Université Catholique de Louvain
Ecole Polytechnique de Louvain

Efficient Algorithms for Table Constraints

Schaus Pierre
Supervisor

Deville Yves
Reader

Mairy Jean-Baptiste
Reader

Thesis submitted for the Master's degree
in Computer Science and Engineering
option Artificial Intelligence
by Jordan Demeulenaere

Louvain-la-Neuve
Academic year 2014 – 2015

Acknowledgments

First and foremost I wish to express my gratitude to my supervisor Pierre Schaus for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

I take this opportunity to thank my parents and sister for the unceasing encouragement, support and attention they gave me during my studies. I am also very grateful to my partner and friends who were always there cheering me up and stood by me through not only the good times but also the bad ones.

Contents

Introduction

Constraint Programming (CP) is a powerful programming paradigm that can solve a multitude of combinatorial problems. One of the key concept of CP is propagation, which allows to reduce the search space associated to a problem without losing any of its solutions. A problem and the corresponding solutions are expressed thanks to variables and constraints between those same variables. There exist many different constraints[1], but ultimately the single *table constraint* allows to express any one of them. Therefore, finding a good propagation algorithm for this constraint is critical and subject to a lot of research.

For this thesis, we were asked to study different table constraint propagators and implement the most efficient ones in OscaR [24]. The best algorithms and implementations known as of writing have been selected and then compared to each other.

Contribution

The main contribution of this work is the improvement of the table constraint in OscaR, which has seen its efficiency drastically increase. Indeed, no less than six different propagators have been added to OscaR for the purpose of this thesis.

The CP community might also benefit from the descriptions of the most sophisticated propagators through our many illustrated examples. To the best of our knowledge, the Compact Table algorithm (CT) implemented in or-tools[2] [10] and presented in Chapter 3 is not yet documented by the scientific community. Given that we consider CT currently as one of the best algorithms for the table constraint, this can also be considered as a contribution of this work.

Structure

The remainder of this document is structured as follows: the first two chapters provide the background necessary for the understanding of this work. Then we present CT, the algorithm for the table constraint implemented in or-tool [10]. The next chapter describes the GAC-4 algorithm [20] and its variant GAC-4R [25], one of the most efficient propagator as of writing. The fifth chapter details MDD-4R [25], a compression-based algorithm that represents its associated table thanks to a Multivalued Decision Diagram (MDD) [31]. Finally, the last chapter compares all state-of-the-art algorithms that were implemented in OscaR for the purpose of this thesis.

[1] The *Global Constraint Catalog* [3] enlists 423 global constraints as of writing.

[2] The gold medal solver of the MiniZinc Challenge 2014 [21].

Chapter 1

Background

This chapter provides the fundamental background necessary for the understanding of this thesis. Constraint Programming and associated concepts are presented in the first section. The next two sections describe the bitset and sparse set data structures. Finally, Multivalued Decision Diagrams are briefly described in the last section. The reader who feels comfortable with those notions should be able to safely skip this chapter.

1.1 Constraint Programming

Constraint Programming (CP) is a programming paradigm used to solve Constraint Satisfaction Problems (CSP). A CSP is fully described by a set of variables, domains and constraints. A domain restricts the different values that can be assigned to a variable. Each constraint declares a condition that must be satisfied between different variables of the CSP. The role of CP is to assign a value to each variable such that every constraint is satisfied. Constraint Programming can also solve Constraint Optimization Problems (COP) that have an additional objective to optimize. The formal definitions and notations used throughout this thesis are as follows.[1]

Definition 1.1.1 (Variable)**.** A variable x is an entity that can be assigned different values. The set of the possible values is called the *domain* of x and is denoted $\mathcal{D}(x)$. There exist many different types of domains. In the remainder of this thesis, we will assume that the domains of the manipulated variables are finite and discrete.

Definition 1.1.2 (Constraint)**.** A constraint $c(x_1, ..., x_n)$ states a *relation* $rel(c)$ that must hold between different variables. The *scope* of the constraint $scp(c) = \{x_1, ..., x_r\}$ is the set of variables restricted by the constraint, where r denotes the *arity* of the constraint.

Definition 1.1.3 (Constraint Satisfaction Problem)**.** A CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is fully defined by:

- A set of n variable $\mathcal{X} = \{x_1, x_2, ..., x_n\}$
- A set of n domains $\mathcal{D} = \{\mathcal{D}(x_1), \mathcal{D}(x_2), ..., \mathcal{D}(x_n)\}$
- A set of e constraints $\mathcal{C} = \{c_1, c_2, ..., c_e\}$ defined on the variables $x \in \mathcal{X}$

Definition 1.1.4 (Instantiation)**.** An instantiation $\mathcal{I} = \{x_1 = a_1, ..., x_n = a_n\}$ is an assignment of a value to each variable $x \in \mathcal{X}$.

Definition 1.1.5 (Solution)**.** An instantiation $\mathcal{S} = \{x_1 = a_1, ..., x_n = a_n\}$ is a *solution* of the CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ if and only if the assignment satisfies every constraint $c \in \mathcal{C}$.

[1] The definitions of this section have been inspired from the notes of [18].

1.1.1 Consistency and propagation

Consider a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ such that each variable $x \in \mathcal{X}$ has d different values in its domain. Then the *search space* consists of d^n possible instantiations, which is impossible to explore within a reasonable time period for large values of n. Propagation allows to reduce the search space by removing inconsistent values from the domain of the variables, i.e. values that will never belong to a solution.

Definition 1.1.6 (Support)**.** A value $a_i \in \mathcal{D}(x_i)$ is *supported* w.r.t. a constraint c if for each other variable $x_j \in scp(c)$ there exists a value $a_j \in \mathcal{D}(x_j)$ such that

$$c(x_1 = a_1, ..., x_i = a_i, ..., x_r = a_r)$$

is satisfied.

Definition 1.1.7 (Constraint consistency)**.** A constraint c is domain-consistent if and only if all the values of each variable $x \in scp(c)$ are supported w.r.t. c.

Definition 1.1.8 (CSP consistency)**.** A CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is domain-consistent if and only if all constraints $c \in \mathcal{C}$ are domain-consistent.

To each constraint $c \in \mathcal{C}$ we associate a propagator that removes values that are not supported regarding c. The propagation of c is triggered each time a specific event happens, for instance when the domain of one of its variables $x \in scp(c)$ is reduced. However removing values only with the help of propagators is not always sufficient to find a solution of the CSP, which is why we need to explore the search space in parallel.

1.1.2 Search

Once the CP solver has removed all inconsistent values through propagation and the CSP is domain-consistent, we have reached a *fix-point* and must search for a solution into the search space. This is achieved by reducing the domain of one or more uninstantiated variables $x \in \mathcal{X}$.

A classic choice made by a solver is either the assignment of a value a to a variable x or the removal of a from $\mathcal{D}(x)$. The choice of the variable and the value are called respectively variable and value heuristics. Reducing the domain of a variable to find a solution is called a *search decision*.

Variable heuristic Any uninstantiated variable $x \in \mathcal{X}$ can be picked as a candidate. There exist many different variable heuristics. Static heuristics will always pick the variables in the same order. Dynamic heuristics choose the variable depending on the current state of the search. For instance, the dynamic `dom` heuristic selects the variable which has currently the smallest domain. A good choice for the variable heuristic depends strongly on the problem we want to solve.

Value heuristic Once we have chosen the variable, we must choose the value to assign or refute. The choice of the value also depends strongly on the problem. An example of a good value for a COP is one that increases (decreases) the objective we want to maximize (minimize).

The CSP may become inconsistent when we reduce the domain of a variable. The propagators thus check their associated constraint if the domain of a variable from its scope was reduced, and remove inconsistent values accordingly. Once this is done, another variable domain is reduced and propagation removes again the inconsistent values. We continue this way until all variables are instantiated and form a solution or until the domain of a variable becomes empty.

Failure and backtracking If the domain of a variable becomes empty after the propagation of a search decision, then no solution can be found and the decision must be revoked. We say that the search has *failed* and the solver *backtracks* (i.e. returns) to the state it was before taking that decision. Once the solver is in its previous state, it ensures to never make the same wrong decision in the future. As an example, supposing that the solver fails right after assigning a to x, the solver will backtrack to the same state it was before the assignment and will remove a from $\mathcal{D}(x)$.

Copy and trailing Classic CP solvers use one of those two strategies to backtrack information:

- A solver can *copy* its current state before making a search decision. If the decision leads to a failure, then the solver can discard the current state and use the clone to continue the search. This is the strategy used in Gecode [9].
- Instead of copying the whole state, the *trailing* strategy only saves the modifications made on the structures modified during the search. When the solver needs to backtrack, the modifications are reverted to restore the state. Trailing is used in most solvers such as or-tools [10], OscaR [24], and Choco [28].

Reversible structures The main structure that needs to be restored when the search backtracks is the domain of each variable. However, some advanced propagators also maintain important structures that need to be restored each time the solver backtracks to a previous state. In the remainder of this thesis, such a structure will be mentioned as *reversible*. For instance, the GAC-4 algorithm presented in chapter 4 maintains a reversible set for each variable value pair (x, a).

1.2 Bitsets

A bitset, also known as *bit vector*, is a compact structure used to implement a set data structure. A bitset of length n is a sequence of n bits and usually represents a set of size n. The ith element is present in the set if and only if the ith bit is equal to 1. In this thesis, we will say that a bit is *set* if and only if it is equal to 1.

Basic operations, called bitwise operations, can be performed on bitsets. The result of those operations is a new bitset.

NOT operation The NOT operation on a bitset b of length n returns a bitset r of length n such that its ith bit is set if and only if the ith bit of b is not set. The operator associated to the NOT operation is denoted "$\sim$":

$$\sim 01101011 = 10010100$$

AND operation The AND operation between two bitsets b_1 and b_2 of length n returns a bitset r of length n such that its ith bit is set if and only if the ith bits of both b_1 and b_2 are set. The operator associated to the AND operation is denoted "&":

$$01101011 \ \& \ 10100110 = 00100010$$

OR operation The OR operation between two bitsets b_1 and b_2 of length n returns a bitset r of length n such that its ith bit is set if the ith bit of b_1 or b_2 is set. The operator associated to the OR operation is denoted "|":

$$01101011 \mid 00100010 = 01101011$$

1.3 Sparse sets

The sparse set data structure was introduced by Briggs and Torczon in [5]. It allows to represent a set data structure for a fixed-size universe with constant time implementation of important operations such as insertion, removal and member checking.

Representation Consider a set that can contain integer values from 1 to n. The sparse set representation is composed of two arrays[2] $sparse$ and $dense$ of size n as well as an integer $size$ ranging from 0 to n. Each of these arrays contains the different values 1 to n. The ith entry of the $sparse$ array contains the position of i in $dense$, i.e. we have

$$dense[sparse[k]] = k \qquad \forall k \in \{1...n\}$$

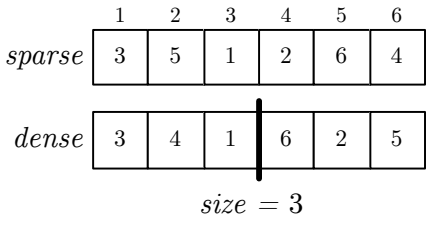
The $size$ integer indicates the current size of the set. Each element i currently in the set is such that its position in $dense$ is smaller or equal to $size$, i.e. i is in the set if $sparse[i] \leq size$. Figure 1.1a illustrates those structures for the set $\mathcal{S} = \{1, 3, 4\}$ that can contain values $\{1...6\}$.

Insertion When a value i is inserted in the set, it is swapped in $dense$ with the value at position $size + 1$, and the corresponding positions in $sparse$ are updated. The $size$ is incremented given that a new element has been inserted. Figure 1.1b shows the operations performed for the insertion of 5 in $\mathcal{S}$. The resulting structures are illustrated in Figure 1.1c.

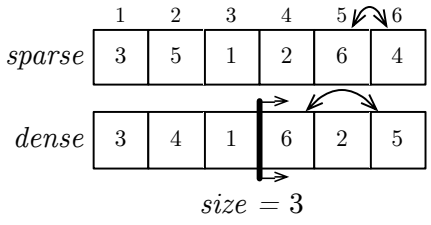
Removal The removal of a value i performs the same kind of operations. We swap i in $dense$ with the value at position $size$, and the corresponding positions in $sparse$ are updated. The $size$ is decremented given that an element has been removed. The removal of 4 from $\mathcal{S}$ is illustrated in Figure 1.1d. The resulting set $\mathcal{S} = \{1, 3, 5\}$ is shown in Figure 1.1e.

Reversibility An important benefit of the sparse set data structure is that it is cheap to trail and restore [7] if the size of the set is monotonically increasing or decreasing during the search. Indeed, consider the sparse set $\mathcal{S} = \{1, 3, 5\}$ in Figure 1.1e after removing 4 from the set. Supposing that the solver backtracks and wants to restore $\mathcal{S}$ to the state it was in Figure 1.1c. Then we only need to restore its previous $size$ to re-add all values that were removed since then. Figure 1.1f illustrates the set after the restoration of $size$.

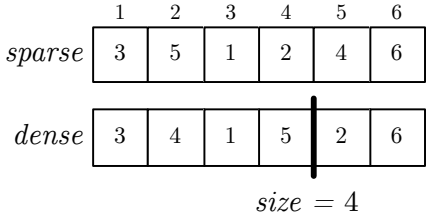
[2]Throughout this thesis, we will suppose that the indexing of an array A starts at 1 and ends at n. The ith entry of the array is denoted $A[i]$.



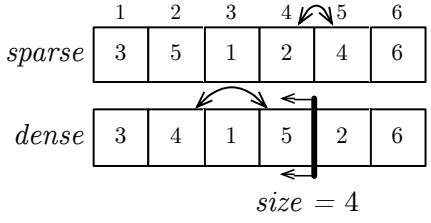
(a) Representation of $\mathcal{S} = \{1, 3, 4\}$



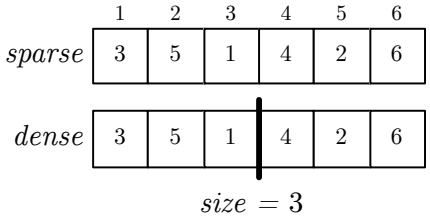
(b) Operations performed to insert 5 in $\mathcal{S}$



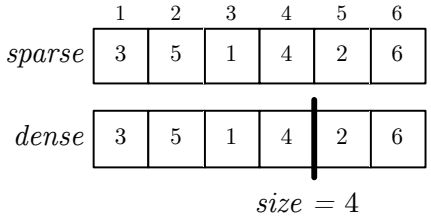
(c) Representation of $\mathcal{S} = \{1, 3, 4, 5\}$



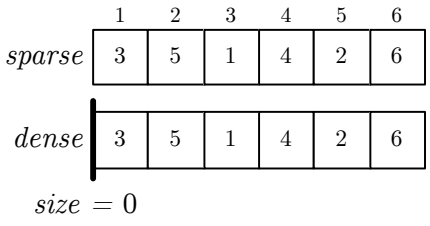
(d) Operations performed to remove 4 from $\mathcal{S}$



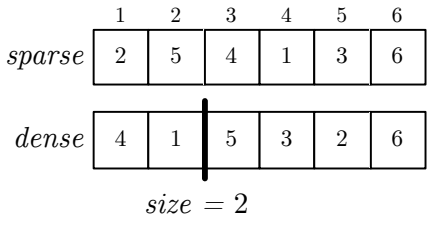
(e) Representation of $\mathcal{S} = \{1, 3, 5\}$



(f) Representation of $\mathcal{S} = \{1, 3, 4, 5\}$ after the restoration of *size*



(g) Representation of $\mathcal{S} = \varnothing$ after clearing it



(h) Representation of $\mathcal{S} = \{1, 4\}$

Figure 1.1: Illustration of the structures used to represent a set $\mathcal{S}$ containing values from the fixed-size universe $\{1...6\}$.

(a) A fully-reduced MDD **(b)** The associated semi-reduced MDD

Figure 1.2: Illustrations of a fully and semi-reduced MDD for $k = n = 2$.

Reset Sparse sets also allow to easily clear a set and restore values previously in the set without losing its reversibility. Indeed, Figure 1.1g shows that clearing $\mathcal{S}$ only requires to set *size* to 0. Supposing that we re-add 4 and then 1 in the set, we obtain $\mathcal{S} = \{1, 4\}$ represented in Figure 1.1h. As we can see by comparing Figures 1.1f and 1.1h, restoring $\mathcal{S}$ to the state it was in Figure 1.1f would only require the restoration of *size* to 4. Clearing a set and re-adding some of its former values is called a *reset*, which is a especially useful strategy introduced in [25] and used to improve most of the best algorithms presented in this thesis.

1.4 Multivalued Decision Diagrams

The definitions of this section are borrowed from [30].

A Multivalued Decision Diagram (MDD) is a directed acyclic graph used to encode a multivalued function $f : \{0...k-1\}^n \to \{0...k-1\}$, based on a given range value $k \in \mathbb{N}$. Given the n possible input variables, the directed acyclic graph representation is designed to contain n levels of nodes, such that each variable is represented at a specific level of the graph. Each node on a given layer has k outgoing edges to nodes in the next (lower) layer of the graph. The final (lowest) layer of the graph is represented by k terminal nodes with values in the range $\{0...k-1\}$.

The function encoded by the decision diagram is evaluated on a given set of n input variables by traversing the graph, starting from the root (i.e., the highest-level variable in the decision diagram), and choosing the outgoing edge at each node corresponding to the input value of the variable represented at the current level of the graph. This traversal continues until a terminal node is reached. The resulting value of the function is indicated by the value of the terminal node reached along this evaluation path.

Fully-reduced MDD A decision diagram is considered to be *fully-reduced* if it does not contain any nodes with all k outgoing edges pointing to the same node and does not contain duplicate nodes for a given level in the directed acyclic graph. An example of fully-reduced MDD for $k = n = 2$ is illustrated in Figure 1.2a.

Semi-reduced MDD If we relax the property that the MDD does not contain any node with all k outgoing edges pointing to the same node, we obtain a *semi-reduced* decision diagram as illustrated in Figure 1.2b.

Chapter 2

Table Constraints

Table constraints are fundamental in Constraint Programming. They can express any type of constraint, which is why finding efficient propagation algorithms is critical. In the remainder of this thesis, we will describe and compare different state-of-the-art algorithms for the table constraint. This chapter defines important associated terms and presents the related work.

2.1 Definitions

A table constraint $\mathcal{C}_{TC}$ is defined by a set of tuples T of arity r. The set of tuples is called a *table* and is denoted by $rel(\mathcal{C}_{TC})$. The constraint $\mathcal{C}_{TC}$ is satisfied if and only if the tuple formed by the variables from its scope $scp(\mathcal{C}_{TC}) = \{x_1, ..., x_r\}$ belongs to the table, i.e. if $(x_1, ..., x_r) \in rel(\mathcal{C}_{TC})$. The ith value of a tuple T is associated to $x_i \in scp(\mathcal{C}_{TC})$ and is denoted by $T[x_i]$. We say that a tuple is *valid* if and only if all of its values belong to the domain of their respective variable, i.e. if $T[x_i] \in \mathcal{D}(x_i)$ for all $x_i \in scp(\mathcal{C}_{TC})$.

Given the definition of the table constraint, a value $a \in \mathcal{D}(x_i)$ is supported w.r.t. $\mathcal{C}_{TC}$ if and only if there is a tuple $T \in rel(\mathcal{C}_{TC})$ such that T is valid and $T[x_i] = a$. We also say that a is *supported* by T and T is a *support* for a.

2.2 Related work

Given the importance of table constraints, a lot of work has been spent to find efficient propagators. This section briefly describes the most important ones. The next chapters give a more detailed description of CT, GAC-4R and MDD-4R [25] that have been implemented in OscaR [24] for the purpose of this thesis.

Generalized Arc-Consistency GAC-3 is a general propagation algorithm and an extension of AC-3 [17] for n-ary constraints. It is a constraint-based algorithm, which means that it receives no specific information when executed. Each call to GAC-3 tests, for each variable value pair (x, a) with $a \in \mathcal{D}(x)$, if a is still supported by a tuple. Unsupported values are removed from their respective domain. Several improvements to fasten the search for a support gave birth to variants such as GAC2001 [4].

GAC-4 [20] is a value-based algorithm. For each variable value pair (x, a), it maintains a *support* set of valid tuples supporting a. Each time a value is removed, all supporting tuples are removed from the associated sets. If the *support* set of (x, a) becomes empty,

then a is removed from $\mathcal{D}(x)$. GAC-4 and its variant GAC-4R will be detailed in Chapter 4.

Simple Tabular Reduction STR2 [14] is an improvement of STR [32], its predecessor. Both are constraint-based algorithms that dynamically maintain the set of valid tuples. Each call to the algorithm removes the invalid tuples from the set. Valid tuples are then used to deduce the supported values of each variable. The improvements brought in STR2 avoid unnecessary operations by considering only a relevant subset of the variables while checking the validity of a tuple.

STR3 [16] is a value-based algorithm totally independent from STR and STR2. For each variable value pair, it first computes an array of tuples that support (x, a). Those arrays are static, they are not maintained during the search. It keeps for each pair a reversible integer *curr* that indicates the position of the last valid tuple in the array. STR3 also maintains the set of valid tuples. Each time a value a is removed from $\mathcal{D}(x)$, all previous supports in the associated array are invalidated and $curr(x, a)$ is updated.

Compressed representations Other algorithms gamble on the compression of the tuples representation to reduce the time needed to ensure domain-consistency. The most promising data structure allowing a more compact representation is the Multivalued Decision Diagram (MDD) [31]. Two notable algorithms using an MDD as main data structure are `mddc` [6] and MDD-4R. The former does not modify the decision diagram and performs a depth-first search of the MDD during propagation to detect which parts of the MDD are consistent or not. MDD-4R dynamically maintains the MDD by deleting nodes and edges that do not belong to a solution. Each variable value pair (x, a) is matched with its corresponding edges in the MDD. When (x, a) has none of its edges present in the MDD, we remove a from $\mathcal{D}(x)$.

Finally, the Compact Table (CT) algorithm implemented in or-tools [10] use bitsets to represent the sets of valid tuples and supports for each variable value pair. The set of valid tuples is recomputed each time the domain of a variable is reduced and the consistency of the variables is checked after all domain reductions have been handled. The following chapter describes this algorithm as well as our own implementation in OscaR.

Chapter 3

Compact Table

This chapter describes the Compact Table algorithm (CT) implemented in or-tools [10] to enforce domain consistency on table constraints. CT maintains a bitset of the valid tuples as well as bitset masks to represent the supports of each variable value pair (x, a). Those are used during propagation to remove inconsistent values from the domain of the variables thanks to bitwise operations. Our experiments show that CT outperforms the current state-of-the-art algorithms STR2, GAC-4R and MDD-4R.

3.1 Principle

As STR2 and STR3, CT dynamically maintains the set of tuples that are valid regarding the current domain of each variable. All tuples are indexed by the order they have in the table given during the initialization of the constraint. Invalid tuples are removed during the initialization as well as values that are not supported by any tuple.

Let n be the number of tuples composing the table of the constraint. A dynamic bitset $validTuples$ of size n represents the validity of the tuples such that its ith bit is set if the ith tuple is still valid, and is equal to 0 otherwise. During the set up of the constraint, CT also computes a static bitset mask of size n for each variable value pair (x, a) such that the ith bit of $mask(x, a)$ is set if and only if the ith tuple is initially a support for (x, a), i.e. if $T_i[x] = a$. Figure 3.1 shows an illustration of those bitsets after the initialization of the following table constraint $\mathcal{C}_{CT}$:

$$\begin{aligned} scp(\mathcal{C}_{CT}) &= \{x, y, z\} \\ rel(\mathcal{C}_{CT}) &= \{(a,a,a), (a,a,b), (a,b,c), (b,a,a), (a,c,b), \\ &\qquad (a,b,b), (b,a,b), (b,b,a), (b,b,b)\} \\ \mathcal{D}(x) &= \{a, b\} \\ \mathcal{D}(y) &= \{a, b, d\} \\ \mathcal{D}(z) &= \{a, b, c\} \end{aligned}$$

The tuple (a, c, b) is initially invalid because $c \notin \mathcal{D}(y)$, and thus will not be indexed. Value d will be removed from $\mathcal{D}(y)$ given that it is not supported by any tuple.

As $validTuples$ is a reversible and dynamically maintained structure, the value of some bits change from 1 to 0 when tuples become invalid or from 0 to 1 when the search backtracks. On the contrary, the masks are only computed at the creation of the constraint and are not maintained during search. It follows from the definition of those bitsets that

T	x	y	z
1	a	a	a
2	a	a	b
3	a	b	c
4	b	a	a
	a	c	b
5	a	b	b
6	b	a	b
7	b	b	a
8	b	b	b

(a) The indexed tuples

$validTuples$	1	1	1	1	1	1	1	1
$mask(x, a)$	1	1	1	0	1	0	0	0
$mask(x, b)$	0	0	0	1	0	1	1	1
$mask(y, a)$	1	1	0	1	0	1	0	0
$mask(y, b)$	0	0	1	0	1	0	1	1
$mask(y, d)$	0	0	0	0	0	0	0	0
$mask(z, a)$	1	0	0	1	0	0	1	0
$mask(z, b)$	0	1	0	0	1	1	0	1
$mask(z, c)$	0	0	1	0	0	0	0	0

(b) The corresponding bitsets

Figure 3.1: Illustration of the data structures after the initialization of $\mathcal{C}_{CT}$. The tuple (a, c, b) will not be indexed and d will be removed from $\mathcal{D}(y)$.

(x, a) has a valid support if and only if

$$(validTuples \ \& \ mask(x, a)) \neq 0_n \tag{3.1}$$

where & stands for the AND bitwise operator and 0_n for the bitset of size n whose bits are all equal to 0. Therefore, each time a tuple becomes invalid, the constraint must check this condition for every variable value pair (x, a) such that $a \in \mathcal{D}(x)$, and remove a from $\mathcal{D}(x)$ if the condition is not satisfied any more. This leads to the propagation algorithm described in the next section.

3.2 Propagation

Values are removed from the domain of some variables during the search of a solution, which can lead to inconsistent values in the domain of other variables. CT ensures the consistency of the constraint in two main steps:

1. It updates *validTuples* so that only valid tuples have their associated bit set to 1.
2. For each variable, it reviews each value in the domain of the variable and checks whether Equation (3.1) is satisfied or not. If it is not, the value must be removed from the domain. As soon as one domain becomes empty, the solver fails and backtracks to a previous state.

3.2.1 Update of *validTuples*

Every time one or more values are removed from any single variable $x \in scp(\mathcal{C}_{CT})$, CT calls the function `update(x)` to update *validTuples* by considering the set of values removed from $\mathcal{D}(x)$ since the last call to `update(x)`. Let's call this set Δ_x. The new value of *validTuples* is obtained by setting to 0 every bit i such that $T_i[x] = a$, for each $a \in \Delta_x$. Indeed, those tuples are no longer valid given that they were supported by (x, a) and a does not belong to $\mathcal{D}(x)$ any more. For each $a \in \Delta_x$, this is performed using two bitwise operations:

$$validTuples = validTuples \ \& \ (\sim mask(x, a))$$

where & and ~ are respectively the AND and NOT bitwise operators. Once the new value of *validTuples* is computed after considering all values in Δ_x, we stand in one of the two following cases:

1. *validTuples* has changed compared to its value when `update(x)` was called. This means that some tuples have been invalidated because of the removal of the values from $\mathcal{D}(x)$. Therefore we have to ensure that every value $b \in \mathcal{D}(y)$ for each variable $y \in scp(\mathcal{C}_{CT})$ is still supported by a valid tuple.

2. *validTuples* has not changed. This means that the tuples invalidated by $a \in \Delta_x$ were already invalid.

In the latter case, nothing more needs to be done and the `update(x)` function terminates. In the former case, a consistency check will be performed by the `propagate` function described in the next section. Before this check, we can also verify that $validTuples \neq 0_n$ to ensure that there is at least one valid tuple remaining. The complete algorithm for the `update` function is given in Algorithm 3.1.

Algorithm 3.1 The `update` function for CT

Pre: $\mathcal{C}_{CT}$ is a CT table constraint
Pre: $x \in scp(\mathcal{C}_{CT})$
Pre: Δ the set of values removed from $\mathcal{D}(x)$ since the last call to UPDATE($\mathcal{C}_{CT}$, x, Δ)
Post: invalidate all tuples supported by (x, a) for each $a \in \Delta$
Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
1:  function UPDATE(C_CT: constraint, x: variable, Δ: set of values)
2:      prevValid ← COPY(C_CT.validTuples)
3:      for all a in Δ do
4:          C_CT.validTuples ← C_CT.validTuples & (~C.mask(x, a))
5:      if prevValid ≠ C_CT.validTuples then
6:          if validTuples = 0_n then
7:              return false
8:          else
9:              return PROPAGATE(C_CT)
10:     return true
```

3.2.2 Consistency check

When the bitset *validTuples* is modified, CT has to check whether the values currently in the domain of each variable $x \in scp(\mathcal{C})$ have at least one valid support, and delete the inconsistent values accordingly. The `propagate` function thus evaluates if the condition of equation (3.1) is satisfied for each variable value pair (x, a) such that $x \in scp(\mathcal{C})$ and $a \in \mathcal{D}(x)$, and remove a from $\mathcal{D}(x)$ if it is not. The `propagate` function is shown in Algorithm 3.2.

3.2.3 Example

As an example, suppose that a is removed from $\mathcal{D}(x)$ after the initialization of the previous table constraint $\mathcal{C}_{CT}$. Given that the domain of x is reduced, the function `update(x)` of CT is called and all tuples supporting any value in $\Delta_x = \{a\}$ will be invalidated. Figure

Algorithm 3.2 The `propagate` function for CT

Pre: $\mathcal{C}_{CT}$ is a CT table constraint
Post: remove all values that are not supported by a tuple with its bit set in $validTuples$
Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
1: function PROPAGATE(C_CT: constraint)
2:     for all x in scp(C_CT) do
3:         for all a in D(x) do
4:             if (validTuples & mask(x, a)) = 0_n then
5:                 remove a from D(x)
6:                 if D(x) = ∅ then
7:                     return false
8:     return true
```

3.2b illustrates the intermediary bitsets used to compute the new value of $validTuples_{out}$ from its old value $validTuples_{in}$ and $mask(x, a)$. Given that its new value differs from the old one, a consistency check has to be performed by the `propagate` function.

During this check, `propagate` computes for each variable value pair (x, a) (with $a \in \mathcal{D}(x)$) the intersection of the associated mask and the current $validTuples$

$$support(x, a) = (validTuples \ \& \ mask(x, a))$$

as shown in Figure 3.2c. Given that $support(z, c) = 0_n$, `propagate` removes c from $\mathcal{D}(z)$ which will in its turn trigger a call to `update(z)`.

Figure 3.2d illustrates the computation of the new value of $validTuples$ by `update(z)`. Given that $validTuples$ does not change, there is no need for a consistency check and the function terminates.

3.3 Improvements

The algorithm of Section 3.2 can be greatly improved, which leads to the efficient algorithm implemented in or-tools.

3.3.1 Delayed consistency check

First we easily notice that there are as many calls to `propagate` as there are to `update`, which is not necessary. Indeed, let's suppose t different variables have their domain reduced at the same time, e.g. by an other constraint or by a search decision. If none of the variables has its domain being emptied during the process, there will be t calls to `update` and `propagate`:

- `update(`x_1`,` Δ_{x_1}`)`
- `propagate()`
- `update(`x_2`,` Δ_{x_2}`)`
- `propagate()`
- ...
- `update(`x_t`,` Δ_{x_t}`)`
- `propagate()`

T	x	y	z
1	a	a	a
2	a	a	b
3	a	b	c
4	b	a	a
5	a	b	b
6	b	a	b
7	b	b	a
8	b	b	b

(a) The indexed tuples

$validTuples_{in}$	1	1	1	1	1	1	1	1
$mask(x, a)$	1	1	1	0	1	0	0	0
$\sim mask(x, a)$	0	0	0	1	0	1	1	1
$validTuples_{out}$	0	0	0	1	0	1	1	1

(b) `update(x)` invalidate tuples supporting (x, a)

$validTuples$	0	0	0	1	0	1	1	1
$support(x, b)$	0	0	0	1	0	1	1	1
$support(y, a)$	0	0	0	1	0	1	0	0
$support(y, b)$	0	0	0	0	0	0	1	1
$support(z, a)$	0	0	0	1	0	0	1	0
$support(z, b)$	0	0	0	0	0	1	0	1
$support(z, c)$	0	0	0	0	0	0	0	0

(c) Consistency check by `propagate` that will remove c from $\mathcal{D}(z)$

$validTuples_{in}$	0	0	0	1	0	1	1	1
$mask(z, c)$	0	0	1	0	0	0	0	0
$\sim mask(z, c)$	1	1	0	1	1	1	1	1
$validTuples_{out}$	0	0	0	1	0	1	1	1

(d) `update(z)` does not invalidate any tuple

Figure 3.2: Illustration of the bitset operations performed by `update` and `propagate` after the removal of a from $\mathcal{D}(x)$.

However, given that multiple calls to `update` invalidate more and more tuples, checking the consistency only after the last one is sufficient to remove inconsistent values from the variables domains. CT can thus call `propagate` only once after every `update` has been handled. This can be achieved in different ways, depending strongly on the solver for which the constraint is implemented. Section 3.4.2 describes our implementation of this improvement in OscaR, the solver used for the experiments in Chapter 6.

3.3.2 Reset

The idea of *reset* has been proposed in [25] and successfully applied to GAC-4 and MDD-4. Instead of studying only the removal of values to maintain the data structures, in our case *validTuples*, we can rebuild it from scratch (*reset* it) if it allows to perform less operations. When handling the removal of a set of values Δ_x from variable x in `update(x)`, we actually have two choices to compute the new value of *validTuples*:

1. set to 0 each bit i in *validTuples* such that the associated tuple T_i is a support for (x, a) and a has just been removed from $\mathcal{D}(x)$, i.e. $T_i[x] = a$ and $a \in \Delta_x$. This is what we did in Section 3.2.1.

2. for all bit i in *validTuples*, set its value to 1 if and only if the associated tuple T_i is still valid and is a support of (x, a) for each $a \in \mathcal{D}(x)$, i.e. if the ith bit of *validTuples* is set and $T_i[x] = a$.

In the former, let's suppose we have computed the bitset *tempDelta* of length n such that its ith bit is set if $T_i[x] \in \Delta_x$. Then we can deduce the new value of *validTuples* thanks to two bitwise operations:

$$validTuples = validTuples \ \& \ (\sim tempDelta)$$

In the latter, following the same logic, let's suppose we have the bitset *tempDomain* of length n such that its ith bit is set if $T_i[x] \in \mathcal{D}(x)$. One bitwise operation is sufficient to get the new value of *validTuples*:

$$validTuples = validTuples \ \& \ tempDomain$$

It remains to compute *tempDelta* or *tempDomain* according to which alternative is the best. This can be achieved using the masks computed by CT during the initialization:

tempDelta $\leftarrow 0_n$ *// or tempDomain*
for all a **in** Δ_x **do** *// or* a ***in*** $\mathcal{D}(x)$
    tempDelta $\leftarrow$ *tempDelta* $\mid mask(x, a)$

We can easily see that deducing the new value of *validTuples* takes $|\Delta_x| + 2$ bitwise operations with the first method and $|\mathcal{D}(x)| + 1$ operations with the other. As a result, it is better to use the first one if $|\Delta_x| < |\mathcal{D}(x)|$ and the second otherwise.

3.3.3 Avoiding unnecessary operations

The third improvement is inspired by a refinement brought to STR by [14] and can be achieved only if the delayed consistency check is implemented, which we will suppose is the case. We know that when `propagate` is called, all values removed from the

variables domains by other constraints (or search decisions) have already been handled by the `update` function, and thus the tuples associated to those removed values have been invalidated. As a reminder, the goal of `propagate` is to check for each variable value pair (x, a) whether a has still a valid support or not. Supposing that there is a pair (x, a) such that $a \in \mathcal{D}(x)$ and a is not supported any more, then we have

$$(validTuples \ \& \ mask(x, a)) = 0_n \tag{3.2}$$

and a needs to be removed from $\mathcal{D}(x)$. However, the removal of a (and maybe other values) from $\mathcal{D}(x)$ will later trigger `update(x)` that will review each removed value and try to compute the new value of *validTuples* in $|\Delta_x| + 2$ operations, as we have seen in the previous section. Unfortunately, those operations are useless given that every tuple supporting those values are already invalidated, otherwise equation (3.2) would not be satisfied. To avoid unnecessary computation, we can use a reversible integer $lastSize(x)$ for each $x \in scp(\mathcal{C})$ in which we store the size of each variable domain at the end of the last `propagate`. When updating *validTuples* in `update(x)`, we can stop immediately if no value has been removed from x after the last call to `propagate`, i.e. if $lastSize(x) = |\mathcal{D}(x)|$.

3.3.4 Last modified variable

The last improvement comes from the observation that it is not necessary to look at a variable during a consistency check if this was the only modified variable since the last check. Indeed, let us assume that right after a call to `propagate`, some values are removed from a variable x and `update(x)` is triggered. The value of *validTuples* will be computed thanks to $\mathcal{D}(x)$ or Δ_x. If there are no other `update` call to handle, i.e. if no other variable was modified, a consistency check will be launched by `update(x)`. However, given that *validTuples* is based only on values from $\mathcal{D}(x)$, we can omit the consistency check of $\mathcal{D}(x)$.

The final algorithm of Compact Table with those improvements is shown in Algorithm 3.3. The reset is implemented at lines 20–23 and the delayed propagation at line 28. For each variable, *lastSize* is updated at line 9 and is used at lines 12–13. Finally, we do not check x if it was the only modified variable since last `propagate` at line 3.

3.3.5 Example

To illustrate this improved algorithm, let's simulate the operations performed by CT on the following table constraint $\mathcal{C}_{CT}$ right after that a is assigned to x.

$$\begin{aligned} scp(\mathcal{C}_{CT}) &= \{x, y, z\} \\ rel(\mathcal{C}_{CT}) &= \{(a, b, a), (b, a, c), (c, c, b), \\ &\qquad (a, a, b), (b, c, a), (c, b, c)\} \\ \mathcal{D}(x) = \mathcal{D}(y) &= \mathcal{D}(z) = \{a, b, c\} \end{aligned}$$

When a is assigned to x, `update(x)` handles the values removed from x, i.e. $\Delta_x = \{b, c\}$. Figure 3.3c shows the computation of *validTuples* thanks to *tempDomain*. Indeed, `update` detects that using *tempDomain* will take $|\mathcal{D}(x)| + 1 = 2$ bitwise operations which is much better than $|\Delta_x| + 2 = 4$ operations with *tempDelta*. Given that *validTuples*

Algorithm 3.3 The final `propagate` and `update` functions for CT

Pre: $\mathcal{C}_{CT}$ is a CT table constraint
Post: remove all values that are not supported by a tuple with its bit set in $validTuples$
Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
1:  function PROPAGATE(C_CT: constraint)
2:      for all x in scp(C_CT) do
3:          if x is not the only variable modified since last propagate then
4:              for all a in D(x) do
5:                  if (validTuples & mask(x, a)) = 0_n then
6:                      remove a from D(x)
7:                      if D(x) = ∅ then
8:                          return false
9:          C_CT.lastSize(x) ← |D(x)|
10:     return true
```

Pre: $\mathcal{C}_{CT}$ is a CT table constraint
Pre: $x \in scp(\mathcal{C}_{CT})$
Pre: Δ the set of values removed from $\mathcal{D}(x)$ since the last call to UPDATE($\mathcal{C}_{CT}$, x, Δ)
Post: invalidate all tuples supported by (x, a) for each $a \in \Delta$
Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
11: function UPDATE(C_CT: constraint, x: variable, Δ: set of values)
12:     if C_CT.lastSize(x) = |D(x)| then
13:         return true
14:     prevValid ← COPY(C_CT.validTuples)
15:     temp ← 0_n
16:     if |Δ_x| < |D(x)| then
17:         for all a in Δ_x do
18:             temp ← temp | mask(x, a)
19:         validTuples ← validTuples & (~temp)
20:     else
21:         for all a in D(x) do
22:             temp ← temp | mask(x, a)
23:         validTuples ← validTuples & temp
24:     if prevValid ≠ C_CT.validTuples then
25:         if validTuples = 0_n then
26:             return false
27:         else
28:             if there are no more calls to update in the propagation queue then
29:                 return PROPAGATE(C_CT)
30:     return true
```

T	x	y	z
1	a	b	a
2	b	a	c
3	c	c	b
4	a	a	b
5	b	c	a
6	c	b	c

(a) The indexed tuples

$validTuples$	1	1	1	1	1	1
$mask(x, a)$	1	0	0	1	0	0
$mask(x, b)$	0	1	0	0	1	0
$mask(x, c)$	0	0	1	0	0	1
$mask(y, a)$	0	1	0	1	0	0
$mask(y, b)$	1	0	0	0	0	1
$mask(y, c)$	0	0	1	0	1	0
$mask(z, a)$	1	0	0	0	1	0
$mask(z, b)$	0	0	1	1	0	0
$mask(z, c)$	0	1	0	0	0	1

(b) The corresponding bitsets

$validTuples_{in}$	1	1	1	1	1	1
$mask(x, a)$	1	0	0	1	0	0
$tempDomain$	1	0	0	1	0	0
$validTuples_{out}$	1	0	0	1	0	0

(c) `update(x)` changes the value of $validTuples$

$validTuples$	1	0	0	1	0	0
$support(y, a)$	0	0	0	1	0	0
$support(y, b)$	1	0	0	0	0	0
$support(y, c)$	0	0	0	0	0	0
$support(z, a)$	1	0	0	0	0	0
$support(z, b)$	0	0	0	1	0	0
$support(z, c)$	0	0	0	0	0	0

(d) Consistency check by `propagate` that removes c from $\mathcal{D}(y)$ and $\mathcal{D}(z)$

Figure 3.3: Illustration of the bitset operations performed by `update` and `propagate` after a is assigned to x.

has changed and no more call to `update` needs to be handled, a consistency check by `propagate` is triggered. Figure 3.3d illustrates the bitsets $support(x, a)$ for each variable value pair (x, a). As explained in Section 3.3.4, there is no need to compute these bitsets for variable x given that it was the only variable modified before the consistency check. Value c will be removed from both $\mathcal{D}(y)$ and $\mathcal{D}(z)$ seeing that $support(y, c) = support(z, c) = 0_n$. Finally, `update(y)` and `update(z)` are triggered but terminate immediately because $lastSize(y)$ and $lastSize(z)$ have not changed since the last consistency check.

3.4 Implementation in OscaR

This section describes the choices made for our implementation of the Compact Table in OscaR, the open source CP solver used for our experiments. OscaR is developed in Scala [22], an object-functional programming language designed to be compiled to Java bytecode and interpreted by the Java Virtual Machine (JVM) [2].

3.4.1 Bitset

The most important data structure in this algorithm is the bitset and its associated bitwise operations. A bitset is implemented as an array of `Long` integers, i.e. an `Array[Long]`. A `Long` is represented by a 64-bit signed integer and is equivalent to Java's `long` primitive type [1], which is the longest bit sequence for a primitive type. Consequently, a bitset of size n is implemented as an `Array[Long]` of size $\lceil n \ / \ 64 \rceil$ and the ith bit of this bitset

is matched with the $(i \ \% \ 64 + 1)$th bit of the $(i \ / \ 64 + 1)$th `Long` of the array.

Bitwise operations To compute the result of a bitwise operation between two bitsets of size n, we thus have to apply the operation between each `Long` that have the same position in their respective array, e.g. in Scala for a bitwise AND operation:

```
def bitwiseAnd(x: Array[Long], y: Array[Long]): Array[Long] = {
        val result = new Array[Long](x.length)
        var i = 0
        while (i < x.length) {
                result[i] = x[i] & y[i]
                i += 1
        }
        return result
}
```

3.4.2 Update and delayed propagation

In order to ensure domain consistency, OscaR allows a constraint to be informed when the domain of a variable is modified in many different ways:

- Each time a value a is removed from one of its variables x, a constraint can ask its function $valRemove(x, a)$ to be triggered. This could be used as the `update` function of CT, but it would directly imply $|\Delta_x| = 1$ which is not a good solution if we want to take advantage of the reset improvement described in Section 3.3.2.

- A constraint can *subscribe* to its variables so that the *propagate*() function is called whenever one or more of these variables are modified. This is exactly what we used for the `propagate` function of CT that checks the consistency of the domains.

- Variables can also be *tracked* by a constraint so that its function *propagate*(*delta*) is triggered when one of its variables x is modified. The *delta* argument is the set of values removed from x since the last time this same function was called. This feature fits perfectly for the `update` function of CT that we use to update the new value of *validTuples*. The computation of *delta* if efficiently managed by OscaR and described in [7].

As explained in Section 3.3.1, CT is more efficient if calls to `propagate` are postponed and activated only after all `update` calls have been handled. Fortunately, OscaR allows to define a priority for each of the functions that are triggered after a change occurred on a variable. Hence, it is sufficient to set the priority of *propagate*(*delta*) higher than the priority of *propagate*() so that all `update` calls will be executed before the consistency checks by `propagate`.

Chapter 4

GAC-4 and GAC-4R

This chapter describes the GAC-4 algorithm and its variant GAC-4R. GAC-4 dynamically maintains the set of tuples supporting each variable value pair (x, a). GAC-4R improves GAC-4 thanks to the reset and recomputation of the supports when necessary.

Both algorithms were implemented in OscaR using reversible sparse sets to represent the supports of each (x, a) pair. Our experiments will show that GAC-4 is competitive with the state-of-the-art algorithms STR2 and MDD-4R, and that the reset idea makes GAC-4R one of the most efficient algorithm as of writing.

4.1 GAC-4

Unlike CT and STR2, GAC-4 does not explicitly maintain the set of valid tuples. Instead, for each variable value pair (x, a), the set $supports(x, a)$ contains the index of the tuples supporting (x, a) that are still valid, i.e. $i \in supports(x, a)$ if $T_i[x] = a$ and $T_i[y] \in \mathcal{D}(y)$ for every $y \in scp(\mathcal{C})$. The idea behind the algorithm is simple:

1. Each time a value a is removed from its associated variable x, every tuple in $supports(x, a)$ is invalidated.

2. When a tuple T is invalidated, it is removed from the associated *supports* sets, i.e. from $supports(x_i, T[x_i])$ for all $x_i \in scp(\mathcal{C})$. If a set $supports(x_i, T[x_i])$ becomes empty, then $T[x_i]$ is removed from $\mathcal{D}(x_i)$.

The tuples are indexed by the order they have in the table given at the creation of the constraint. Invalid tuples as well as unsupported values are removed during the initialization of the sets. Figure 4.1 illustrates each *supports* set after the initialization of the following constraint $\mathcal{C}_{GAC4}$:

$$\begin{aligned} scp(\mathcal{C}_{GAC4}) &= \{x, y, z\} \\ rel(\mathcal{C}_{GAC4}) &= \{(a,a,a),(a,a,b),(a,b,c),(b,a,a),(a,c,b), \\ &\qquad (a,b,b),(b,a,b),(b,b,a),(b,b,b)\} \\ \mathcal{D}(x) &= \{a, b\} \\ \mathcal{D}(y) &= \{a, b, d\} \\ \mathcal{D}(z) &= \{a, b, c\} \end{aligned}$$

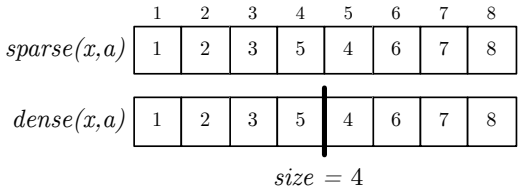
T	x	y	z
1	a	a	a
2	a	a	b
3	a	b	c
4	b	a	a
	a	c	b
5	a	b	b
6	b	a	b
7	b	b	a
8	b	b	b

(a) The indexed tuples

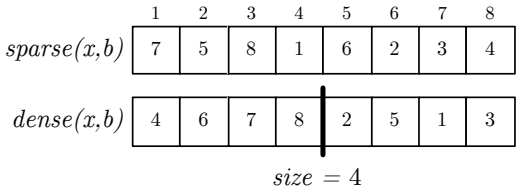
$$
\begin{aligned}
supports(x,a) &= \{1,2,3,5\} \\
supports(x,b) &= \{4,6,7,8\} \\
supports(y,a) &= \{1,2,4,6\} \\
supports(y,b) &= \{3,5,7,8\} \\
supports(y,d) &= \varnothing \\
supports(z,a) &= \{1,4,7\} \\
supports(z,b) &= \{2,5,6,8\} \\
supports(z,c) &= \{3\}
\end{aligned}
$$

(b) The corresponding sets

Figure 4.1: Illustration of the sets after the initialization of $\mathcal{C}_{GAC4}$. The tuple (a, c, b) will not be indexed and d will be removed from $\mathcal{D}(y)$.



(a) Structures for $supports(x, a)$



(b) Structures for $supports(x, b)$

Figure 4.2: Example of the sparse sets structures for the supports of (x, a) and (x, b).

4.1.1 Representation

As presented in Chapter 1, the reversible sparse set data structure allows to represent reversible sets of a fixed-size universe efficiently [5]. As a reminder, the reversible sparse set is composed of a reversible integer *size* and two arrays *sparse* and *dense*. The *sparse* array contains, for each element of the set, its position in *dense*. The number of elements currently in the set is *size*. Each member of the set is such that its position in *dense* is inferior or equal to *size*, i.e. k belongs to the set if $sparse[k] \leq size$. As explained in [7], this representation allows a cheap restoration of the values in the set upon backtracking, which makes it a perfect candidate for our *supports* sets. Figure 4.2 shows an illustration of those sets for the variable x of the previously initialized constraint $\mathcal{C}_{GAC4}$.

Shared sparse sets Assuming that the number of tuples in the table is n, each *supports* set is represented by a reversible integer and two arrays of size n. However, given that a tuple T_i can support only one value for each variable $x \in scp(\mathcal{C})$, its index i can be only in one *supports* set associated to x. For instance, the tuple $T_3 = (a, b, c)$ can be in either $supports(x, a)$, $supports(y, b)$ or $supports(z, c)$ but it will never be in $supports(x, b)$ or $supports(y, a)$ given that a single tuple can only support one value per variable. As explained in [19], this allows us to share the *sparse* array between all sparse sets associated to the same variable, and therefore to spare memory. Figure 4.3 illustrates this improved representation for the *supports* sets associated to the variable x in $\mathcal{C}_{GAC4}$.

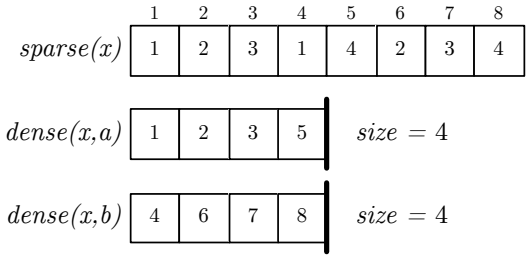


Figure 4.3: Example of the shared sparse sets structures for the supports of (x, a) and (x, b).

4.1.2 Maintenance of the supports

Each time a value a is removed from a variable domain $\mathcal{D}(x)$, all tuples in $supports(x, a)$ are no longer supported and they must be invalidated. This is the goal of the function `valRemove(x,a)` that is called whenever a value a is deleted from $\mathcal{D}(x)$.

When we invalidate a tuple T_i, we remove i from all associated *supports* sets, i.e. we remove i from $supports(y, T_i[y])$ for all $y \in scp(\mathcal{C})$. Whenever a set $support(y, T_i[y])$ becomes empty we remove $T_i[y]$ from $\mathcal{D}(y)$.

Algorithm 4.1 shows the function `valRemove(x,a)` as well as `invalidateTuple(i)`, which invalidates the ith tuple.

Algorithm 4.1 The `valRemove` and `invalidateTuple` functions for GAC-4.

Pre: $\mathcal{C}_{GAC4}$ is a GAC-4 table constraint
Pre: $x \in scp(\mathcal{C}_{GAC4})$
Pre: a a value removed from $\mathcal{D}(x)$
Post: invalidate all tuples supporting (x, a)
Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
1: function valRemove(C_GAC4: constraint, x: variable, a: integer)
2:     for all i in C_GAC4.supports(x, a) do
3:         if invalidateTuple(C_GAC4, i) = false then
4:             return false
5:     return true
```

Pre: $\mathcal{C}_{GAC4}$ is a GAC-4 table constraint
Pre: i the index of a tuple to invalidate
Post: remove i from all *supports* sets associated to T_i
Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
6: function invalidateTuple(C_GAC4: constraint, i: integer)
7:     for all x in scp(C_GAC4) do
8:         remove i from supports(x, T_i[x])
9:         if D(x) = ∅ then
10:            return false
11:    return true
```

T	x	y	z
1	a	a	a
2	a	a	b
3	a	b	c
4	b	a	a
5	a	b	b
6	b	a	b
7	b	b	a
8	b	b	b

(a) The indexed tuples

$$\begin{aligned} supports(x,a) &= \{2,3,5\} \\ supports(y,a) &= \{2,4,6\} \\ supports(z,a) &= \{4,7\} \end{aligned}$$

(b) The *supports* sets modified by the invalidation of T_1

$$\begin{aligned} supports(x,a) &= \{3,5\} \\ supports(y,a) &= \{4,6\} \\ supports(z,b) &= \{5,6,8\} \end{aligned}$$

(c) The *supports* sets modified by the invalidation of T_2

$$\begin{aligned} supports(x,a) &= \{5\} \\ supports(y,b) &= \{5,7,8\} \\ supports(z,c) &= \varnothing \end{aligned}$$

(d) The *supports* sets modified by the invalidation of T_3 which removes c from $\mathcal{D}(z)$

$$\begin{aligned} supports(x,a) &= \varnothing \\ supports(y,b) &= \{7,8\} \\ supports(z,b) &= \{5,6,8\} \end{aligned}$$

(e) The *supports* sets modified by the invalidation of T_5. Nothing happens given that $a \notin \mathcal{D}(x)$

Figure 4.4: Illustration of the different *supports* sets modified by the invalidation of each tuple supporting (x, a).

4.1.3 Example

As an example, let's suppose that a is removed from $\mathcal{D}(x)$ after the initialization of the $\mathcal{C}_{GAC4}$ constraint introduced earlier. When this happens, the function `valRemove(x,a)` is triggered to invalidate each tuple supporting (x, a), i.e. the tuples T_i such that $i \in supports(x, a) = \{1, 2, 3, 5\}$. Figure 4.4 illustrates, for each of those tuples, the different *supports* sets modified by `invalidateTuple`. When the tuple T_3 is invalidated in Figure 4.4d, $supports(z, c)$ becomes empty and c is thus removed from $\mathcal{D}(z)$. Figure 4.4e shows that $supports(x, a)$ becomes empty when invalidating its last tuple T_5, as expected.

4.2 GAC-4R

GAC-4 was one of the first algorithm to benefit from the reset improvement [25]. This section briefly describes GAC-4R, the improved version of GAC-4 that we also implemented in OscaR.

Before invalidating all tuples in $supports(x, a)$ for all pairs (x, a) such that a was removed from $\mathcal{D}(x)$, we first check the number of tuples that will be invalidated. If we invalidate more than half the number of tuples currently valid, then it is better to clear all *supports* sets and re-fill them thanks to the tuples that are still valid. Algorithm 4.2

shows the `reviseGAC-4R` function of GAC-4R.[1] In practice, the modifications brought to GAC-4 are as follows.

- Instead of calling `valRemove(x,a)` each time a value a is removed from $\mathcal{D}(x)$, GAC-4R calls `reviseGAC-4R(deletionSet)` when one or more of its variables are modified. This is the main function of the algorithm that determines the operations to perform. The `deletionSet` argument is the set of variable value pairs (x, a) such that a was removed from $\mathcal{D}(x)$ since the last call to `reviseGAC-4R`.

- Given that we use the set of valid tuples to recompute the supports, we have to maintain a reversible set *validTuples* of the tuples currently valid which is updated each time we invalidate a tuple.[2]

- Each time `reviseGAC-4R(deletionSet)` is triggered we compute for each variable x the number of tuples $\#\Delta T(x)$ that will be invalidated because of values removed from $\mathcal{D}(x)$ (lines 2-6 of the algorithm):

  for all $x \in scp(\mathcal{C}_{GAC4R})$ **do** $\#\Delta T(x) \leftarrow 0$

  for all $(x, a) \in$ `deletionSet` **do** $\#\Delta T(x) \leftarrow \#\Delta T(x) + |supports(x, a)|$

  $x_{MAX} \leftarrow arg(\max_x\{\#\Delta T(x)\})$

- If $\#\Delta T(x_{MAX}) \leq \dfrac{|validTuples|}{2}$ then we simply invalidate all tuples in $supports(x, a)$ such that $(x, a) \in$ `deletionSet`, as we do with GAC-4. This corresponds to lines 30-39 in the algorithm.

- If $\#\Delta T(x_{MAX}) > \dfrac{|validTuples|}{2}$ then we perform a reset of the data structures (lines 7-28). We first clear the *validTuples* set and we re-compute its new value from $\mathcal{D}(x_{MAX})$ as shown in lines 9-13. Next, we clear all *supports* sets and we fill them with the current valid tuples (lines 14-21). Finally, lines 22-28 remove all unsupported values from their respective domain.

As we will see in Chapter 6, the reset makes GAC-4R one of the most competitive algorithm for table constraints.

[1] This algorithm is taken from Algorithm 4 of [25] and reproduced in this thesis for the reader's convenience.

[2] As we do with STR2.

Algorithm 4.2 The `reviseGAC-4R` function for GAC-4R.

Pre: $\mathcal{C}_{GAC4R}$ is a GAC-4R table constraint

Pre: *deletionSet* the list of variable value pairs (x, a) such that a was removed from $\mathcal{D}(x)$

Post: remove invalid tuples from *validTuples*

Post: remove inconsistent values from the domains of the variables

Post: return **false** if the domain of a variable becomes empty, **true** otherwise

```
1: function reviseGAC-4R(C_GAC4R: constraint, deletionSet: list)
2:     for all x ∈ scp(C_GAC4R) do
3:         #ΔT(x) ← 0
4:     for all (x, a) ∈ deletionSet do
5:         #ΔT(x) ← #ΔT(x) + |C_GAC4R.supports(x, a)|
6:     x_MAX ← arg(max_x{#ΔT(x)})
7:     if #ΔT(x_MAX) > |C_GAC4R.validTuples|/2 then
8:         // Clear the valid tuples
9:         C_GAC4R ← ∅
10:        // Re-fill the valid tuples
11:        for all a ∈ D(x_MAX) do
12:            for all T ∈ supports(x_MAX, a) if T is valid do
13:                add T to C_GAC4R.validTuples
14:        // Clear the supports sets
15:        for all x ∈ scp(C_GAC4R) do
16:            for all a ∈ D(x) do
17:                supports(x, a) ← ∅
18:        // Re-fill the supports sets
19:        for all T ∈ C_GAC4R.validTuples do
20:            for all x ∈ scp(C_GAC4R) do
21:                add T to supports(x, T[x])
22:        // Remove unsupported values
23:        for all x ∈ scp(C_GAC4R) do
24:            for all a ∈ D(x) do
25:                if supports(x, a) = ∅ then
26:                    remove a from D(x)
27:                    if D(x) = ∅ then
28:                        return false
29:    else
30:        // Classical GAC-4 deletion process
31:        for all (x, a) ∈ deletionSet do
32:            for all T ∈ supports(x, a) do
33:                remove T from C_GAC4R.validTuples
34:                for all y ∈ scp(C_GAC4R) do
35:                    remove T from supports(y, T[y])
36:                    if supports(y, T[y]) = ∅ then
37:                        remove T[y] from D(y)
38:                        if D(y) = ∅ then
39:                            return false
    return true
```

Chapter 5

Multivalued Decision Diagrams for Table Constraints

The MDD-4R algorithm was introduced by Perez and Régin in [25]. Unlike `mddc` [6], MDD-4R modifies and restores its associated Multivalued Decision Diagram (MDD) during the search. Multivalued Decision Diagrams allow a compact representation of a table by taking advantage of identical prefixes or suffixes between the tuples. Each path in the MDD corresponds to a valid tuple, and the consistency of the variables is maintained by deleting invalid nodes and edges.

This chapter briefly reminds the fundamentals of the algorithm and describes our implementation in OscaR. The performance of MDD-4R compared to other states-of-the-art algorithms is discussed in Chapter 6.

5.1 Principle

5.1.1 Representation

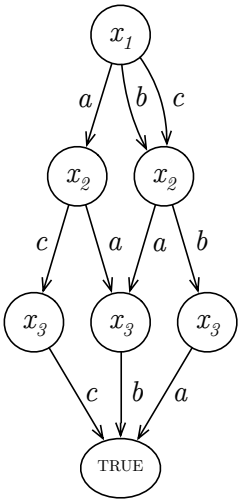
Supposing a table constraint $\mathcal{C}_{MDD}$ of arity r, MDD-4R maintains a semi-reduced MDD[1] with a height equal to $r+1$. Each variable $x_i \in scp(\mathcal{C}_{MDD})$ is indexed by its order in the tuples given at the initialization of the constraint. For $1 \leq i \leq r$, the nodes in the ith layer of the MDD are associated to x_i. The first layer consists of a single node associated to x_1 and the last layer of one single node TRUE. An edge e going from a node in the ith layer to a node in layer $i+1$ is labeled by a value a that x_i can have in one or more tuples. We also say that such an edge is in layer i. Each path from the x_1 node to the TRUE node corresponds to a single valid tuple. Figure 5.1 illustrates a semi-reduced MDD and its corresponding tuples. The construction of an MDD from a table will be discussed in Section 5.3.

In addition to the MDD, the *edges* sets are computed and maintained for each variable value pair (x_i, a). The set $edges(x_i, a)$ contains the index of the edges in the ith layer labeled a and which are still present in the MDD. In other words, $k \in edges(x_i, a)$ if the edge e_k is associated to (x_i, a) and is still part of a path from the x_1 node to the TRUE node. The index of an edge is assigned during the construction of the MDD. Figure 5.2a illustrates the MDD from Figure 5.1 with the edges labeled by their index. The corresponding *edges* sets are shown in Figure 5.2b.

[1] See Section 1.4 for the definition of semi-reduced MDD.

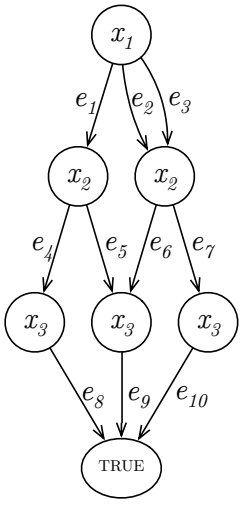
x_1	x_2	x_3
a	a	b
a	c	c
b	a	b
b	b	a
c	a	b
c	b	a

(a) The tuples



(b) The semi-reduced MDD

Figure 5.1: Illustration of a semi-reduced MDD and the associated tuples.

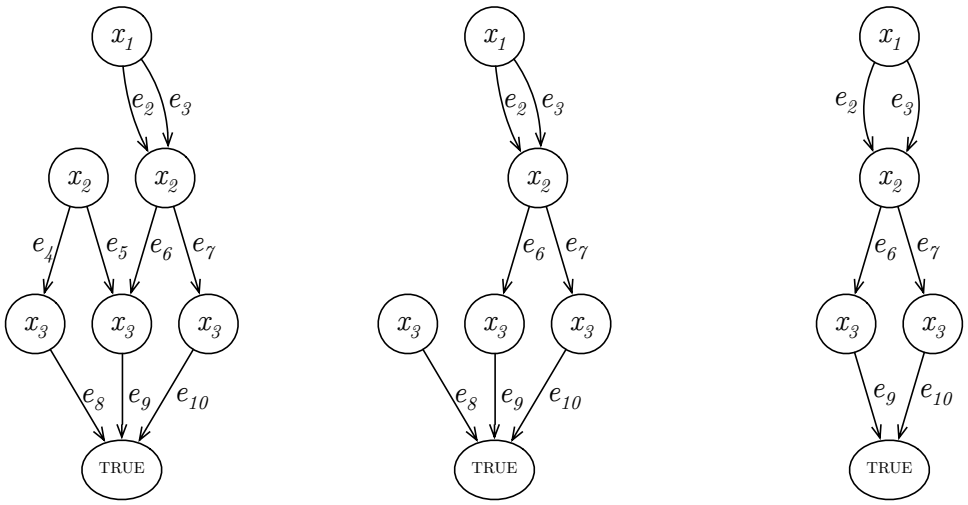


(a) The MDD with the edges labeled by their index

$$
\begin{aligned}
edges(x_1, a) &= \{e_1\} \\
edges(x_1, b) &= \{e_2\} \\
edges(x_1, c) &= \{e_3\} \\
edges(x_2, a) &= \{e_5, e_6\} \\
edges(x_2, b) &= \{e_7\} \\
edges(x_2, c) &= \{e_4\} \\
edges(x_3, a) &= \{e_{10}\} \\
edges(x_3, b) &= \{e_9\} \\
edges(x_3, c) &= \{e_8\}
\end{aligned}
$$

(b) The *edges* sets for each variable value pair (x, a)

Figure 5.2: Illustration of the *edges* sets associated to the MDD of Figure 5.1.



(a) The MDD after removing e_1. This makes the first node in the second layer invalid.

(b) The MDD after removing the invalid node and its adjacent edges e_4 and e_5. We remove c from $\mathcal{D}(x_2)$.

(c) The final MDD after removing the invalid node of the third layer. We remove c from $\mathcal{D}(x_3)$.

Figure 5.3: Illustration of the different states of the MDD during the propagation after a is removed from $\mathcal{D}(x_1)$.

5.1.2 Propagation

When values are removed from the variables, the nodes and edges that became invalid are deleted from the MDD. We call this process the maintenance of the MDD [25]. Each time a value a is removed from a variable domain $\mathcal{D}(x_i)$, we delete in the MDD all associated edges $edges(x_i, a)$.

However, deleting an edge can make a node invalid, i.e. a node that has zero ingoing or outgoing edge. Such a node will never be part of a path from the first layer to the last, hence it needs to be removed. When removing a node, all remaining edges adjacent to that node must be deleted too, which may invalidate other nodes. The process of nodes and edges deletion continues until all nodes have at least one ingoing and outgoing edge. When we delete an edge in the MDD, we also remove it from the associated *edges* set. If a set $edges(x_m, b)$ becomes empty during the propagation, then b is removed from $\mathcal{D}(x_m)$.

5.1.3 Example

As an example, let's keep the table constraint represented in Figure 5.1 and suppose that a is removed from $\mathcal{D}(x_1)$. Figure 5.3 illustrates the different states of the MDD during the propagation. When a is removed from $\mathcal{D}(x_1)$, the edges in $edges(x_1, a) = \{e_1\}$ are deleted. This leads to the MDD shown in Figure 5.3a. We directly see that the first node in the second layer becomes invalid, given that it has no ingoing edges. Therefore, this node is removed from the MDD as well as each of its remaining adjacent edges e_4 and e_5, which gives the MDD of Figure 5.3b. Finally, the first node of the third layer is removed as well as its single outgoing edge e_8. The resulting MDD is shown in Figure 5.3c. During the process, edges e_4, e_5 and e_8 are removed respectively from $edges(x_2, c)$, $edges(x_2, a)$ and $edges(x_3, c)$. Value c is then deleted from $\mathcal{D}(x_2)$ and $\mathcal{D}(x_3)$ given that $edges(x_2, c) = edges(x_3, c) = \varnothing$.

5.1.4 Reset improvement

In practice, MDD-4R handles a set of values Δ_{x_i} removed from $\mathcal{D}(x_i)$ (instead of a single value a) and processes each layer one by one. Indeed, it first deletes all the edges of layer i that are associated to each value $a \in \Delta_{x_i}$. Those deletions can invalidate nodes from either layer i or layer $i+1$. If it does invalidate some nodes of layer i, then some edges between layers $i-1$ and i will need to be removed with those nodes. Those new deletions could in their turn invalidate nodes from layer $i-1$, and so on. The same reasoning is applied with nodes from layer $i+1$. Consequently, the algorithm processes the different layers of the MDD in the following order:

$$i \to i-1 \to i-2 \to \ldots \to 1 \to i+1 \to i+2 \to ... \to r+1$$

The authors of [25] have also introduced the idea of *reset*. It consists of sometimes recomputing the data structures from scratch instead of computing them incrementally. In the case of MDD-4R, we can take advantage of this processing by layer when we have to delete m edges in the ith layer. Supposing that there are a total of M edges before removing m of them, then there will be $M-m$ remaining after the removal. If $M-m < m$ it is more efficient to clear the ith layer and re-add the $M-m$ edges that were not supposed to be deleted.

As an example, let's take again the MDD represented in Figure 5.1 and suppose that a and b are removed from $\mathcal{D}(x_2)$. The associated edges e_5, e_6 and e_7 must be removed from the second layer. However, given that e_4 will be the only remaining edge, it is more efficient to clear the whole layer and re-add e_4 instead of deleting those three edges one by one. The following section describes the data structures that allow to put this improvement in practice.

5.2 Implementation in OscaR

This section describes our implementation of MDD-4R in OscaR. The different data structures and classes were inspired by the implementation of the same algorithm by Régin and Perez in or-tools [10]. The pseudo-code of the algorithm can be found in Algorithm 5 of [25].

5.2.1 Data structures

The most important structure in the algorithm is obviously the Multivalued Decision Diagram. Because we maintain the MDD dynamically, we need a reversible structure that can restore its previous value when the search backtracks. A Multivalued Decision Diagram being a directed graph, many representations could be used. The remainder of this section describes the different structures used in our implementation. The literals N and E denote respectively the number of nodes and edges of the MDD.

Static edges An array *edges* of size E stores the information about the edges of the MDD. The ith entry $edges[i]$ is of type `Edge` and contains the following information about the ith edge e_i:

- *value*, the unique value associated to this edge ;
- *start*, the index of the node that the edge is leaving ;

- *end*, the index of the node in which the edge is entering.

Those data never change during the search, therefore there is no need to use any reversible structure.

Reversible nodes An array *nodes* of size N stores the information about the nodes. The ith entry of the array is of type `Node` and is composed of two different sets associated to the ith node of the MDD:

- *inEdges* the set of edges entering this node ;
- *outEdges* the set of edges leaving this node.

Given that edges are deleted in the MDD by the propagation algorithm, those sets need to be reversible so that we can restore their previous value when the search backtracks. For this reason, we use reversible sparse sets which share their $sparse$ array as explained in Chapter 4. Indeed, each edge i can enter in only one node and leave only one other, which means that we can use only two arrays $sparseIn$ and $sparseOut$ respectively for all *inEdges* and *outEdges* sets.

When the algorithm removes an edge e_i from the MDD, we actually remove it from the associated *inEdges* and *outEdges* sets of its two adjacent nodes. If a node has either its *inEdges* or *outEdges* set empty, the nodes is invalid and all remaining adjacent edges must be removed.

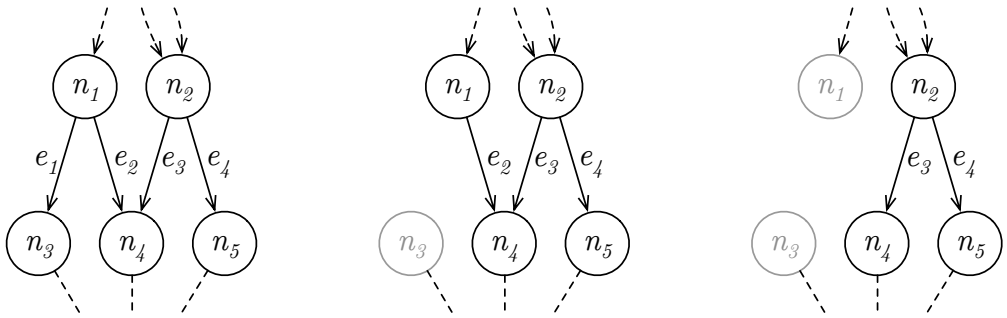
Supports sets As explained in Section 5.1.1, all edges associated to the variable value pair (x_i, a) are stored in the dynamic set $edges(x_i, a)$. When an edge is removed from the MDD, it is also removed from the associated set. If the set $edges(x_i, a)$ becomes empty during the process, then we remove a from $\mathcal{D}(x_i)$. Once again, we can use reversible sparse sets for each $edges(x_i, a)$ and share their $sparse$ array, given that an edge can be in at most one of them.

Layer nodes The algorithm also maintains $r + 1$ additional sets $layerNodes(i)$ used to store the nodes currently valid for each layer i. Those sets, also implemented with a reversible sparse set data structure, are especially useful to collect the nodes that have been invalidated when we delete some edges or when we reset an entire layer. The example in the following section illustrates the use of this structure during propagation.

5.2.2 Example

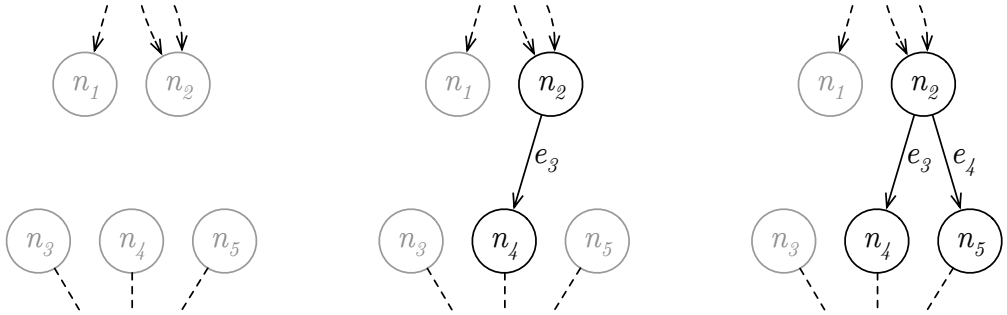
Let's take as an example the MDD partially represented in Figure 5.4a. The nodes and the edges are labelled by their unique index. For the sake of clarity, we did not show neither the variables associated to the nodes nor the values associated to the edges. We assume that the edges are in the ith layer, which means that n_1 and n_2 are associated to variable x_i and the other nodes to x_{i+1}. We will now simulate the operations performed by MDD-4R supposing that it has to remove e_1 and e_2 by incremental deletion of the edges or by resetting the whole layer. Before removing the edges, we have:

$$layerNodes(i) = \{n_1, n_2\}$$
$$layerNodes(i+1) = \{n_3, n_4, n_5\}$$



(a) Layer i of the MDD **(b)** Removing e_1 invalidates n_3 **(c)** Removing e_2 invalidates n_1

Figure 5.4: Illustration of the partial states of an MDD during the removal of some of its edges.

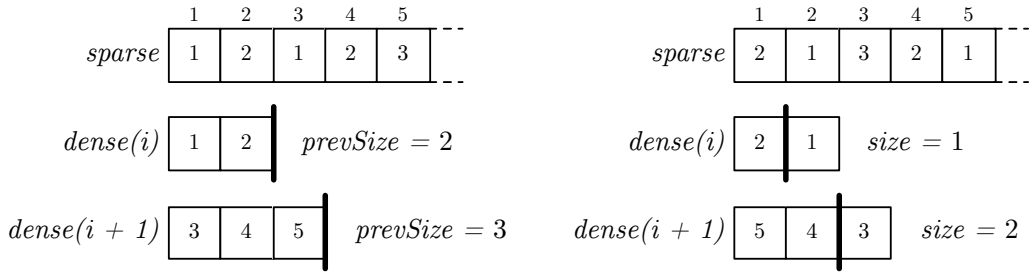


(a) Layer i is entirely reset **(b)** Restoring e_3 validates n_2 and n_4 **(c)** Restoring e_4 validates n_5

Figure 5.5: Illustration of the partial states of an MDD during the reset and restoration of some of its edges.

Incremental deletion Figure 5.4b shows the MDD after that the algorithm has first removed the edge e_1. As we can see, e_1 was the only ingoing edge of n_3, which means that $n_3.inEdges$ is now empty and n_3 must be removed from $layerNodes(i+1)$. Next we delete e_2 from the MDD and thus from $n_1.outEdges$ and $n4.inEdges$. This implies that n_1 has no more outgoing edges, i.e. $n1.outEdges = \varnothing$, hence it must be deleted from $layerNodes(i)$ as shown in Figure 5.4c.

Reset of the layer Figure 5.5 illustrates the operations performed if we use the reset method. First, we reset the ith layer by clearing the sets $layerNodes(i)$ and $layerNodes(i+1)$, the $outEdges$ sets of n_1 and n_2, and finally the $inEdges$ sets of n_3, n_4 and n_5. The result is shown in Figure 5.5a. Next, we restore e_3 in the MDD by re-adding it in the sets $n_2.outEdges$ and $n_4.inEdges$. Given that both those sets were empty before the restoration of e_3, n_2 and n_4 are respectively re-added to $layerNodes(i)$ and $layerNodes(i+1)$. Finally, Figure 5.5b shows the restoration of e_4 in $n_2.outEdges$ and $n_5.inEdges$. Given that the latter was empty before the addition of e_4, it is restored in $layerNodes(i+1)$.



(a) The sparse sets structures before the removal of the invalid nodes

(b) The sparse sets structures after the removal of the invalid nodes

Figure 5.6: Illustration of the arrays representing the shared sparse sets $layerNodes(i)$ and $layerNodes(i+1)$ before and after the invalidation of n_1 and n_2.

Retrieving the invalid nodes As we can see by comparing Figures 5.4c and 5.5c, both methods lead to the same $layerNodes$ values:

$$layerNodes(i) = \{n_2\}$$
$$layerNodes(i+1) = \{n_4, n_5\}$$

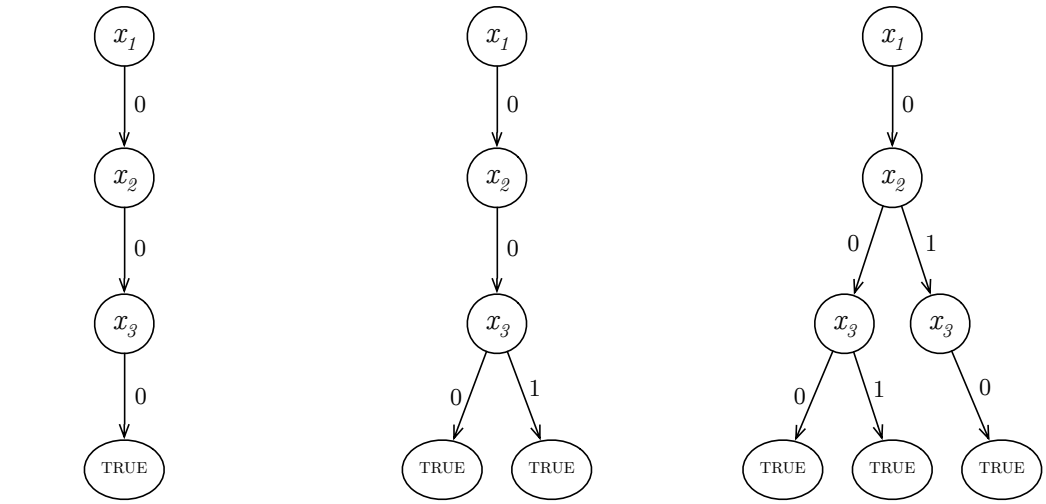
Given that some nodes from layers i and $i+1$ have been invalidated, their remaining adjacent edges need to be removed as well. However, we first have to retrieve those nodes removed during the previous step. Fortunately, given that the $layerNodes$ sets are implemented with a (shared) sparse set, we can get them easily provided that we know the size of the $layerNodes$ sets before the removal of the invalid nodes. As an example, Figure 5.6 illustrates the different arrays representing the $layerNodes$ sets before and after the removal of n_1 and n_3 respectively from $layerNodes(i)$ and $layerNodes(i+1)$. Figure 5.6b shows that the elements in the $dense(i)$ array with their index between $size$ and $prevSize$ correspond to the nodes removed from the associated set $layerNodes(i)$.

Propagation to the previous layer Once that we have retrieved the invalid nodes in layer i, in our case n_1, we have to delete all edges from layer $i-1$ entering n_1. This new edges removal could invalidate new nodes in layer $i-1$ which would imply the removal of edges in layer $i-2$, and so on. We continue this process until we reach the first layer or until no node has been removed in the current layer.

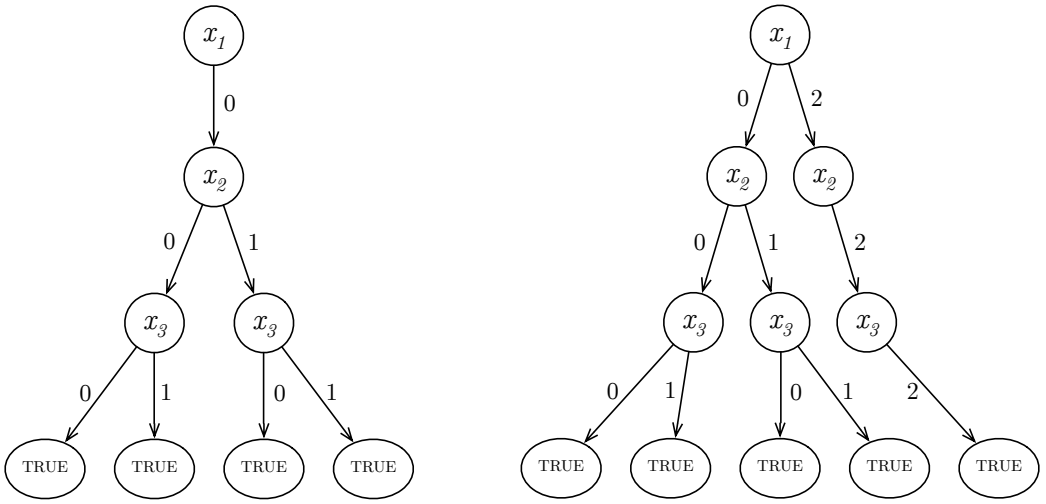
Propagation to the next layer Following the same logic, the invalidation of nodes from the layer $i+1$ will involve the removal of the edges of layer $i+1$ going out from those nodes. This continues until we reach the last layer or until no more node is invalidated by the deletions of the edges.

5.3 Construction of an MDD

As we will show in Chapter 6, the construction of a Multivalued Decision Diagram is an operation that takes a significant amount of time when using an MDD-based propagator for table constraints. Cheng and Yap introduced an algorithm in [6] to build an MDD from its associated table. This algorithm is implemented in or-tools and used for the implementation of MDD-4R in this same solver. On the other side, the construction of an



(a) Initialization of the trie with the tuple $(0, 0, 0)$ **(b)** Addition of the tuple $(0, 0, 1)$ **(c)** Addition of the tuple $(0, 1, 0)$



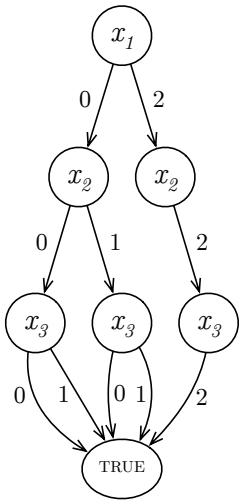
(d) Addition of the tuple $(0, 1, 1)$ **(e)** Addition of the tuple $(2, 2, 2)$

Figure 5.7: Illustration of the states of a trie during its construction.

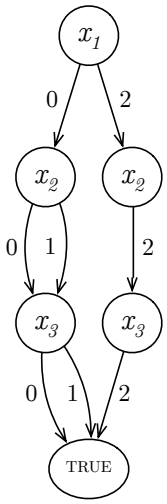
MDD in OscaR is performed by our own faster algorithm that we will briefly describe in this section. A comparison of the algorithms will be presented in the following chapter.

Both algorithms first build a trie that is then reduced into an MDD. The main difference between our strategy and the one from [6] is that the latter is a recursive algorithm that first reduces the root of the MDD and then recursively reduces its children down to the leaves (top-down) while ours starts from the leaves and reduces each layer one by one up to the root (bottom-up).

Construction of the trie The first step of the algorithm is the construction of a *trie* associated to the tuples from the table. A trie can be seen as an MDD without the constraint that it can not have two similar nodes in the same layer. This structure can compress the representation of a table thanks to the prefix sharing of similar solutions [6]. Figure 5.7 illustrates the construction of a trie associated to the tuples $\{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (2, 2, 2)\}$.



(a) Merge the TRUE nodes



(b) Merge of identical nodes from the x_3 layer

Figure 5.8: Illustration of the states of the trie during its reduction into an MDD.

Reduction of the trie Once the trie is entirely constructed, we can merge identical nodes of the trie from bottom to top to reduce the trie into a (semi-reduced) MDD. Two nodes are identical if all their outgoing edges point to the same nodes for the same values. The TRUE leaves are considered identical as well. Figure 5.8 illustrates the different steps performed to reduce the previous trie. We first merge all TRUE nodes and we then look for identical nodes from the layer associated to x_3. The two first nodes of this layer are identical given that they point to the same nodes for all their values, hence they are merged. Finally, both nodes from the second layer are different, so we do not merge them. We never check for identical nodes in the first layer given that it will always be composed of the root only.

An important step in the reduction of the MDD is to find an efficient method for detecting identical nodes in the same layer. Our strategy uses a hash table [33] $nodesHash$ where nodes currently visited are stored. Supposing that we have a hash function $hash$ such that $hash(N_1) = hash(N_2)$ if and only if two nodes N_1 and N_2 are identical, then we can use the algorithm shown in Algorithm 5.1 to efficiently reduce a trie. Different hash functions could be used as long as they satisfy the previous condition.

Algorithm 5.1 The `reduceTrie` function used by OscaR to reduce a trie into a semi-reduced MDD.

Pre: n is the number of variables composing the tuples of the trie
Pre: $layerNodes$ is an array such that $layerNodes[i]$ is the set of nodes in the ith layer
Pre: $hash$ is a hash function of type {Node $\rightarrow$ String} such that $hash(N_1) = hash(N_2)$ if and only if N_1 is identical to N_2
Post: Identical nodes are merged
Post: The trie is reduced into a semi-reduced MDD

```
1:  function REDUCETREE(n: integer, layerNodes: array, hash: hash function)
2:      nodesHash ← initialize a hash table of type {String → Node}
3:      l ← n + 1 // layer currently reduced
4:      while l > 1 do
5:          for all node ∈ layerNodes[l] do
6:              h ← hash(node)
7:              if h ∉ nodesHash then
8:                  nodesHash(h) ← node
9:              else
10:                 merge node with nodesHash(h)
11:         l ← l − 1
```

Chapter 6

Experimental Results

As we have seen, there are many different propagation algorithms for the table constraint. From a developer point of view, it can be difficult to evaluate which algorithm is the best for his CP solver. This chapter aims to rigorously assess the performances of the current state-of-the-art propagators. As we will see, the Compact Table algorithm described in Chapter 3 clearly outperforms its contestants. We would thus advise to prefer this algorithm as the default table constraint propagator for a CP solver.

6.1 Context

Machine All experiments have been performed on a MacBook Pro with a Intel Core i5 2.4 GHz CPU and 8 GB RAM, running under OS X Yosemite 10.10.2.

Solvers and implementations The algorithms CT, GAC-4, GAC-4R and MDD-4R described in the previous chapters have been implemented in OscaR for the purpose of this thesis. The default propagator for the table constraint in OscaR is currently STR2 and was already implemented. However, we optimized its implementation and reduced the time taken by STR2 to solve any instance by a factor 2. Although STR3 was also implemented for this thesis, it is not part of the experiments given that the results obtained with this algorithm were not good enough to be competitive with the others. Moreover, we also implemented a variant of CT by implementing the *validTuples* and *mask* structures with sparse sets instead of bitsets, but the performance of this implementation was not satisfying enough to be presented in this thesis. All implementations brought to OscaR have been rigorously profiled and optimized to obtain the most representative results for each algorithm.

In order to have an external work for comparison, we also measured the performance of the CT implementation in or-tools, the solver maintained by Google and written in C++. This solver also comes with an implementation of MDD-4R by Perez and Régin [25]. Unfortunately, the current code has bugs[1] and does not always generate the expected search space obtained with the other propagators. For this reason only, we decided not to include it in our main results. However, a comparison of this implementation and our own for the instances correctly solved by or-tools will be briefly discussed in Section 6.3.3.

[1]Even though this implementation is available with the source code, we remind the reader that it is not activated by default and does not influence the overall quality of the official or-tools binaries.

For the reader interested in reproducing our experiments, those are the specific versions used for both solvers:

- OscaR fork available at [23]; branch `mdd4r`; commit **8fe9560**.
- or-tools available at [11]; branch `master`; commit **5c35352**.

Benchmarks The instances selected for our measures were taken from [13]. Two sets of instances *cw-m1c-ogd* and *cw-m1c-uk* come from the crossword problem, which goal is to fill a grid with words from a given dictionary. The tables composing those instances have an arity between 4 and 20. Given that the crossword problem comes from "the real world", we also selected two other sets of instances generated randomly *rand-8-20-5* and *rand-10-20-2*, which respectively have a maximum table arity of 8 and 10. Those four sets were chosen because they involve table constraints only.

6.2 Experiments

Measures The total number of instances from all different sets is 170. Appendix A reports, for each propagator p and for each instance i, the following data:

- INIT the time required by p to set up all table constraints composing i ;
- SEARCH the search time taken by p to find at least one solution of i ;
- TOTAL the total time, i.e. TOTAL = INIT + SEARCH ;
- NFAILS the number of times the solver had to backtrack after a failure ;
- NBRANCHES the number of search decisions taken by the solver ;
- NSOLS the actual number of solutions found by p.

The set up, search and total time are expressed in milliseconds. We also reported the data for the MDD-4R implementation of or-tools to show its inconsistency with the other propagators. Indeed, the number of failures and search decisions for a single instance should obviously be the same for all implementations, and it is not always the case with this one.

Taking into account the set up time of the constraints is often neglected for this kind of benchmark, because it is in general not relevant regarding the total time taken to solve a problem. However, in our case, we decided to measure it given that the construction of a Multivalued Decision Diagram from its associated table is a time consuming task that can not be overlooked while comparing the performance of MDD-4R with other algorithms.

Limits Given the large number of instances to solve for each of the propagators, we set a time limit of 5 minutes. The search was stopped if the solver did not find any solution within this time, which is reported as T.0. in the appendix. For difficult instances with high arity constraints, the propagators also stopped after finding a first solution. However, instances with small arity constraints had to find a required number of solutions n before stopping. Indeed, the first solution of such problems were often found in a few milliseconds, hence we decided to increase the number of solutions to have relevant computing times. Finally, instances that took less than 2 seconds to be entirely solved by one of the algorithm were discarded. Unsatisfiable instances are reported as N.S. in the NSOLS column of Appendix A.

Search strategy The variable and values heuristics used during search have been borrowed from [6] and [25]. The variable that appears in most constraints is selected first. In case of tie, the variable with the smallest index in the table is selected. This variable is assigned its minimum value and, in case of failure, the value is removed from the domain of the variable.

6.3 Results

Once all measures have been performed, different solutions exist to represent and compare our results. Performance profiles [8] offer a powerful interpretation tool for benchmarking optimization software, and were chosen for our analysis of the propagators.

6.3.1 Definitions

Let $t_{i,p}$ be the total time taken by a propagator p to solve instance i and $\mathcal{P}$ be the set of propagators. We first note the *performance ratio*

$$r_{i,p} = \frac{t_{i,p}}{\min\{t_{i,p} : p \in \mathcal{P}\}}$$

as the ratio between the time taken by p to solve i with the best time (i.e. smallest) taken by any propagator to solve i.
Let n_i be the total number of instances and $\mathcal{I}$ the set of all instances, then we define

$$\rho_p(\tau) = \frac{1}{n_i}\text{size}\{i \in \mathcal{I} : r_{i,p} \leq \tau\}$$

as the probability for propagator $p \in \mathcal{P}$ that a performance ratio $r_{i,p}$ is within a factor $\tau \in \mathbb{R}$ of the best possible ratio for instance i [8]. The term *performance profile* of the propagator p denotes the distribution function of $\rho_p(\tau)$ for different values of τ.

Advantages We chose to use performance profiles for their many advantages. First, it is always easier to analyze metric results on a graph than on an extensive table. Moreover, this method allows to easily take into account the case where some propagators fail to find a solution within the imposed time. Indeed, it is not always clear how to aggregate results for a specific set of instances when a timeout occurred. Finally, performance profiles allow to have a global view of the propagators efficiency given that all instances have the same impact on the result. For example, large instances that take a lot of time to process do not have the same influence on the performance profiles as they would have with an arithmetic or geometric mean of a set of instances.

6.3.2 Propagators performance profiles

Figure 6.1 illustrates the performance profiles of CT, STR2, GAC-4, GAC-4R and MDD-4R with OscaR as well as CT with or-tools.

The illustration shows many information. Let's first ignore the curve associated to or-tools. We directly notice the large difference between CT and the other algorithms. In addition to be more efficient, CT is also very robust because there was no instance that was solved by any other algorithm and not by CT. Secondly, the graph shows that GAC-4 is clearly improved in GAC-4R thanks to the reset idea given that it is globally faster and allows to solve more instances. Moreover, STR2 seems to stand between those two

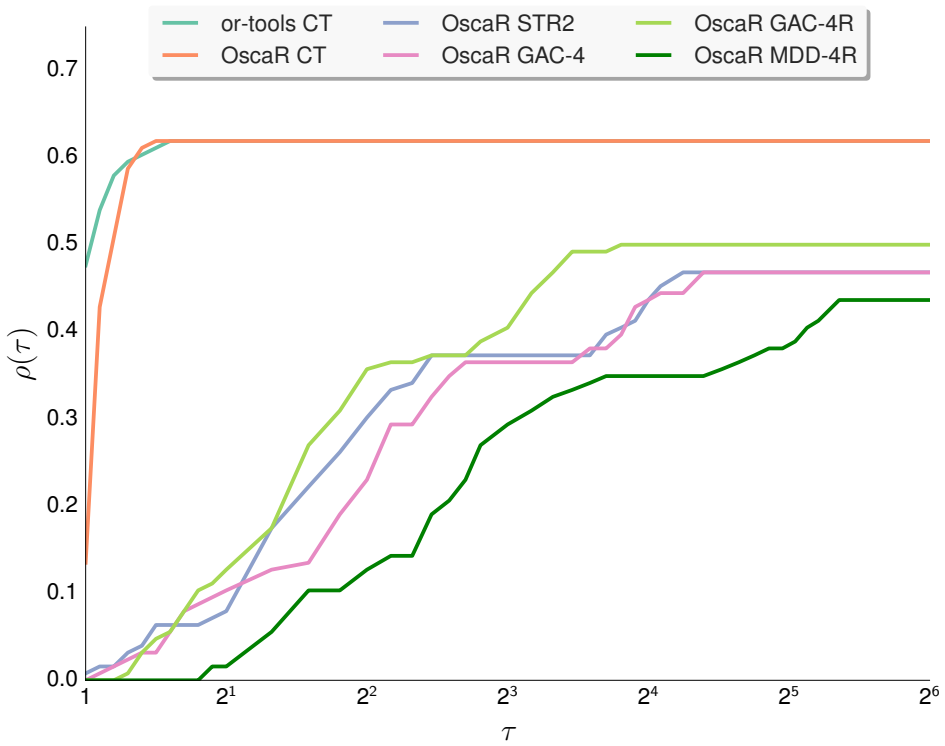


Figure 6.1: Illustration of the performance profiles for each algorithm on the solving time for all instances of *cw-m1c-ogd*, *cw-m1c-uk*, *rand-8-20-5* and *rand-10-20-10* from [13] within a time limit of 5 minutes.

algorithms. The gain obtained by the reset in GAC-4R suggests that a similar idea could be a good way of improving STR2.

Finally, the MDD-4R curve is quite low compared to the other algorithms. This can be due to the complexity of the algorithm and the number of operations performed during propagation. Moreover, MDD-4R is also designed to solve very large problems with a high compression rate and implying diagrams that take a few gigabytes in memory [26]. Unfortunately, we did not have the required resources to perform experiments on such problems.

Comparing two propagators from different solvers is a delicate task. Indeed, the difference in computing time between those can have origins external to the propagators themselves, such as the language in which each solver is implemented or the operations performed during the search outside of propagation. However, we can see that the implementation of CT in OscaR is really close to the one in or-tools, with a slight advantage for the latter. We suspect that this difference is due to the fact that or-tools is implemented in C++, which is expected to be a bit more efficient than Scala [12].

High arity table constraints As Figure 6.1 shows, approximately 40% of the instances could not be solved by any algorithm within the time limit of 5 minutes (given that $\max_p\{\lim_{\tau\to\infty} \rho_p(\tau)\} \simeq 0.6$). The unsolved instances are generally composed of tables with arity between 7 and 13. In order to compare the different algorithms regarding those instances, we decided to associate their performance profiles to the number of search decision performed by the solver. Indeed, the number of search decisions taken by a solver

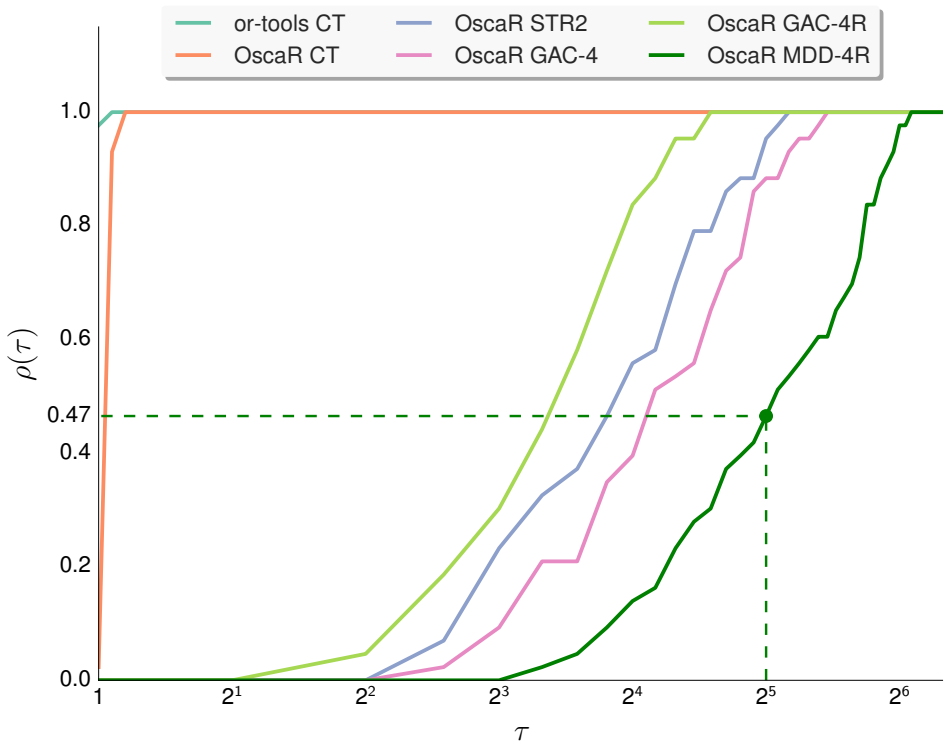


Figure 6.2: Illustration of the performance profiles for each algorithm on the number of decisions taken by the solver for the unsolved instances of *cw-m1c-ogd*, *cw-m1c-uk*, *rand-8-20-5* and *rand-10-20-10* from [13] within a time limit of 2 minutes.

is a good indication for the progression of the search. We can then evaluate the relative performance of two propagators on a same instance by comparing the number of search decisions taken by the solver when used with each of the propagators, within a same time limit.

Therefore, for each of the unsolved instances, we measured the number of search decisions taken by each of the algorithms within 2 minutes. Those are reported in Appendix B. Let $d_{i,p}$ be the number of search decisions taken by p during the search of a solution for instance i, we can then redefine the performance ratio $r_{i,p}$ as

$$r_{i,p} = \frac{\max\{d_{i,p} : p \in \mathcal{P}\}}{d_{i,p}}$$

The definition of $\rho_p(\tau)$ remains unchanged.

Figure 6.2 shows the performance profiles of the different algorithms on the number of search decisions taken by the solver within 2 minutes on the large instances that were not solved previously. As we can see, CT is again far ahead its contestants and the difference between OscaR and or-tools is barely noticeable. The relative performance of the other algorithms remains the same as for the small instances, where GAC-4R is quite efficient compared to STR2 and its predecessor GAC-4. Moreover, the illustration shows that CT took at least 32 times more search decisions than MDD-4R for 53% of those large instances. Based on those results, we can conclude that CT is a very efficient propagation algorithm. It was chosen currently as the default propagator for table constraints in OscaR and it is also the default one in or-tools.

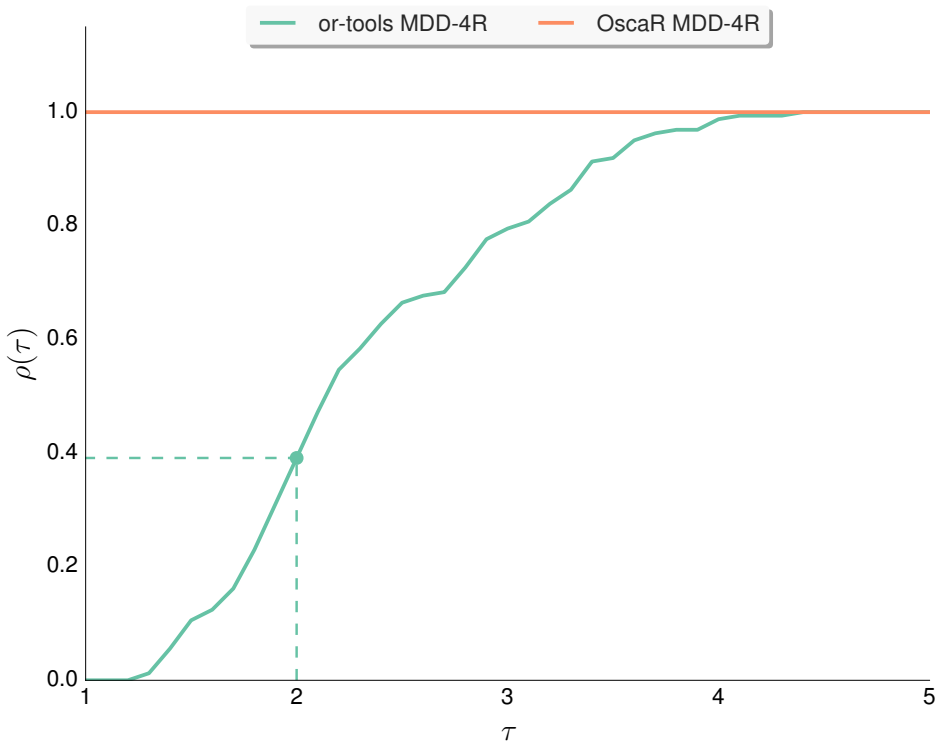


Figure 6.3: Illustration of the performance profiles of OscaR and or-tools on the construction of the MDD associated to all instances of *cw-m1c-ogd*, *cw-m1c-uk*, *rand-8-20-5* and *rand-10-20-10* within a time limit of 5 minutes.

6.3.3 MDD-4R comparison

Even though the MDD-4R implementation of or-tools does not always generate the correct search space, we decided to compare it with ours for relevant instances. As a reminder, MDD-4R in OscaR implements our own algorithm for constructing an MDD from the associated table, as opposed to or-tools that implements the algorithm described by Cheng and Yap in [6]. A recent paper by Régin and Perez [27] introduces a new linear algorithm for the construction of an MDD but we unfortunately did not find the time to implement it in OscaR.

Initialization Figure 6.3 illustrates the performance profiles of OscaR and or-tools on the initialization of the MDD for all 170 instances. Indeed the implementation of the algorithm from Cheng and Yap in or-tools is functional, hence we decided to compare the initialization for all instances to have a big, representative set. As we can see, our algorithm is much more efficient. Indeed, the graph shows that OscaR is always faster than or-tools to construct the MDD given that $\rho_{oscar}(\tau = 1) = 1.0$. Moreover, or-tools requires more than twice the time taken by OscaR for approximately 60% of the instances.

Search The comparison of the algorithms for the search is more delicate given that the implementation of MDD-4R in or-tools is inconsistent. Comparing the search space generated by two algorithms is a complex task, hence it is difficult to find out for which instances the or-tools implementation was not deficient. However, if two algorithms have the same number of failures and search decisions for a same problem, it is reasonable to

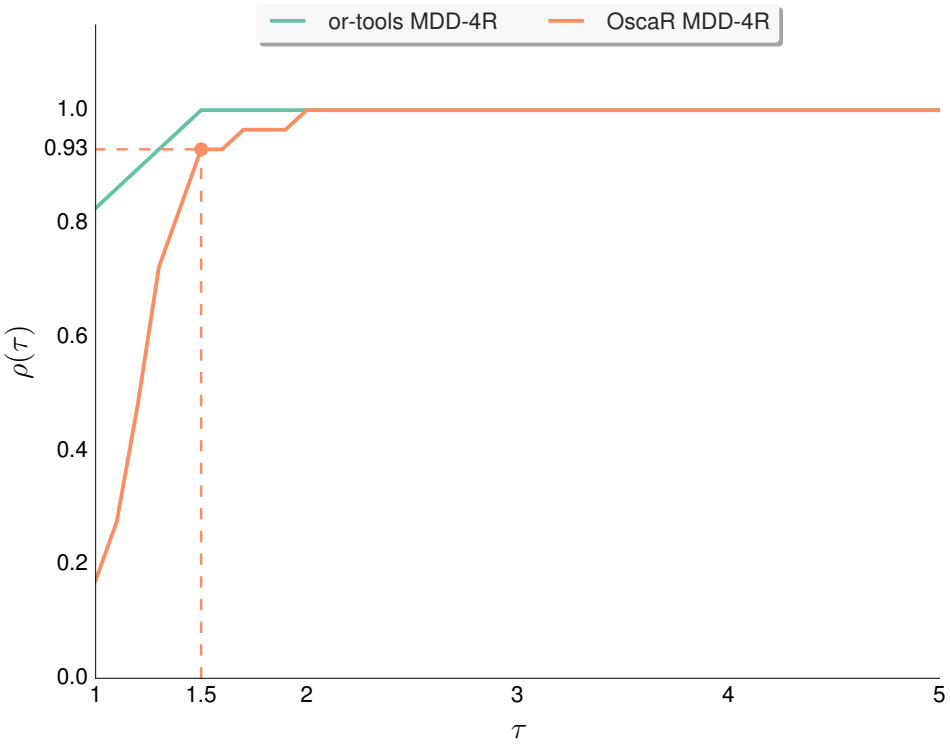


Figure 6.4: Illustration of the performance profiles of OscaR and or-tools on the search of a solution with MDD-4R for the instances of *cw-m1c-ogd*, *cw-m1c-uk*, *rand-8-20-5* and *rand-10-20-10* from [13] correctly solved by or-tools within a time limit of 5 minutes.

assume that both explored the same search space. Following this logic, we decided to compare OscaR and or-tools on the 46 instances from the different sets such that the number of failures and search decisions made by MDD-4R of or-tools were the same as with any other correct algorithm.

Figure 6.4 shows the performances profiles of OscaR and or-tools on the search of solutions with MDD-4R for those instances. In this case, the implementation of or-tools is a bit faster than ours. However, more than 90% of those instances are solved with OscaR in a time with a ratio less or equal than 1.5 compared to the time required by or-tools, i.e. we have $\rho_{oscar}(\tau = 1.5) > 0.9$. As for the comparison of both CT implementations, this small difference of performance can come from different factors.

Finally, an interesting result is the overall comparison of the implementations by taking into account both the initialization of the MDD and the search of a solution. Figure 6.5 shows the performance profiles of MDD-4R in OscaR and or-tools on the construction of the MDD and the search of solutions for the 46 instances correctly solved by or-tools. As we can see, OscaR is slightly above or-tools thanks to its improved initialization algorithm, even if it is a bit slower during search. This concludes that the construction of an MDD from a table should never be neglected while evaluating the performance of an algorithm based on Multivalued Decision Diagrams.

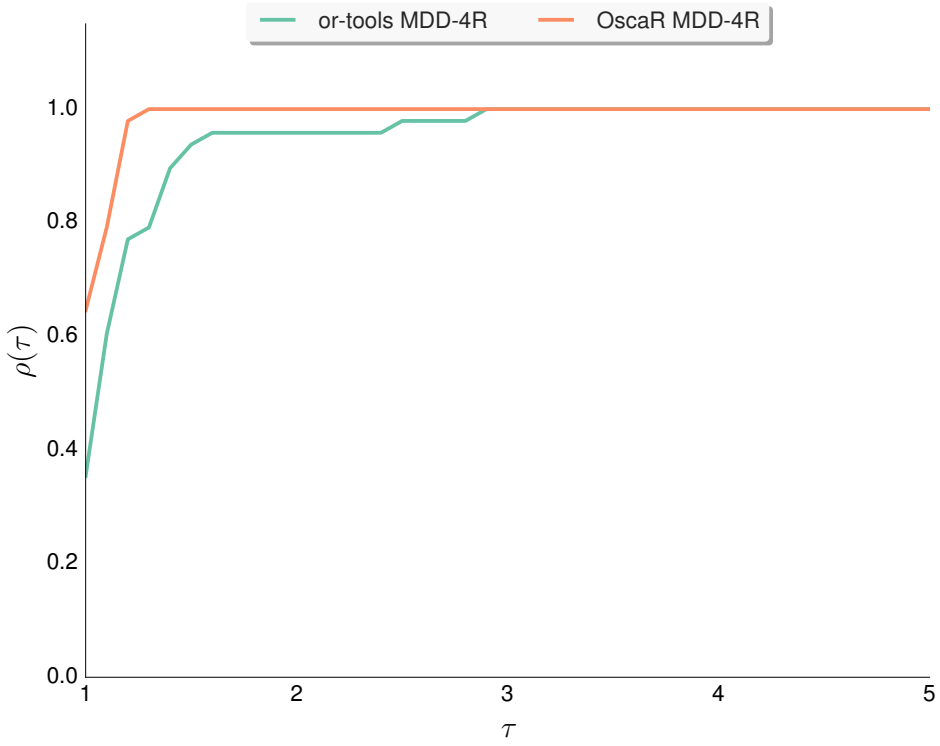


Figure 6.5: Illustration of the performance profiles of OscaR and or-tools on the initialization and search of a solution with MDD-4R for the instances of *cw-m1c-ogd*, *cw-m1c-uk*, *rand-8-20-5* and *rand-10-20-10* from [13] correctly solved by or-tools within a time limit of 5 minutes.

Conclusion

In this thesis, we presented and compared the most efficient propagation algorithms for the table constraint. On our large set of experiments, CT obtained the best results. Although it does not mean that there is no possible instance where it would not be the case, we would recommend this algorithm to be the default propagator for table constraints in any solver.

All algorithms added to OscaR were carefully implemented and documented so that they will be easily modified if some of those algorithms are improved in the coming years. We also proposed a simple yet efficient method for the construction of an MDD from the associated table.

Finally, we hope that our detailed explanations of CT, GAC-4(R) and MDD-4R will help the community to better understand those sophisticated algorithms.

Future Work

A few additional experiments could be performed as an addition to this thesis. First, we would have liked to compare CT with algorithms that are really efficient for small arity constraints such as $AC3_{rm}$ [15] and the different algorithms proposed in [19]. Moreover, the comparison between our strategy for the construction of an MDD and the algorithm recently proposed in [27] would be of great interest. We also think that the reset idea should be applied to STR2 given the overall performance improvement it gave to the other algorithms.

This thesis studied the performance of the different propagators in a practical way, by measuring the results of each one of them on a large set of instances. However, a deep complexity analysis of each of those propagators is missing and would go hand in hand with this work.

Even though our results with MDD-4R were not as good as expected, a study of the variable ordering inside the MDD would certainly improve the efficiency of MDD-4R. Finally, the Multivalued Decision Diagram is a very powerful data structure and we think that it could be of great use for the implementation of the *regular* constraint [29].

Bibliography

[1] Scala standard library 2.11.6. http://www.scala-lang.org/api/current/index.html#scala.Long.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 1996.

[3] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. 2005. Available from http://sofdem.github.io/gccat/gccat/.

[4] 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.

[5] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):59–69, 1993.

[6] Kenil CK Cheng and Roland HC Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[7] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[8] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[9] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

[10] Google. Google optimization tools, 2015. Available from https://developers.google.com/optimization/.

[11] Google. Google's Operations Research tools, 2015. Available from https://github.com/google/or-tools.

[12] Robert Hundt. Loop recognition in c++/java/go/scala. In *Proceedings of Scala Days 2011*, 2011.

[13] Chirstophe Lecoutre. Instances of the constraint solver competition, 2015. Available from http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html.

[14] Christophe Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[15] Christophe Lecoutre, Fred Hemery, et al. A study of residual supports in arc consistency. In *IJCAI*, volume 7, pages 125–130, 2007.

[16] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland Yap. A path-optimal gac algorithm for table constraints. In *20th European Conference on Artificial Intelligence (ECAI'12)*, pages 510–515, 2012.

[17] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[18] Jean-Baptiste Mairy and Yves Deville. LINGI2365: Constraint programming. University Lecture, 2015.

[19] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.

[20] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *8th European Conference on Artificial Intelligence (ECAI'88)*, pages 651–656. Pitmann Publishing, 1988.

[21] Optimisation Research Group NICTA. The MiniZinc Challenge, 2015. Available from `http://www.minizinc.org/challenge.html`.

[22] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[23] OscaR. OscaR fork, 2015. Available from `https://bitbucket.org/jdemeulenaer/oscar/`.

[24] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[25] Guillaume Perez and Jean-Charles Régin. Improving gac-4 for table and mdd constraints. In *Principles and Practice of Constraint Programming*, pages 606–621. Springer, 2014.

[26] Guillaume Perez and Jean-Charles Régin, 2015. Mail exchange.

[27] Guillaume Perez and Jean-Charles Régin. Relations between mdds and tuples and dynamic modifications of mdds based constraints. *arXiv preprint arXiv:1505.02552*, 2015.

[28] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014. Available from `http://www.choco-solver.org`.

[29] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In *Principles and Practice of Constraint Programming-CP 2006*, pages 751–755. Springer, 2006.

[30] Michael Rice and Sanjay Kulhari. A survey of static variable ordering heuristics for efficient bdd/mdd construction. *University of California, Tech. Rep*, 2008.

[31] Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95. IEEE, 1990.

[32] Julian R Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177(18):3639–3678, 2007.

[33] Wikipedia. Hash table — wikipedia, the free encyclopedia, 2015. Available from `http://en.wikipedia.org/w/index.php?title=Hash_table&oldid=664152780`.