UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Multivalued decision diagrams for optimization

Dissertation presented by
Romain HENNETON

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Pierre SCHAUS

Reader(s)
Yves DEVILLE, Hélène VERHAEGHE

Academic year 2016-2017

Abstract

Constraint programming is a well known efficient programming paradigm sometimes called smart brute-force. By using a branch and bound algorithm associated with domain propagation, it allows to solve hard combinational problems. One of the strengths is its expressiveness via the constraint modelization. However two major weaknesses of the paradigm are the lack of usage of memory (branch and bound is a simple DFS) and the lack of communication between the involved constraints.

The multivalued decision diagram is a structure that allows to represent the search tree in a condensed way. This allows to circumvent the communication problem by increasing the memory usage. With a better communication between constraints, the time used for the propagation decreases and the number of failures in the search tree lowers as well.

Multiple types of decision diagrams are presented in this thesis and all of them are implemented in Oscar - a constraint programming framework. The multivalued decision diagram sub-framework is designed to be flexible and allow the addition of new modules and algorithms in it. Most of the decision diagram algorithms are competitive with the corresponding state of the art propagators. Moreover, when multiple constraints are mixed inside the framework, the combination of state of the art methods is outperformed by the associated multivalued decision diagram.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, Pr. Pierre SCHAUS for his support, patience and advices throughout the year.

Secondly, I would like to take this opportunity to sincerely thank my parents, my brother and more recently my partner for their constant support during all those years. They were always by my side, and were even more present in the bad moments.

Then, I would like to thank Alexandre Malingreau and Hélène Verhaeghe for making me insightful remarks about the redaction of this thesis.

Finally, I would like to thank my friends from the *David Parnas'* room at the INGI departement for the friendly and hard-working atmosphere that was present in the room throughout the year.

CONTENTS

ACRONYMS AND SYMBOLS

MDD	Multivalued Decision Diagram
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
COP	Constraint Optimization Problem
O	Big O complexity
DFS	Depth First Search
BFS	Breadth First Search
GAC	Generalized Arc Consistency
CT	Compact Table
STR	Simple Tabular Reduction
DFA	Deterministic Finite Automaton
Regex	Regular Expression
$\delta^{in}(v)$	Set of edges having node v as destination
$\delta^{out}(v)$	Set of edges having node v as source
$\phi_{(u,v)}$	Value of the edge (u, v)
TSP	Traveling Salesman Problem
TSP-TW	Traveling Salesman Problem with Time Windows
NP	Nondeterministic, polynomial time complexity

INTRODUCTION

Constraint programming is a programming paradigm used to solve hard combinational problems. Nowadays, we hear lots about the optimization of decisions for processes and companies, in order to limit the costs or the time needed to produce a certain product. The power of CP relies on the fact that once a problem has been modeled with constraints, you do not need to know how to solve the problem, a solver does the job for you. Therefore, one important aspect of constraint programming is the creation of very efficient algorithms used as propagator in the CP framework. Unfortunately, the propagators usually work in isolation, meaning that the constraints do not communicate a lot, which is one of the biggest drawbacks of the current state of the art CP solvers. In this thesis, we explore a new kind of propagator toolbox, the multivalued decision diagram, that allows to combine constraints, increase the communication and reduce the time needed to solve complex problems.

The binary decision diagrams were introduced by Lee in [19] to represent switching circuits. Then, the tool was extended to the representation of any boolean expression. After several years of research on binary decision diagrams, the idea was further extended into multivalued-decision diagrams (MDD) by Kam in [15]. In 2007, in [1], H.R. Andersan et al. proposed the idea of a constraint store based on multivalued decision diagrams. Since then, several teams around the world have tried to use MDD's in constraint programming. More recently, David Bergman, Andre A. Cire, Willem-Jan van Hoeve and John Hooker have published [2], a book referencing the current possible usages of this object.

The structure of this thesis is articulated around the creation of a big MDD-framework in which it is easy to add new blocks to improve the scope of the decision diagrams. First we define some concepts, useful for the understanding of the following chapters. Then, we present a generalised propagator based on a built decision diagram. The chapters 3, 4 and 5 describe how to build this decision diagram. Several methods are visited such as table constraint based MDD, diagrams constructed from a finite state automaton or even from a custom set of constraints. Finally, the last chapter presents a way of adding dynamic constraints to an MDD in order to increase the propagation even further. The benchmarks relative to the different sections come at the end of their respective chapter.

CHAPTER 1

PRELIMINARIES

1.1 Constraint programming

Constraint programming (CP) is a programming paradigm used to solve hard combinational problems. It can be used to find solutions to Constraint Satisfaction Problems (CSP) and Constraint Optimization problems (COP). In the CP model, we define a set of variables and their domains (set of values) plus a certain number of constraints on the variables. A solution to the CSP problem is an assignation of each variable to a value from its domain, such that all the constraints are enforced. Concerning optimization problems, the aim is to find the solution optimizing an objective function and respecting the constraints.

A CP framework is usually splitted in two parts:

- *The model*: This part of the paradigm corresponds to the phase were the user transforms a problem (Traveling Salesman, Vehicle Routing, scheduling, ...) into a set of variables, each of them having a specified domain. Then, the user can use those variables and apply constraints on them to describe the solution. The asset of this paradigm is that it allows to find solutions of a problem without knowing how to solve it, only the description of the solution is needed.
  Several types of constraints exist but in the literature, they are separated in two categories: the binary constraints that operate on two variables and the global constraints that operate on a set of variables.
  - The classic example to present a problem easily modelled by CP is the Sudoku game. In that problem, the model is fairly easy and defined by the rules of the game:
    - ⋆ Each cell of the grid is a *variable*.
    - ⋆ The *domain* of each variable is {1,2,...,9} (except for the values that are already known, for those variables, their domain is a singleton: a single value).
    - ⋆ The *constraints* are the rules of the sudoku: all the variables in a 3x3 cell, in a row and in a column should have different values.
- *The Solver*: A solver is a smart piece of code in which you put your model and it finds all the possible instantiations such that all the constraints are enforced.
  Its basic concept (simplified) is the exploration of a search tree (in a depth first way) in which each branch is an assignation of a value to a variable. The computation of this tree is an repetition of two steps:

- Propagation: Reduce the size of the variables domains by removing the values that are inconsistent with the constraints based on the current domains. This process takes each constraint in isolation, removes the inconsistent values and put every constraints affected by the removal of this values in a queue. The step is repeated until no more value can be suppressed (= Fix point algorithm) and the queue is emptied.
- Branching: Take a decision after the propagation ; 3 options are possible:
  - ⋆ A variable has an empty domain, leading to no possible assignation, and a backtrack in the tree is needed (= Failure leaf).
  - ⋆ The domain of each variable is a singleton, then this assignation is a solution. Then a backtrack is needed to explore the remaining parts of the super-problem (= Solution leaf).
  - ⋆ All domains are non empty and some of them have more than one value. In that situation, the algorithm branches, meaning that it takes a decision on the domains and solves the subproblems induced by this decision. In order to have a complete search, all the possible decisions at a certain node must be tried (= Internal node).

1.2 Trailing and sparse set

A branching decision is a temporary modification of the domain. There is the need of recovering from this decision once the induced sub-problem is fully solved (in order to explore all the potential decision and therefore ensure a complete search).

To circumvent this difficulty, we use special object called reversible structures and more particularly the reversible Integer object. This object contains an integer that is restored when going back up in the tree thanks to stack structures. The mechanism of this object and a basic implementation of a solver are available in [17].

A more interesting usage of the trailing mechanism to analyze is how domains are stored in a CP solver. In OscaR [31], the domains are represented as a sparse set [5]. The object is a set of size n containing the set of values $\{0, 1, ..., n-1\}$ when created. Afterwards, values can be removed from the set (but not be added, the size of the set must be monotically decreasing). However, as explained above, the domains should be restored to their previous state when a backtrack is executed. 3 components are needed to allow this behaviour:

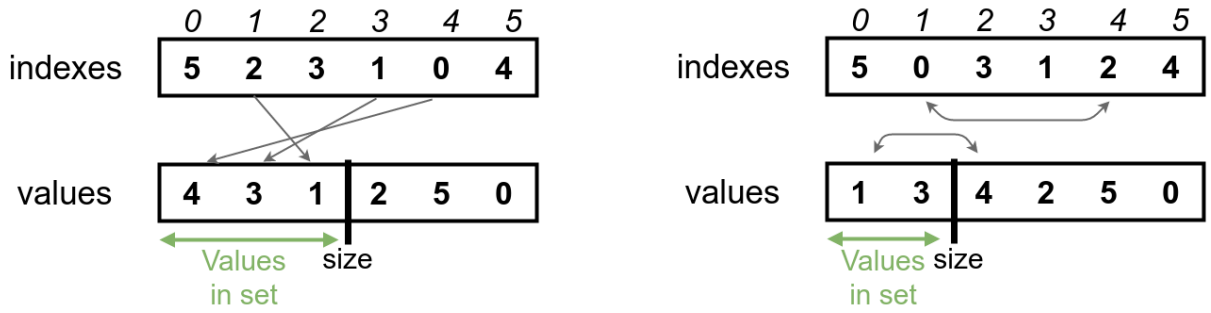
- `values`: array of size n containing a permutation of $\{0, 1, ..., n-1\}$
- `indexes`: array of size n giving the position of each value in the array value. We have the invariant $\texttt{values}[\texttt{indexes}[k]] = k \quad \forall\ k \in \{0, 1, ...n-1\}$
- `size`: a reversible integer giving the number of items still present in the set

The creation of a set having those three components allows to easily perform the contains, remove and reset methods on the structure by ensuring a constant time complexity $O(1)$ [30].

`Contains:` The figure 1.1 presents a sparse set in a certain state, with its active values underlined. We can verify the invariant that a value is present in the set if `index[value]` $< size$. In our case, the size is 3, the elements present in the set are 4, 3 and 1. Those values have an index of 0, 1 and 2 respectively. The `contains` method on value a returns true only if `indexes[a]` is smaller than size.

`Remove:` If the value 4 is removed from the set, we must enforce that its index is greater than size, and that the elements present (resp. absent) in the set are still present (resp. absent) after the deletion. We start by analyzing what happens in the values array. The deleted item (4) is swapped with the item in position $size - 1$, which is 1. Then, the size is decremented to remove 4 from the set. The only operation that is relevant for the update of the indexes is the swap, and in fact, a swap in the values array correspond to a swap in the indexes array as well. All the operations described in this part are in $O(1)$ time. Therefore a removal is performed in constant time as well. We can see that if the $size$ is backtracked to its previous value $size + 1$, the element re-added in the set is 4, which is the last value removed. The inactive values are in fact sorted by their deletion order, which allows to recover them in order on backtrack by simply adapting the size.

`Clear:` All the items are removed from the set by setting the `size` to 0.



(a) State of a sparse set, containing the values 4, 3 and 1. The values 2, 5 and 0 are out of the set

(b) Sparse set of figure 1.1a after the removal of value 4

FIGURE 1.1 – Representation and behaviour of a sparse set

1.3 Multivalued Decision Diagram

An MDD is a directed acyclic graph that can be used to represent a compressed version of a search tree. It is composed of a root node on the first layer, an end node on the last (deepest) layer and internal nodes grouped by layers. Each edge goes from a node of layer i towards a node of layer $i + 1$ and is associated to a value assignation to a variable. One important point corresponds to the fact that the variables are ordered in a static way, meaning that each layer of the graph is associated with a fixed variable. Therefore, a path from the root to the end represents a specific assignation of values to variables.

For example, the figure 1.2 presents an MDD of 3 variables. The possible instantiations are referenced in table 1.1. In fact, the number of paths in this kind of layered directed acyclic graph can grow exponentially with respect to the number of edges. Therefore, instead of representing all the paths in a table, we have a compressed way of expressing the same idea.

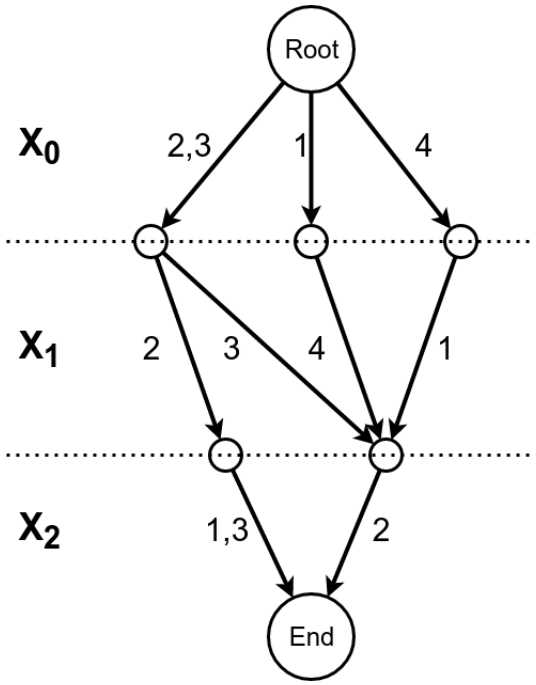


FIGURE 1.2 – Example of Multivalued Decision Diagram

x_0	x_1	x_2
2	2	1
2	2	3
3	2	1
3	2	3
2	3	2
3	3	2
1	4	2
4	1	2

TABLE 1.1 – Table representing all the valid paths in figure 1.2

Note that various types of MDD's exist. In this work, we limit ourselves to some variations of those decision diagrams. They differ by their usage and how they are built but share some similarities. For example the diagrams that we study have in common that they contain at least all the solutions of a problem respecting all the constraints applied on the diagram. A solution in an MDD is a path from the root to the end node such that each edge used in a layer assigns a value to the variable corresponding to this layer and respecting all the constraints of the problem.

The following definitions are borrowed from the work of Bergman et al. [2], Perez & Régin [23], Demeulenaere [10] and Rice & Kulhari [29]. As the decision diagram framework is quite large, several variants and namings exist and sometimes overlap. The aim of this section is to clarify some definitions.

Exact decision diagram: exact representation of feasible solutions to a discrete optimization problem. In this kind of diagram, each solution corresponds to a path in the MDD and more importantly, each path corresponds to a solution.

Relaxed decision diagram: diagram in which the set of paths is a superset of the paths present in the exact decision diagram corresponding to the problem. Therefore, a relaxed decision

diagram contains all the solutions but may also contain paths that do not correspond to any solution.

For the two following definitions, we consider that the edge values can take k values, ranging from 0 to k-1.

Fully reduced decision diagram: 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. Two nodes are considered to be duplicates if their outgoing edges have the same values and lead to the same destinations nodes. We do not allow them in order to restrict the size of the MDD to a minimum. In fact, those duplicate nodes can be merged without any modification of the paths contained in the MDD.

Semi-reduced decision diagram: This kind of diagram relaxes the full-reduction property by allowing nodes having k outgoing edges pointing to the same node. However, we keep the property that the diagram can not have duplicate nodes.

As an example, imagine that we have the constrained problem defined below:

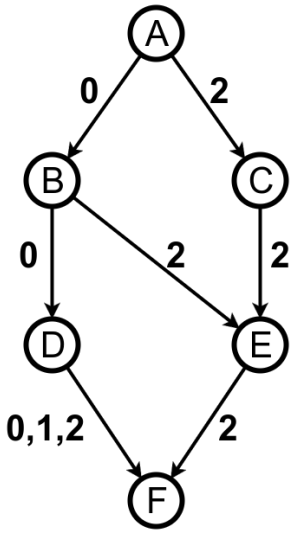
$$D(x_0) = D(x_1) = D(x_2) = \{0, 1, 2\}$$

$$\text{Constraints:} \begin{cases} x_0 \neq 1 \\ (x_0 + x_1)\%2 = 0 \\ x_0 \leq x_1 \leq x_2 \end{cases}$$

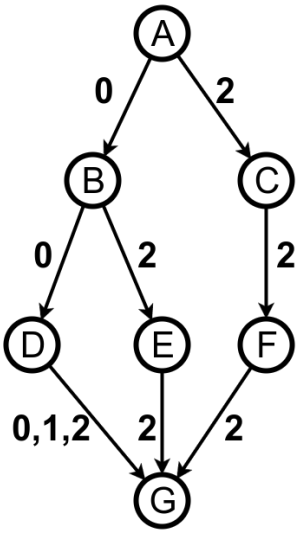
This custom problem can be represented by many decision diagrams. We give some examples (Figure 1.3) to give you more insight about the differences between the types of diagram.

- Figure 1.3a represents the exact semi-reduced diagram corresponding to the problem. We can verify that each path corresponds to a solution and that each solution is a path (= exact). Moreover, the diagram does not contain any duplicate node (semi-reduced).
- This is the difference with the diagram presented on figure 1.3b that is still exact but not semi-reduced. Indeed, if we look at node E and F, they both have an out edge with value 2 that leads to node G. Those two nodes should be merged to obtain a semi-reduced diagram.
- Figure 1.3c shows an exact and fully reduced diagram. Compare it with the one on figure 1.3a and notice that the only difference lies in the fact that we removed the node D. This is due to the fact that the edges leaving D represent all the domain of x_2 and they all lead to node E. We can therefore compress the diagram to its maximum to obtain the one on figure 1.3c.
- The last diagram is the relaxed one, presented on figure 1.3d. We can see that this diagram contains the path 2-0-1 that does not respect all the constraints. However, all the solutions are present in the diagram. Therefore, the paths set is still a superset of the solution set. As no two nodes are duplicate, we call this diagram a relaxed semi-reduced version of the problem mentionned above.

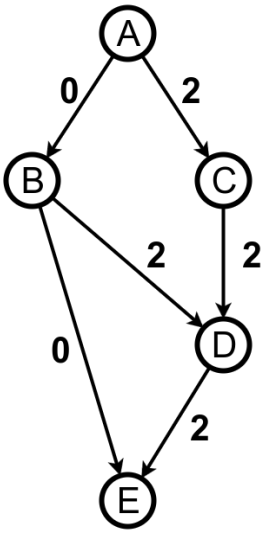
In sections 3.1 and 4 we analyze how the semi-reduced MDD for the table and regular constraints are built. The chapter 5 introduces how to build a relaxed MDD with a limited width, enforcing some global constraints. Those MDD are first created as static objects, and plugged into a CP solver as global constraints afterwards. The hope is that the information



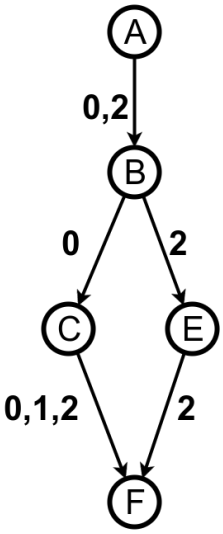
(a) Exact semi-reduced diagram



(b) Exact diagram



(c) Exact fully reduced diagram



(d) Relaxed semi-reduced diagram

FIGURE 1.3 – Variants of multivalued decision diagrams

provided by the MDD can prune more and faster that the existing constraints. However that the strength of the MDD lies in the communication between constraints and in the compression of the graph. Therefore, if only one constraint is applied on the MDD (one table, one regular, one alldiff,...), the propagation gain will be low compared to domain-based CP propagators.

1.4 Performance profiles

This thesis introduces some propagators based on multivalued decision diagrams. Those can be plugged afterwards into a Constraint Programming framework. In order to assess the performances of those propagators, we run benchmarks on several instances to analyze the relative improvement. However, one problem that often arises with benchmarking comes from the fact that an approach can be better for some instances and less efficient in other cases. In [13], Fleming explained why using the arithmetic mean was not relevant and why the geometric mean was the correct way to compare implementations. However, the mean is a simple number that does not allow to visually grasp which method is the best in which case. An other approach is to

use log-log diagram representing the time used for 2 methods. A 45° line is then drawed in order to compare which method is faster than the other. The drawback of this method lies in the fact that it only allows to compare 2 algorithms in a graph.

The benchmark analysis that we use in the thesis is the performance profiles. The method, introduced by Dolan and Moré in [12] is based on performance ratios. For each instance we compute the minimum computation time across all the algorithms. Then, for each algorithm we compute the relative score.

Considering that P is the set of problems/instances, S is the set of solvers and $t_{p,s}$ the computing time needed to solve problem p with solver s. We define the performance ratio as the ratio between the time taken by the propagator p to solve the problem s and the best time taken by all the propagators to solve that same problem p.

$$r_{p,s} = \frac{t_{p,s}}{\min_{i \in S} t_{p,i}} \tag{1.1}$$

Let n_p be the number of problems, we define $\rho_s(\tau)$ as the probability for solver s to have a performance ratio within a factor τ of the best possible ratio. The performance profile of set of solvers S on a set of problems P is the plot of the evolution of $\rho_s(\tau)$ $\forall s \in S$ compared to τ.

$$\rho_s(\tau) = \frac{1}{n_p} size\{p \in P : r_{p,s} \leq \tau\} \tag{1.2}$$

As an example, you can find times obtained for 3 solvers running on 10 problems on the left side of table 1.2. The $\perp$ sign means that the solver reached the maximum allowed time to solve the problem (150 for example in our case). For those times, we compute the performance ratios thanks to the equation 1.1. Finally, based on those ratios, we plot the performance profile presented on figure 1.4. We can see on that graph that the solver A is faster than solver B in general, but if we allow more time to the solver, the solver B solves more instances than solver A. the solver C is in general slower than A and B, and is able to solve less problems than both other solvers.

	Time S_A	Time S_B	Time S_C	Ratio S_A	Ratio S_B	Ratio S_C
p_1	10	30	20	1	3	2
p_2	20	25	30	1	1.25	1.5
p_3	15	45	45	1	3	3
p_4	10	35	15	1	3.5	1.5
p_5	30	60	$\perp$	1	2	∞
p_6	25	70	10	2.5	7	1
p_7	5	15	6	1	3	1.2
p_8	100	50	$\perp$	2	1	∞
p_9	$\perp$	20	$\perp$	∞	1	∞
p_{10}	$\perp$	10	$\perp$	∞	1	∞

TABLE 1.2 – Analysis of benchmarks results

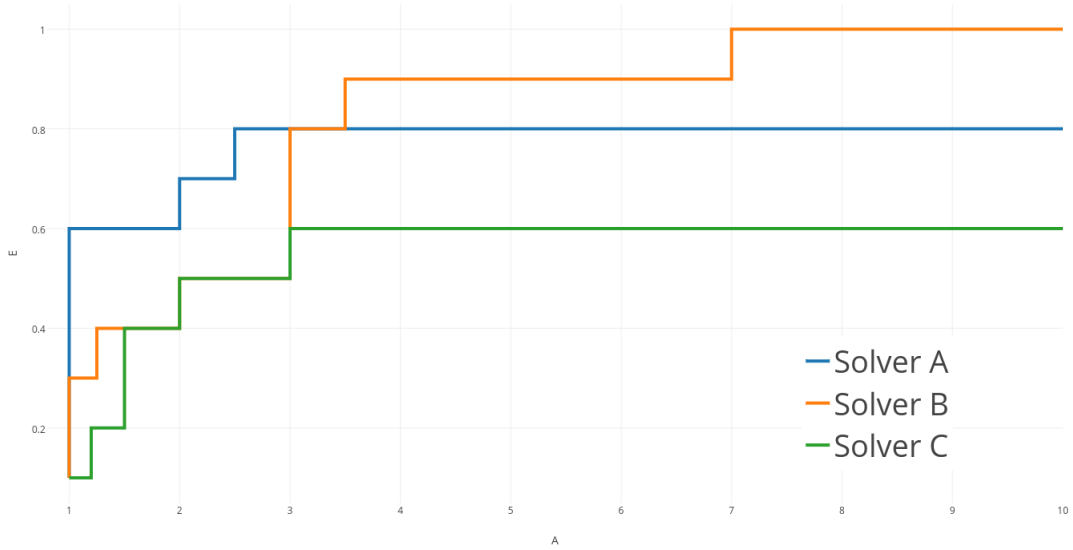


FIGURE 1.4 – Performance profile example relative to results of table 1.2

1.5 OscaR

OscaR [31] is a Scala [21] toolkit developped by the BeCool research lab, CETIC and n-Side in Louvain-la-Neuve. The framework is used to solve Operations Research problems and is composed of several packages, including a constraint programming section that we improve in this thesis. More informations and the source code of the project can be found on `https://bitbucket.org/oscarlib/oscar/wiki/Home`.

CHAPTER 2
THE GLOBAL MDD CONSTRAINT

The MDD-4R algorithm was proposed by Perez and Régin in [23]. The basic idea is that an instantiation of the variables must correspond to a path from the root to the end node in the diagram. The propagation works by deleting edges when values are removed from the domain of a variable. The removal of those edges can induce the invalidation of nodes that leads to a suppression of other edges. Then, if for a certain variable there are no more edges with a certain value present in the MDD, we can remove this value from the domain of the considered variable.

2.1 Structure

As explained above, we deal with decision diagrams containing at least all the solutions. With this property in mind, we can already define a global constraint in a CP framework like OscaR that allows us to understand how the propagation on this structure is handled, without knowing how the decision diagram is built. The various ways of constructing the MDD objects are detailed chapters 3, 4 and 5.

Static MDD: The global constraint receives a static MDD (a graph like the one on figure 1.2) and turns it into a reversible one. In our architecture, any static MDD is composed of five methods that are needed in order to build the reversible MDD. The abstract structure of the object is the one below. The aim of the framework was to standardise some operations on the structure while still allowing a lot of flexibility for the construction of the MDD. Therefore, it is easy to build a custom MDD. Then, all the functions of the framework are directly usable on it.

```
abstract class StaticMDD {

  /**
    * 4 functions used to modify the paths present in the MDD. The usage
    * of these functions is explained in chapter "Incremental MDD for table constraint"
  */
  def contains(tuple: Array[Int]): Boolean

  def reduce(): Unit

  def addTuple(tuple: Array[Int]): Unit

  def removeTuple(tuple: Array[Int]): Unit

    /**
    *   Allows to add constraints on the MDD and to refine it afterwards.
```

```
    *   Those steps are explained in chapter "Constraints on an MDD"
    */
  def addConstraint(cstr: StaticMDDConstraint): Unit

  def refine(maxWidth: Int): Unit

    /**
    *   Those helper functions allow to build the reversible MDD based on any static
        MDD
    *   They map the id's of the edges and nodes from 0 and create the reversible
    *   structure used in chapter "The global MDD constraint"
    */

  def mapNodes(nodes: Array[ReversibleMDDNode], reversibleContext:
      ReversibleContext): util.HashMap[Long, Int]

  def mapEdgesAndLink(edges: Array[ReversibleMDDEdge], nodes:
      Array[ReversibleMDDNode], nodesMapping: util.HashMap[Long, Int]):
      util.HashMap[Long, Int]

  def getArity(): Int
}
```

Reversible MDD: Once the static MDD is fully finished, we can put it as a constraint. The MDD is reversible because during the search, edges and nodes can be deleted from the MDD and need to be restored on backtrack. The MDD needs to maintain several reversible structures to be able to propagate efficiently:

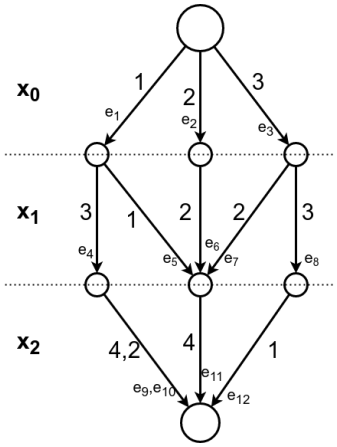
- `support`: For each variable-value association, we maintain the set of edges still present in the MDD, corresponding to the association.
- `in/out-edges`: For each node, a sparse set allows to retrieve all the in/out edges that are still valid.

2.2 Propagation

The main idea of the propagation is to remove all the edges that do not belong to any path from root to end, based on the current domains. When a value a is removed from the domain of a variable x, the edges still active (present in `support[x,a]`) are removed from the MDD (deleted from the in/out edge sets of the corresponding nodes). When the edges are removed, some nodes may have no more in-edge or out-edges. We say that the nodes are invalidated. If a node has no more in (resp. out) edges, all of its out (resp. in) edges must be deleted. This invalidation triggered by the new edge deletion can invalidate other nodes, that can invalidate other edges,... This snowball effect is performed in a BFS or DFS way, and stopped when no more edges need to be deleted or when one domain becomes empty (= Failure).

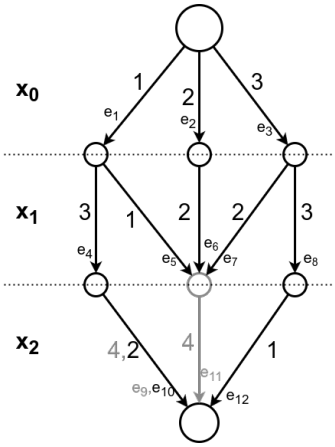
During this propagation phase, some values can be invalidated, meaning that the CP domain of a variable can be reduced. This happens when the `support[x,a]` becomes empty. In that case, the value a must be removed from the domain of x.

As an example, take the MDD and the corresponding supports, presented on the figure 2.1a. Let us imagine that the value 4 is removed from the domain of x_2. Let us see how the propagation works:



Variable	Value	Support
x_0	1	e_1
	2	e_2
	3	e_3
x_1	1	e_5
	2	e_6,e_7
	3	e_4,e_8
x_2	1	e_{12}
	2	e_{10}
	4	e_9,e_{11}

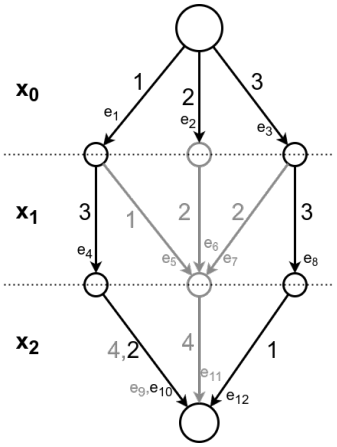
(a) Initial MDD constraint-graph



Variable	Value	Support
x_0	1	e_1
	2	e_2
	3	e_3
x_1	1	e_5
	2	e_6,e_7
	3	e_4,e_8
x_2	1	e_{12}
	2	e_{10}
	4	e_9,e_{11}

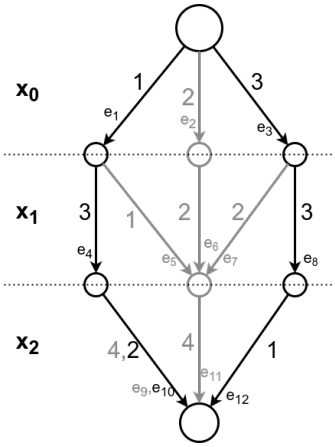
(b) Removal of the edges corresponding to $x_2 = 4$

FIGURE 2.1 – Propagation in decision diagram (Part 1)



Variable	Value	Support
x_0	1	e_1
	2	e_2
	3	e_3
x_1	1	e_5
	2	e_6,e_7
	3	e_4,e_8
x_2	1	e_{12}
	2	e_{10}
	4	e_9,e_{11}

(a) First step of the propagation



Variable	Value	Support
x_0	1	e_1
	2	e_2
	3	e_3
x_1	1	e_5
	2	e_6,e_7
	3	e_4,e_8
x_2	1	e_{12}
	2	e_{10}
	4	e_9,e_{11}

(b) Second step of the propagation

FIGURE 2.2 – Propagation in decision diagram (Part 2)

1. Delete the edges e_9 and e_{11}, those are the ones present in `support[`x_2`,4]`: Figure 2.1b
2. By deleting these edges, one node has no more out edges (the one that is in grey in figure 2.1b). No path goes from the root to the end and pass through this node. It is then invalidated and all of its edges must be deleted
3. We delete e_5, e_6 and e_7: the in-edges of invalidated node and remove them from their support set. At this point, we notice that the `support[`x_1`,1]` is empty and so is `support[`x_1`,2]`. We can remove the values 1 and 2 from the domain of x_1. (Figure 2.2a)
4. We continue the propagation and invalidate the other nodes that have no more out-edges.
5. This triggers the deletion of e_2 (Figure 2.2b). At this point, `support[`x_0`,2]` becomes empty. The value 2 is then removed from the domain of x_0.
6. At this point (Figure 2.2b), the propagation of the MDD constraint is done and the fixed point can continue with other constraints, or the solver can branch if no more constraint are active.

2.3 Reset improvement

The authors of [23] noticed that it would be better to process value removal in batch instead of value per value. Indeed, in the previous algorithm, when 10 values were removed from a domain, the propagation function was called 10 times. In this version, we consider the edge deletion layer per layer.

2.3.1 Principle & example

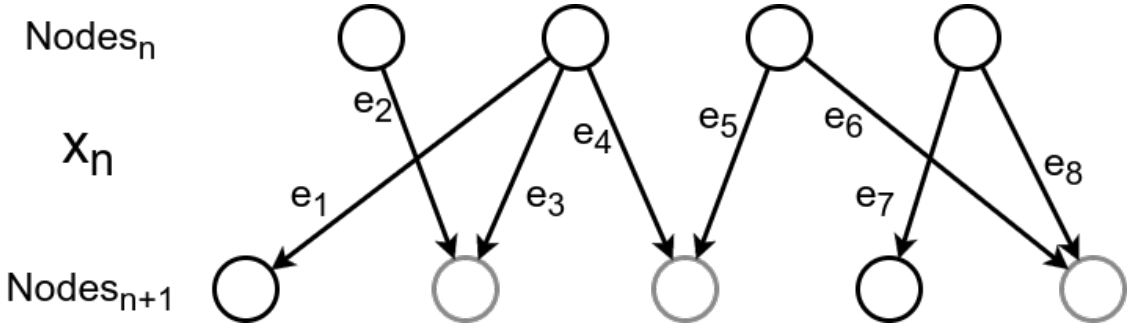
Usually, when the propagation takes place, several values have been removed from the domain of a variable. The reset improvement takes that fact into account and chooses between deleting the edges one by one or clean the whole layer and re-add the edges that are still active afterwards. This layer-per-layer deletion can be done because if we delete edges on layer i, it can invalidate nodes on layer i and layer $i+1$. Then, the deletion of nodes of layer i can only lead to dead-ends and deletion of edges of layer $i-1$, which can lead to deletion of nodes of layer $i-1$, edges of layer $i-2$,...This observation leads us to consider the removal of edges layer by layer, in a breadth first search way.

Considering the deletion layer per layer allows to develop an other way of deleting the inconsistent edges. Indeed, if the layers contains 100 edges and only 2 must be kept, it is better to invalidate all the edges of the layer in batch (remember that clearing a sparse set is performed in O(1)), and then re-add those 2 edges and re-validate the nodes that were affected by this operation. This process is called the reset and is a useful improvement to the propagation as shown in the experimentation phase (section 2.4). In our case, and for the sake of simplicity, we consider that a reset must be performed if the number of edges to remove is greater than the number of edges to keep.

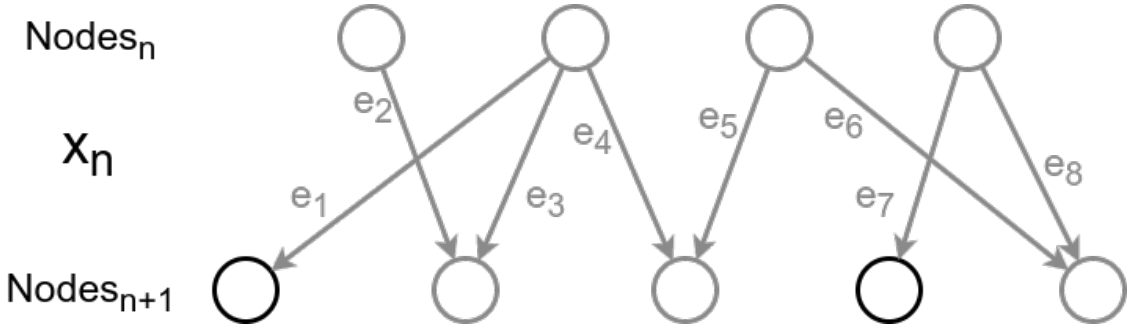
Let us take a look at the reset of a layer. On figure 2.3b, we can see that 3 nodes of the layer $n+1$ have been removed (in grey). All their in-edges are then be deleted as well. As we have 6 deleted edges and 2 that are not, a reset is performed.

The first step is the invalidation of all the edges and the nodes of layer n (Figure 2.3a). Then, we traverse all the active nodes from layer n+1 and re-validate their in-edges. Putting edges back

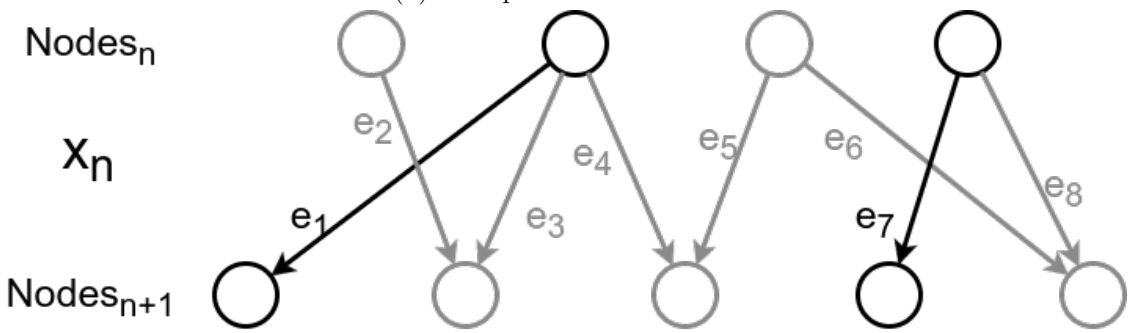
also triggers the re-validation of nodes of layer n. On figure 2.3c, the edges e_1 and e_7 have been re-inserted as their destination node is still valid. Their source node is reconsidered as active.



(a) Inital situation, need to propagate the deletion of the nodes of layer n+1



(b) First phase of the reset



(c) Final state after the reset and the re-insertion of edges and nodes

FIGURE 2.3 – Different steps of a reset on a specific layer of a multivalued decision diagram

2.3.2 Implementation details

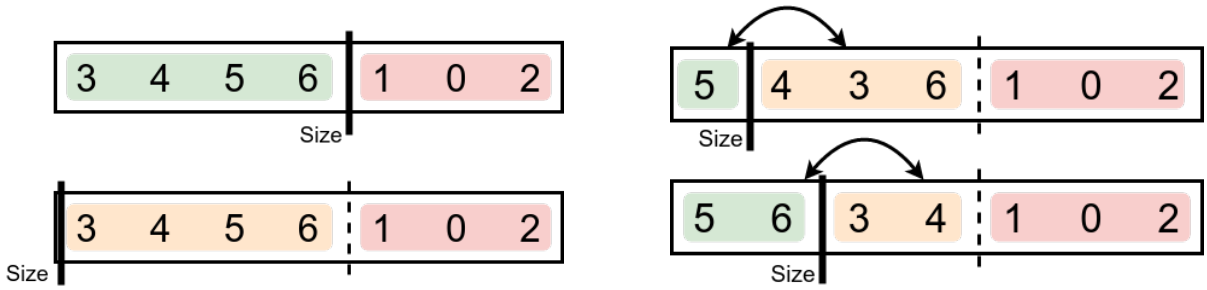
To use the reset trick, some reversible structures must be maintained in order to compute efficiently some parameters of the MDD during the propagation.

- `edgesPerLayer`: For each layer, we maintain the number of active edges. This is simply used to check if a reset is needed or not. This is done by keeping a reversible integer per layer.
- `activeNodes`: For each layer of nodes, a sparse set specifying which nodes are still active in the MDD.

The *activeNodes* sparse set is useful to recover efficiently the invalidated nodes of the following layer after a reset. For example, imagine that the values of the sparse set corresponding to the active nodes of the considered layer are the ones on figure 2.4a.

The first step of the reset is to clear the active node set, which is simply done by setting the size to 0. The result of this operation is shown on the second array of figure 2.4a, where the initial size is represented with a dashed line. This initial size is used in the next steps of the algorithm.

Then, imagine that the nodes 5 and 6 are re-inserted as valid after putting back some edges (Figure 2.4b). It is interesting that the re-activated nodes are in positions $< size_{t_{end}}$, while the invalidated nodes are 3 and 4, which are located between the new size and the previous size. Thanks to the *activeNodes* sparse set, it is then possible to efficiently retrieve the invalidated nodes in linear time and without re-checking the whole layer.



(a) Initial set and cleared set (1^{st} step of the reset)

(b) Set after the re-insertion of 5, then 6 (2^{nd} step of the reset)

FIGURE 2.4 – Propagation after the removal of edges

2.4 Experimentation

The experimentation process was quite basic, we generated 180 custom instances of table constraint with arities ranging from 3 to 20 and tables of size ranging from 500 to 10 000 tuples. At this point we only consider that the MDD is already built and analyze its performances in the CP framework.

The results shown on figure 2.5 show the difference between the classic MDD propagation (one edge at a time, in a DFS or BFS way) and the version using the reset idea, with a reset threshold of 50% (a reset is performed if more than 50% of the active edges of the layer have to be deleted). We can see that the MDD4-R algorithm outperforms the simple MDD4 as predicted.

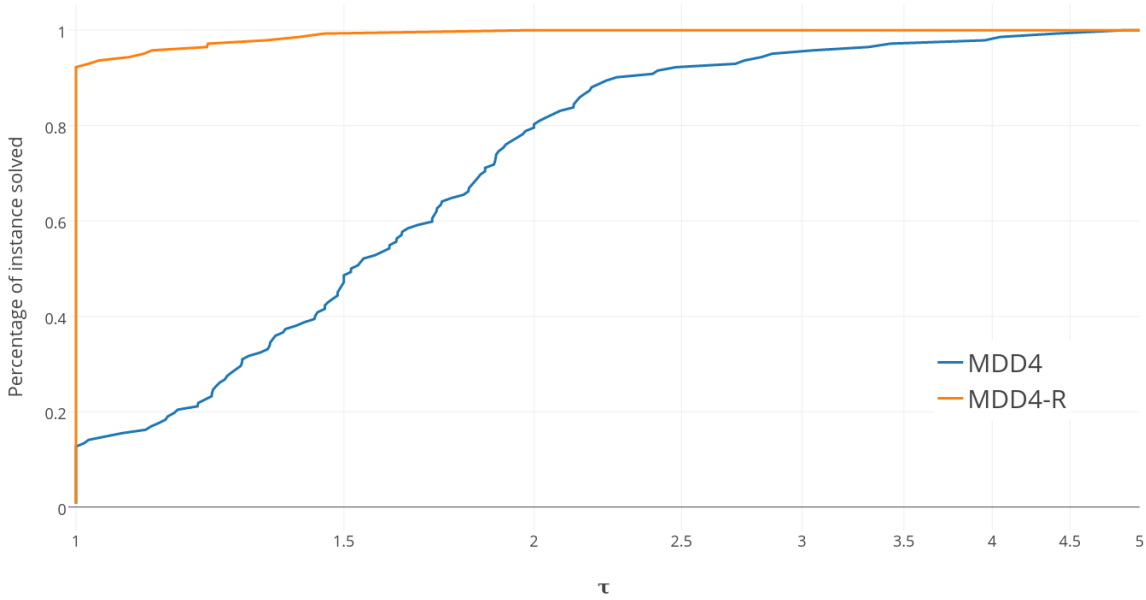


FIGURE 2.5 – Illustration of the performance profiles of the basic propagation algorithm inside a multivalued decision diagram (MDD4) and the improved version using the reset principle (MDD4-R)

CHAPTER 3

INCREMENTAL MDD FOR TABLE CONSTRAINT

3.1 MDD from table

The table constraint is a way to limit solutions of a problems to at most a certain set of tuples. Those tuples represent the valid combinations of instantiations of a set of variables. This constraint is very useful and can be used to represent any combination of constraints. The efficient propagation of table constraint has been studied a lot as they offer a way to decompose any constraint. Several algorithms have been developed such as GAC-4 [20], GAC2001 [4], STR [32] and its improved STR2 version [18]. Finally, Demeulenaere et al. presented the compact table (CT) algorithm [11] that outperforms the previously cited algorithms. Note that all those algorithms have exactly the same filtering, the performances differ only by the speed at which they achieve the consistency.

The interesting point to note is that a table constraint can be directly transformed into a multivalued decision diagram. The basic idea of the algorithm to build the corresponding MDD is to first build a trie from the tuples and then to perform a bottom up pass on the nodes and merge the ones having the same out edges leading to the same destination nodes. The steps of the process are be detailed in the two following subsections.

3.1.1 Cheng and Yap

In [6], Cheng and Yap proposed a simple method to build the semi-reduced MDD corresponding to a table constraint. The first step is to build the trie corresponding to the table with all paths ending at the same node. The second step is the reduction of the MDD by recursively reducing children. The author of [10] proposed a faster alternative that consists in doing this operation in a bottom up way, avoiding the cost of the recursivity. The idea is to get an identifier of the out-edges of a node (in our case we use a string). Then, this node identifier is put into a map linking the identifiers to the nodes. If the identifier is already present in the set, we merge the node that we tried to add to the map with the one already present in the map. This operation is performed by using an HashMap structure. Algorithm 1, borrowed from [10] presents the reduction operation.

Algorithm 1: Bottom up reduction algorithm to turn an MDD into a semi-reduced MDD

Pre : *hash* is a hash function of type Node -> String such that $n_1.hash() = n_2.hash()$ if and only if n_1 and n_2 are duplicates

Post : the diagram is turned into a semi-reduced diagram

Input : *arity* is the depth of the MDD, meaning the number of variables associated with it. Note that the number of layers of nodes is equal to $arity + 1$

Input : *layers* is an array of Sets representing the layers of nodes of the MDD

```
1  nodeHash ← new HashMap[String,Node]
2  l ← arity
3  while l ≥ 1 do
4  |  for node ← layers(l) do
5  |  |  h ← node.hash()
6  |  |  if h ∉ nodeHash.keys() then
7  |  |  |  nodeHash(h) ← node
8  |  |  end
9  |  |  else
10 |  |  |  merge node with nodeHash(h)
11 |  |  end
12 |  end
13 |  l ← l − 1
14 end
```

To illustrate this algorithm, let us take an example with the table 3.1. This constraint has an arity of 4, and the corresponding MDD has then 4 layers of edges. The first step is presented on figure 3.1, where we can see the associated trie, having all of its leaves merged into the end node.

x_0	x_1	x_2	x_3
1	2	4	1
1	3	3	4
1	3	2	5
2	2	4	1
2	3	1	2
3	2	3	4
3	2	2	5

TABLE 3.1 – Example of table constraint

The second step is the bottom-up merging. We begin by considering the nodes on the layer between x_2 and x_3. We can see that the node J and N have the same out edges (one edge with value 4 and destination End) and they can be merged. The same approach is used to merge I-L and K-O. The result of this pass on the layer of nodes are shown on figure 3.2a. Then, we do the same operation with the node layer above, the one between x_1 and x_2. In this pass, the node F and D can be merged, but also E and H, as they both have an edge with value 3 leading to J and one other with value 2 leading to K. The result after this step is presented on figure 3.2b and finally, no more nodes can be merged on the layer between x_0 and x_1. The final semi-reduced MDD corresponding to the table constraint of table 3.1 is then the one shown on figure 3.2b.

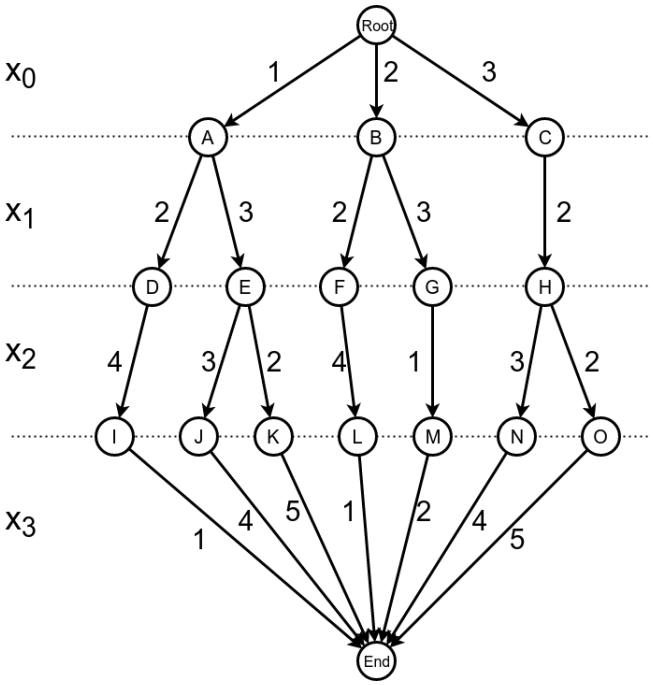
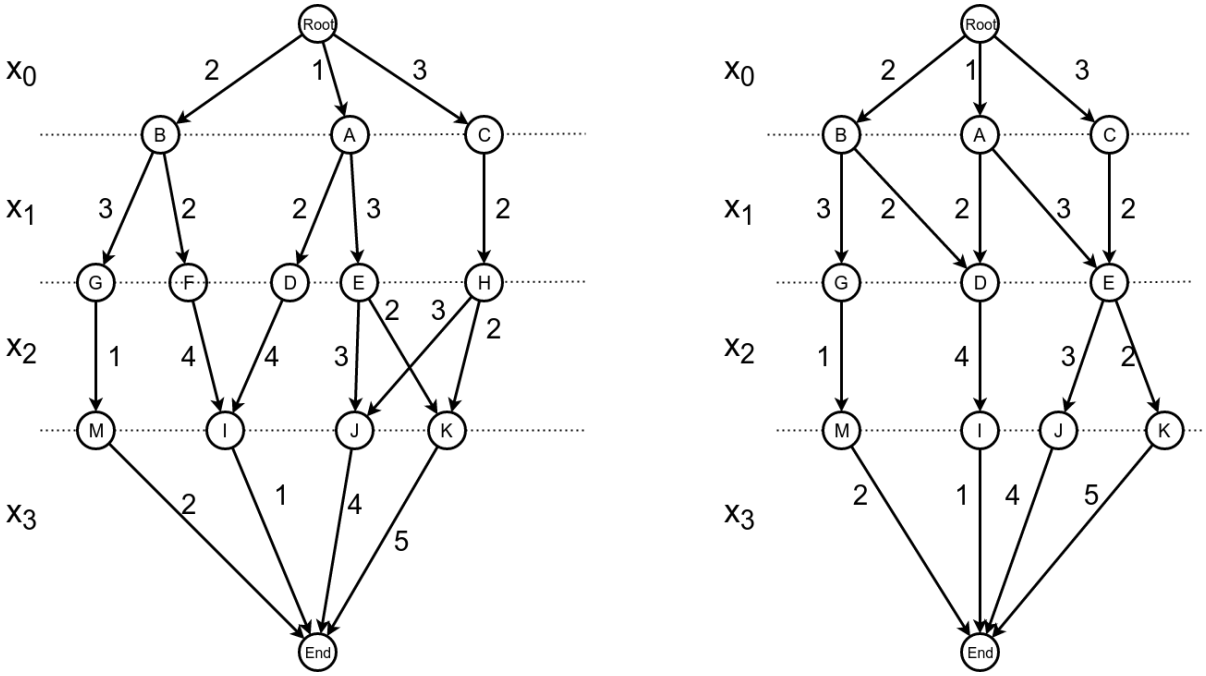


FIGURE 3.1 – Trie corresponding to the table 3.1



(a) MDD after first merging

(b) State after second merging

FIGURE 3.2 – MDD bottom up reduction

3.1.2 Perez & Régin

The algorithm presented by Cheng and Yap has been recently improved in [24] by Perez and Régin. The idea is the same as the previous one, except for the construction of the trie.

The trick used is to sort the tuples in the table, using a radix sort at each node of the trie. This improvement gives a global complexity that is linear with respect to the size of the final trie. Moreover, the random accesses needed to put the tuples in the MDD are reduced to a

minimum. In [6], the tuples are added one by one, and even if one is nearly fully present, in order to add it, we have to start from the root and follow the present path, which implies having more random accesses. By using a radix sort, the algorithm presented in [24] allows to have a good improvement in the creation of the MDD as presented in the experimentation (section 3.3.1).

3.1.3 Complexity analysis the algorithms

Let us use those symbols to describe the complexity:

- r: Arity of the table constraint
- d: Domain size of the table constraint (minimum value up to maximum value)
- N_n: Number of nodes in the trie
- N_e: Number of edges in the trie
- N_t: Number of tuples
- h: Random access time for a hashMap
- H_n: Time needed to compute the hash of a certain node (used to find duplicates)

Building the trie Cheng & Yap: The complexity to build the trie with the basic technique is

$$O(N_t \cdot r \cdot h)$$

because each tuple is added one by one and needs a descent down the whole trie each time, with one random access per value of the tuple.

Building the trie Régin & Perez: The total complexity of the improved algorithm to build the trie is

$$O(N_e \cdot h + N_t \cdot d \cdot r)$$

with the first part of the sum being the creation of the MDD via the random accesses to build the edges. The second part represent the radix sort that is performed on all the tuples at each layer.

Bottom up reduction: As those two complexities are quite hard to compare, we implemented both methods and compared them in section 3.3.1. However, this corresponds to the first part of the algorithm and the construction of the trie. For both algorithm, the bottom up reduction used is the same and has a complexity of

$$O(N_n(h + H_n))$$

because for each node, we compute the hash, and then look in the hash table of the layer if there is a duplicate. Then, if there is a duplicate, we merge both nodes. We do not consider the computation of the merging in the MDD as it is the same as the one needed to compute H_n (in fact, we simply unlink all the out-edges, and redirect the in-edge towards the other node. Both H_n and the merging have a complexity linear regarding the number of out-edges, which is the same than).

3.2 Incremental tuple addition / deletion

The MDD is a complex structure and adding a specific path to an existing graph can be complex. In this section, we present a way to customise an already built MDD by adding and removing specific tuples from the graph, without rebuilding the whole structure.

In order to perform the addition and deletion, Perez has introduced the principle of path isolation in [24]. The complexity of this problems comes from the fact that the MDD should stay semi-reduced after the addition/removal. Moreover, adding a tuple should only add one path and not create more than this path. The two constraints imply that the addition and removal operations cannot be performed greedily.

3.2.1 Tuple deletion

The process used too delete a tuple from an MDD is divided in 3 steps:

1. Isolate the path of the tuple to be deleted
2. Delete this path
3. Reduce the MDD again in a bottom-up way

However, doing a full reduction on the whole diagram after each deletion has a huge time consumption. Hopefully, it is possible to reduce the MDD by applying the reduction only on nodes that were on the path, and altered by the deletion. As the hashing and the merge are performed in nearly constant time, we can delete a tuple efficiently, with a total complexity that is linearly dependent of the arity of the diagram.

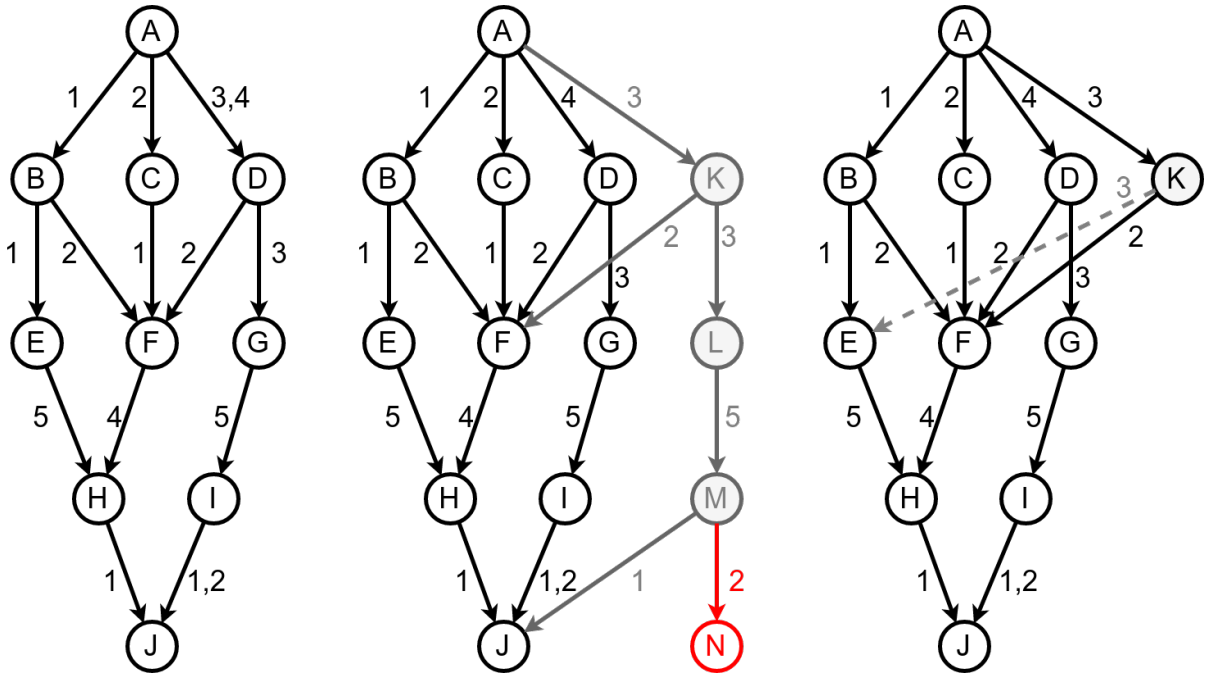
Imagine that we have a static MDD corresponding to the one on figure 3.3a and we want to delete the tuple (3,3,5,2). The path isolation is presented in figure 3.3b, and note that allf paths that were present on 3.3a are still in 3.3b. This step is then simply a decompression of one path of the MDD. Then, the edge going from M to N is deleted, and if M had not had any more out-edges, the node M would have been deleted, triggering a propagation like the ones already seen in the previous section (invalidated node invalidates edges, which can invalidate nodes, ...). We can see that the only nodes that have to be merged again are the ones on the extracted path: K, L and M. We start from the bottom and notice that we can merge M into H. Then L into E. K cannot be merged, which gives us the final result on figure 3.3c.

3.2.2 Tuple addition

The same ideas works for the tuple addition. However a small modification is needed, we have to isolate the path as long as it is possible. Since the tuple is not supposed to be already present in the diagram, at some point, the algorithm reaches a dead end. Once this dead end reached, we build the rest of the path, based on the values of the tuple being added. Then, we can apply the reduction on all the nodes of the path, plus the ones that were modified during the extraction.

3.3 Experimentation

We have seen two methods to build a multivalued decision diagram from a table. Then, we saw how to incrementally modify the tuples/existing paths of the diagram. This section provides a short experimentation of the different operations to see if they are effective.



(a) Original MDD (b) MDD after path isolation (c) MDD after removal and bottom up reduction

FIGURE 3.3 – Deletion steps of a specific tuple in a multivalued decision diagram

3.3.1 Cheng vs Régin

The first benchmark presented is the one that compares the efficiency of the algorithms of Cheng and Yap with the one of Perez and Régin. The table instances were generated randomly with several different arities, MDD-final width and number of tuples. The results are shown on figure 3.4, where we can confirm the results of Perez. Indeed, in the paper, the performances of their algorithm used to build the trie outperformed the one of Cheng & Yap. This is confirmed by our benchmark, even if on some instances, the basic version is still faster than the optimized one.

3.3.2 Table vs incremental tuple addition

The second benchmark is a bit more complex. It is used to show the efficiency of the incremental addition of tuples to an existing MDD. The experiment was done with a basic MDD of arity 7, containing 200 000 tuples. Then, we added a certain number X of tuples to this MDD, either by recreating the diagram from scratch with the 200 000 + X tuples, either by using the already created diagram, and by incrementally adding the X tuples to it.

The results of this experiment are presented on figure 3.5, and we can see that both constructions grows linearly with the number of tuples added (a linear regression has been performed to convince you that it is the case). However, we can see that after a certain threshold of 30% (60 000 tuples), it is faster to rebuild the MDD from scratch with the method proposed by Perez & Régin. Even if this is the case, we could imagine an application where one thread is running a specific model, yielding solutions to an other thread that maintains the MDD corresponding to those solutions. In this case, even if the creation from scratch would be faster, it could be more interesting to build the diagram progressively in another thread, rather than waiting for all the solutions before starting the construction of the structure.

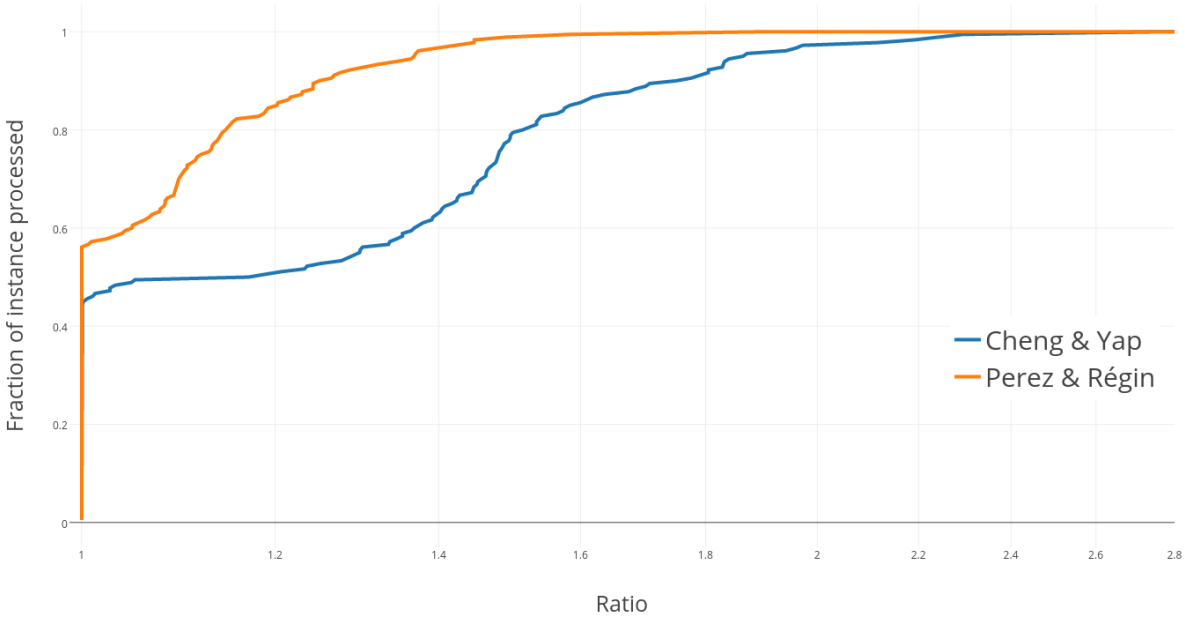


FIGURE 3.4 – Comparison of ways to build an MDD from a table

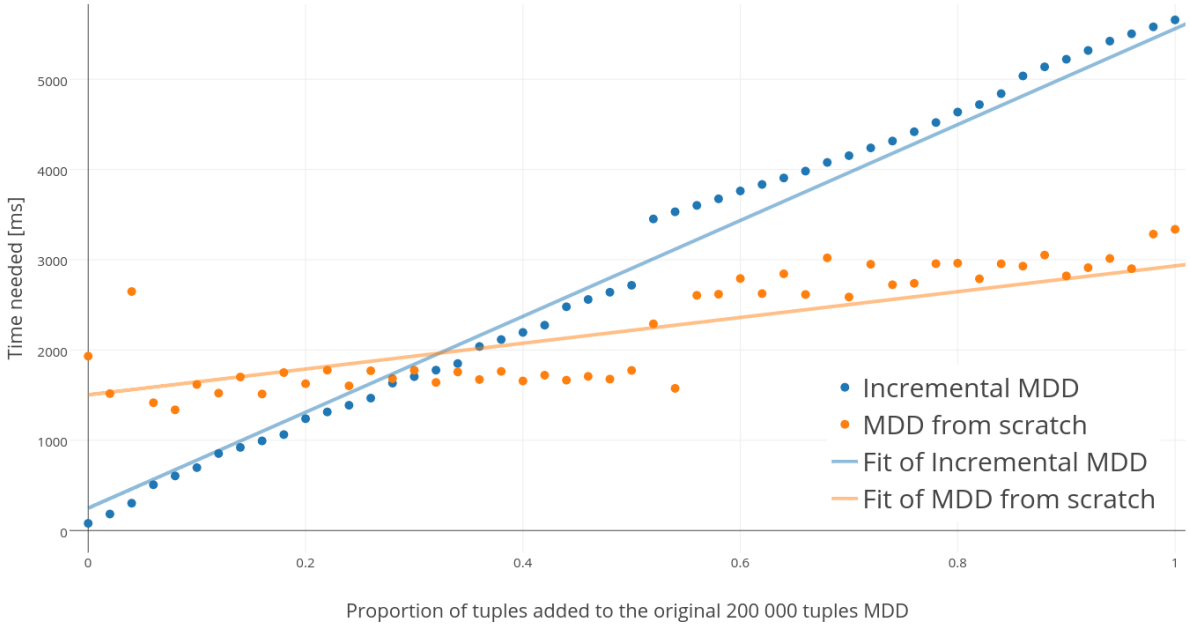


FIGURE 3.5 – Analysis of the time needed to build a MDD of arity 7, with (200 000 + x) random tuples, where x is the proportion added, based on the 200 000 initial tuples. The lines show the difference between adding tuples to the existing MDD of size 200 000 incrementally, or rebuilding the MDD of size (200 000 + x) from scratch.

CHAPTER 4

MDD FOR REGULAR CONSTRAINT

The previous chapter offered us a way to build a semi-reduced multivalued decision diagram from a table constraint. This object contained all the solutions as paths and only the solutions. This section shows that it is also possible to build efficiently a decision diagram from a regular constraint that also contains all the solutions as paths and only the solutions. The idea of representing a regular constraint with a layered directed acyclic graph was introduced by Gilles Pesant in [25] without using the term MDD for the underlying structure used by the propagator.

4.1 Deterministic Finite Automaton

A DFA is defined as a 5-tuple containing:

- A finite set of states (nodes)
- A finite set of symbols (the alphabet)
- A transition function (the edges)
- An initial state
- A set of accepting states

A DFA is in general used to represent any regular expression (Regex). Let's take the following regular expression as an example.

$$[a(ca)^*] \mid [(ca)^* \; b \; a^* \; c] \tag{4.1}$$

One possible representation of this regex is on figure 4.1 and corresponds to the following 5-tuple.

- States: 0, 1, 2, 3
- Alphabet: {a, b, c}
- Transition function f(state, letter) = destination state:
  - f(0,a) = 1
  - f(0,b) = 2
  - f(1,c) = 0
  - f(2,a) = 2

- f(2,c) = 3

- An initial state: 0
- The set of accepting states: {1, 3 }

This automaton (figure 4.1) defines the set of sequences of letter from the alphabet {a,b,c} that can be accepted by the automaton. As an example, the sequences aca, acbc, acbaaaac, ... are accepted, while aa, acc, acba, ... are not.

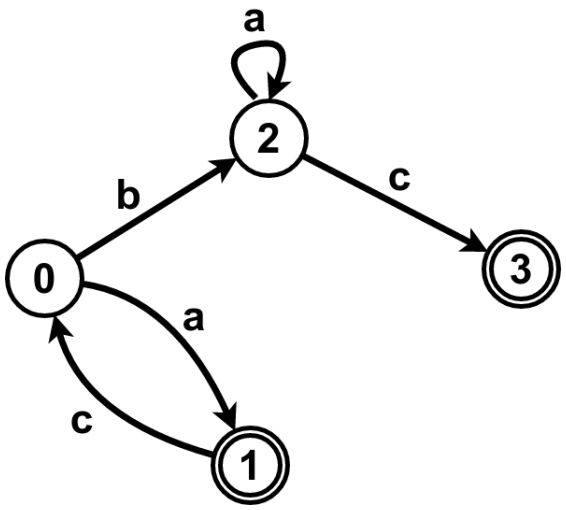


FIGURE 4.1 – DFA representing the regex of equation 4.1

4.2 Regular constraint and MDD

Based on a deterministic finite automaton, we can build what is called a regular constraint. A regular constraint on a sequence of variables enforces that when all the variables are bound, the resulting sequence is accepted by the automaton. The current state of the art propagator for the regular constraint is the one developed by Quimper & Walsh in [27]. In this work, they describe how to represent a regular grammar or a DFA by a conjunction of table constraints which are then propagated by a state of the art table constraint propagator like *Compact Table* [11].

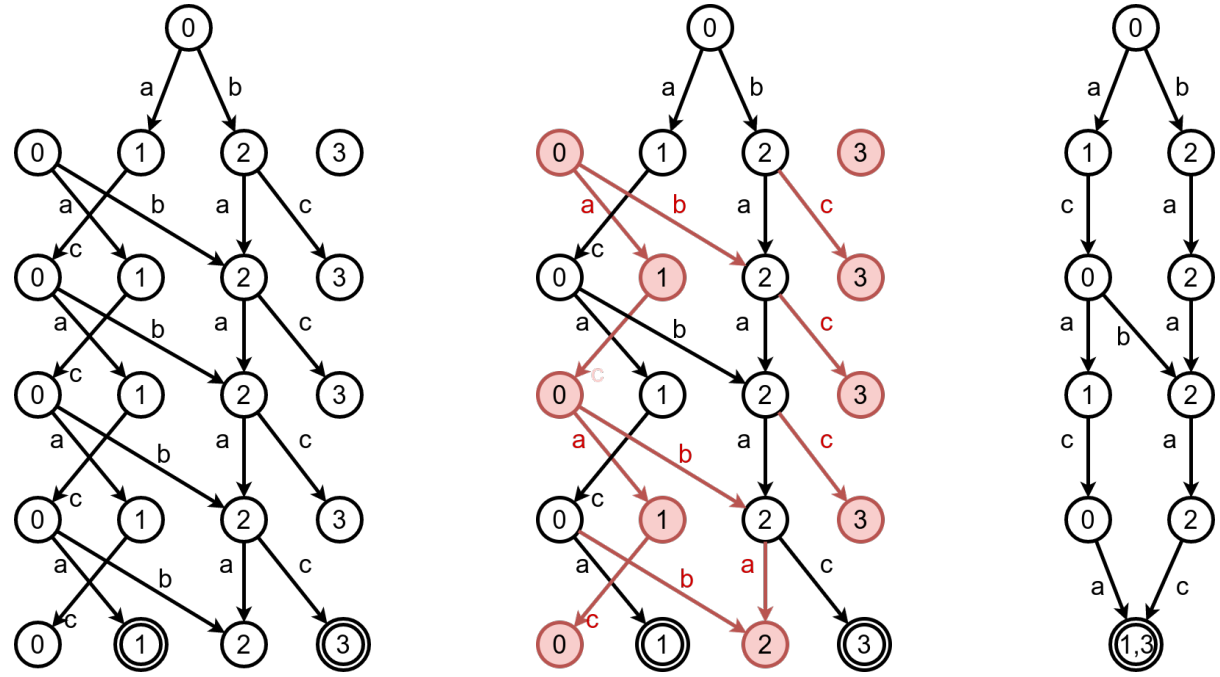
In this chapter, we propose a way to represent the regular constraint via a decision diagram. This static MDD can be combined with other MDD constraints (as explained later), and be injected in a global constraint. The idea is that the other table algorithms are really poor when combined with other constraints.

As the constraint is applied on a sequence of finite size, we must also specify the length of the built MDD. The idea is to build the regular layer by layer. Each layer (except the first) contains all the possible states. Next, we connect each state with its direct successors of the next layer. Then, for the last layer, we keep only the nodes corresponding to accepting states and delete the others. This creates dead ends (edges leading to no valid node). We delete those edges and propagate the deletion. The same is performed for the edges without any top node in the first layer. The total complexity of this creation is

$$O(r(N_{states} + N_{transitions}))$$

with r being the arity of the MDD, N_{states} the number of states in the automaton and $N_{transitions}$ being the number of transitions in the automaton.

The figure 4.2 shows the different steps of the process applied on the exemple DFA presented in figure 4.1, with a depth of 5. Note that it is also possible to build the MDD by performing a breadth-first search inside the automaton, starting from the initial node.



(a) Basic regular representation

(b) Remove useless nodes and transitions

(c) Final MDD-regular

FIGURE 4.2 – MDD bottom up reduction

4.3 Experimentation: Nonogram

Japanese crossword, Hanjie or nonogram is a type of problem that involves logic and coloring. The aim of the game is to fill a grid by coloring it, based on information on the rows and columns. The game is usually played in black and white, but can also contain colors. In our case, we focus on black and white nonograms but the framework could be easily augmented to handle such nonograms. An example of a solved instance is shown on figure 4.3. The rules are simple, take for example the first row, which is 1 2. This means that at some point on the row, we must have one black square, followed by at least one blank square and after, a group of 2 black squares. From that perspective, a nonogram is very like a sudoku, we have cells to fill and constraints between those cells. Fortunatly, those row/line constraints can be expressed with DFA's and their associated regular constraints.

The automaton used to represent the 1 2 row constraint is the one on figure 4.4 , where the 1's represent a black square and the 0 a white one. To extend the framework to other colors, simply add more possible values other than 0. The automaton starts with 0,1,2 or more white square, then one black square that has to be followed by at least one white cell (mandatory 0 and self loop to allow more than one 0). Then, 2 black cells must come, and at the end, we can put as many white cells as needed to finish the row (accepting state).

Now that we have defined how to build a constraint for a row or a column, the model is fairly

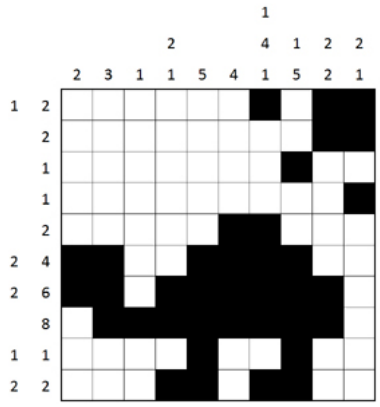


FIGURE 4.3 – Example of a solved nonogram

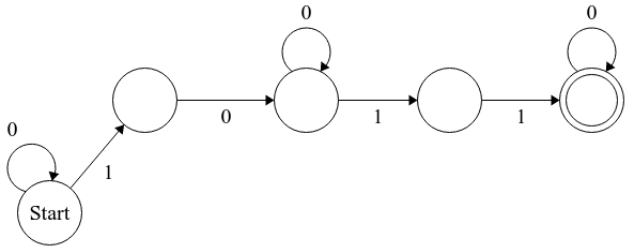


FIGURE 4.4 – Automaton used to represent a row of a nonogram instance: 1 2 constraint

easy to define. We build a matrice of variables that can take the value 0 or 1. Then, we add the regular constraints on the corresponding rows and lines.

The results of figure 4.5 show that the multivalued decision diagrams used as global regular constraint are competitive with the state of the art propagator even if a bit slower in our implementation. Still, the chapter 6 shows that regular combined with constraints (static) or costs (dynamic) outperforms the table implementation that does not allow much communication between the involved constraints.

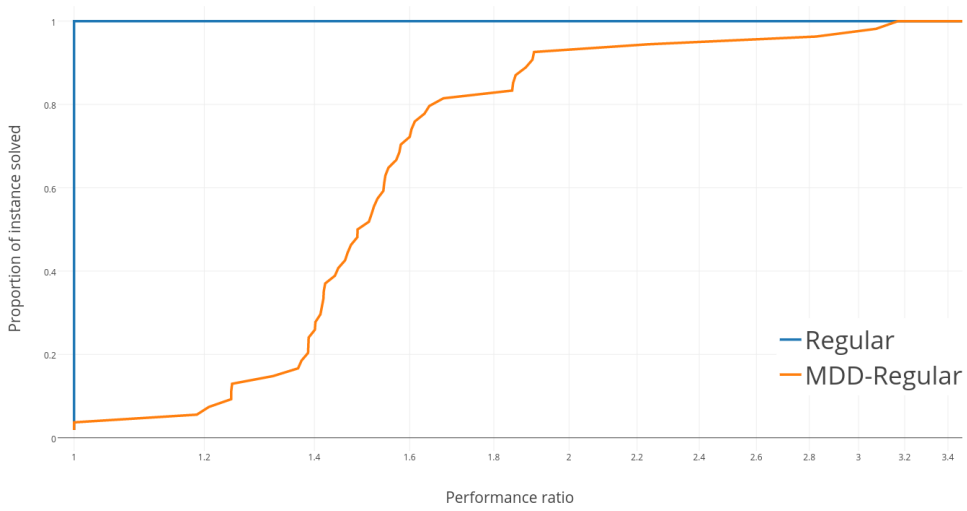


FIGURE 4.5 – Performance profile comparing the efficiency of table-based regular versus MDD-based regular

CHAPTER 5

CONSTRAINTS ON AN MDD

5.1 Preliminaries

In the previous sections we saw how to build an MDD corresponding to a specific constraint. However, we restricted ourselves to MDD for which each path is a solution and each solution has its path. The issue is that we want to use the MDD to solve very hard problems that can have an exponential number of solutions, meaning that the number of paths inside the decision diagram can become exponential as well. We know that we want to use those MDD as global constraint in a CP solver, and those solvers are using polynomial propagation to solve non polynomial problems in a faster way. However, if we want to compute what is called the *exact diagram* for a problem we can have a non polynomial complexity.

Therefore, we restrict ourselves to limited-width MDD introduced by Bergman, A. Cire, N. Hooker and van Hoeve in [3], [8] and [14]. In these works, we begin with an MDD representing the cartesian product of the domains. Then, we increase the number of nodes on each layer in a heuristic process called *refinement*, by limiting the maximum number of nodes on a layer to a constant (passed as input). Moreover, we suppress the edges inconsistent with the constraints to refine even more. Finally, we can plug this static MDD containing some information about the constraints as a global CP constraint that can improve the propagation of classical other constraints.

Note that the MDD constraint is GAC when compared to the static MDD. However, as the static MDD is not always exact, it is not GAC with respect to the constraints added onto the decision diagram. So when modelling the problem with CP, we still have to use the full CP model. The MDD is only a redundant constraint that may reduce the number of backtracks and speed up the propagation.

Initial MDD: The first step to build an MDD with constraints is to start with a diagram of width 1 containing the cartesian product of all domains as possible paths.

Refinement: The second step is called the refinement and is a fixed-point iterative process. At each step of the fixed point, the algorithms does a pass on all the layers in a top-down way. If the number of nodes in the layer is smaller than the maximum allowed width, the nodes with more than 1 in-edge are splitted to reach that maximum width. The out-edges of the new nodes are pruned. If no more node is splitted and no edge is deleted on a pass, the iteration stops and the diagram can not be more refined. The object *splitted node* corresponds to a virtual node built with only one incoming edge. Then the list of splitted node objects is sorted heuristically

and the real nodes of the layer are splitted greedily based on the sorting, until the maximum width is reached.

Node states: Each constraint computes certain information about the state of the nodes of the MDD. This step is done at the beginning of the fixed point loop. The states are constraint-dependent and allow to define if an edge should be deleted.

Edge deletion: The process of deleting an edge is based on node states. An edge (u,v) is processed by a constraint that will analyze the states of u and v. Based on those states, the constraint can tell if (u,v) does not belong to any path respecting this constraint. For example, if all the paths leading to u contain the value 1 and an all different is applied on the MDD, we now that no edge with value 1 can come out of the node u. If such an edge exists, it should then be removed.

A small example

Imagine that we have a set of variable with the domains presented here:

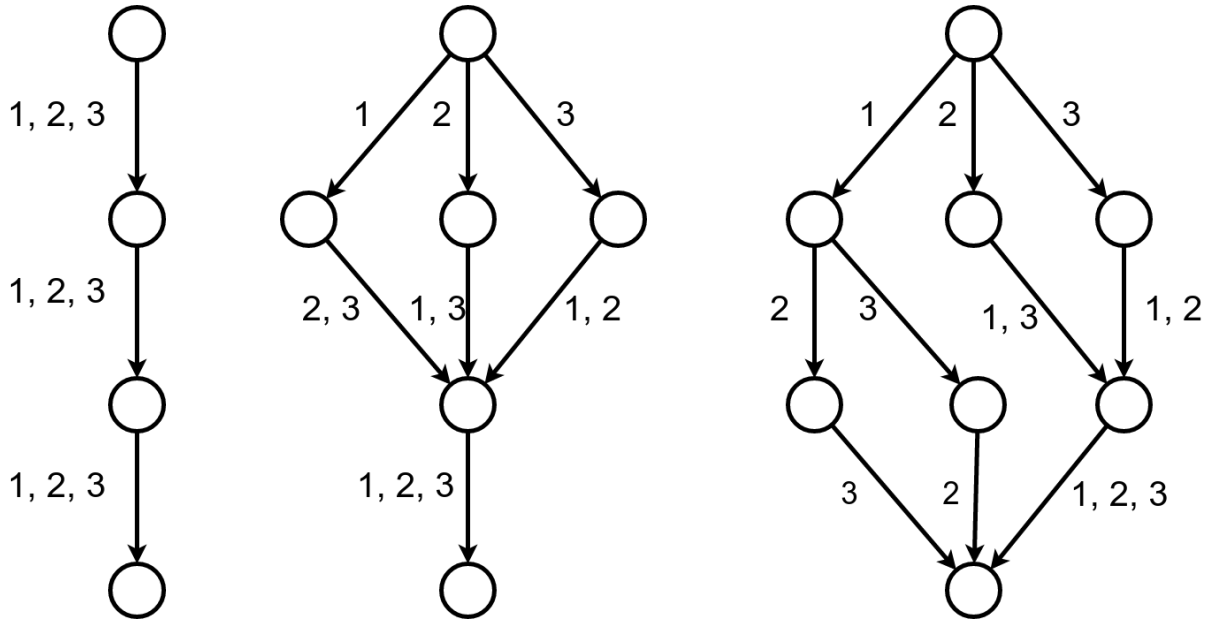
$$x_0 = \{1, 2, 3\}$$
$$x_1 = \{1, 2, 3\}$$
$$x_2 = \{1, 2, 3\}$$

and the only constraint that is applied is an `AllDifferent(`x_0`, `x_1`, `x_2`)`. For this example, we will use a maximum width of 3.

Initial MDD : The initial MDD is presented on figure 5.1a. As explained above, the possible paths corresponds to the cartesian product of the variable domains. In our case, there will be 6 solutions to the problem. However, the MDD currently contains 27 paths. We refine it based on the constraint and see how this process restricts the number of possible paths.

Refinement: The first step is to expand the first layer. Figure 5.1b shows the splitting. Then, we can suppress some edges based on the fact that they are inconsistent with the all different constraint. When the path start with a 1, it cannot contain any 1 afterwards (we will see later how specific constraints can work, currently we only think logically). Then, we split the second layer. The problem is that there are 6 edges, corresponding to 6 potential nodes while the maximum width is 3. The solution is to greedily pick 2 edges and *extract* them. The result of this step is presented on figure 5.1c, where no more edges can be pruned and no more layer can be expanded.

In this final diagram, we can see that only 14 paths are present. We say that the MDD contains more information about the constraints. We started with 27 paths and now we have only 14, meaning that we removed already 13 inexact paths, only by allowing a width of 3. This idea is the basis of the refinement and increasing the maximum width provides a MDD containing more information. However, building a larger MDD needs more computational resources, meaning that we have a trade-off between the strength and the time complexity. One benefit of the constraint-MDD is that it allows to have constraints of custom strength based only on the maximum width. Another asset of this process is that all constraint are used simultaneously, decisions are anticipated by expanding nodes and a better communication between constraints occurs as presented in the later experiments. Note that the algorithm does not find the exact decision diagram with the correct width if it exists. For example, the diagram on figure 5.2 shows an exact decision diagram corresponding to the problem above, and with the correct width of 3. However, finding that diagram with the refinement method is computationally too complex and we limit ourselves to the greedy refinement method with the hope that it reaches a decision diagram as exact as possible.



(a) Basic MDD representation based on domains.

(b) MDD after splitting of the first layer and inconsistent edges deletion.

(c) MDD after splitting of the second layer and inconsistent edges removal.

FIGURE 5.1 – MDD refinement process example

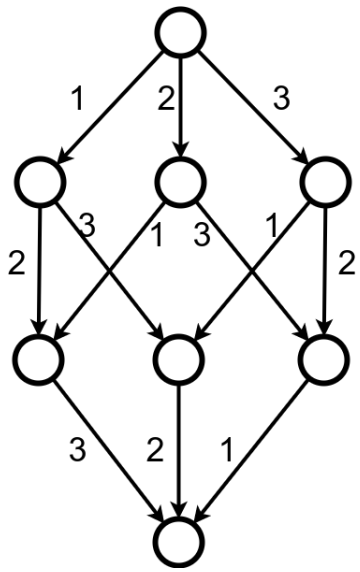


FIGURE 5.2 – An exact semi-reduced MDD corresponding to the problem of 3 variables with a domain of 3 values and an all different constraint on top of those 3 variables

5.2 Some constraints

In this section, we describe 2 global constraints on MDD and how they allow to prune edges based on their associated states. The constraints were implemented in our MDD-based framework and can be added easily to a refinable MDD. Later, we analyze how a specific scheduling constraint was implemented. The refine function manages itself the added constraints and refine the MDD accordingly as developped in the refinement algorithm above.

5.2.1 Knapsack

The first constraint that we analyze is the lose knapsack, in order to understand the basic process of state computing and edge deletion. This propagator was presented in [1] and we describe briefly the maths behind it.

Let X be an array of n variables x_i having a certain domain $D(x_i)$. For each variable, each value of the domain is associated to a cost $f_i(x_i)$, or more precisely, each edge (u,v) is associated to a weight $w_{(u,v)}$. We also define a maximum cost B on the array/path. A correct instantiation of the n variables to values must follow the constraint:

$$\sum_{i=0}^{n-1} f_i(x_i) \leq B \tag{5.1}$$

In order to use this constraint in the MDD, we must define the states associated with it and the rules used to define if an edge is deletable.

States: We define 2 states, associated to the shortest path from the root or end node to the node of interest. The first state is the less costly path from the root to node v and is defined recursively as:

$$S_{\downarrow}(v) = \begin{cases} 0 & \text{if v is the root} \\ \min\limits_{(u,v)\in\delta^{in}(v)} S_{\downarrow}(u) + w_{(u,v)} & \text{otherwise} \end{cases} \tag{5.2}$$

In the same way, we can define the less costly from the current node to the end as

$$S_{\uparrow}(v) = \begin{cases} 0 & \text{if v is the end} \\ \min\limits_{(v,w)\in\delta^{out}(v)} S_{\uparrow}(w) + w_{(v,w)} & \text{otherwise} \end{cases} \tag{5.3}$$

Those two states can be computed in a top-down and a bottom up pass, both being allowed in our framework.

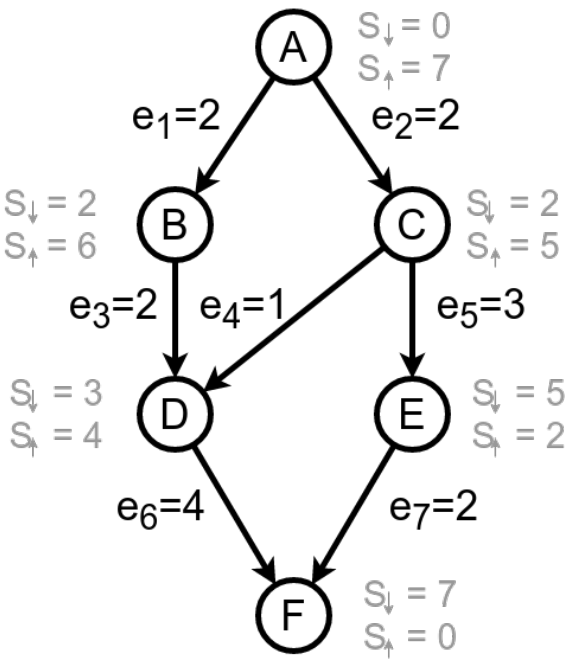
Edge pruning: To prune an edge (u,v), we must check if at least one path going through this edge allows to have a total weight smaller than B. Based on the computed states, it is fairly simple to observe that an edge should be removed. If the shortest path from the root to the end that uses this edge has a value greater than B we prune it (because no path going through that edge is able to meet the constraint). The pruning rule that defines if an edge (u,v) should be deleted is then:

$$S_{\downarrow}(u) + w_{(u,v)} + S_{\uparrow}(v) > B \tag{5.4}$$

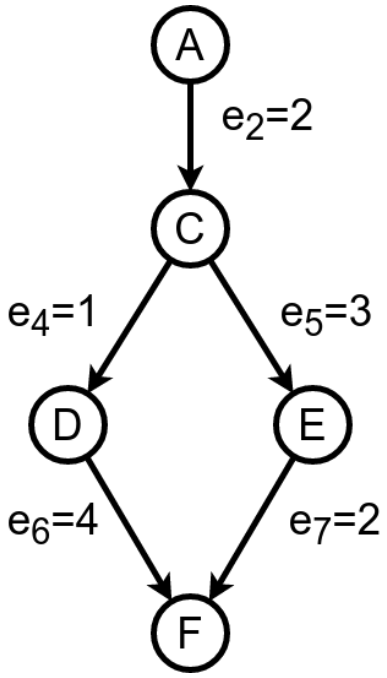
Take a look at figure 5.3a where the edges have their associated weight shown (we consider that the value of the edge is their weight for simplicity and clarity). The states $S_{\uparrow}$ and $S_{\downarrow}$ have been computed for each node. Imagine that we put B = 7, meaning that the paths must have a sum of at most 7. Based on the edge pruning rule, we can observe that the edges e_1 and e_3 must be deleted. This can be verified by noticing that those edges belong only to the path A-B-D-F, that has a total weight of $8 > 7$. The other edges can be kept, because they belong to either A-C-D-F that has a total weight of 7, either A-C-E-F that has a total cost of 7 as well (Figure 5.3b for the pruned MDD).

5.2.2 All different

Now that we have studied an easy constraint, we can start to analyze more complex ones. A very useful constraint in CP is the *All different* constraint, the one that enforces that a solution



(a) Weighted MDD



(b) Weighted MDD of figure 5.3a after pruning

FIGURE 5.3 – Effect of pruning using the cost constraint and an upper bound of 7

only contains different values. For domain propagation, Régin proposed a method based on value graphs in [28] and the MDD propagator is presented in [1]. In order to tackle this problem in the MDD framework, we compute 2 types of state, each one having a top-down and bottom-up component. This means that we have 4 different states.

States: The first type of state is the set A, which contains all the values that *must* have been taken from the start to the node, in order to reach that node. A way to define this mathematically is the following, where the first state can be computed in a top down pass while the second is computed in a bottom up pass on the graph. Note that $\phi_{(u,v)}$ corresponds to the value of the edge (u, v).

$$\begin{cases} A_v^{\downarrow} & = \cap\{A_u^{\downarrow} \cup \{\phi_{(u,v)}\} : (u,v) \in \delta^{in}(v)\} \\ \\ A_v^{\uparrow} & = \cap\{A_w^{\uparrow} \cup \{\phi_{(v,w)}\} : (v,w) \in \delta^{out}(v)\} \end{cases} \tag{5.5}$$

Figure 5.4 presents the A states computed for each node of a certain decision diagram. If we look at the $A_{\downarrow}$ set of node C, we can say that any path coming from node A contains at least a 3. This means that no edge leaving D can contain those values. Otherwise, the all different constraint would be violated. This rule allows us to remove the inconsistent edge value 3 between C and E. The formal definition of this rule will be given later in this section and this small example is only present to give you the pruning spirit that goes with the states and the all different constraint.

The second type of state used is the set S, which is the set of values that belong to a least one path from root (or end) to the considered node v. Its mathematical definition is quite comparable to the A_v states:

$$\begin{cases} S_v^{\downarrow} & = \cup\{S_u^{\downarrow} \cup \{\phi_{(u,v)}\} : (u,v) \in \delta^{in}(v)\} \\ \\ S_v^{\uparrow} & = \cup\{S_w^{\uparrow} \cup \{\phi_{(v,w)}\} : (v,w) \in \delta^{out}(v)\} \end{cases} \tag{5.6}$$

Node	$A_\downarrow$	$A_\uparrow$
A	$\emptyset$	$\emptyset$
B	$\emptyset$	$\emptyset$
C	$\{3\}$	$\{4\}$
D	$\emptyset$	$\emptyset$
E	$\{3\}$	$\{4\}$
F	$\emptyset$	$\emptyset$

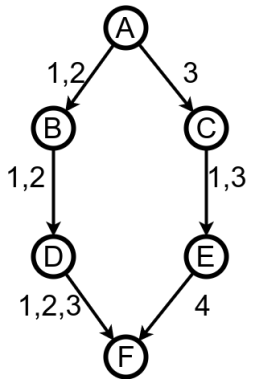


FIGURE 5.4 – Example of usage of state A in all different constraint

The idea for that type of state is a bit more complex. It also relies on the fact that a value must have been taken when a certain node is reached and therefore cannot be present as out-edge value. For example, on figure 5.5 (same diagram than the one on figure 5.4, with the edge (C,E) with value 3 removed as explained above), we have a multivalued decision diagram for which we have computed the S states. Let us take a look at node D and the $S_\downarrow$ associated with it. It contains the values 1 and 2. To reach node D, we have to assign values to 2 variables before: x_0 and x_1. If there are only 2 values that can be used for 2 variables and we have the all different constraint, this means that the 2 values have already been used on the 2 previous variables (because of the all different constraint). Therefore, we cannot assign one of those 2 values to an edge leaving E. This leads to the deletion of the edges (D,F) with values 1 and 2. This pruning also works in a bottom up fashion with the same idea: If n values are assigned to n variables with an all different on the variables, that instantiation is a permutation of the n values, and they cannot be used later. This rule is formalized below.

Node	$S_\downarrow$	$S_\uparrow$
A	$\emptyset$	$\{1,2,3,4\}$
B	$\{1,2\}$	$\{1,2,3\}$
C	$\{3\}$	$\{1,4\}$
D	$\{1,2\}$	$\{1,2,3\}$
E	$\{1,3\}$	$\{4\}$
F	$\{1,2,3,4\}$	$\emptyset$

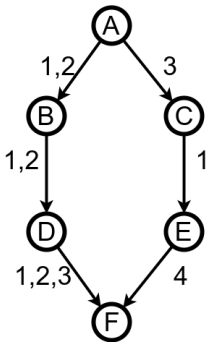


FIGURE 5.5 – Example of usage of state S in all different constraint

Edge pruning: We have shown 2 examples of all different pruning in an MDD above. The deletion rules associated with the states and the MDD all different are listed below: An edge (u,v) with value $\phi_{(u,v)}$ must be removed from the MDD if:

1. $\phi_{(u,v)} \in A_u^\downarrow$
2. $\phi_{(u,v)} \in A_v^\uparrow$
3. $|S_u^\downarrow| = d(root, u) \wedge \phi_{(u,v)} \in S_u^\downarrow$
4. $|S_v^\uparrow| = d(v, end) \wedge \phi_{(u,v)} \in S_v^\uparrow$

with $d(u, v)$ being the distance (in number of edges) between node u and v which is equal to $layer(v) - layer(u)$.

5.3 Experimentation: disjunctive scheduling

5.3.1 Introduction

In previous chapters and the section above, we have defined a complete framework for the mutlivalued decision diagrams, let us try to analyze the possible improvement that this new method can give to the current constraint programming framework.

This section tries to reproduce the benchmarks done by Andre A. Cire and Willem-Jan van Hoeve in [7] and [16]. The results obtained looked promising and we want to see if our implementation gives similar results when implemented in the OscaR framework.

The problem that we tackle is the disjunctive scheduling or unary resource problem. The idea is simple, we have a set of activities to schedule based on some rules. There is a transition time associated with each pair of activities and everything is disjunctive. This means that we cannot schedule two activities/transitions at the same time. You can imagine this as the day schedule of a worker that has to perform several jobs at different places in a building. Each job takes some time, has to be performed in a time-window and the worker needs time to go from one job to the next one. Each activity is defined by 3 features.

- r_i, the earliest possible start time of the activity i
- d_i, the latest possible finishing time of the activity i
- p_i, the time needed to perform activity i

Note that we must have $r_i+p_i \leq d_i$ to ensure that it is possible to do the job in the time window.

Two common objective functions associated with the disjunctive scheduling problem and that needs to be minimized:

- Makespan: The makespan is the total time separating the beginning of the first activity and the end of the last one.
- Setup time sum: Each transition from an activity to another is associated to a setup time cost. The objective to minimize is then the sum of those setup times.

As presented in [7], the MDD gives an improvement when trying to minimize the setup time. Therefore, we only try to reproduce the good results associated with the setup time minimization in OscaR.

In order to perform benchmarks on this type of problem, we test our code on several instances of the Traveling Salesman Problem with Time Window while minimizing the setup/travel time. We have a set of clients that must be visited by a delivery truck for example, in a certain time window. The truck must reach each client in the time window and there is a time to drive from one client to another. The aim is to minimize the total driving time, which corresponds to the minimization of the setup time. This application can be seen as a way to minimize the time during which the truck drives and thus to limit the consumption of oil.

In the following parts, we also consider that the truck must start and come back at the depot after the tour. A trick used to enforce this is to create a new virtual client that is located at the starting position and enforce that it must also be visited as first and last client.

5.3.2 Model

We described earlier how to solve a problem with constraint programming. First, define a model representing the problem with variables, domains and constraints. Then give this to a solver that finds all the solutions. In our case, as we want to minimize an objective function, we only look for a solution having the lowest possible objective value.

There are two possible models to solve a TSP-TW, we briefly describe how they work and how we can integrate the MDD in those existing models.

Permutation model

The first way to model these instance is by representing the order of activities as an array of variables. Once all the variables are set to a value, we have the global order of activities and the activities are put as soon as possible based on the finishing time of the previous activity and on their earliest start time.

If Perm is the permutation array of size n, containing the domains that are initially [0,.. n-1], the model can be resumed as:

- The first and last activities must be the depot
- The start time of client i plus the duration of the job and the transition time towards client $i+1$ should be smaller than the start time of client $i+1$
- The start time should be greater than r_i, the earliest start time
- The activity i must be completed before d_i, the deadline associated with it.
- All the activities must be performed, and only once. This is enforced with a global constraint all different.

Subsection 5.3.3 shows that this model allows to add an MDD constraint efficiently and make a good mapping between the permutation variables and the ones involved in the layers of the decision diagram.

Successor model

The successor model is more complex and has been studied for the unary resource with transition time problem by the ICTEAM at UCLouvain in [9] and [33]. It is based on the fact that the traveling salesman problem (TSP) is in fact a cycle. In a cycle, each node has only one successor and this is how we define our array of variables in our case.

The table 5.1 presents the two models corresponding to the TSP of figure 5.6. The permutation is the array of visits, first activity 0, then activity 4,... The successor model shows that the successor of 0 is 4, the successor of 4 is 1,...

Model / activity	0	1	2	3	4
Permutation	0	4	1	3	2
Successor	4	3	0	2	1

TABLE 5.1 – Array representing the models associated with the TSP presented on figure 5.6

The model is however fairly easy to define once that notion of successor array is grasped.

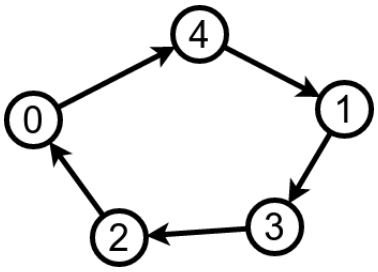


FIGURE 5.6 – TSP

- The successor of the last activity is the first activity (depot)
- The start time of client i plus the duration of the job and the transition time towards client successor(i) should be smaller than the start time of client successor(i).
- The start time should be greater than r_i, the earliest start time
- The activity i must be completed before d_i, the deadline associated with it.
- The array successor should represent a circuit (all different plus the constraint that there should be only one connected component).

The next section shows that this model can not really be adapted to the MDD model. In order to be able to still use the MDD in that case, we do something called chanelling, which means using both model and linking them. The successor array can be linked to the permutation by enforcing that:

$$Permutation(i) == Successor(Permutation(i-1)) \ \forall \text{ i} \in \text{[0,n-2]}$$

5.3.3 A specific MDD-constraint

We have already defined the all different constraint enforcing the permutation on a MDD. Hopefully, [7] defines a way to enforce the disjunctive constraint and the time windows. Without going into too much details, this constraint maintains 2 states per MDD node.

- E_v represents the earliest end time at node v and more precisely the minimum end time amongst all partial assignement from root node to node v.
- F_v represents the maximum end time at node v over all partial sequences from root node to node v.

The mathematical definition of those two state is detailed in [7], and the formulas obtained are

$$\begin{cases} E_v & = \min\{E'_{(u,v)} : (u,v) \in \delta^{in}(v)\} \\ E'_a & = \max\{r_{\phi_{(u,v)}}, E_u + \min\limits_{b \in \delta^{in}(u)} \{t(\phi_b, \phi_{(u,v)})\}\} + p_{\phi_{(u,v)}} \\ & \\ F_v & = \min\{\max\limits_{a \in \delta^{in}(v)} d_{\phi_a}, \max\limits_{a \in \delta^{out}(v)} F'_a\} \\ F'_{(v,w)} & = F_w - \min\limits_{b \in \delta^{in}(v)} \{t(\phi_b, \phi_{(v,w)})\} - p_{\phi_{(v,w)}} \end{cases} \tag{5.7}$$

Then, based on those states, we can define that an edge (u,v) should be deleted if:

$$\max\{r_{\phi_{(u,v)}}, E_u + \min_{(w,u) \in \delta^{in}(u)} t(\phi_{(w,u)}, \phi_{(u,v)})\} + p_{\phi_{(u,v)}} > \min\{d_{\phi_{(u,v)}}, F_v\} \tag{5.8}$$

5.3.4 Experimentation results

We have seen above that the representation used in the paper to represent the tour was the permutation model (see [16]). The figure 5.7 shows that we have the same results: a huge improvement when multivalued decision diagrams are added. Moreover, we can see that the performances increase with the breadth of the MDD. The section 5.4 shows how this maximum width can influence the performances. Note that as presented in the paper, the speed up can be as huge as a factor of 1000. The number of instances solved also increased with the MDD width. For this experiment (and the following ones), we have set a time limit of 3600 seconds and removed the instances that were solved too quickly by all method (threshold of 2 seconds).

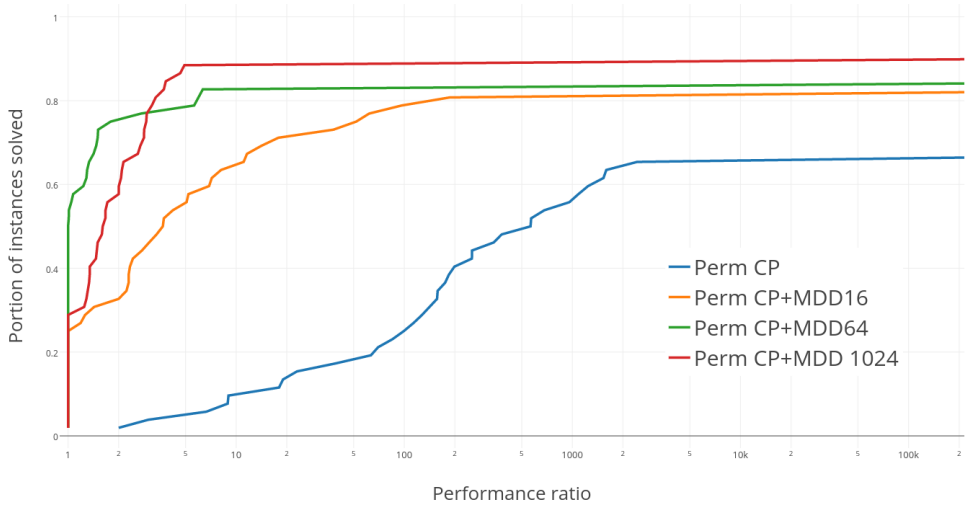


FIGURE 5.7 – Performance profiles on Dumas instances with permutation representation of the activities ordering, the comparison is performed on the time needed to solve the instances.

Yet, the figure 5.8 shows less motivating results. They depict the performance profile when the successor model is used. We can see that adding the MDD to the basic model adds no improvement to the successor model and even reduces the performances. This can be partially explained by the fact that the MDD is applied on a permutation model. This means that we have to add a chanelling between permutation and successor variables, which gives an overhead on the propagation and slows down the fixed point.

We have seen that multivalued decision diagrams are useful on the permutation model but not on the successor one. Figure 5.9 is a comparison between the successor and permutation performances. We can see that the successor model is the best one, even with the MDD improvement added to the permutation model. This analysis moderates the optimist results of [7]. However, this moderation comes from the fact that MDD's are not fully compatible with the successor model and that a chanelling between successor and permutation is needed when using the MDD on successor model.

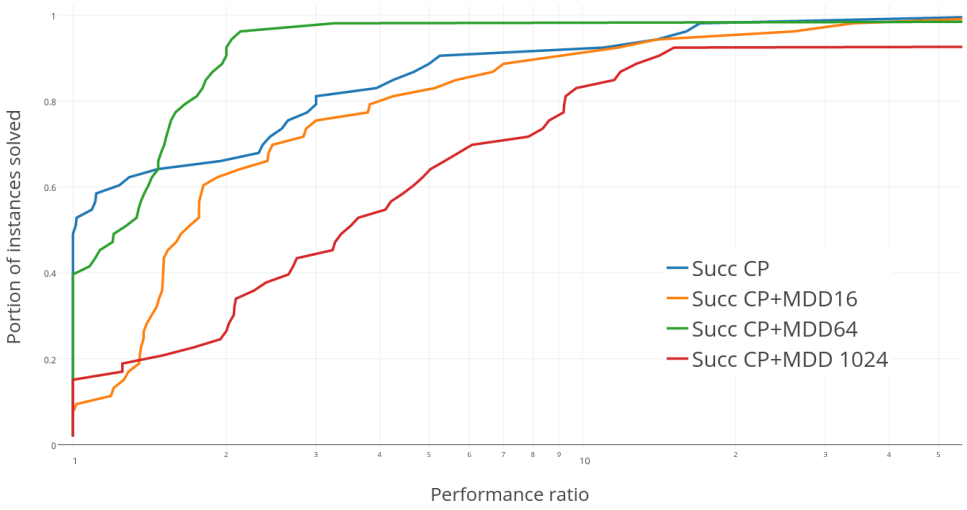


FIGURE 5.8 – Performance profiles on Dumas instances with successor representation of the activities ordering, the comparison is performed on the time needed to solve the instances.

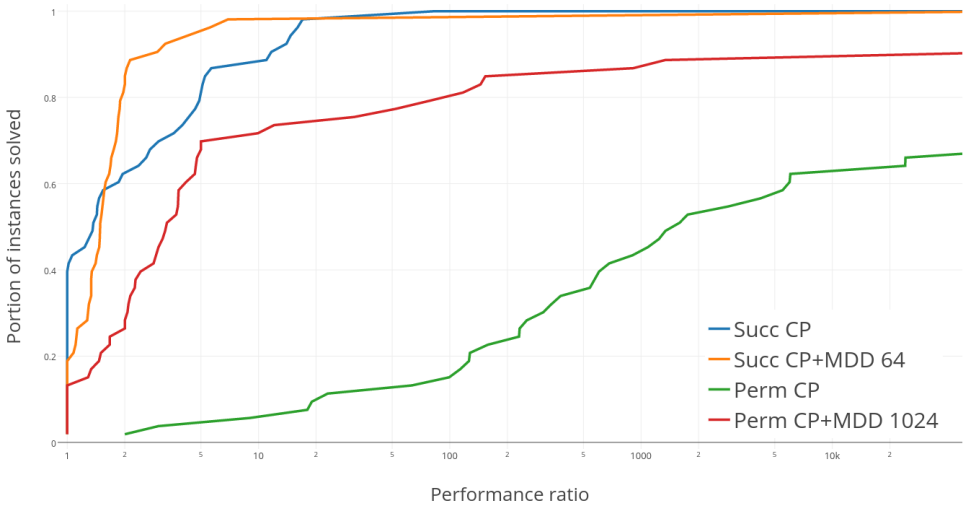


FIGURE 5.9 – Performance profiles on Dumas instances with permutation and successor representation of the activities ordering, the comparison is performed on the time needed to solve the instances.

5.4 Influence of the maximum width on the performances

In the previous section, we have seen that we can build a MDD based on constraints. However, if the MDD is not complete, it means that some paths correspond to non-solutions. This means that the basic constraints must still be enforced during the regular CP search. The MDD-constraint can then be seen as a heuristic improvement that can lower the number of failures.

However, usually, the strength of a constraint is defined by its consistency. For example, Generalized Arc Consistency enforces that before each branching, each value has a support for each constraint. In other words, having a support means that if we assign a variable x to a value a, it is still possible to assign all the other variables with one value from their current domain,

such that the considered constraint is satisfied. Some other consistencies are stronger, other are less powerful. But all kinds are important because in general, a stronger consistency means more computational time to prune the domains, but also a better pruning. The aim is then to find the best consistency, or the one that is the best trade-off between wrong branching decisions and huge propagation time.

This tradeoff is also present in the limited-width multivalued decision diagrams. If the maximum-width is 1, the resulting MDD does not contain much information about the constraints applied on it. However, if the maximum width is infinite, we reach the complete MDD and only solution paths are present in the structure, meaning a perfect pruning. The issue that arises is that we try to solve NP problems. Then, computing the complete MDD corresponds to finding all the solutions of the problem. As the problem is non-polynomial, we know that computing the complete MDD is also NP. Therefore, we must try to limit the time to refine the MDD, and therefore, the width.

The width describes the strength of the resulting heuristic constraint. A MDD of width 1000 is supposed to be stronger (or at least have the same power) than the same MDD but with a limited width of 10. The interesting point to note is that usually, consistencies are discrete in a way. The number of consistencies for a constraint is limited to the number of algorithms to enforce that consistency. In our case, we can vary the width of the MDD to build MDD having a custom consistency and measure that consistency with the width. The multivalued decision diagrams allow then to have a better control on the consistency complexity but also to play with it easily and without the need to re-design a brand new algorithm each time we want to change the consistency.

This section is purely experimental and tries to measure the impact of varying the width of the MDD in concrete examples. The figure 5.10 shows the time needed to solve 3 different disjunctive scheduling instances when put in perspective of the maximum size allowed. We can see that at first, increasing the breadth of the diagram improves the total time, but after a certain point, the time needed to build the MDD and the overhead given by the propagation inside the MDD lower the performances. When using an MDD model in an application, it is then useful to find a way to dimension the maximum width to reach the optimal solving time.

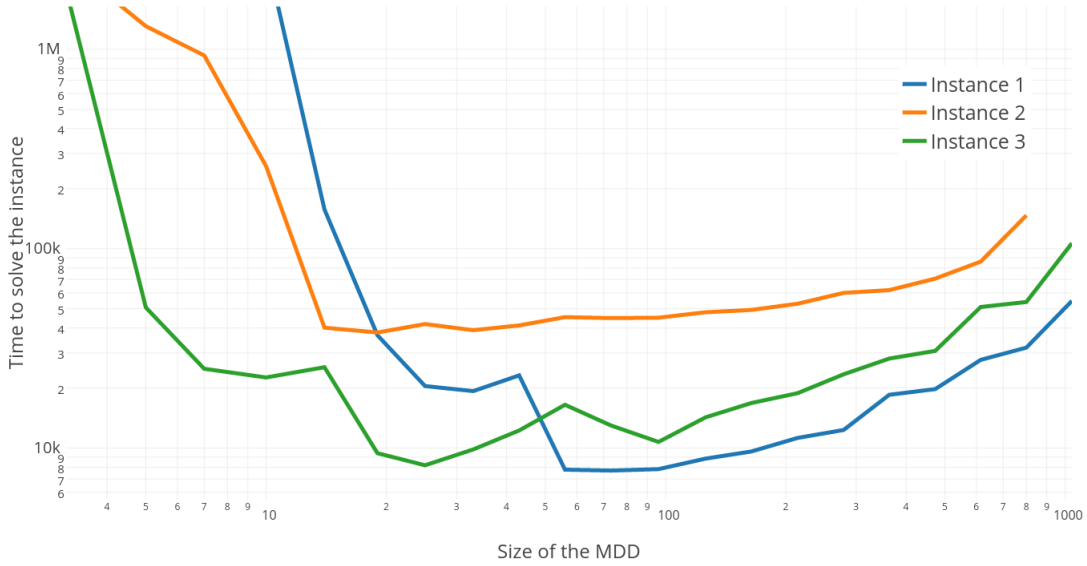


FIGURE 5.10 – Time needed to solve disjunctive scheduling instances while varying the maximum width of the MDD

CHAPTER 6

ADDING DYNAMIC COST-CONSTRAINTS TO MDD

The precedent chapters have presented static MDD on which we can apply constraints and refine them. The result of this refinement produces a static diagram, which is then injected in a CP constraint, building a reversible structure allowing to propagate efficiently that static information. One problem relies in the fact that those static MDD are built heuristically. If the heuristic of node-splitting was poorly designed, the MDD won't add much propagating power and moreover add computation time to maintain its consistency. Moreover, the static MDD has plenty of information in the firsts layer, but a bit less in the last one (as the number of paths from root to node v increases as we go down the graph). This section tries to analyze how we can add dynamic constraints efficiently to the reversible MDD constraint in order to add more pruning during the search. This dynamic idea means that the states defined in the static version of the constraint is dynamically recomputed when needed and add more pruning.

6.1 Cost constraint

In [22], Perez and Régin presented a way to add the dynamic cost constraint to a decision diagram. This constraint corresponds exactly to the static *Knapsack* constraint presented in section 5.2.1. The aim is to find assignments of variables such that the sum of the cost associated with the values of the variables is lower (resp. greater) than a certain value.

In the same way than for the knapsack, the dynamic constraint maintains the longest (resp. shortest) path from the root to the node and from the node to the end node to be able afterward to prune edges based on the lower (resp upper) bound. To simplify the text and the presentation, we only detail the upper bound constraint, which aims at keeping the total cost under a certain value. However, it is easy to turn this upper bound constraint into a lower bound constraint by using the same ideas.

As for the knapsack constraint, the algorithm maintains for each node, the costless path from root to node ($S_\downarrow$ array) and from node to end ($S_\uparrow$ array). The same rule applies and an edge (u,v) must be deleted if

$$S_\downarrow(u) + w_{(u,v)} + S_\uparrow(v) > B$$

The problem now is that those costs must be recomputed dynamically after each edge removal. And this recomputation must be efficient. In other words, we cannot afford to recompute the shortest path for each node after each edge deletion. The idea to solve this problem efficiently arises from the fact that the shortest path of a node must be only recomputed in two cases:

- If an edge incident to this node is deleted and was on the shortest path of the node
- If the shortest path value of one of the neighbouring node of the considered one has its value increased.

To know which nodes must be recomputed, we maintain queues of nodes and mark the nodes already present in the queues. The idea is to empty this queue and recomputing the values of node states when needed. When a value is modified, the neighbors nodes are enqueued, until a fixed-point is reached. At this moments, all the values are correct for the current state of the MDD and all the useless edges inconsistent with the bound are deleted.

As presented in [22], an other key point is that a node must be marked and enqueued, only if the edge that was deleted was responsible of the shortest path value. Therefore, if we delete the edge (u,v) having a cost $w_{(u,v)}$, we mark u only if:

$$S_{\uparrow}(u) = w_{(u,v)} + S_{\uparrow}(v)$$

Similarly, we mark v only if:

$$S_{\downarrow}(v) = w_{(u,v)} + S_{\downarrow}(u)$$

For example imagine that we have a portion of a cost-MDD corresponding to the one present on figure 6.1. In this diagram (during the CP fixed point), the edge A-D is removed.

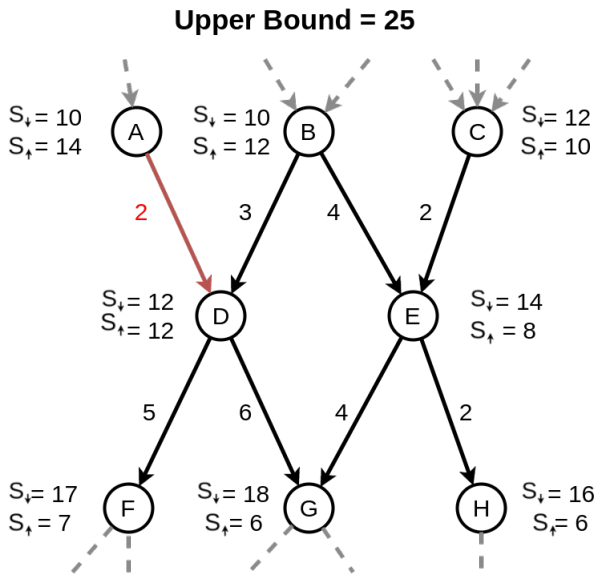


FIGURE 6.1 – Original cost MDD, with states computed. The edge (A,D) with value 2 is deleted.

We can see on figure 6.2 the effect of deleting edge (A,D). Note that the state up of the node B has been updated (imagine that an edge above B was deleted, modifying the state). We know that we have to recompute the $S_{\downarrow}$ value of D, since $S_{\downarrow}(A) + 2$ was equal to $S_{\downarrow}(D)$ and the new value is 14. Since the value of $S_{\downarrow}(B)$ and $S_{\downarrow}(D)$ was modified, we have to check if their out edges should be deleted or not:

- (B,D): 11+3+12 = 26 > 25, edge (B,D) should be deleted
- (B,E): 11+4+8 = 23 $\leq$ 25, edge (D,E) can be kept

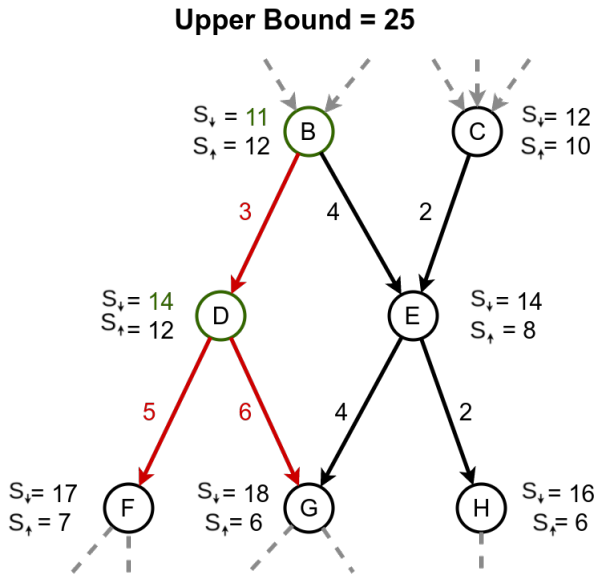


FIGURE 6.2 – Original cost MDD, with states computed. The edge (A,D) with value 2 is deleted.

- (D,F) and (D,G) yields value 26 and should be deleted as well. Note that as the node D did not have any more in-edges, it would have been invalidated, leading to the suppression of its out-edges.

The figure 6.3 presents the MDD after applying the cost fixed point. Node that all the values $S_{\uparrow}(v)/S_{\downarrow}(v)$ are consistent and that no more edge can be deleted based on the cost constraint.

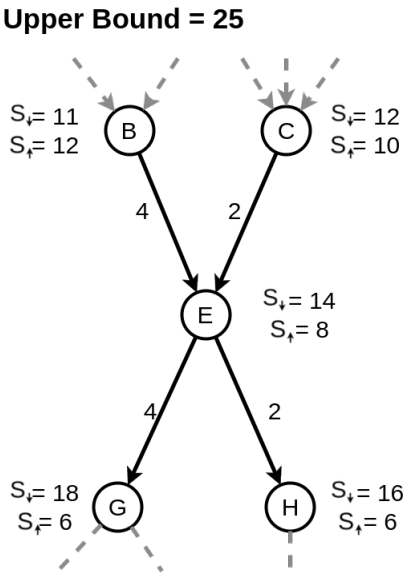


FIGURE 6.3 – Original cost MDD, with states computed. The edge (A,D) with value 2 is deleted.

6.2 Experimenting regular with costs: Longest path

The longest simple path problem consists of finding the longest path that visit each node at most one time. This problem is NP-hard and can be reduced to finding an hamiltonian cycle in a graph which is NP-complete.

In [26], Pham presented a way to solve this problem with constraint programming. The model uses the successor representation (see the disjunctive scheduling section for more information about the successor representation) with the trick that if the node v is not on the path, then its successor is a special value $\perp$. This approach works for small graphs but takes too much time when dealing with bigger graphs.

Our model is based on the permutation and is simpler than the one presented in [26]. We simply take the graph, represent it as an automaton: we add a dummy node for the beginning and then label the edges based on their destination node. We can then apply a basic regular constraint on this graph. Note that the current implementation of the regular constraint uses tables allowing the possible transitions. The permutation is a sequence of nodes on which we an apply an all different. Then, we optimize the objective function corresponding to the sum of the edges values.

We can also represent this as a succession of edges as in the MDD, by simply using the MDD-based regular constraint. We implemented an allDifferentWithValue in the MDD framework because each edge in the graph is associated with an *id*. This *id* can give us the cost but also the destination. Therefore it is possible to apply the all-different on the MDD on the destination nodes of the edges and not on the edge id. We also add a dynamic cost constraint on the reversible MDD for which we can change the lower bound. When a solution is found, the lower bound is updated and the MDD is pruned accordingly.

We implemented those 3 models in OscaR and obtained the results presented on figure 6.4. Those results show that the regular model is stronger than the one presented by Pham. Moreover, we can see that there is even a bigger improvement when using the cost-MDD-regular constraint. This comes from the fact the the propagation of the sum is inefficient when using the basic regular constraint because is consists of many tables of size 2 taken in isolation. With cost-MDD, when taking a certain decision, the potential paths that can happen are described more precisely, the bound can have a strong impact on the pruning of the MDD and therefore on the pruning of the domains.

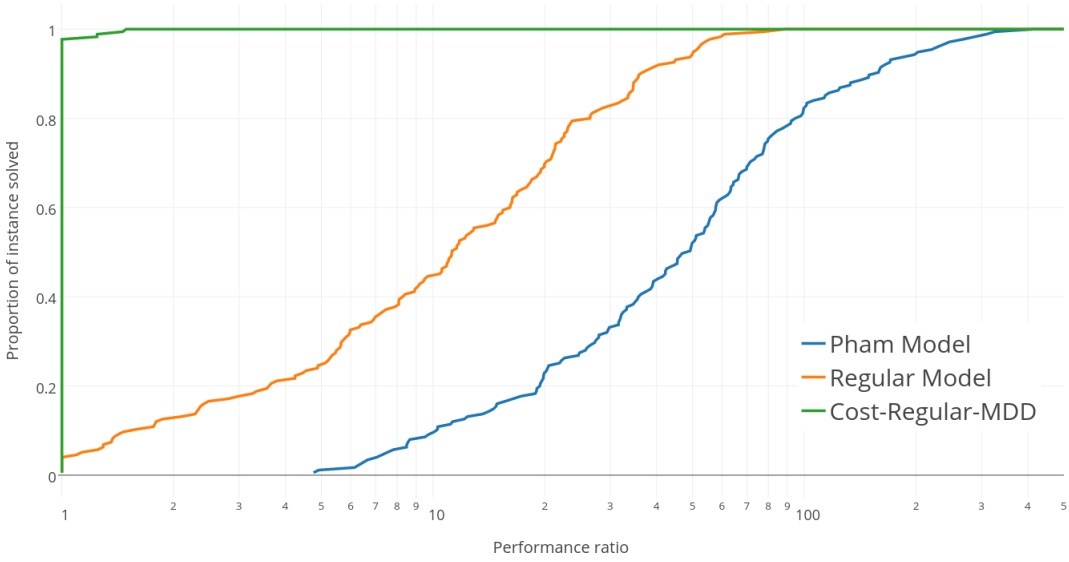


FIGURE 6.4 – Performance profile comparing the various CP algorithms used to solve longest path instances

CHAPTER 7

CONCLUSION

The major workload of this master thesis laid in the design of a correct, efficient and modular framework based on the multivalued decision diagram. For example, our modularity allows us to easily build an MDD from a table constraint, then apply other constraints such as `alldifferent` and even add or remove custom tuples in the same diagram, before turning this customised constraint into a CP propagator. Also, the MDD propagators have been fully integrated in Oscar and unit tested in order to ensure that any modification does not remove any feature of the initial implementation.

Concerning the algorithmic part, we implemented several versions of the propagation inside the MDD with MDD4, MDD4R and cost-MDD. The benchmarks showed the improvement added by the reset idea and the huge improvement that is brought by the dynamic cost constraint. We also compared the time needed to build a decision diagram based on a table constraint with the incremental MDD built by adding tuples one by one. The regular constraint was easily added to the framework and the performances were competitive with the state of the art propagator. We also added the possibility to refine a decision diagram from a set of custom constraints. This creates an MDD-based constraint programming framework to build a diagram that is used afterwards as a propagator to speed up another CP framework (Oscar in our case).

More importantly, the assumption that the MDD adds communication between constraints by increasing the memory consumption is verified and we see a huge improvement of efficiency when a dynamic cost constraint is combined with an `all different` and `regular` constraints. This opens the way for new MDD-constraints to be developed in order to tackle more efficiently all the problems that have a lack of communication between constraints. For example, we showed that the communication problem for the sum/cost constraint is reduce by an important factor when decision diagram are used.

As potential further work, we would like to notice that one drawback of our diagrams comes from the fact that they are built statically and heuristically before launching the propagation with the pre-built MDD. One way to improve the power of the constraints would be to allow a dynamic refinement on the global MDD structure during the search, by doing splitting and pruning at each propagation call. This could reduce the number of fails in the search tree by a huge factor. However, keep in mind that the refinement is also costly and could balance the diminution of backtracks with an increase of propagation time.

BIBLIOGRAPHY

[1] Henrik Reif Andersen et al. “A constraint store based on multivalued decision diagrams”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 118–132.

[2] David Bergman et al. *Decision Diagrams for Optimization*. 2016.

[3] David Bergman et al. “Optimization bounds from binary decision diagrams”. In: *INFORMS Journal on Computing* 26.2 (2013), pp. 253–268.

[4] Christian Bessière et al. “An optimal coarse-grained arc consistency algorithm”. In: *Artificial Intelligence* 165.2 (2005), pp. 165–185.

[5] Preston Briggs and Linda Torczon. “An efficient representation for sparse sets”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 2.1-4 (1993), pp. 59–69.

[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”. In: *Constraints* 15.2 (2010), pp. 265–304.

[7] Andre A Cire and Willem Jan van Hoeve. “MDD Propagation for Disjunctive Scheduling.” In: *ICAPS*. 2012.

[8] Andre A Cire and Willem-Jan van Hoeve. “Multivalued decision diagrams for sequencing problems”. In: *Operations Research* 61.6 (2013), pp. 1411–1428.

[9] Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. “The unary resource with transition times”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 89–104.

[10] Jordan Demeulenaere. “Efficient Algorithms for Table Constraints”. In: *Master thesis*. EPL. 2015.

[11] Jordan Demeulenaere et al. “Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 207–223.

[12] Elizabeth D Dolan and Jorge J Moré. “Benchmarking optimization software with performance profiles”. In: *Mathematical programming* 91.2 (2002), pp. 201–213.

[13] Philip J Fleming and John J Wallace. “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Communications of the ACM* 29.3 (1986), pp. 218–221.

[14] Willem-Jan van Hoeve. “Decision diagrams for optimization and scheduling”. In: *12th Workshop on Models and Algorithms for Planning and Scheduling Problems*. 2015, p. 11.

[15] Timothy Kam. “Multi-valued decision diagrams: Theory and applications”. In: *Multiple-Valued Logic* 4.1 (1998), pp. 9–62.

[16] Joris Kinable, Andre A Cire, and Willem-Jan van Hoeve. “Hybrid optimization methods for time-dependent sequencing problems”. In: *European Journal of Operational Research* (2016).

[17] Laurent Michel, Pierre Schaus, Pascal Van Hentenryck. *Learning Constraint Programming step-by-step with mini-cp*. Available from `www.info.ucl.ac.be/ pschaus/minicp`. 2016.

[18] Christophe Lecoutre. “STR2: optimized simple tabular reduction for table constraints”. In: *Constraints* 16.4 (2011), pp. 341–371.

[19] Chang-Yeong Lee. “Representation of Switching Circuits by Binary-Decision Programs”. In: *Bell Labs Technical Journal* 38.4 (1959), pp. 985–999.

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

[21] Martin Odersky and al. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. Lausanne, Switzerland: EPFL, 2004.

[22] Guillaume Perez and Jean-Charles Régin. “Building efficient soft and cost MDD constraints”. In: (2016).

[23] Guillaume Perez and Jean-Charles Régin. “Improving GAC-4 for table and MDD constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 606–621.

[24] Guillaume Perez and Jean-Charles Régin. “Relations between MDDs and Tuples and Dynamic Modifications of MDDs based constraints”. In: *arXiv preprint arXiv:1505.02552* (2015).

[25] Gilles Pesant. “A regular language membership constraint for finite sequences of variables”. In: *International conference on principles and practice of constraint programming*. Springer. 2004, pp. 482–495.

[26] Quang Dung Pham and Yves Deville. “Solving the longest simple path problem with constraint-based techniques”. In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2012, pp. 292–306.

[27] Claude-Guy Quimper and Toby Walsh. “Decomposing global grammar constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 590–604.

[28] Jean-Charles Régin. “A filtering algorithm for constraints of difference in CSPs”. In: *AAAI*. Vol. 94. 1994, pp. 362–367.

[29] Michael Rice and Sanjay Kulhari. “A survey of static variable ordering heuristics for efficient bdd/MDD construction”. In: *University of California, Tech. Rep* (2008).

[30] Vianney le Clément de Saint-Marcq et al. “Sparse-sets for domain implementation”. In: *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*. 2013, pp. 1–10.

[31] OscaR Team. *OscaR: Scala in OR, 2012*.

[32] Julian R Ullmann. “Partition search for non-binary constraint satisfaction”. In: *Information Sciences* 177.18 (2007), pp. 3639–3678.

[33] Sascha Van Cauwelaert et al. “Efficient Filtering for the Unary Resource with Family-Based Transition Times”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 520–535.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**