de Programmation par Contraintes (JFPC19)*, 2019.

[VNP+19a] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. In *The*

25th International Conference on Principles and Practice of Constraint Programming (CP2019), 2019.

[VNP+19b] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. In Katrien Beuls, Bart Bogaerts, Gianluca Bontempi, Pierre Geurts, Nick Harley, Bertrand Lebichot, Tom Lenaerts, Gilles Louppe, and Paul Van Eecke, editors, *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019*, volume 2491 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[VNP+20a] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. pages 1–25. Springer, 2020.

[VNP+20b] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming (extended abstract). In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4765–4769. ijcai.org, 2020.

[VNP+21] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Apprentissage d'arbres de décision optimaux grâce à la programmation par contraintes. In *Seizièmes journées Francophones de Programmation par Contraintes (JFPC21)*, 2021.

[Wal72] DI Waltz. Generating semantic description from drawings of scenes with shadows, aitr-271, 1972.

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

[WXYL16] R. Wang, W. Xia, R. Yap, and Z. Li. Optimizing Simple Tabular Reduction with a bitwise representation. In *Proceedings of IJCAI'16*, pages 787–795, 2016.

[XY13] Wei Xia and Roland HC Yap. Optimizing str algorithms with tuple compression. In *International Conference on Principles and Practice of Constraint Programming*, pages 724–732. Springer, 2013.