Decision diagrams : constraints and algorithms

Guillaume Perez

UNIVERSITÉ CÔTE D'AZUR

UNIVERSITÉ COTE D'AZUR

Doctoral thesis

Decision Diagrams: Constraints and Algorithms

Defended by

Guillaume PEREZ

to obtain the title of

Doctor of Science

Specialty : ARTIFICIAL INTELLIGENCE

Thesis Advisor: Jean-Charles RÉGIN

prepared at I3S Sophia Antipolis, MDSC Team
Doctoral School STIC

defended on September 29, 2017

Jury :

Reviewers :	Roland YAP	-	National University of Singapore
	Nicolas BELDICEANU	-	IMT Atlantique
	Willem-Jan VAN HOEVE	-	Carnegie Mellon University
President :	David COUDERT	-	INRIA Sophia Antipolis
Examinators :	Pierre SCHAUS	-	Université Catholique de Louvain
	François PACHET	-	Sony CSL
	Michel BARLAUD	-	Université Nice - Sophia Antipolis
	Arnaud MALAPERT	-	Université Nice - Sophia Antipolis
Advisor :	Jean-Charles RÉGIN	-	Université Nice - Sophia Antipolis

Abstract

Multivalued Decision Diagrams (MDDs) are efficient data structures widely used in several fields like verification, optimization and dynamic programming. In this thesis, we first focus on improving the main algorithms such as the reduction, allowing MDDs to potentially exponentially compress set of tuples, or the combination of MDDs such as the intersection of the union. We go further by designing parallel algorithms, and algorithms handling non-deterministic MDDs. We then investigate relaxed MDDs, that are more and more used in optimization, and define the notions of relaxed reduction or operation and design efficient algorithms for them. The sampling of solutions stored in a MDD is solved with respect to probability mass functions or Markov chains. In order to combine MDDs with constraint Programming, we design the propagators of all the types of MMDD constraints in solvers, and introduce a new one, the channeling constraint. These new propagators outperform the existing ones and allow the reformulation of several other constraints such as the dispersion constraint, and even to define new ones easily. We finally apply our algorithm to several real world industrial problems such as text and music generation and geomodeling of a petroleum reservoir.

Résumé

Les diagrammes de décision Multi-valués (MDD) sont des structures de données efficaces et largement utilisées dans les domaines tels que la vérification, l'optimisation et la programmation dynamique. Dans cette thèse, nous commençons par améliorer les principaux algorithmes tels que la réduction de MDD, permettant aux MDD de potentiellement compresser exponentiellement des ensembles de tuples, ou la combinaison de MDD, tels que l'intersection ou l'union. Ensuite, nous proposons des versions parallèles de ces algorithmes ainsi que des versions permettant de travailler avec la version non déterministe des MDD. De plus, dans le domaine des MDD relâchés, un domaine de plus en plus étudié, nous définition les notions de réduction et combinaison relâchés, ainsi que leurs algorithmes associés. Nous résolvons le problème de l'échantillonnage des solutions d'un MDD avec respect de loi de probabilité tels que des fonctions de probabilité de masse ou des chaines de Markov. Pour permettre d'utiliser les MDD dans les solveurs de programmation par contraintes, nous proposons de nouveaux propagateurs pour toutes les contraintes basées sur des MDD, améliorant les performances des algorithmes existants, puis nous en introduisons une nouvelle contrainte, la contrainte de channeling. Grâce à eux, nous montrons que nous pouvons reformuler plusieurs contraintes et en définir de nouvelles tout en étant basés sur des MDD. Finalement nous appliquons nos algorithmes à des problèmes industriels réels de génération de texte et musique, et de modélisation de réservoir de pétrole.

Acknowledgments

I would like to thank my advisor Prof. Jean-Charles Régin. For having bet on me even years before my PhD, and for all his useful advices and encouragements during this thesis. Jean-Charles is one of the most intelligent person I had the opportunity to met in my life, and its ability to constantly share its knowledge is wonderful.

Many works inside of this PhD could have never existed without several researchers and the great collaborations we had. I would like to thanks them all. First Laurent Perron, at the beginning of my PhD. Then François Pachet, Pierre Roy and Alexandre Papadopoulos, which have provided me many good advices and insights on many fields. Also, I had the chance to work with Pierre Schaus and Christophe Lecoutre, which have provided me a great view of real world problems.

Inside of my laboratory, I had the luck to work with Michel Barlaud and Lionel Fillatre. I particularly want to thank them, for their patience and for having provided me so much useful knowledge on mathematical optimization. I am happy to thank Arnaud Malapert, for its many useful comments constantly allowing me to take a step back on most of my works, and for the work we have already done together, or for the next coming. I would like to thanks the many people in my lab, for the right working environment and the great atmosphere they have provided me. So thank you Sandra Devauchelle, Enrico Formenti, Benjamin Miraglio, Jonathan Behaegel, Ophelie Guinaudeau, and the many others.

I have a lot of gratitude for my friends. First, Mehdi Ahizoune, my best friend, and Yoan Kraria, Nicolas Huin, Anthony Palrmieri, Heytem Zitoun, Yassine Ferkouch, Jean-Michel Diaz Vaz, Sami (y) Lazreg. I thank them for the support they have provided me these last years.

I have a big thank for my family, for my mom Veronique, my brother and sisters, Virginie, Yannick and Doreenda. I would not be without them; they are my models for so many reasons. I would also thanks my step-parents, Isabelle and Jean, for their wonderful support.

Finally, and the most important, I would like to thank my Wife, Marion. She is my principal inspirational source. She has supported me, motivated me and encouraged me all along these years.

I dedicate this thesis to Her.

Contents

Chapter 1

Introduction

Contents

1.1 Introduction and Motivation

Computers are used to solve many applications like scheduling a product line of cars in a factory, designing elevator maps in tall buildings, detecting diseases in DNA and most important, helping you decide which movie you are going to watch next. Problems become harder every day, but computer scientists always design faster algorithms to fit with the ever evolving amount of data.

Constraint Satisfaction and Optimization Problems (CSPs and COPs) are general-purpose definitions of problems. They allow to define a problem by expressing its structure (constraints) over variables associated with domains, and often by giving some objectives to optimize. Such a definition is then taken by a solver which tries to find a solution satisfying the constraints while optimizing the objectives. These solvers are more and more efficient, allowing to always solve new problems.

However, many hard problems remain unsolved. There are many reasons for that. The first one is problem size: indeed many problems involve hundreds or thousands of variables. Second, many problems are too complex and contain particular structures, that we are unable to exploit. Finally, many problems are easily defined by sub-problems, but combining these sub-problems is hard in practice (Principle of compositionality).

In order to solve these issues, we need to find data structures that can represent huge and complex problems and that can be combined. These data structures must efficiently: 1) represent discrete problem solutions, without necessarily enumerating them. 2) express problems and efficiently combine

them. 3) be integrated in solvers via fast and incremental algorithms. All of that while aiming at solving a broader range of problems.

Many data structures has been used to represent problems or their solutions. Consider first table constraints, often called extensional constraints. A table is defined by the list of all the allowed tuples. Using such a representation, the size of the data structure storing the solution is linear in the number of solutions, but this number can be exponential. Thus, many compressed data structures have been proposed, such as Global Cut Seeds (GCS) [Focacci 2001]. A GCS is defined by a vector of set of values, each set is associated with a variable, and each combination of the Cartesian product of the sets is an allowed tuple. Such a representation can gain an exponential factor in memory, but in practice the Cartesian product is too strong for representing a constraint, thus a large amount of GCSs is needed. Improved versions has been proposed, such as tuple sequences [Régin 2005] for negative tables, tables containing prohibited tuples. Another one is the smart table [Mairy 2015], i.e. a table containing smart tuples, which allow a finer grain description of the tuples compared to the GCS. But once again, the range of application is limited to few constraints. More importantly, these representations are hard to combine. That is why automaton based constraints [Beldiceanu 2004a, Pesant 2004] have been proposed. These constraints can be defined by regular expressions, regular languages or directly by automatons.

Automaton based constraints have been a major step in constraint expressivity and they solve several of the previous issues. However, the definition of automatons is often challenging. Automatons can contain cycles which allow them to accept words of different sizes and this can cause several issues. First in CP, the number of variables of a constraint is fixed. To overcome this difficulty, propagators need to unroll the cyclic automaton over the variables, in order to enforce the accepted words to have a fixed size. The result can be seen as an new non necessarily minimal acyclic automaton. Second, the definition or combination of automatons, that can generate words of different sizes, is often harder than solving the problem for a fixed size. A simple example is the application of a simple unary constraint forbidding a value for a given variable can build an automaton of exponential size (see chapter 2, section 2.2.1).

We need to find a data structure that, in addition to the previously defined requirements, efficiently represents solutions of fixed size. That's why we focus in this thesis on Multi-valued Decision Diagrams.

Binary and Multi-valued Decision Diagrams (BDDs or MDDs) are efficient data structures that represents functions or sets of tuples. An MDD, defined over a fixed number of variables, is a layered rooted Directed Acyclic Graph

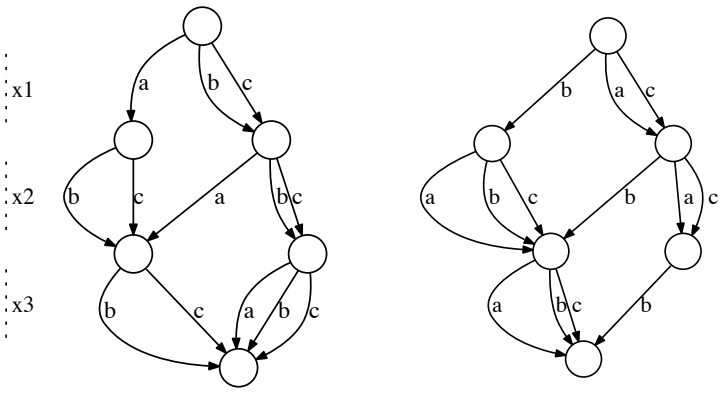


Figure 1.1: On the left, the MDD, defined over three variables (x1,x2,x3) associated with the domain (a, b, c), representing the constraint *at most one* a. On the right the MDD, defined over three variables (x1,x2,x3) associated with the domain (a, b, c), representing the constraint *at least one* b.

(DAG). It associates a variable with each of its layers. MDDs have an exponential compression power and are widely used in problem solving. An MDD has a root node, and two potential terminal nodes, the true terminal node `tt`, and the false terminal node `ff`. Each node, associated with a variable, can have at most as many outgoing arcs as there are values in the domain of the variable, the arcs are labeled by these values. Each path from the root node to the `tt` (resp. `ff`) node is said to be valid (resp. invalid). The label vectors of the valid path's arcs represent the valid tuples.

BDDs are well known for their use in the logic area, verification and model checking [Bryant 1986, Bryant 1992]. An unrolled automaton can be seen as a not reduced MDD. Furthermore, BDDs and MDDs are more and more used in optimization. During the last ten years, many works shown how to efficiently use them in order to model and solve several optimization problems [Bergman 2016b, Bergman 2011, Hooker 2007]. An advantage of MDDs is that they have a fixed number of variables, and often a strong compression ratio. However MDDs can have an exponential size, and it effectively occurs in practice.

Example:

Consider the problem of generating sequences of three letters, defined on three variables (x1,x2,x3) associated with the domain $\{a, b, c\}$, that contain at most one a and the problem of generating sequences having at least one b. Consider the MDDs from Figure 1.1. The left one represents

the constraint *at most one* a, a famous constraint enforcing that the value a can appear at most once per solution. Note that the MDD represents all the solutions of this constraint. Thus every solution of this MDD, which are the paths from the highest node to the lowest node, do not contain more than one a. The right MDD represents the constraint *at least one* b, which enforce that at least one b must appear in the solution. Thus every solution of this MDD contains a b. The path (a, c, c) is a solution of the left MDD, but not of the right MDD. The path (a, a, b) is a solution of the right MDD but not of the left MDD.

Many of the advantages of MDDs come from these three operations:

- **Creation**: MDDs allow to build many existing problems, without enumerating all their solutions. This very useful when modeling sub-problems containing huge amount of solutions, many values and variables.

- **Reduction**: MDDs have a strong compression power. They allow for example, to model a complex sub-problem having 10^{90} solutions, based on more than one hundred of variables, using an MDD having slightly more than 600,000 arcs (see chapter 18).

- **Combination**: Finally, one of the most fundamental aspect of MDDs is their ability to be easily combined. They offer an efficient way to combine constraints that are difficult otherwise. This allows to solve many problems by building MDDs for sub-problems and combine them. These are the reasons why we are going to focus on how to efficiently use MDDs for solving problems.

Example:
Consider the problem of generating sequences of three letters, defined on three variables (x1,x2,x3), using the alphabet $\{a, b, c\}$, that contain at most one a and at least one b, thus the combination of the problems from the previous example. Consider the MDD from Figure 1.2. This MDD is the intersection of the MDDs representing the two constraints *AtMost one a* and *AtLeast one b* from Figure 1.1. Thus all the solutions of this MDD do not contain more than one a but they all contain at least one b. This MDD represents the combination of the two constraints.

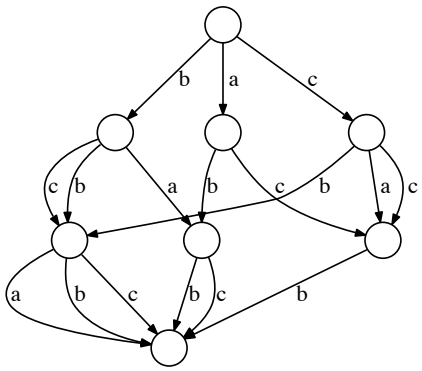


Figure 1.2: The intersection of the two MDDs from Figure 1.1. Thus all the solutions do not contain more than one a but do contain at least one b.

In order to efficiently manipulate MDDs, we need efficient algorithms for these three operations. Most of the existing methods fail to handle large domains, while many applications have large size domains (greater than 10,000 values). A good example is the MaxOrder problem [Papadopoulos 2014] (Chapter 17), it has a domain size of 11,000 in some of our instances. The existing MDD algorithms fail to solve it in a reasonable amount of time. The first goal of this thesis is to improve these algorithms. Thus we propose new algorithms for each of the operations. The main idea is to use the facts that an MDD is defined over a fixed set of variables, and that the domains can be large but sparse for the nodes. Moreover, most of the existing algorithms use complex internal data structures that, as we will show, slow down the computation and restrict the addition of features. We propose to remove these data structures and design conceptually simpler and more efficient algorithms.

Creation. One good reason for using MDDs is that they can be created from many sources, for example, they can be built from Boolean formulas or dynamic programming [Hooker 2013]. Thus in this thesis we focus on improving some of the existing conversion, like the ones from tables or automaton, and we propose new creation methods for several existing compressed data structures like the global cut seed [Focacci 2001] and the tuple sequences [Régin 2011].

Reduction. MDDs are often used because of their strong compression power, which mostly comes from the reduction of the MDD. The reduction of an MDD is an operation which merges *equivalent* nodes. Throughout the years, several algorithms have been proposed [Andersen 1997, Brace 1991, Bryant 1986]. In this thesis, we propose a linear reduction algorithm, which

is especially well suited for large set of different values.

Combination. Consider a problem containing p constraints, building an MDD for each constraint and intersecting them lead to an MDD containing all the solutions of the problem. Even if such a method is not always possible, combining MDDs often drastically improves the resolution time. Combining MDDs is a well studied topic, and as for the reduction, several algorithms have already been proposed [Andersen 1997, Bergman 2014b, Brace 1991, Bryant 1986, Bryant 1992]. In this thesis, we propose to study the existing algorithms, enlightening some of their weakness and proposing new versions strengthening them.

Advanced operations Thanks to the simple definition of our new algorithms for the reduction and combination, we are able to improve and extend them: 1) by designing parallel algorithms. 2) by considering relaxed MDDs. 3) by considering non deterministic MDDs.

Parallel versions of algorithms for BDDs or MDDs have been studied [Bergman 2014a, Kimura 1990, Stornetta 1996], but most of these works were limited because of the complex data structures they used for building, combining or reducing MDDs. Since the algorithms we propose for the reduction and combination of MDDs avoid such data structures, a new parallel algorithm can be design for both these algorithms. These parallel algorithms are designed to dispatch independent work between workers and are lock-free.

The relaxation of MDDs have been successfully applied in optimization and constraints solving [Andersen 2007, Bergman 2016b, Bergman 2011, Cire 2013, Cire 2014b, Hadzic 2008, Hoda 2010, Hooker 2007]. The authors designed several methods for building relaxed MDDs, i.e. MDDs representing a super set of the solutions. These methods have been integrated into solvers, for example for extracting lower bounds of solution cost, and some solvers are even fully based on relaxed MDDs. In this thesis, we propose to take a look inside the algorithms. Existing works mostly focus on relaxing the creation of MDDs. Firstly, we propose to improve these creation methods. Secondly, we define the notion of relaxed reduction for MDDs, and we give an associated algorithm. Thirdly, we define the notion of relaxed combination and design two associated algorithms. Finally, we propose to analyze the available possibilities (e.g. relax creation, relax combination...) for the modeler while constructing its problem relaxation.

Non deterministic Finite Automaton (NFA) are well know in automata theory for their efficient representation that can gain an exponential factor in space against classical Deterministic Finite Automaton (DFA). This idea has been applied to MDDs several times [Bollig 1999, Finkbeiner 2001], but almost all the works have focused on restricted versions of non-determinism.

We propose in this thesis to study the simplest non-deterministic version of MDDs that allows a node to have several outgoing arcs labeled by the same value. Using this representation, we define combination algorithms that output both deterministic and non deterministic MDDs. Thanks to this algorithm, we will be able to design a new and original propagator handling the cost-MDD constraint in CP solvers.

These three features are non exclusive and can be combined. The resulting algorithms are parametric algorithms handling both deterministic and non deterministic MDDs, applying relaxation if necessary, and running in parallel.

Sampling In the context of artificial intelligence, sampling is a useful technique. Sampling usually consists in randomly generating sequences or values, with respect to a probability distribution. Sampling under constraints is at least as hard as solving the involved CSPs, thus several works focus on a restricted subsets of constraints [Jurafsky 2014, Papadopoulos 2014, Papadopoulos 2015]. MDDs are well suited for combining constraints. We study in this thesis how to sample solutions of an MDD with respect to a probability distribution, that can be given either by a Markov chain or by a Probability Mass Function (PMF).

MDDs and solvers Constraint Programming mostly focuses on solving discrete CSPs and COPs. CP solvers, allow to define problems by their sub-problems (constraint) and the propagation mechanism combines them during the search. Embedding MDDs into solvers is one of the challenges for benefiting of their efficient compression and expressivity power. Several works have been focused on designing efficient propagator algorithms allowing to use MDDs in constraint programming solvers [Cheng 2010, Cheng 2008, Gange 2011]. Moreover, even a state of the art modeling language allows to directly define MDD constraints [Boussemart 2016]. In this thesis, we propose to define a new MDD propagator for constraint solvers, which is simple, has a good complexity, and which almost always improves the resolution time compared to the existing ones. Furthermore, this MDD algorithm has been implemented in several state of the art CP solvers, such as Or-Tools and OscaR [Perron 2013, OscaR Team 2012].

MDDs can also represent optimization problems, usually this is done by adding a cost to the arcs of the MDDs, the resulting MDDs are called cost-MDDs. Several works have focused on optimization and MDDs, and in the context of constraint programming, several algorithms handle these cost-MDDs [Demassey 2006, Gange 2013]. In this thesis we propose a new algorithm for handling cost-MDD, which improves existing algorithms in all tested instances. Furthermore, we propose a new technique that allows to

convert cost-MDD constraints into simple MDD constraints, by using the non-deterministic operations. This allows any simple MDD propagator to handle cost-MDD constraint. Note that these converted cost-MDDs can also be used into other areas than constraint programming, like satisfiability solvers.

Several problems do not contain any solution, they are called over-constrained problems. Consider for example the problem of generating a word, which has to contain at least one a, one b and two n, but whose length has to be less than 3. This problem is unfeasible. In constraint programming, over-constrained problems are often defined using soft constraints. A soft constraint is a constraint which can be violated, but with respect to a violation measurement. The goal is then to find the solution minimizing the total amount of violations. Soft constraints are well known in constraint programming, but soft MDD algorithms have never been investigated. Thus in this thesis we focus on designing several algorithms allowing to handle soft MDD constraints.

Modeling with MDDs The size of the MDD representing the allDifferent constraint grows exponentially. This implies that building an MDD representing all the solutions of a problem is not always a good idea. Intersecting two constraints allows to extract all the solutions respecting both of the constraints. But for intersecting two constraints using MDDs, we first need to define the two MDDs, which can already be exponential for some constraints such as the allDifferent constraint. Then, the intersection of two MDDs can lead to an MDD having as size the product of the size of two MDDs. Thus several work focus on kind of relaxed version of the intersection of constraints onto MDDs [Hoda 2010] in order to avoid these memory issues.

In this thesis we design a channeling constraint for MDDs. This constraint allows to link the values and the variables of sub-sets of arcs of an MDD with other constraints. This new method allows to define new constraints in CP solvers based on MDDs. Our best example is the Allen constraint. This constraint aims at constraining the values of variables occurring during temporal sequences. Constraining variables using temporal sequences is often hard to define because the indexes of the variables do not necessarily correspond to their temporal positions. The propagator of this constraint uses our MDD propagators, incremental version of combinations of MDDs that we designed and the channeling constraint for MDDs. This model allows to solve instances orders of magnitude faster than other methods.

Constraint programming solvers are efficient because they have a specific algorithm for each sub-problem (constraint). But this implies a lot of code and time since the number of constraints is huge [Beldiceanu 2012]. Several works

focus on reformulating constraints into others, allowing to implement and define only a sub-set of algorithms and using them to handle many constraints [Lhomme 2012]. In this thesis, we propose to reformulate existing constraints, like the dispersion and thus spread constraints [Pesant 2005, Schaus 2014, Pesant 2015] into MDD constraints.

Thanks to the proposed methods of this thesis, several models have been designed for solving the three following industrial applications, and excellent results have been obtained. First the geomodeling of a petroleum reservoir, based on probability and knapsack constraints. Second, the Maxorder problem, aiming at generating sequences avoiding plagiarism. Finally, a musical synchronization problem, solved using the proposed propagators and our model, which outperforms by orders of magnitude other methods.

1.2 Contributions and Outline

1.2.1 Inside this thesis

Most of the work of this thesis consists in improving the definition of models for solving problems and in designing their associated algorithms. Next chapter is a reminder of the state of the art about decision diagrams and gives several notations used all along. The thesis is divided into five parts.

Part I focuses on the three fundamental operations for MDDs: The reduction (Chapter 3), by giving a new algorithm that is very efficient in practice. The creation (Chapter 4), by improving several existing MDD constructors like the one from table and by proposing several new ones. The combination (Chapter 5), by proposing a new algorithm based on set operations instead of composition of functions, which is simple and efficient, and avoids complex data structures. Note that these improvements solve one of the problems defined in the application part of this thesis (Chapter 17).

Part II presents efficient parallel versions of the reduction and combination algorithms (Chapter 6). Lock-free algorithms having independent work loads, allowing to distribute the load over several computers are given. Then it deals with modification of our algorithms in order to handle non-deterministic MDDs (Chapter 7). Next it focuses on the existing relaxation for MDD (Chapter 8) and proposes to take a look inside them and to design new algorithms in order to define efficient relaxations. This is done by designing operations

like the reduction or the combination producing relaxed MDDs. Note that these three modifications are non exclusive.

This part also focuses on designing efficient sampling algorithms for MDDs (Chapter 9), by considering the two most used statistical distribution, the Markov process and the Probabiliy Mass Function (PMF). Thus this chapter defines algorithms for both of them, incremental modifications and parallel updates.

Part III focuses on the algorithms used in CP solvers for handling pure MDD constraints. First the GAC4R algorithm is presented (Chapter 10). This new algorithm allows to efficiently handle table constraints, and is used to design MDD4R, the algorithm handling MDDs inside CP solvers that we proposed. Second, the cost version of MDDs is studied (Chapter 11), and the cost version of MDD4R is proposed, all along with another technic converting cost MDD constraints into classical MDD constraints. Third, the soft version of the MDD constraint is studied (Chapter 12), and we propose three different methods for handling them, all with different efficiencies and levels of consistency. Finally, we propose a channeling constraint for MDD allowing to constrain only sub-part of the MDD (Chapter 13), by constraining the allowed (or prohibited) values and variables

Part IV focuses on defining existing or new constraints using the proposed MDD algorithms. First the Allen constraint is defined (Chapter 14), a constraint enforcing temporal constraints on variables. Then, an MDD version of the well known Spread and dispersion constraints is given (Chapter 15), all along with several others statistical constraints based on Markov processes and PMF.

Part V is about the applications we solved during this PhD thesis. The first one is a problem of generating sequences following a Markov generation process but avoiding plagiarism from a corpus (Chapter 17). This application is mostly solved using the operation defined in Part I. The second one is another generation problem consisting in generating sequences of musical notes for several instruments (Chapter 18). This problem imposes temporal synchronization points as constraints. Furthermore, the generated sequences have to follow a Markov generation process. Our model uses the constraints defined in both Part III and Part IV. The last application is about the geomodeling of a petroleum reservoir (Chapter 19), our model use several of the statistical constraints defined in Chapter 15.

The work presented in this thesis mostly comes from the following publications: [Perez 2017b, Perez 2016, Perez 2015a, Perez 2014, Perez 2017a, Perez 2015b, Perez 2017c, Roy 2016]. Chapters 6, 7, 8 and a part of 13 are still under submission.

1.2.2 Other Contributions

During my PhD I had the chance and opportunity to work with many other researchers about different topics other than MDDs. This section is dedicated to them.

In the context of machine learning and optimization, I worked with M. Barlaud and his team on the design and implementation of algorithms for numerical optimization, on splitting algorithms and on the design and implementation of and algorithm for the projection onto the simplex and the L_1 ball. This collaboration led to the writing of both a paper in the French conference of the domain and a journal paper currently under submission.

In the context on constraint programming, I worked with several researchers (all the authors of [Demeulenaere 2016]), about the design of an efficient table constraint propagator. This collaboration led to a publication to the CP conference of 2016 too. Moreover, this algorithm is becoming one of the state of the art algorithm for table constraints.

In parallelism and constraints, with my supervisor J.-C. Régin. I designed and implemented a search strategy selector based on active learning algorithms (i.e Bandit, UCB) for the paper [Palmieri 2016].

Last but not least, I had worked with Anthony Palmieri, another PhD student currently at Huawei Paris, and a long time friend, about optimization, search strategies and search hybridization. This collaboration led to an article under submission.

Chapter 2

Definitions & Related Work

Contents

2.1 Definitions and Notations

2.1.1 Constraint Programming

Constraint Programming (CP) is a problem-solving method. In CP, a problem is first modeled, using variables and constraints. Usually, each variable is defined by its domain, corresponding to its set of possible values. Each constraint defines a property that must be satisfied by a subset of the variables.

A Constraint Satisfaction Problem (CSP) is a couple $P = (X, C)$, where $X = x_1, x_2, ..., x_N$ is a set of variables and $C = C_1, C_2, ..., C_m$ is a set of constraints. Each variable x_i is associated with its domain $D(x_i)$, representing all its possible values. A constraint C_i associated with a set of all allowed tuples $T(C_i)$ defined over a subset of variables $S(C_i) \subseteq X$.

A solution is a tuple of values $(a_1, a_2, ..., a_k)$ such that the assignment $x_1 = a_1$, $x_2 = a_2$ ..., $x_N = a_N$ satisfy all the constraints.

The resolution of a CSP generally involves a Depth First Search (DFS) algorithm using backtracking building a search tree. At each node of this tree, a propagation algorithm is run. This algorithm removes the inconsistent values with respect to the constraints. This is done by running a specific filtering algorithm associated with each constraint, called the filtering algorithm (or propagator). This filtering algorithm removes values that cannot belong to a solution of the constraint and so reduces the search space. The DFS algorithm is driven by a search strategy, usually choosing the next couple variable value to affect.

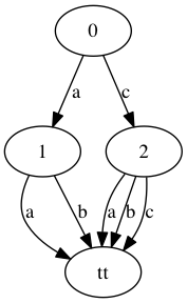


Figure 2.1: An MDD representing the tuple set {(a,a),(a,b),(c,a),(c,b),(c,c)}

Constraints In constraint programming, a constraint can be defined in several ways, the simplest one is to define the constraint by all its allowed tuples, thus by a table T having $\lambda = |T|$ tuples. But a constraint can be defined by relation between variables, for example by enforcing that $x_1 < x_3$. Moreover, in CP, complex constraints exist, like the allDifferent [Régin 1994], enforcing that the value taken by the variables are pairwise different. Or the atMost constraint preventing a value to appear more than a given number of times.

Applications Constraint programming is often used in product line scheduling for industrial factories [Régin 1997, Bergman 2014b]. Moreover, CP is often used in crew and nurse scheduling in hospitals [Pesant 2004, Demassey 2006, Schaus 2009b]. While this list is not exhaustive, CP is often used in industrial application of Bin-Packing, etc [Schaus 2009a, Schaus 2012, Bent 2004]. In addition, CP is used in transportation, by managing crews, gates and flights or in configuration problems [Sabin 1996, Hadzic 2004].

Most of the applications presented in this thesis are mainly from the Artificial Intelligence area, based and the seminal works of Pachet and his team in the context of content generation for entertainment [Barbieri 2012, Pachet 1999, Pachet 2014, Pachet 2001, Pachet 2011, Papadopoulos 2014]. Thus chapters 17 and 18 present applications that have mainly been realized with them. The last application presented in this thesis came from an industrial problem about the geomodeling of a petroleum reservoir.

2.1.2 Multi-valued Decision Diagrams

Multi-valued decision diagram (MDD) is a multiple-valued extension of BDDs [Bryant 1986, Akers 1978]. MDD is a rooted directed acyclic graph (DAG) often used to represent some multi-valued function $f : \{0...d-1\}^r \rightarrow \{true, false\}$, based on a given integer d. Given the r input variables.

The DAG representation is designed to contain $r+1$ layers of nodes, such that each variable is represented at a specific layer of the r first layer of the graph, and such that the last layer represent both the *true* and *false* terminal nodes (the *false* terminal node is typically omitted). Each node on a given layer has at most d outgoing arcs to nodes in the next layer of the graph. Each arc is labeled by its corresponding integer.

When the MDD is used to represent a function f, there is an equivalence between $f(v_1, ..., v_r) = true$ and the existence of a path from the root node to the *true* terminal node whose arcs are labeled $v_1, ..., v_r$. Figure 2.1 shows an example of MDD defined on two variables.

Notation An MDD $G = (N, E)$ has $n = |N|$ nodes and $m = |E|$ edges. A node u has a list $\omega^+(u)$ of outgoing arcs and a list $\omega^-(u)$ of incoming arcs. $\omega^+(u)[i]$ is the i^{th} arc and $|\omega^+(u)|$ is the number of arcs in the list . An arc is a triplet $e = (u, v, a)$, where u is the emanating node, v the terminating node and a the label. If the MDD is a cost-MDD, an arc is a quadruplet $e = (u, v, a, c)$, where u is the emanating node, v the terminating node, a the label and c the cost.

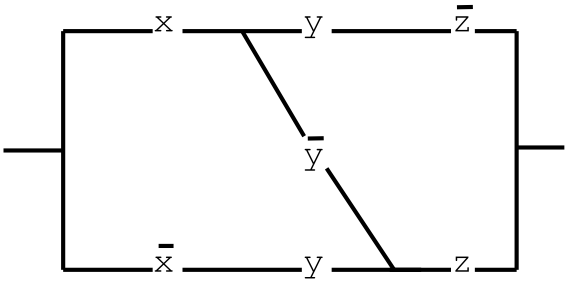


Figure 2.2: A simple switching circuit from [Lee 1959]

2.2 Related Work

One of the seminal work on design and analysis of circuits comes from Shannon [Shannon 1938, Shannon 1949]. The famous shannon decomposition considers that a Boolean function $f(x_1, x_2, ...x_k)$ can be recursively decomposed into $x_1 f(1, x_2, ...x_k) + \overline{x_1} f(0, x_2, ...x_k)$.

For example, consider the function $f(x_1, x_2, x_3) = \overline{x_1} x_2 x_3 + x_1 \overline{x_2} x_3 + x_1 x_2 \overline{x_3} + x_1 x_2 x_3$. Applying the Shannon decomposition on the first variable gives :

$$\begin{aligned} f(x_1, x_2, x_3) &= x_1 f(1, x_2, x_3) + \overline{x_1} f(0, x_2, x_3) \\ f(x_1, x_2, x_3) &= x_1(\overline{x_2} x_3 + x_2 \overline{x_3} + x_2 x_3) + \overline{x_1}(x_2 x_3) \\ f(x_1, x_2, x_3) &= x_1(x_2 + x_3) + \overline{x_1}(x_2 x_3) \end{aligned}$$

The origin of Binary Decision Diagrams come from the Binary Decision Programs defined by Lee [Lee 1959] for representing switching circuits and in order to "compare its representation with the algebraic representation of Shannon".

In such circuit, a switch can have a value, either 0 or 1, corresponding to its current state. Figure 2.2 shows an example, coming from [Lee 1959], of such a circuit. In this circuit, x indicates that $x = 1$ and $\overline{x}$ indicates that $x = 0$.

Some voluntary long Shannon decomposition of the switching circuit from

Figure 2.2 can be given as follows:

$$
\begin{aligned}
F(x,y,z) &= xy\overline{z} + x\overline{y}z + \overline{x}yz \\
F(x,y,z) &= xF(1,y,z) + \overline{x}F(0,y,z) \\
F(x,y,z) &= x(y\overline{z} + \overline{y}z) + \overline{x}(yz) \\
F(x,y,z) &= xyF(1,1,z) + x\overline{y}F(1,0,z) + \overline{x}yF(0,1,z) + \overline{x}\overline{y}F(0,0,z) \\
F(x,y,z) &= x(y(\overline{z}) + \overline{y}(z)) + \overline{x}(y(z) + \overline{y}(0)) \\
F(x,y,z) &= x(y(z(0) + \overline{z}(1)) + \overline{y}(z(1) + \overline{z}(0))) + \overline{x}(y(z(1) + \overline{z}(0)) + \overline{y}(z(0) + \overline{z}(0))) \\
F(x,y,z) &= x(y(\overline{z}(1)) + \overline{y}(z(1))) + \overline{x}(y(z(1)))
\end{aligned}
$$

A Binary Decision Program is a set of conditional instructions, whose, using the (inverse of) today's standard of programming, can take the form:

$$T \rightarrow x?\ A : B$$

Meaning that if the value of x is 0 then go to instruction A, otherwise go to instruction B. Note that each instruction is associated with a variable.

Thus for the switching circuit of Figure 2.2, we obtain the following binary decision program:

$$
\begin{aligned}
1 &\rightarrow x?\ 2 : 4 \\
2 &\rightarrow y?\ F : 3 \\
3 &\rightarrow z?\ F : T \\
4 &\rightarrow y?\ 3 : 5 \\
5 &\rightarrow z?\ T : F
\end{aligned}
$$

Such a program describes all the possible paths of the circuits. As we can see, compared to the Shannon decomposition, the function z is shared.

Several years later, Akers [Akers 1978] introduce the Binary Decision Diagram, as a graphical structure. These BDDs were used to represent switching functions extracted from the switching networks. As said before, the Binary Decision Diagrams can have two outgoing arcs, labeled by either 0 or 1, directed to the next layers and two terminal nodes 0 and 1. The arcs labeled by 0 are usually dashed in graphical representation.

Consider for example the BDD from Figure 2.3 that represented the Binary Decision Program defined for the switching circuit of Figure 2.2. This BDD shows the sharing of the node 3.

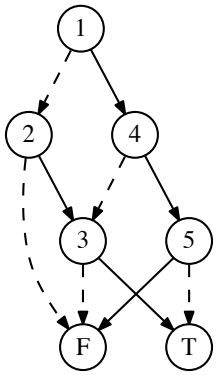


Figure 2.3: A BDD representing the switching circuit from Figure 2.2

In 1986, Bryant [Bryant 1986] published a groundbreaking work about BDDs. Bryant proposed to impose a total ordering on the variables of the BDD, introducing the Ordered Binary Decision Diagrams. Thanks to this, Bryant proposed many algorithm allowing to efficiently combine BDDs. Furthermore, using this total ordering, Bryant was able to reduce the OBDD into a canonical form giving the Reduced Ordered Binary Decision Diagrams (ROBDDs) which are widely used, and most of the time, BDD stands for Bryant's ROBDDs. In its works, Bryant distinguished the three important operations for BDDs which are the creation, the reduction and the combination.

The creation of BDDs and MDDs have been studied many times, as shown for building BDDs from Boolean formulae [Bryant 1986, Andersen 1997]. But since several years, BDDs and MDDs are built from many other sources, consider for example the work of Cheng and Yap [Cheng 2008, Cheng 2010, Cheng 2005] which builds MDDs from set of tuples or sub-problems. More about their work is given in chapter 4.

Moreover, several researchers aimed at building MDDs from dynamic programming [Hooker 2013, Bergman 2016b], by extracting a set of states and a transition function. Furthermore, they have made a seminal work on compiling constraint satisfaction problems (CSPs) and constraint optimization problems onto MDDs and in the analysis of their complexity [Hadzic 2008, Andersen 2007]. For example, in using an MDD as a domain store instead of classical set of values, which has helped solve several hard combinatorial problems.

Andersen et al. [Andersen 2007] proposed to limit the size of the MDD by limiting the number of nodes. Such MDDs are called the *relaxed* MDDs, since they are discrete relaxations of the constraints they represent. These relaxed MDDs usually represents super set of the solution, instead of the exact set of solutions. They can be built while compiling the CSPs, like in [Hadzic 2008], or by separation of the constraints [Ciré 2014a]. Another relaxation named the *restricted* Decision Diagrams have been introduced [Ciré 2014a] to represent a subset of the solutions, instead of a super set. One of the advantages of such relaxation is that the cost of a solution found in this *restricted* DD is a lower-bound of the best solution.

In the context of constraint programming, many works can be found. The first MDD propagator has been given by Cheng and Yap in [Cheng 2008, Cheng 2010], then several MDD propagators has been designed [Gange 2011, Perez 2014], all these works allow CP solvers to directly handle an MDD storing all the satisfying or prohibited tuples. Nowadays, almost all the CP solvers have an MDD constraint and MDDs are even part of the standard XCSP format [Boussemart 2016], and MDD where also available since 2000 in SICStus [Carlsson] and used to implement the transition constraints associated to the reformulation of an automaton, but without minimization.

In addition to these propagators, several works focus on constraining the MDD himself [Hoda 2010, Bergman 2014b], by, for example, enforcing that the cost of all the paths is in a given interval. Andersen et al. [Andersen 2007] have introduced the notion of MDD consistency, which enforces that each arc belongs to at least one solution.

Several distinct subjects are studied in this thesis, thus many chapters start with a related work section focusing on the exact subject of the chapter.

2.2.1 Automaton

Automaton have been often used in Constraint Programming. A deterministic finite automaton [Hopcroft 2006] can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with :

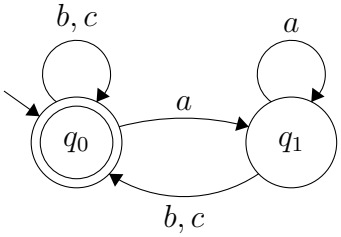
- A finite set of states Q
- A finite set of symbols Σ (the alphabet)
- A transition function δ between states. Q x $\Sigma \rightarrow Q$
- An initial state q_0

- A set of accepting states F.

An automaton *accepts* a word (sequence of symbols s_1,s_2...$s_n \in \Sigma$) if there exists a set of transition $\{(q_0, s_1, q_1), (q_1, s_2, q_2), ..., (q_{n-1}, s_n, q_n)\}$ and that q_n is an accepting state. The set of words accepted by an automaton is called the *language* of the automaton. The minimal automaton for a given language is the automaton recognizing the language and having the smallest number of state.

Example:

Let $\Sigma = \{a, b, c\}$. Consider the following automaton preventing *accepted* words to finish by an a:

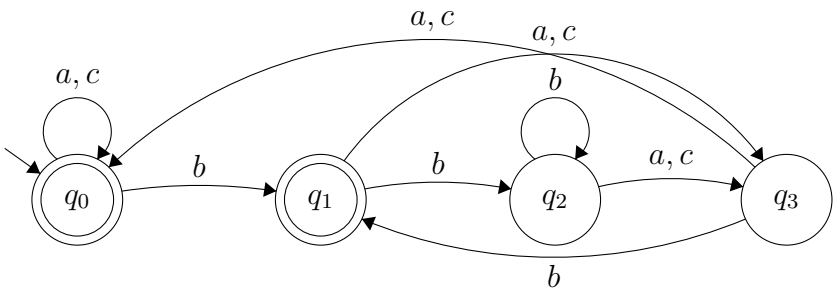


This automaton is not really complicated, and is readable.

It can be challenging to define automatons. This is often due to their ability to accept words of arbitrary sizes.

Example:

Consider the following automaton preventing *accepted* words to have a b in the *before last* position:



This automaton is not trivial anymore.

Combining automaton accepting words of arbitrary sizes is often harder than solving the problems with MDDs having fixed size.

Example:

Consider the automatons of the two previous examples. The product automaton of these automatons is given in Figure 2.4. This automaton represents the words that cannot finish by an a and not having a b in the *before last* position. It is almost unreadable for human.

These two simple unary constraints can be easily enforced with MDDs. Figure 2.5 shows the resulting MDD while applying the constraints over 3 variables. This MDD is simple and readable. This example shows that MDDs are well suited for fixed size words generation.

Even avoiding the fact that defining automaton is challenging, some simple unary constraints can lead to automatons having an exponential size, while the MDD has a linear size.

Example:

Consider the generalization of the previous examples, the language $(a|b|c)^*a(a|b|c)^n$. The DFA representing this language has 2^{n+1} nodes. The case with $n = 0$ and $n = 1$ are the two previous examples. A Non-deterministic Finite Automaton (NFA) can represent this language using $n + 1$ nodes. An MDD defined over k variables represents this language using k nodes, see Figure 2.5.

This example shows that an MDD can have comparable compression power to NFA. Furthermore, NFA can have an exponential compression power against DFA. The proposed example shows an MDD having an exponential factor of compression against DFA.

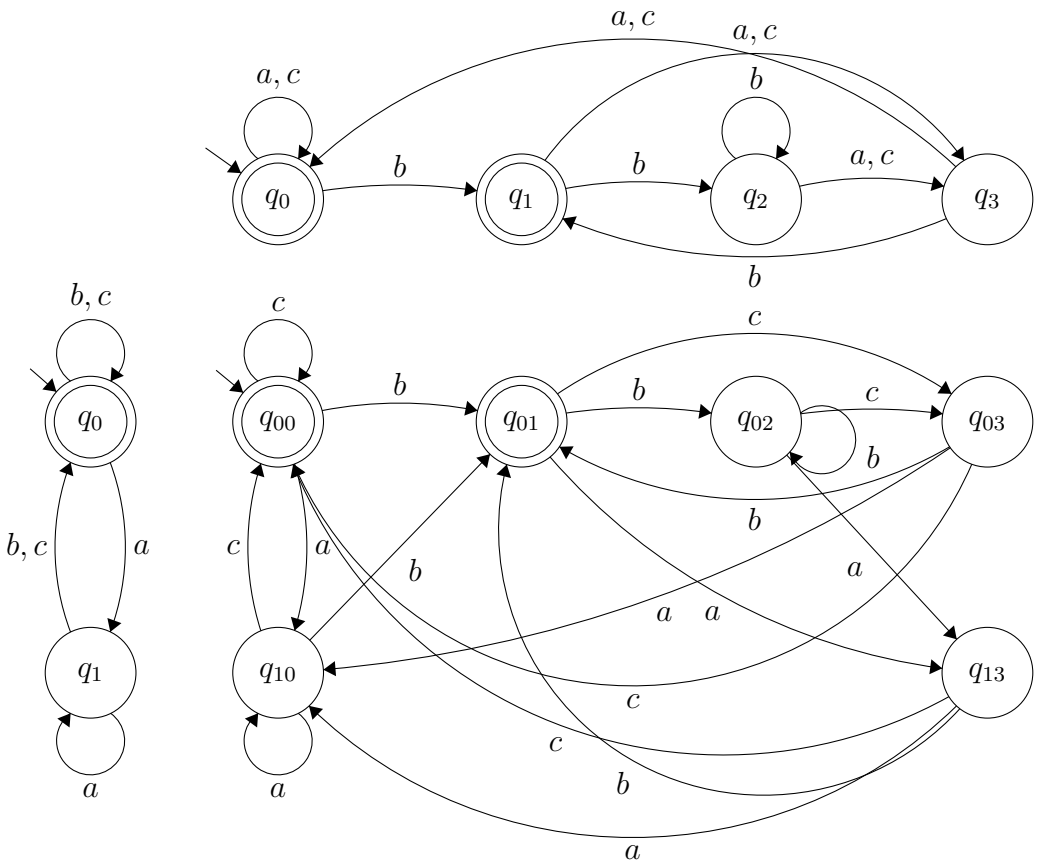


Figure 2.4: Intersection of two automatons representing unary constraints.

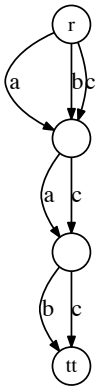


Figure 2.5: MDD representing solutions of two unary constraints over three variables.

Part I

MDDs: Fundamental Algorithms

Chapter 3

Reduction

Contents

3.1 Introduction

One of the main advantages of MDDs is their compression. MDDs are able to gain an exponential factor in representation space, but to do so MDDs have to be *reduced*. Reduction is an operation which consists of transforming an MDD into its smallest canonical form, for a given variable ordering [Bryant 1986]. This is one of the most important operations for MDDs.

Reduction operation merges equivalent nodes. Two nodes are equivalent if they have the same outgoing labeled paths.

Definition 1 *Two nodes u and v are equivalent, denoted by $u \equiv v$, if:*

$$|\omega^+(u)| = |\omega^+(v)| \wedge \forall (u, w, a) \in \omega^+(u), \exists (v, w, a) \in \omega^+(v) \qquad (3.1)$$

Nodes that have the same outgoing arcs can be easily merged. This is done by redirecting all incoming arcs of all the nodes to only one, and then removing the ones that do not have any more incoming arcs.

By using this definition for merging nodes, in a bottom up fashion, we can find all the equivalent nodes.

Proof If two nodes at layer i are equivalent, after reduction of the layer $i + 1$, they have the same outgoing arcs. This property is true for the last layer, and recursively true for the other layers.

Note that the idea is close to the tree isomorphism [Aho 1974].

Example:

Consider the MDDs from Figure 3.1. This example shows the merging of the two equivalent nodes, e and c.

In the left MDD, both the nodes c and e have arcs labeled by 0 and 1 and directed to the node `tt`, these two nodes are equivalent. This implies that we can merge them. The right MDD shows the MDD after the merging of nodes, the resulting node being the node ce. As we can see in this MDD, even if both a and b nodes have arcs labeled by 0 and 1 directed to the node ce, these two nodes are not equivalent since the node a has an arc labeled by 2 and directed to the node d.

When no more nodes can be merged, an MDD is said to be *reduced*. Note that the order in which nodes are merged has no impact since a reduced MDD, for a given variable ordering, is on a canonical form [Bryant 1986].

We can denote a Reduced Ordered MDD by the acronym ROMDD. In this thesis, for the sake of clarity, the acronym MDD is going to be used instead of ROMDD.

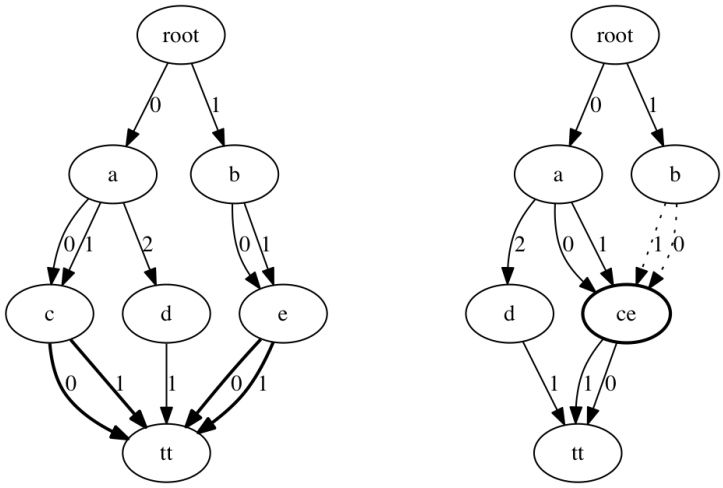


Figure 3.1: Example of reduction.

The problem Define a reduction algorithm finding all the equivalent nodes efficiently.

Plan This chapter is split in three parts. The first one describes the state of the art methods for reducing MDDs or BDDs. The second one introduces pReduce, a reduction algorithm, linear on the number of arc, and one of the contributions of this thesis. The third one proposes ipReduce, an incremental reduction version of pReduce.

3.2 Related Work

Several algorithms for reducing MDDs and BDDs exist [Bryant 1986, Cheng 2010, Andersen 1997, Brace 1991]. This section describes some of them.

One of the classical representations for MDDs is to represent each node by an array of outgoing arcs (cf Appendix A.1). The size of this array is fixed and is the size of the domain (d). Using this representation, the existence of an arc labeled by i is given by the value of the i^{th} cell of this array. If the value is `ff` then the arc does not exist, otherwise the cell contains the terminating node of the arc.

The access of the outgoing arc of node u labeled by i is denoted by $u[i]$. Using this representation, two nodes u and v are equivalent iff:

$$\forall i \in [1, d], u[i] = v[i] \tag{3.2}$$

Example:

Considering the MDD from Figure 3.1, the nodes c, d and e have the following arrays of outgoing arcs:

Node	0	1	2
c	`tt`	`tt`	`ff`
e	`tt`	`tt`	`ff`
d	`ff`	`tt`	`ff`

The nodes c and e have the same line (array of nodes), that is why they are equivalent.

Main idea The main idea behind a lot of reduction algorithms is to perform a search on the MDD and to merge equivalent nodes by memorizing all the already visited nodes in a data structures.

These algorithms usually process a DFS inside the MDD. During the post-visit, they use a dictionary-like data structure to search for a similar node of the current node. This kind of method has a complexity of $O(n * D)$ with

Algorithm 1 Reduction of an MDD using the classical DFS and a Dictionary.

```
REDUCE(M)
    define D
    root(M) ← reduceDFS(root(M),D)
REDUCEDFS(u, D)
    if ∃v ∈ D, u ≡ v then return v
    for each i ∈ [0, d] do
        u[i] ← reduceDFS(u[i],D)
    if ∃v ∈ D, u ≡ v then return v
    Add(D,u)
    return u
```

D the complexity of using a dictionary for finding equivalent nodes. The Algorithm 1 is a possible implementation of such algorithm.

Dictionary There are two ways for implementing such a dictionary that deserves some attention: by a radix tree or by a hash table.

A radix tree is a tree used to store words such that each node contains an array of outgoing edges of size $|\Sigma|$, where Σ is the alphabet of the words. To check if a word belongs to the radix tree, we have to check if a path using the letters of the word exists. An example of radix tree is given in Figure 3.2.

The use of a radix tree with words of size d having n different values gives a tree having d layers such that each node is an array of size n (n is the number of nodes of the MDD). Using this radix tree, looking for an equivalent node can be done in $O(d)$, and the insertion of a node may lead to the creation of d nodes of size n. This prevents us from using such a data structure.

That's why most algorithms use hash tables. In this case we cannot ensure reaching a $O(d)$ time complexity but we can expect the search in the table to be close to $O(1)$ once the hash code of the key has been computed, which is in $O(d)$. Such a result can be obtained by using a table whose size is greater than n when n elements are involved. The drawback of this approach in practice is that it may be time consuming to compute an efficient hash code and it needs a large table when we do not know n in advance.

Advantages This reduction method can be useful while performing operations like the Apply operator (see Chapter 5) on MDDs because it can reduce the MDD while performing the operation.

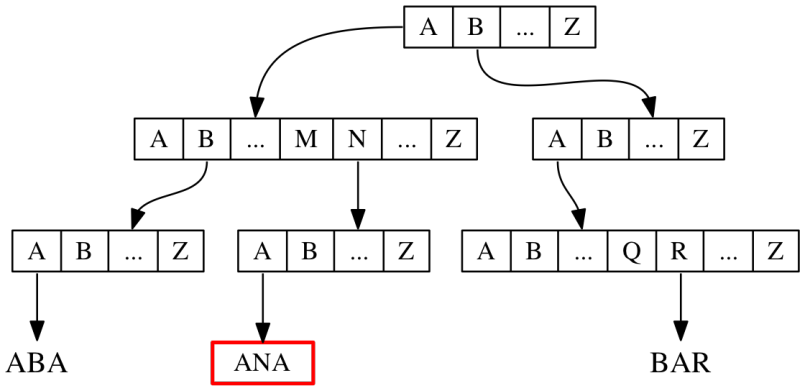


Figure 3.2: A radix tree containing the words ANA, ABA and BAR. The word ANA is obtained by following the path A->N->A in the tree.

Example:

Consider the MDDs from Figure 3.1. This example shows the application of the classical reduction method to the MDD on the left.

First, the algorithm processes a DFS and the post visit of the DFS stores the nodes in a Hash map. Starting from the *root* node. Using the values in the lexicographic order, this leads us to node c. Node c has $0tt1tt$ as signature, and thus is put in the associate cell of the hash map. The algorithm then goes at node d and put this node in the cell associated to signature $1tt$. Then the algorithm goes to node a and put it in cell $0c1c2d$. The next studied node is e with signature $0tt1tt$. Node c already has this signature, thus this two nodes are merged. This is done by returning c instead of e. The next processed node is b with signature $0c1c$, no node has the same signature, thus b is put in the Hash map. The result is the MDD on the right.

More Bryant [Bryant 1986] proposed an algorithm for BDDs that associates a unique key to each node based on their outgoing arcs. The nodes are then sorted and the algorithm checks for each two-consecutive nodes if they are equivalent (i.e. if their unique key is the same). With $d = 2$, a unique key based on the outgoing arcs can easily be generated, but generating a unique key to any $d > 2$ is costly. The next section shows how to do it incrementally without necessarily check all the arcs.

3.3 pReduce, a linear reduction operator

This section presents pReduce, named from "pack reduce", a reduction algorithm whose time complexity per node is bounded by its number of outgoing arcs and not by the number of possible values (d). In addition, the time and memory complexities are both linear on the size of the MDD $O(n + m)$.

This algorithm is close to the algorithm proposed for acyclic deterministic automaton [Revuz 1992] which performs a kind of lexicographic sort of nodes using bucket sort. But here, each bucket is going to be split by looking at the next arc, and thus the complexity is bound by the number of arcs instead of being quadratic.

This algorithm can be seen as an incremental version of the algorithm first proposed for BDDs [Bryant 1986]. Note that the direct application would have used a raddix sort, using as base the number of possible values, but the complexity would have been quadratic $O(d * n)$.

Representation The pReduce algorithm uses the ω^+ list implementation of an MDD (Appendix A.2). The particularity of this list of outgoing arcs is that they are ordered by their label. Thanks to several creation and operation algorithms presented in this thesis, we can assume that this property is always ensured.

Example:

Considering the MDD from Figure 3.1, the nodes c, d and e have the following lists of outgoing arcs:

Node	ω^+
c	{(0,tt),(1,tt)}
e	{(0,tt),(1,tt)}
d	{(1,tt)}

Main idea Instead of checking for each node if there exists an equivalent node in the MDD, pReduce tries to build clusters of equivalent nodes.

From equation (3.1), two nodes are equivalent if they have the same number of outgoing arc, and for each pair $(label, destination)$ from the ω^+ of the first one, there is a pair $(label, destination)$ in the second one.

Let the equal $=$ and not equal $\neq$ operators between two arcs be operators comparing both the label and the destination. Since the ω^+ lists of arcs are sorted by the label, we can apply the function of equivalence given in Algorithm 2.

Algorithm 2 Equivalence of two nodes.

```
EQUIVALENT(u, v)
    if |ω+(u)| ≠ |ω+(v)| then
        return False
    for each i ∈ 1..|ω+(u)| do
        if ω+(u)[i] ≠ ω+(v)[i] then
            return False
    return True
```

This equivalent function compares the outgoing arcs of two nodes while they are the same and stop at the first difference.

The pReduce algorithm performs this equivalence function incrementally over all the nodes at the same time. pReduce puts all the nodes of a layer into a set (i.e. a pack). Then it splits this set into several sets by comparing their first outgoing arc. Then for each of these new sets, it splits the nodes by comparing their second outgoing arc. The same process is applied iteratively over all the arcs until obtaining a node alone or having checked all the arcs of the nodes. When two or more nodes have all their outgoing arcs checked and they are still in the same pack, they are equivalent.

Pack A pack is a data structure. A pack p contains a set S of nodes and a position t. A pack ensures that all the nodes in S have the same prefix of t arcs in their ordered ω^+ list. Formally we have:

$$\forall u, v \in S, \forall (u, w, a) \in \omega^+(u)[1, t], \exists (v, w, a) \in \omega^+(v)[1, t] \tag{3.3}$$

As defined in section 2.1, $\omega^+(v)[1, t]$ denotes the sub-list of the t first elements of $\omega^+(v)$. A pack contains all the nodes having the same first t arcs (prefix). We denote by $|p|$ the number of nodes inside the pack p.

Splitting a pack During the processing of a pack, pReduce splits the pack into several packs according to their $t + 1^{th}$ arc. A pack is created for each pair of values (v, a) in the $t + 1^{th}$ arc (x, v, a) of the nodes.

pReduce Algorithm 3 is a possible implementation of the pReduce algorithm. It uses V_A and N_A two arrays of sets in order to perform the split operation of a pack p in $O(|p|)$. Array V_A is indexed by the values and array N_A is indexed by the nodes. Note that these two arrays contain only empty sets at the beginning and at the end of the operation. It also uses V_{list} and

N_{list} two lists of elements that are used to save the entries that are not empty in the arrays. At the end of the algorithm the lists are empty.

Algorithm 3 has two phases. First, it splits the current pack p into the array of sets V_A according to the value labeling the arc at position $\texttt{pos}(t)+1$. The second phase considers each set computed in the first phase and splits its elements into the array of sets N_A according to the terminating node of the arcs. The algorithm also modifies the computed sets in order to detect nodes that can be merged and to define packs and put them into the queue Q. The time complexity of this algorithm depends only on the number of neighbors of a node, because thanks to the lists we never reach an empty cell. In addition each arc is traversed only twice: one for the value and one for the node. The space complexity is in $O(n+d)$. The operation $\omega^+(x)[i+1]$ can be performed by keeping the last checked arc for each node.

The reduction of the whole MDD is made by applying a BFS from the bottom to the top and by calling Function REDUCELAYER for each layer with L the list of nodes to merge as a parameter.

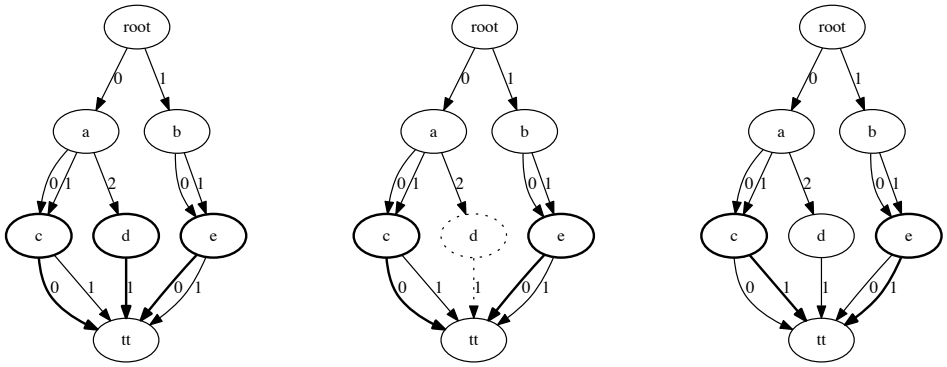


Figure 3.3: Application of the pReduce algorithm on the last layer.

Example:

Consider the MDDs from Figure 3.3. This example shows the application of the pReduce algorithm to the MDD from Figure 3.1.

First, all the nodes of the layer of the last variable (c,d,e) are put in a pack. Then their first arc is studied. This arc is (0,`tt`) for c and e and is (1,`tt`) for d. This is shown on the MDD on the left, remember that the arc are sorted by their label. Two packs are thus created, one containing c and e and one containing only d. The packs that contain only one element

are removed, thus the pack containing d is removed, MDD on the middle. The algorithm now processes the second arc of the pack (c,e), MDD on the rigth. This two nodes are on the same pack while no more nades has to be processed, they are equivalent.

The algorithm now processes the layer of nodes a and b, they are on the same pack for their two first arcs $(0,c)$ and $(1,c)$, but they are splitted since b does not have any arc to process and a is alone in its pack thus not processed.

Improvement While the complexity is already linear, we can try to improve the efficiency of this reduction algorithm by first splitting the pack of the nodes of a layer by considering the size of the ω^+ list of arcs.

Complexity Since pReduce only considers the common prefix of the outgoing arcs list, the complexity can be defined as the sum of the common prefix of the nodes. The worst-case complexity is bounded by $O(n + m + d)$.

More This notion of pack is important, using it, we are going to modify the pReduce algorithm in order to deal with incremental modification of MDDs (next section) and parallel version of the algorithms (chapter 6).

3.3.1 ipReduce, Incremental reduction

Some algorithms presented in this thesis in chapter 5 modify MDDs that was already reduced. After these modifications we want to reduce the obtained MDDs. A simple method is to apply the existing reduction algorithms. But classical reduction operations consider the whole MDD, even when small modifications occur.

ipReduce is an incremental adaptation of the pReduce algorithm, allowing to save time while reducing previously reduced MDDs after their modifications.

Main idea After the modification of an MDD, only modified nodes, newly created nodes or nodes having arcs directed to such nodes can lead to a merge. The idea is to consider only the packs that contain such nodes.

In order to be able to know whose nodes have been modified since the last reduction, we need to store this information inside the nodes. Let m be the field of nodes that contains a stamp whose value is the stamp of the last modification.

The algorithm 4 is a possible implementation of this algorithm. As you can see, the first line of the `iReducePack` function check if the current pack contains or not a node whose m value is equal to the last modification stamp m_g. Note that this can be maintained by keeping in each pack the `max` value of the m values of the nodes while building it.

While the worst-case complexity of this algorithm remains the same as the non-incremental version, the advantage in practice is important. Also, this incremental version of the reduction can become the only one in an implementation of an MDD package since it considers all the nodes at the creation of an MDD.

An application of this algorithm is shown in the chapter 5 section 5.3.

Algorithm 3 pReduce of an MDD.

```
pReduce(L)
    define V_A, N_A, V_list, N_list
    for each i from r − 1 to 0 do
        reduceLayer(L[i], V_A, N_A, V_list, N_list)

reduceLayer(Layer, V_A, N_A, V_list, N_list)
    delete nodes without outgoing neighbors
    define the pack p with Layer, 0
    Q ← ∅
    reducePack(p, V_A, N_A, V_list, N_list, Q)
    while Q ≠ ∅ do
        pick and remove p from Q
        reducePack(p, V_A, N_A, V_list, N_list, Q)

reducePack(p, V_A, N_A, V_list, N_list, Q)
    i ← pos(p)
    for each x ∈ p do
        v ← value(ω⁺(x)[i + 1])
        if V_A[v] = ∅ then add v to V_list
        add x to V_A[v]
    for each v ∈ V_list do
        for each x ∈ V_A[v] do
            y ← node(ω⁺(x)[i + 1])
            if N_A[y] = ∅ then add (v, y) to N_list
            add x to N_A[y]
        V_A[v] ← ∅   for each (v, y) ∈ N_list do
            if |N_A[y]| > 1 then
                define a pack p′ with ∅, i + 1
                M ← {x ∈ N_A[y]/|ω⁺(x)| = i + 1}
                merge all elements of M together
                add N_A[y] − M to p′; add p′ to Q
            N_A[y] ← ∅
        N_list ← ∅
    V_list ← ∅
```

Algorithm 4 ipReduce of an MDD.

```
ipReduce(L)
    define V_A, N_A, V_list, N_list
    for each i from r − 1 to 0 do
        iReduceLayer(L[i], V_A, N_A, V_list, N_list)

iReduceLayer(Layer, V_A, N_A, V_list, N_list)
    delete nodes without outgoing neighbors
    define the pack p with Layer, 0
    Q ← ∅
    iReducePack(p, V_A, N_A, V_list, N_list, Q)
    while Q ≠ ∅ do
        pick and remove p from Q
        iReducePack(p, V_A, N_A, V_list, N_list, Q)

iReducePack(p, V_A, N_A, V_list, N_list, Q)
    if ∄ u ∈ p, m(u) = m_g then return
    i ← pos(p)
    for each x ∈ p do
        v ← value(ω⁺(x)[i + 1])
        if V_A[v] = ∅ then add v to V_list
        add x to V_A[v]
    for each v ∈ V_list do
        for each x ∈ V_A[v] do
            y ← node(ω⁺(x)[i + 1])
            if N_A[y] = ∅ then add (v, y) to N_list
            add x to N_A[y]
        V_A[v] ← ∅   for each (v, y) ∈ N_list do
            if |N_A[y]| > 1 then
                define a pack p′ with ∅, i + 1
                M ← {x ∈ N_A[y]/|ω⁺(x)| = i + 1}
                merge all elements of M together
                add N_A[y] − M to p′; add p′ to Q
            N_A[y] ← ∅
        N_list ← ∅
    V_list ← ∅
```

type of problem	pReduce	`Bryant`
rand-5-12-12-200-p12442	8.2	46.7
rand-8-20-5-18-800	74.5	191.8
crossword-m1c-uk-vg	50.2	668.2
crossword-m1c-ogd-vg	103.5	724.6
crossword-m1c-lex-vg	5.0	97.0
bdd-21-133-18-78	110.9	244.0

Table 3.1: Average creation time for each table constraint (ms) involved int the solver competition XCSP.

	1000 tuples			
	arity			
d	6	8	10	12
12	11	20.6	26.7	29.8
30	32.3	47.5	57.8	54.7
60	80.6	84.6	76.1	79.1

Table 3.2: Time gain factor of pReduce while growing the domain size and the arity with 1000 tuples.

3.4 Experiments

In these experiments, we will consider not reduced MDDs given and we will compare time needed to reduce them.

Configuration All these algorithms have been implemented in C++ and run on a 6 cores server (Inter 3930) having 64 GB of memory and running under Windows 7.

XCSP Competition The first problems are from the Solver Competition archive [Lecoutre 2009]. For each type of problem, the geometric mean of the reduction times of all the instances is show for the pReduce algorithm and State of the Art `Bryant` algorithms based on Hash + DFS. We obtain the results from Table 3.1 which clearly show the advantage of pReduce.

Random instances The methods are also compared on random table constraints. The tables 3.2 and 3.3 show the **gain factor** of pReduce against `Bryant`.

	arity = 12		
	tuples		
d	1K	10K	100K
4	8.2	5.3	5.9
8	20	18.7	18.6
12	29.8	25,8	38.6
30	54.7	40.1	110.6
60	79.1	55.1	150.0

Table 3.3: Time gain factor of pReduce while growing the domain size and the number of tuples with an arity of 12.

	Out-place		In-place	
instances	deletion	reduction	deletion	reduction
30*300K-300K	35,4	4.2	24.8	1.8
300K - 1K	5.3	0.7	1.2	0.6
90K-30K	2.1	0.2	1.6	0.2
300K-10	4.7	0.6	0.002	0.2

Table 3.4: Arity 12, domain size 10. Average deletion and reduction times (s) for random instances.

ipReduce on random instances The table 3.4 show the efficiency of the ipReduce

MaxOrder Problem The results given here are from the problem defined in chapter 17. The MDDs of these experiments have a domain size close to 11k. Only pReduce and ipReduce have successfully reduced these MDDs. The MDD (reduced) from the second operation has more than 1.2 millions of nodes and close to 200 millions of edges. The deletion operation are from the chapter 5. The table 3.5 show how the incremental version of the reduce can be efficient.

	Classic			In-place		
	deletion	reduction	total	deletion	reduction	total
First Operation	2	1.7	3.7	**1.3**	**0.9**	**2.2**
Second Operation	23.9	14.6	38.5	**1.5**	**6.3**	**7.8**

Table 3.5:

Conclusion This section has presented an efficient algorithm for reducing MDDs. This algorithm is linear, can be transformed into an incremental algorithm. Moreover, this algorithm can be performed in parallel (see Chapter 6). The experimental results show that it is efficient in practice. Furthermore these algorithms are easy to implement.

Chapter 4

Constructions

Contents

4.1 Introduction

Building an MDD is obviously the first step before using MDDs. Several creation methods for MDDs already exist, from Boolean formula [Bryant 1992], extensional table [Cheng 2008], regular expression [Pesant 2004] ... The advantage of representing data using MDDs is that MDDs are reduced after their creations and arc consistency algorithms may benefit from this compression.

Building an MDD can often be done while avoiding the enumeration of the solutions during the construction, like the creation made from dynamic programming [Hooker 2013, Trick 2003]. But they offer access to all of these solutions once the MDDs are created. Moreover using MDDs for representing several different constraints allow to use only one propagator in constraint

solvers for these constraints, thus it prevents to implement a bunch of different propagators. Finally, as presented in Chapter 5, representing constraints with MDDs allow the user to combine them aiming at building more powerful and advanced models.

Even if many transformations from existing data representations to MDDs exist, several data structures still do not have any transformation or efficient transformation. This chapter aims at giving a *good* transformation for tables, GCSs [Focacci 2001], sequences of tuples [Régin 2011] and automaton [Pesant 2004].

Related Works One of the first Decision Diagrams creations came from Boolean formula [Bryant 1992, Andersen 1997, Bryant 1986]. This transformation was successfully applied to formal verification and circuit verification. But BDDs and MDDs can be defined from many data structures or even directly from the problems by compiling CSPs or constraints [Cheng 2008, Pesant 2004, Andersen 2007, Hadzic 2008, Cheng 2012]. Consider for example building MDDs from dynamic programming [Hooker 2013, Cire 2013]. For this kind of problem, the Dynamic Programming problem is considered as a set of state and transition. But one of the problem of dynamic programming is that, compared to automata, the transition table is huge and cannot always fit in memory.

In order to build the MDD, they start with a initial state s_1 and unroll the transition state graph by computing it on the fly. Arcs of this graph depend on the state of the emanating node. Consider the knapsack problem [Trick 2003], let s_j be an intermediate state, let v_i be the cost of taking or not the i^{th} item, we can consider as the initial state the scalar 0, and the transition function is given by:

$$\delta(s_j, v_i) = s_j + v_i \tag{4.1}$$

Finally, some works focus on compiling CSPs on data structures close to MDDs [Koriche 2015, Cheng 2005, Mateescu 2008], the MDD here is different since it can have more information on its arc and special nodes like for the And/Or decision diagram.

Since there is many existing transformation, only a subset of the existing transformations are going to be presented, and for each transformation, it is explained before the new version.

Plan This chapter first describes state of the art methods for building an MDD from a trie and a table. Then the first linear algorithm for building an MDD from a table, a Global cut seed or a tuple sequence is presented. In addition, after having presented existing work about the transformation from

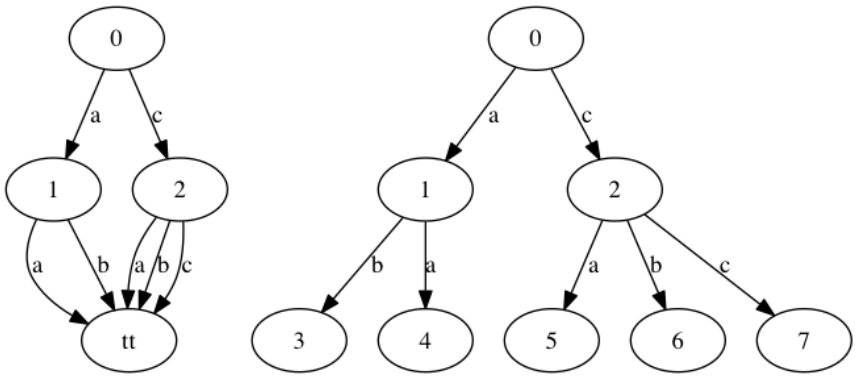


Figure 4.1: An MDD (left graph) and a trie (right graph) representing the tuple set {{a,a},{a,b},{c,a},{c,b},{c,c}}

an automaton or Dynamic Programming to an MDD, a simple and efficient algorithm doing it is presented. Finally, the experimental section shows the efficiency of these algorithms on industrial and random problems.

4.2 Table and Trie

4.2.1 Trie

A trie is a data structure used in Constraint Programming that can be used for compressing a tuple set [Gent 2007]. Each path from the root node to a leaf represents an allowed tuple.

A trie representing a set of T tuples will have exactly $|T|$ leaves. Each variable corresponds to a layer of the trie. A node has a maximum of d children, where d is the size of the domain of the corresponding variable of the node. An example of trie is given in Fig. 4.1. A trie can be transformed into an MDD by merging all the leaves into the terminal node tt and by applying the reduction operation [Cheng 2010]. See Chapter 3 for reduction algorithms. By using the algorithm proposed in Chapter 3, the complexity of the transformation is linear on the size of the trie.

4.2.2 Table

A table is a data structure where each row represents a tuple and where each column corresponds to a value of a variable. The table constraint [Lecoutre 2011, Lecoutre 2012a, Lhomme 2005, Mohr 1988, Bessiere 1997] is one of the most important constraints because it can represent any other

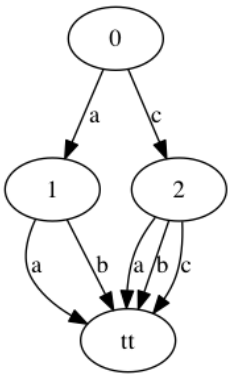
constraint. Moreover, the study of a good algorithm is still a hot topic [Demeulenaere 2016, Verhaeghe 2017].

Example:

Let T be the following table defined for two variables:

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

The MDD representing this table must have all these tuples as paths. The following MDD is the result:



The main advantage of building an MDD from a table is to gain space. Furthermore, since there exist several efficient MDD propagators that can be linear over the size of the MDD [Cheng 2008, Gange 2011, Perez 2014], MDDs can improve the computation time while using a constraint solver.

The classical method for building an MDD from a table come from Cheng and Yap [Cheng 2010]. The algorithm builds an MDD from a table by first defining a trie and then transforming the trie into an MDD, as defined in section 4.2.1. The transformation from a trie to an MDD is linear, thus the important part lies in building the trie.

Building the trie In order to build the trie, an incremental algorithm successively adds the tuples in the trie. To do so, a common root node is first created. Then paths, corresponding to tuples, starting from the root are created. The rooted sub-paths common to several tuples are merged together in order to be represented only once.

In order to merge the common sub-paths starting from the root, and since a trie cannot have more than d outgoing arcs, when the algorithm is at node

table						sorted table						trie				
a	a	c	a	a		a	a	b	a	b		a	a	b	a	b
a	b	a	b	b		a	a	b	a	c						c
a	a	b	a	c		a	a	c	a	a				c	a	a
a	a	b	a	b		a	b	a	a	b			b	a	a	b
a	b	a	a	b		a	b	a	b	b					b	b

Table 4.1: Creation of an MDD from a table of tuples.

u and needs to find the arc labeled by c, it starts by looking if such an arc exists or not. As presented in Appendix A, an array implementation is required if we want a random access to an arc using its label. Using the array implementation, the memory cost of a node is $O(d)$, and so is the creation cost. If the trie has n nodes, the complexity is $O(nd)$.

4.2.3 Linear table transformation

In this section, a simple method transforming a table into an MDD with a linear time and space complexity $O(n+m)$ is presented.

Main idea Since the costly operation in building a trie from a table is the need for a random access, we are going to preprocess the data in order to release this need. This pre-processing is a simple increasing lexicographic sort.

When a table is sorted, all the tuples having a common prefix (sub-path in the trie) are contiguous. Furthermore, consider the two tuples t_i and t_{i+1}, contiguous in the sorted table. By definition of the increasing lexicographic sort:

$$\exists k, t_i[k] < t_{i+1}[k] \wedge \forall j < k, t_i[j] = t_{i+1}[j] \tag{4.2}$$

Thus, using a sorted table, there is no need for a random access, the look for the arc only need to consider the last arc.

Sorting the table Sorting a table can be performed in a linear time because a tuple can be viewed as numbers having r digits where a digit can take up to d values. Thus we can sort a table containing t tuples in $O(r(t+d))$ by using a radix sort [Cormen 2001]. While the size of a table is $r * t$, the sort is linear in the size of the table. An example is given in Table 4.1.

Adding the tuples in the trie Starting from the root, the algorithm looks at the last arc in the ω^+ list, if the label of this arc is equal to the value of the

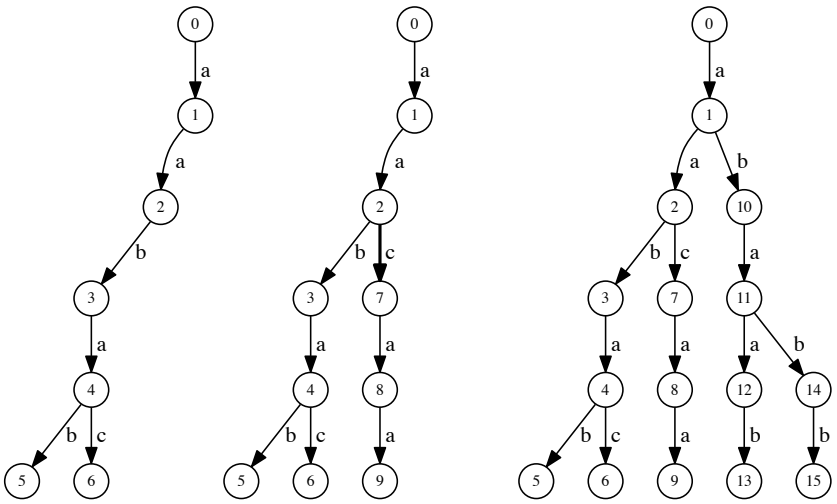


Figure 4.2: Construction of the trie from the tuple set given in Figure 4.1.

tuple, it uses the arc, otherwise it builds a new arc, whose label is the value of the tuple, and puts it in the last position of the ω^+ list. Algorithm 5 is a possible implementation.

Example:

Figure 4.2 shows the construction of the trie from the tuple set given in Figure 4.1. The most left trie contains only the first two tuple (a, a, b, a, b) and (a, a, b, a, c). The middle trie shows the addition of the tuple (a, a, c, a, a), as we can see, for node 2, the algorithm checks if the last arc is equal to c, it is not so the algorithm builds an arc labeled by c starting at 2 and the algorithm continues. The right most trie shows the final trie containing all the tuples.

Complexity This method is divided into three steps. The first step is the sort, which is linear in both space and time. The second step is adding the tuples in the trie, which is also linear in both space and time. The third step is to transform the trie into and MDD, which is linear in both space and time. Finally, we obtain a linear algorithm in both time and space for building an MDD from a table.

The experimental section at the end of this chapter shows the efficiency of this method.

Algorithm 5 Transformation of a table into an MDD.

```
TableToMDD(T)
    Sort(T)
    trie ← new Trie()
    trie.root ← new node()
    for each t ∈ T do
        u ← trie.root
        for each i ∈ 1..r do
            if label(end(ω+(u))) ≠ t[i] then
                pushBack(ω+(u), t[i], new node())
            u = dest(end(ω+(u)))
    Merge all the leaves
    return Reduce(trie)
```

4.3 Global Cut Seed and Tuple Sequences

4.3.1 Definitions

Compressed tuples improve the expressiveness of table constraints and try to reduce the complexity of the filtering algorithms by saving space. Thanks to such tuples, we can express more easily set of tuples. Therefore, it is interesting to represent them by MDDs in order to reinforce the compression and/or to allow the combination of these representations with other MDDs.

GCS A GCS (Global Cut Seed) is a compact representation of a tuple set [Focacci 2001]. A GCS is defined by a vector of value sets: $\{\{v_{1,1}, v_{1,2}, ..., v_{1,k_1}\}, ..., \{v_{n,1}, v_{n,2}, ..., v_{n,k_n}\}\}$, where each value set represents the set of values that can be taken by a variable. The Cartesian product of these sets defines the represented tuples.

Example:
Given D ={1,2,3,4}, the GCS $c = \{D, D, D, D\}$ represents the tuple set:

1	1	1	1
1	1	1	2
...	...	...	...
4	4	4	3
4	4	4	4

One GCS can represent an exponential number of tuples. However not

all tuple sets can be compressed by only one GCS. Two tuples can be represented by the same GCS if they have a Hamming distance equals to 1. For instance, the tuples {1,1,1} and {1,1,2} may be compressed into {1,1,{1,2}}. By contrast the tuples {1,1,1} and {1,2,2} have an Hamming distance equals to 2 and so cannot be represented by only one GCS. So, the compression of a table by a set of GCSs may require a huge number of GCSs.

In order to remedy this problem, tuple sequences have been introduced [Régin 2011]. They generalize GCSs.

Tuple sequences A tuple sequence encapsulates a GCS g and two tuples: t_{min} a minimum tuple, and t_{max} a maximum tuple. It bounds the lexicographic enumeration of the tuples of the GCS by these two tuples.

Example:

Let $D = \{1, 2, 3, 4\}$ then the tuple sequence : $g = \{D, D, D, D\}, t_{min} = \{1, 2, 2, 2\}, t_{max} = \{3, 1, 3, 2\}$, represents the tuple set:

1	2	2	2
1	2	2	3
1	2	2	4
1	2	3	1
...	...	...	...
3	1	3	1
3	1	3	2

Both of these data structures can be used in current Constraint Programming solvers. The next section provides transformation for both of them.

4.3.2 Transformations

Since a tuple sequence is a restriction of a GCS, we propose to first show how to transform a GCS, and then show how to restrict it.

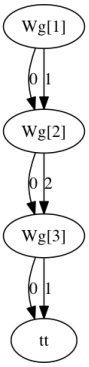
GCS The transformation of a GCS into an MDD is simple, since all the values of the ith set can be taken independently of the previous choice, the MDD representing a GCS is an MDD having 1 node by layer. We call such a node a wild card node.

Wild card nodes Let g be a GCS. There is at most one wild card node per layer i per GCS g which is denoted by $w_g[i]$. The wild card nodes are linked

together. All the arcs outgoing from $w_g[i]$ are incoming arcs of node $w_g[i+1]$ and all arcs outgoing of node $w_g[n-1]$ are incoming arcs of tt. Let $g[i]$ be the value set of g for the layer i, all the outgoing arcs of node $w_g[i]$ are labeled by values in $g[i]$.

Example:

Let $g = \{\{0,1\},\{0,2\},\{0,1\}\}$, the wild card of this GCS is :



Wild card nodes are a simple method for representing a GCS.

Tuple sequences While we are able to build an MDD representing a GCS, thanks to the wild card nodes, we want to build an MDD representing a tuple sequence. The idea is to use the t_{min} and t_{max} tuples to bound the MDD representing the GCS.

Let $s = (g, t_{min}, t_{max})$ be a tuple sequence. The MDD representing s is built in three steps:

1. The paths corresponding to t_{min} and t_{max} are created.
2. Arcs from the nodes of the paths previously created to wild card nodes are created as follows. Consider the path created for t_{min}. For each layer i, let $g[i]$ be the value set of g for the layer i. For each value $a \in g[i]$ such that $a > t_{min}[i]$ we create an arc from the node n_i of the path representing t_{min} to the wild card node $w[i+1]$. We repeat this process for the path created for t_{max}. In addition, we add a particular treatment when a node is shared by the two initial paths: instead of considering all values of $g[i]$, we consider only the values in the interval $g[i] \cap]t_{min}[i], t_{max}[i][$.
3. From nodes $w[i]$ to node $w[i+1]$ we add as many arcs as there are values in $g[i+1]$.

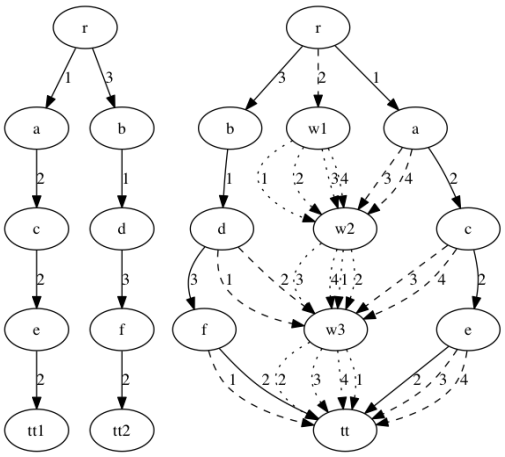


Figure 4.3: Creation of an MDD from the tuple sequence $g = \{D, D, D, D\}$, with $D = \{1, 2, 3, 4\}, t_{min} = \{1, 2, 2, 2\}, t_{max} = \{3, 1, 3, 2\}$

Example:

Fig. 4.3 shows the resulting MDD for the tuple sequence, $g = \{D, D, D, D\}, t_{min} = \{1, 2, 2, 2\}, t_{max} = \{3, 1, 3, 2\}$ with $D = \{1, 2, 3, 4\}$. The left graph contains the two paths from the first step representing the minimum and maximum tuples. The right graph represents with dashed lines the added arcs to wild card nodes. For instance, for node a each value in {1,2,3,4} greater than 2 labels an arc to node w_2. Arcs joining wild card nodes together and with tt are represented by dotted lines.

Complexity Let r be the number of involved variables. The number of nodes of the obtained MDD is bounded by $3(r-1)+2$. There are $2r$ arcs for the paths corresponding to t_{min} and t_{max}. There are at most $|g[i]|$ arcs from nodes of the t_{min} (resp. t_{max}) path to wild card nodes; There are $|g[i+1]|$ arcs from node $w[i]$ to node $w[i+1]$. Thus, there are at most $2\sum_{i=1}^{r}|g[i]|+2r$ arcs in the MDD. This is equivalent to the number of values of the tuple sequence.

Set of tuple sequences We can consider successively each tuple sequence and build for each sequence an MDD with the previous algorithm. Then, there are two possibilities. Either the tuple sequences are disjoint or not. The former case arises frequently (for instance when the tuple sequences represent a set of forbidden tuples). We just have to merge the MDDs. This can be easily done because they are disjoint. The resulting MDD has a space complexity equivalent to the set of tuple sequences and we have:

Property 1 *A set of disjoint tuple sequences can be represented by an MDD having an equivalent space complexity.*

The latter case is more complex. A set of disjoint tuple sequences may be computed from a set of non disjoint tuple sequences and each disjoint tuple sequence can be represented by an MDD. Nevertheless, it may create an exponential number of tuple sequences [Régin 2011] so an exponential number of MDDs.

4.4 Automaton

Recall from section 2.2.1.

4.4.1 Definition and related work

A deterministic finite automaton [Hopcroft 2006] can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with :

- A finite set of states Q
- A finite set of symbols Σ (the alphabet)
- A transition function δ between states. Q x $\Sigma \rightarrow Q$
- An initial state q_0
- A set of accepting state F.

An automaton *accepts* a word (sequence of symbols s_1,s_2...$s_n \in \Sigma$) if it exists a set of transition $\{(q_0, s_1, q_1), (q_1, s_2, q_2), ..., (q_{n-1}, s_n, q_n)\}$ and that q_n is an accepting state. The set of words accepted by an automaton is called the *language* of the automaton.

Example:

The Figure 4.4 represents an example of automaton. As we can see the state r is the initial one, and states a, b and c are accepting states.

The transition function δ of the automaton from 4.4 is:

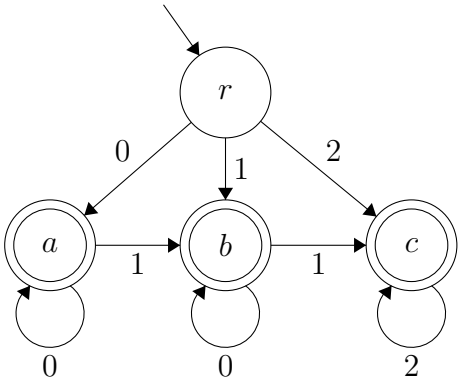


Figure 4.4: An example of automaton.

Q_s	Q_e	v
r	a	0
r	b	1
r	c	2
a	a	0
a	b	1
b	b	0
b	c	1
c	b	1
c	c	2

Q_s (resp. Q_e) denotes the starting (resp. ending) state of the transition.

Constraint The automata and regular constraints [Beldiceanu 2004a, Pesant 2004] deal with automatons and ensure that the solutions of the constraint belong to the *language* of the automaton.

The automata constraint [Beldiceanu 2004a] can be defined using a set of ternary transition constraints : $T(\delta, Q_i, x_i, Q_{i+1})$, where δ is the transition function of the automaton, Q_i and Q_{i+1} two state variables whose domain is Q and x_i is the variable of the constraint.

The Regular constraint [Pesant 2004] allows the use of regular language to constrain the variables, since a regular language can be represented by an automaton, we can use the automaton to constrain the variable.

Building a DAG from an automaton Existing methods for automaton generally unroll the automaton and builds a directed acyclic graph [Pesant 2004, Trick 2003]. At the first layer, the root represents the initial state. Then arcs and nodes representing a specific state are created, and fi-

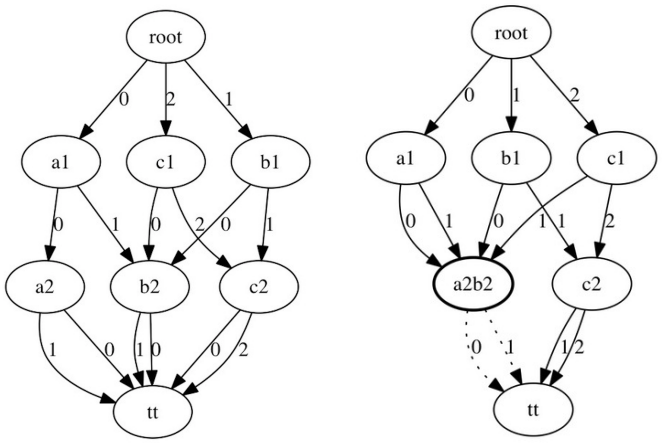


Figure 4.5: On the left, a Directed Acyclic Graph (DAG) representing the automata from Fig.4.4.On the right, an MDD representing the automata from Fig.4.4. An important remark is that an MDD can reduce the size of the DAG representing the automata.

nally only accepting states are created for the last layer. An example is given in Figure 4.5.

The exact same technique in addition with a reduction scheme can be used in order to obtain an MDD [Hooker 2013, Cire 2013].

4.4.2 New method

The method presented in this section is a simple and efficient method for building an MDD from an automaton when the transition table is huge. See the chapter 17 for an application with a big transition table, and the experimental section of this chapter.

Main idea Using the same trick as for tables, sorting the transition table δ by the label allows to prevent the need for a random access. The following algorithm uses this idea.

1. The first step is to index the transition according to their transition value. This can be done with a linear complexity over the transition table, for example by using a counting sort.

2. For each layer i of the MDD, build as many nodes as there are states s in the automaton. Such nodes are denoted by p_s^i.

3. For each transition $t = q_s, q_e, v$ in the indexed transition table:

- If q_s is the initial state, then create an arc between the root node and the node $p^1_{q_e}$ labeled by v.
- For each layer $i \in$ [1,k-2] create an arc between $p^i_{q_e}$ and $p^{i+1}_{q_e}$ labeled by v.
- If q_e is an accepting state, then create an arc between $p^{k-1}_{q_e}$ and the true terminal node tt.

4. For each layer from the top layer to the bottom layer, nodes without incoming arcs are removed. Finally, reduce the obtained MDD.

Example:
The Figure 4.5 shows both of the methods for handling the automata from Figure 4.4. On the left, a classical unrolling scheme, on the right, an MDD. As we can see, the reduction operation has merged the nodes $a2$ and $b2$.

Complexity The complexity of this algorithm can be worse than unrolling method, since for each layer, the whole table of arc is built. The complexity is $O(|\delta| * r)$. In practice, in many random and existing benchmark, the proposed method is at least twice faster than classical method.

Furthermore, it is important to remark that each layer of the MDD cannot have more arcs than the number of tuples in the transition constraint. So, using an MDD explicitly does not introduce any additional costs if the arc consistency algorithms of the ternary constraints do not share a global transition table.

Finally, merging nodes and using MDDs can drastically improves the result while solving problems [Cheng 2010].

4.5 Experiments

4.5.1 Table

MaxOrder The Maxorder problem using MDDs, defined in Chapter 17, contains an MDD named MDD_4 representing all the sequences of 4 words from a corpus of books. The domain size of this problem is close to 11,000 values. The classical creation method for building an MDD from the corpus takes 13 213 ms. The method presented in this chapter takes 77 ms, including the sorting times. The resulting MDD contains more than 90 thousand nodes and more than one millions of arcs.

	min	max	average
gain factor	2.0	5.3	4.1

Table 4.2: Gain factor grid creation.

XCSP competition this experiment studies the performance of the new table creation algorithms against the classical method using the table from the XCSP competition. The times for sorting the elements are included into our results. Next is a table who gives the results for the most representative ones (Boolean, bigger domain size, random ...). "sorted" corresponds to the algorithm described here, "unsorted" is the classical creation method.

instances	creation	
	sorted (ms)	unsorted (ms)
crossword-m1c-ogd	**31.5**	66.2
crossword-m1c-uk-vg	**9.6**	23.1
nonogram-gp	**25.1**	34.5
rand-10-60-20-30	**70.9**	179.9
bdd-21-2713	**8.1**	11.6
bdd-21-133	**98.23**	122.3

Random On the other hand, random instances have been tested. Instances having 22 variables, 1,000 tuples and increasing the domain size is given in Fig. 4.6. This figure is important because it shows that the domain size does not influence the creation time.

We can see that even if the number of tuples or the number of variables increase, the sorted creation algorithm outperforms the existing one. we have also tested instances for all the combinations with domain size in the set {2, 4, 8, 12, 20, 25, 30, 45, 60}, arity in the set {6, 10, 14, 18, 22, 25, 30} and number of tuples in the set{30, 100, 150, 200, 250, 300, 500, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 7500, 10000, 12500, 15000, 17500, 20000, 24000, 28000, 30000}. For all these cases, The new method was better.

4.5.2 Automaton

MaxOrder The Maxorder problem using MDDs, defined in Chapter 17, contains an MDD named MDD_m representing a huge Markov transition function extracted from a corpus of books. The first method that unrolls this MDD takes 3 503 ms to build the MDD. The method define in this chapter takes 733 ms, including the indexing and reducing time.

pentominoes-int This experiment uses all the regular constraints defined from the pentominoes-int problems of the 2014 Minizinc Challenge, because an efficient algorithm is required to solve them. We compare the creation from a regular constraint that we propose versus the creation of the graph (which is a kind of MDD) performed by classical methods. The Table 4.2 shows the gain factors we obtain.

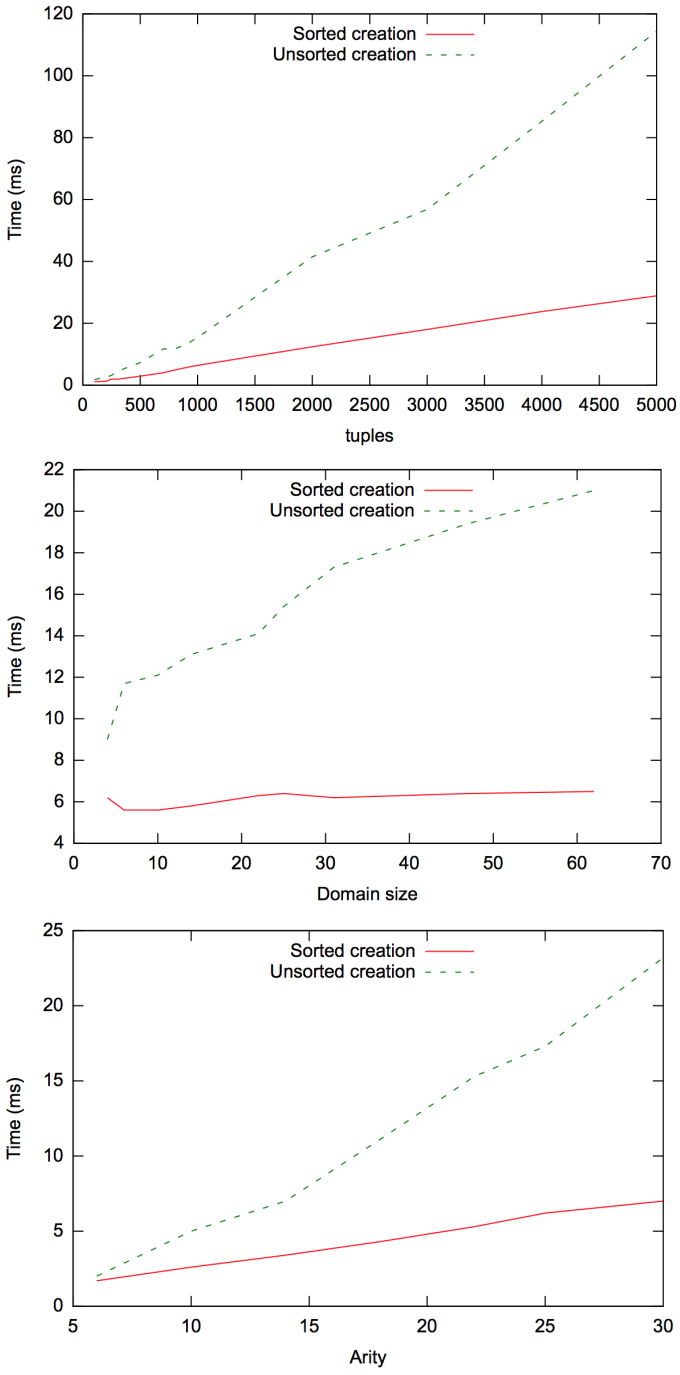


Figure 4.6: Sorted vs unsorted creation

Chapter 5

Operations

Contents

MDDs are efficient data structures for storing functions, tuples, etc. An advantage of representing data using MDDs is that they can be combined. These combinations allow the composition of functions, the intersection of set of tuples, the union of languages. The combination of MDDs is one of the most important operations and has been studied many times [Andersen 1999, Bryant 1986, Bryant 1992, Bergman 2014a, Brace 1991, Miller 1998, Srinivasan 1990].

Consider for example Constraint Programming, since we can convert several constraints into MDD and even any sub-problem, the intersection of two MDDs representing distinct constraints gives an MDD representing the conjunction of these constraints, which can improve the resolution.

Thanks to these combinations, modelers are able to build more efficient models. For example, operation between MDDs has been used in circuit verification [Andersen 1999], in compilation of product configuration [Hadzic 2004] and even during the search as filtering algorithms for constraints inside CP solvers [Bergman 2014b, Roy 2016]. Furthermore, all the models proposed in the Application part of this thesis use operation between MDDs in order to solve problems.

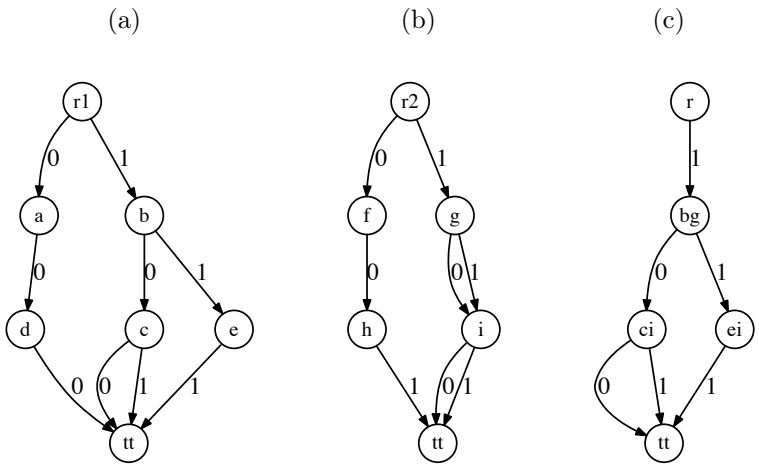


Figure 5.1: The MDD (c) represents the intersection of the two left MDDs. The tuples on MDD (c) are present in both of the left MDDs.

Example:

Consider the MDDs from Figure 5.1. The MDD (a) represents the tuples {(0,0,0),(1,0,0),(1,0,1),(1,1,1)}. The MDD (b) represents the tuples {(0,0,1),(1,0,0),(1,0,1),(1,1,0)(1,1,1)}. Le MDD (c) represents the intersection of the two MDDs (a) ad (b), this MDD contains the tuples {(1,0,0),(1,0,1),(1,1,1)}

This chapter is organized as follows, first a state of the art presenting the most used methods for performing an operation between is presented. Then a new version of this operation scheme is proposed and compared. Finally, an incremental version of this new algorithm is given for modifying MDDs.

5.1 Related Works

One of the most famous methods for combining BDDs is the Apply operator from Bryant [Bryant 1986, Bryant 1992]. This Apply operation, defined for BDDs, generates a BDD representing a Boolean function, depending on two other BDDs.

5.1.1 BDD Apply

The Apply operation for BDD is based on the Shannon expansion:

$$F = v \cdot F_v + \bar{v} \cdot F_{\bar{v}} \tag{5.1}$$

This equation holds for any Boolean operator $\oplus$:

$$F \oplus G = v \cdot (F_v \oplus G_v) + \bar{v} \cdot (F_{\bar{v}} \oplus G_{\bar{v}}) \tag{5.2}$$

Main idea Using the equation 5.2, the Apply operation builds a BDD by recursively applying the equation, while the nodes are not terminal nodes. Finally, when the nodes are both terminals, their values are 0 or 1, the $\oplus$ operator is applied.

The algorithm 6 is a possible pseudo-code for the Apply operation. I have chosen in this pseudo-code to use a loop from 0 to 1, because the MDD generalization of this algorithm will be easier to see.

Note: The algorithm's behavior is the same independently of the function $\oplus$. For each pair, if the nodes are both terminals then it applies the operator $\oplus$ to the values, otherwise, it recursively builds the outgoing arcs of the current node.

Unique table and Processed table In order to gain times and to not repeat the same process again and again, the algorithm keeps a data structure storing the processed pairs of nodes from both BDDs. This data structure keeping the already processed pair is named P in the algorithm.

In the same manner as defined in Chapter 3 about reduction, the Apply operator keeps a data structure storing the nodes in order to find equivalent nodes during the operation. This data structure prevents having twice the same node in the resulting BDD, implying that the BDD is reduced. The name H is used in Algorithm 6 to denote this data structure.

Set of pairs P The first thing that the algorithm does while processing a pair of nodes, is to check in the already processed set of pairs P if the current pair exists. If such a pair already exists, then this pair is returned, otherwise, this is the first time we reach this pair, then we can process it and add it to P. The lines 1 and 5 from Algorithm 6 show the implementation of this method.

The set P contains all the combinations of nodes considered during the processing of the algorithm, thus we have the following equation:

$$|P| \geq |F \oplus G| \tag{5.3}$$

There are two ways for computing this set. The first one uses a 2-dimensional array, using this representation implies an initialization with time and space complexity of $\Omega(|F| \cdot |G|)$ but a random access to each cell during the algorithm. The second method is to use a dynamic hash table with a specific hashing function. Using a hashing function, we need to first build a "Big enough" array and it may be required to extend this array. The complexity is not constant anymore for this last one.

Algorithm 6 Apply operator for BDD.

```
APPLY(F, G, ⊕)
    init(P)
    ApplyDFS(F,G,⊕)
APPLYDFS(u, v, ⊕)
    if u is terminal ∧ v is terminal then
        return u ⊕ v
1   if P contains (u,v) then
        return P.entry(u,v)
    w ← new Node
2   for each i ∈ [0, 1] do
        w(i) ← ApplyDFS(u(i),v(i),⊕)
3   if H contains w' s.t w' ≡ w then
        w ← w'
    else
4       Add w to H
5   Add {(u,v) ⇒ w} in P
    return w
```

Unique table H Lines 3 and 4 maintain the unique table. The unique table contains the set of nodes defined by their pairs (low, high). A node $u = (low_u, high_u)$ has an equivalent node in H if there exists a node $v = (low_u, high_u)$ in H.

The unique table in this algorithm allows the Apply operator to build a reduced BDD. By definition a BDD is reduced if for any nodes u and v in the BDD, then $u \not\equiv v$. During the construction of the resulting BDD, a new node is returned if and only if no equivalent node has been already defined, line 3. Thanks to this, during the operation, the current result is reduced.

With BDDs, this unique table is generally implemented using a perfect hash function, and a "big enough" array. For example, an array of 15,485,863 cells has been proposed [Andersen 1997]. This perfect hashing function is defined over the two values (low,high) of a node. Note that while processing the Apply operation, this array may need to be extended.

Loop over [0,1] On line 2, the loop is the main part of the algorithm. The node is represented by its Shannon expansion, equation 5.2. To do so, for each of the possible values, in a BDD only 0 and 1, the sub-BDD starting from the node and restricted to this value is computed. This computation is

then recursively done.

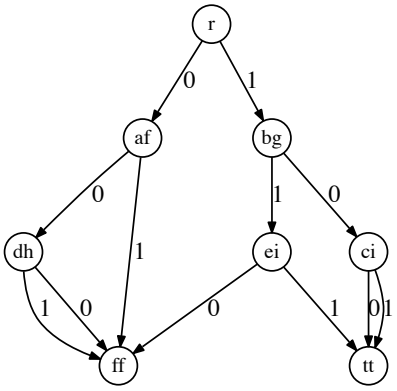
Example:

Consider the two MDDs on the left of Figure 5.1. We apply the Bryant's algorithm for processing the intersection.

First, we define the H and P data structures (both Hash table). Then starting from the pair containing the node *r1* from the first MDD and node *r2* from second MDD, thus (*r1*,*r2*). (*r1*,*r2*) is not in P, we can follow the arcs labeled by 0 starting from both nodes on their respective BDDs and obtain the pair (a,f). This new pair is not in **P** thus we follow the arcs labeled by 0 in both nodes and find the pair (d,h). This new pair is not in **P**, we can follow the arcs labeled by 0 and find the pair (`tt`,`ff`). Since both of the nodes are terminal nodes, we can apply the Boolean operator, which is a `and`. We obtain *False*, and return to the pair (d,h), we follow the arcs labeled by 1 and find the pair (`ff`,`tt`). They are both terminal nodes, thus we apply the operator and obtain *False* again. We return to pair (d,h), since all its children are directed to a *False* node, the pair is set to `ff` and the pair (d,h) is added into (d,h).

The same work is made for pair (a,f). We are now back on pair (*r1*,*r2*) and follow the arcs labeled by 1. We find the pair (b,g) which is not in P. Starting from this pair we follow the arcs labeled by 0 and reach the pair (c,i). This pair is not in P thus we start following the arcs labeled by 0 and reach the pair (`tt`,`tt`). This pair contains only terminal nodes, thus we apply the operator and obtain a *True*. Then, starting from (c,i), we follow the arcs labeled by 1 and reach (`tt`,`tt`) again, which is equal to True. We have processed all possible values (0 and 1), thus we can add the pair (c,i), associated to its just built node, into both P and H. We return at node for the pair (b,g) and follow the arcs labeled by 1. We obtain the pair (e,i), which is not in P, thus we follow first the arcs labeled by 0, obtain the pair (`ff`,`tt`) equals to *False* and then we follow the arcs labeled by 1 and obtain (`tt`,`tt`) thus *True*. The DFS return to the root by adding all the remaining nodes into both P and H.

The BDD processed during this example is given below:



5.1.2 BDD to MDD

An adaptation of the BDD Apply operator for MDDs has been done by [Srinivasan 1990, Miller 1998]. The main difference between MDDs and BDDs is the number of outgoing arcs, which is no longer 2 but d.

The Shannon expansion from (5.2) with d values by nodes becomes:

$$F = \sum_{i=0}^{d-1} (v = i) \cdot G_i \tag{5.4}$$

And for any Boolean operator $\oplus$:

$$F \oplus G = \sum_{i=0}^{d-1} (v = i) \cdot (F_i \oplus G_i) \tag{5.5}$$

The adaptation of the algorithm mainly changes in the loop over the values. Here the algorithm needs to iterate over the d values. A pseudo-code for this MDD adaptation of BDD apply is given in Algorithm 7.

As we can see, in the pseudo-code of the adaptation, only one line is changed. But even if the modification is small, the complexity has strongly changed.

Loop over d values The loop, which was previously defined over 2 values, now has to deal with d values. This implies that the complexity by nodes is now in $O(d)$, and so the overall worst-case complexity is in $O(|F \oplus G| * d)$.

Unique table H The previous implementation of the unique table was depending on two values, so a perfect hashing function was defined and the use of a "big enough" array for the hash table was sufficient. The problem is

Algorithm 7 Apply operator for MDD.

```
APPLY(F, G, ⊕)
    init(P)
    ApplyDFS(F,G,⊕)
APPLYDFS(u, v, ⊕)
    if u is terminal ∧ v is terminal then
        return u ⊕ v
    if P contains (u,v) then
        return P.entry(u,v)
    w ← new Node
    for each i ∈ [0, d] do
        w(i) ← ApplyDFS(u(i),v(i),⊕)
    if H contains w' s.t w' ≡ w then
        w ← w'
    else
        Add w to H
    Add {(u,v) ⇒ w} in P
    return w
```

that the more d increases, the bigger this table has to be. This implies that with d values, it is generally not possible to use a perfect hashing table. A dictionary of words can be used, but as described in section 3.2, the complexity is not linear.

5.1.2.1 Other Algorithms

ITE operator The ite operator defined in [Brace 1991] can define all the two variables Boolean operations. This operator computes for three BDDs F, G and H: If F then G else H.

$$ite(F, G, H) = F \cdot G + \bar{F} \cdot H \tag{5.6}$$

The algorithm for this expansion is close to the BDD Apply algorithm. The difference is: when the node n from F is terminal (True of False), then the result is G if $n = 1$ and H otherwise.

Case operator The adaptation of the Apply operation to MDDs of the Algorithm 7 considers Boolean terminal nodes. Let m be the number of terminal values. The Case operator [Srinivasan 1990] can define most of the

useful operations on MDDs. This operator takes as argument F, a function (MDD), and m other MDDs (G_0,...G_{m-1}). Let H = Case(F,G_0,...G_{m-1}), if $F(X) = i$, then H(X) = $G_i(X)$. For the case operator, the algorithm is close to the MDD Apply except that it performs the DFS over $m + 1$ different MDDs, F and the m G_i MDDs. The difference is again when the node n of F is terminal, then if $n = i$ the result is the current node from G_i.

Melding In its book [Knuth 2011], Knuth proposes to refine the process of the function based apply by not expending nodes which will be reduce to either the `tt` node or the `ff` node. To do that, he uses the information coming from the operation, for example, while performing an intersection, we should stop expending pairs composed of at least one `ff` node.

This work can be seen as a motivation of building algorithm leaded by the wanted operation. To do so, we are going to consider MDDs as tuple sets instead of functions. Thanks to that, we are going to define an algorithm performing set operations, which will be fully leaded by the operation.

5.2 Graph-Based Apply

Motivations As presented before, existing algorithms are based on applying the $\oplus$ operator on the leaves: we can say that they are function based operators. These operators process in a recursive way which is often associated with a DFS over the resulting BDD/MDD. Consider the two MDDs from Figure 5.2. When we apply classical operators in order to build the intersection, we obtain an empty MDD. But the algorithm processes the DFS graph from Figure 5.3. Thus modified of these algorithm has to be used.

The most important part it that the complexity of the existing operators mainly depends on the complexity of the data structures used for P and H. If the programmer of a BDD or MDD package does not pay enough attention to these data structures, the package can be orders of magnitude slower than with efficient data structures.

These are the motivations for this chapter. It proposes another method for performing the classical operation on MDDs or BDDs. This new method does not need to wait for the leaves for making decisions and does not need any specific or complex data structure.

The content of this section is twofold. First the graph-based version of the apply operation is presented. Second the modification allowing no need of any specific or complex data structure is presented.

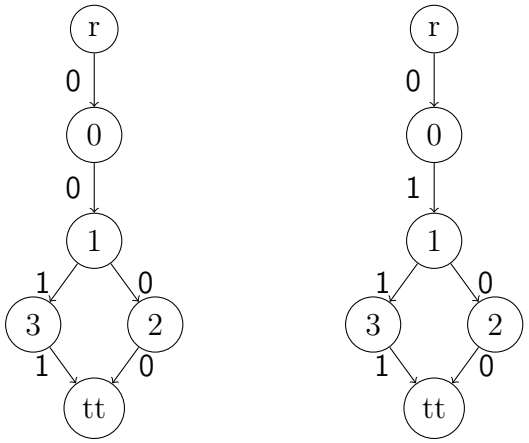


Figure 5.2: The left MDD represents the set {(0,0,0,0),(0,0,1,1)}. The right MDD represents the set {(0,1,0,0),(0,1,1,1)}.

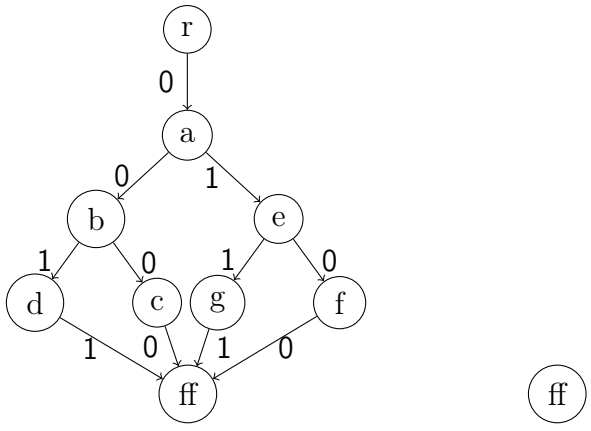


Figure 5.3: The left MDD represents the processing of classical Apply operator over the two MDDs from Figure 5.2. The result is the empty MDD on the right.

5.2.1 Graph-Based Algorithm

MDDs can be seen as a compressing tuple store data structure. When a node is reached, the labels vector of the path used from the root node to the current node is a prefix of at least one tuple contained in the MDD. This implies that extracting the tuples from an MDD consists of enumerating all the paths from the root to the `tt` node.

Since an MDD is considered as a tuple store, it does not need to have arcs directed to `ff`. The algorithm proposed here used the List implementation described in Appendix A. This implies that each node contains an ordered list named ω^+ of outgoing arcs.

Main Idea Using this representation, the information maintained and processed by the operator is a pair of two nodes from the input MDDs. But this is not the only information, since the paths labels used in both MDDs or BDDs are the same. This implies that both MDDs contain a tuple with this labels vector as a prefix.

The Graph-based Apply is going to use this information. Instead of processing a function of the leaves, it processes the result using the graph structure of the MDDs. More precisely, the decisions are made using the arcs instead of the leaves.

Today's algorithm for MDDs tends to converge to Graph-Based algorithm for operations. For example, in [Bergman 2014a] a kind of graph-based algorithm has been provided for the intersection. This algorithm builds the outgoing arcs of a node if and only if both of the nodes have an arc with the same label.

The rest of this chapter explains how the graph-based apply works. First the decision-making process is given, then the algorithm.

Decisions An arc has 3 information, both its ends and its label. While considering a pair of nodes in the algorithm, the decision-making considers only the arcs label. But the following assumption is made by definition of the algorithm: the starting point of the arcs have at least one common prefix.

The decision of building or not an arc labeled by a in the resulting MDD depends on the existence of an arc labeled by a in the pair nodes. This offer 4 possibilities: does it exist an arc labeled by a in the first node or not; does it exist an arc labeled by a in the second node or not.

Definition 2 *Let $(u, v) = w$ be a pair of nodes from both the MDDs operand A and B. Let a be any possible label value in $[0, d]$. The four possible cases for the decision of building an arc labeled by a in w are:*

	op[0] $\neg a_1 \wedge \neg a_2$		op[1] $\neg a_1 \wedge a_2$		op[2] $a_1 \wedge \neg a_2$		op[3] $a_1 \wedge a_2$	
layer	[1..r-1]	r	[1..r-1]	r	[1..r-1]	r	[1..r-1]	r
$A \cap B$	F	F	F	F	F	F	T	T
$A \cup B$	F	F	T	T	T	T	T	T
$A - B$	F	F	F	F	T	T	T	F
$A \Delta B$	F	F	T	T	T	T	T	F
$\overline{A \cup B}$	T	T	T	F	T	F	T	F
$\overline{A \cap B}$	T	T	T	T	T	T	T	F

Table 5.1: Different configuration of the *op* vector depending on the desired operation. A distinction is made depending on the layer of the resulting MDD.

$$\begin{array}{llll}
\exists a_1 \in \omega^+(u) \mid \ell(a_1) = a & \wedge & \exists a_2 \in \omega^+(v) \mid \ell(a_2) = a & \Longrightarrow 3 \\
\exists a_1 \in \omega^+(u) \mid \ell(a_1) = a & \wedge & \nexists a_2 \in \omega^+(v) \mid \ell(a_2) = a & \Longrightarrow 2 \\
\nexists a_1 \in \omega^+(u) \mid \ell(a_1) = a & \wedge & \exists a_2 \in \omega^+(v) \mid \ell(a_2) = a & \Longrightarrow 1 \\
\nexists a_1 \in \omega^+(u) \mid \ell(a_1) = a & \wedge & \nexists a_2 \in \omega^+(v) \mid \ell(a_2) = a & \Longrightarrow 0
\end{array}$$

Let op be the vector of decisions for building the arc depending on the four possible cases. The values of op[i] defining the binary operations are defined in Table 5.1 for the different combinations. More information are given in Figure 5.9.

Example:

Consider the intersection operation. An arc labeled by a in the resulting MDD is built if and only if both of the nodes have an arc labeled by a. As shown in Figure 5.4, the application of these rules of intersection builds first the outgoing arc from r labeled by 0 because both of the other root nodes contain an arc labeled by 0. But for the two pairs of nodes $a = (0, 0)$, with the node 0 from the left MDD and 0 from the middle MDD, does not have any arc with the same label. This implies that the algorithm stops at this node.

Now consider the union operation. An arc labeled by a in the resulting MDD is built if and only if one of the nodes has an arc labeled by a. The Figure 5.5 shows the application of this decision-making. From the node r, both have an arc labeled by 0 thus we can create the arc $(r, a, 0)$ with $a = (0, 0)$, then only the first node has an arc labeled by 0 thus we create the arc $(a, b, 0)$ with $b = (1, ff)$. Then for 1 only one node has an arc labeled by 1 thus we create the arc $(a, e, 0)$ with $e = (ff, 1)$. The algorithm

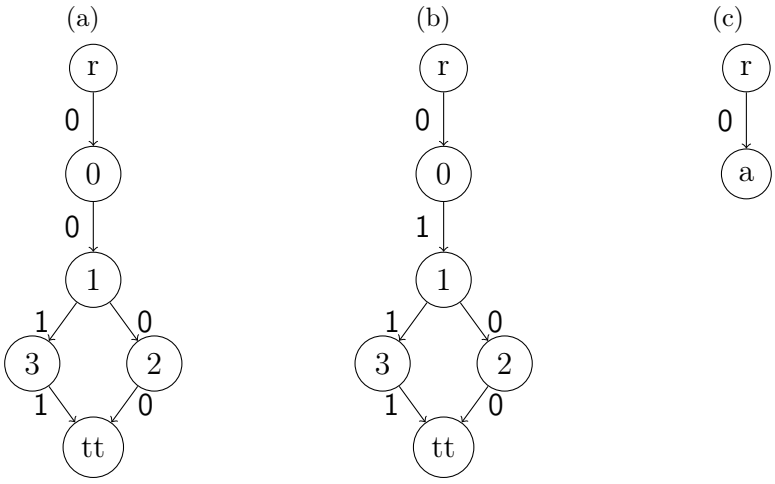


Figure 5.4: The MDD (a) represents the set {(0,0,0,0),(0,0,1,1)}. The MDD (b) represents the set {(0,1,0,0),(0,1,1,1)}. The MDD (c) is the application of the construction of the intersection of the two MDDs (a) an (b) using the rules define by Definition 2.

> then unfolds both of the MDDs using the $op[1]$ and $op[2]$.

Efficient implementation Since the nodes have an outgoing arc list ω^+ which is sorted by the label, an efficient decision-making can be implemented. In the same way as the merge used in fusion sort [Cormen 2001], the algorithm iterates over both of the lists and considers the smallest label.

When the algorithm is processing this pair, it makes the following steps:

1. Define $a_1 \leftarrow first(\omega^+(u))$ and $a_2 \leftarrow first(\omega^+(v))$.
2. Define $l = min(label(a_1), label(a_2))$. Switch :
   - $label(a_1) = l \wedge label(a_2) = l$: build the arc by considering $op[3]$
   - $label(a_1) = l \wedge label(a_2) \neq l$: build the arc by considering $op[2]$
   - $label(a_1) \neq l \wedge label(a_2) = l$: build the arc by considering $op[1]$
3. If $label(a_1) = l$ then $a_1 \leftarrow next(a_1)$. If $label(a_2) = l$ then $a_2 \leftarrow next(a_2)$.
4. If $a_1 \neq \text{null} \vee a_2 \neq \text{null}$ then return to step 2.

The maximum number of operations for a pair of nodes (u, v) is now bounded by $O(|\omega^+(u)|+|\omega^+(v)|)$, against $O(d)$ previously. As shown in step 2, it is impossible to reach the state $label(a_1) \neq l \wedge label(a_2) \neq l$. This particular case occurs for example while processing operations like $\overline{A \cup B}$. If one needs

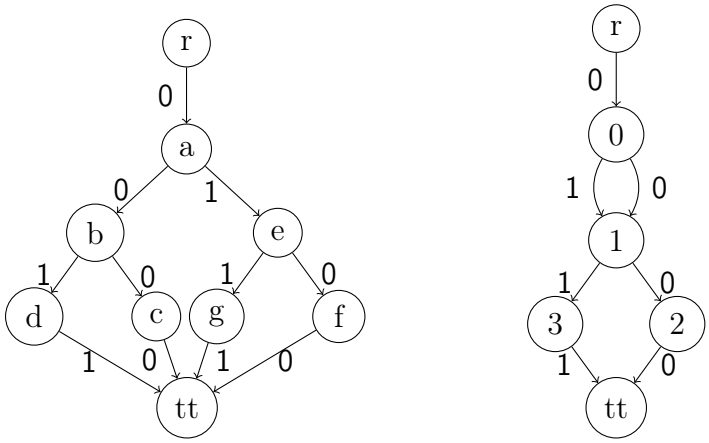


Figure 5.5: The left MDD represents the application of the union of the two MDDs on the left of Figure 5.4. The right MDD is the left MDD after the application of the reduction operator.

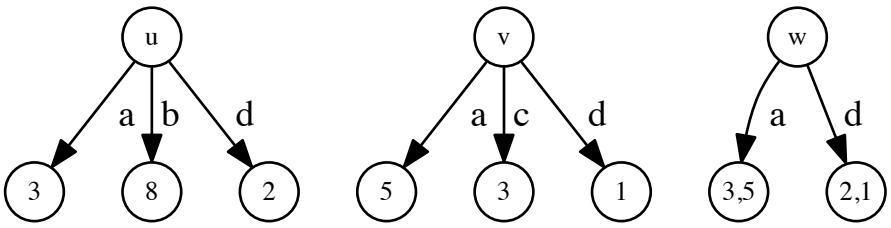


Figure 5.6: Nodes u, v and w from the example of the intersection.

to perform this operation, the classical iteration over the $[0, d]$ interval has to be performed.

Example:

Consider a pair of nodes $w = (u, v)$ with $\omega^+(u) = ((a, 3), (b, 8), (d, 2))$ and $\omega^+(v) = ((a, 5), (c, 3), (d, 1))$. Using this method, the algorithm is going to process the decisions $a_1 = a \wedge a_2 = a$, $a_1 = b \wedge a_2 \neq b$, $a_1 \neq c \wedge a_2 = c$ and $a_1 = d \wedge a_2 = d$. If the algorithm was processing an intersection, then both the arcs labeled by a and d are created. The Figure 5.6 shows the nodes u, v and w.

Building the new pairs Since only the $op[3]$ contains both of the nodes for building a pair, the three other pairs creations miss at least one node. When a node is missing, this implies that the algorithm has taken a prefix such that there is no tuple in the MDD starting with this prefix. In fact, in a representation having both the `tt` and `ff` nodes, this implies that the algorithm reaches the `ff` node. So when the algorithm reaches such a node, the pair is made using a `ff` node which is a node having no outgoing arc.

Let $M = (V, E)$ be an MDD and $[0, d]$ be possible values. Node `ff` is a node that does not have any outgoing arcs. Thus for each value, when the algorithm looks for an arc, the answer is always false.

Complement case The algorithm presented here is able to perform the complement of an MDD. But to do so, the operator needs to use the wild card nodes defined in Chapter 4.

Wild card nodes are special nodes depending on a Global Cut Seed (GCS) g and whose outgoing arcs are all directed to the wild card node of the next layer or to `tt`. The labels of the outgoing arcs of the wild card node of layer i are the values of $g[i]$.

Consider $D = \{0, 1, ..., d\}$ and the GCS $g = \{D, D, ..., D\}$. The MDD T representing g with $d = 1$ is the MDD from Figure 5.7 and containing all the possible tuples. Using this MDD, it is easy to build the complement of A by processing $T - A$. The right MDD of Figure 5.7 is the application of this method to the middle MDD.

The algorithm A possible implementation of this algorithm is given in Algorithm 8. As we can see a parameter V is given. If this parameter is given then the algorithm uses this set of values for the label testing part. Otherwise the union of the possible labels is made.

This algorithm performs a Breath First Search instead of the classical Depth First Search. The search is different because it will allow us not to need any data structure for both P and H. As we can see, the algorithm does not reduce the MDD during the processing but at the end. The explanation of this choice is also given in the next section.

5.2.2 Avoiding Data structures

Avoiding the P The P set contains all the pairs processed during the operation. When a new pair is created, the algorithms generally look into the P set in order to determine if such a pair has already been processed.

Using a Breath First Search, the pairs from layer i are processed only when all the pair from layer $i - 1$ have been processed. Furthermore, once a

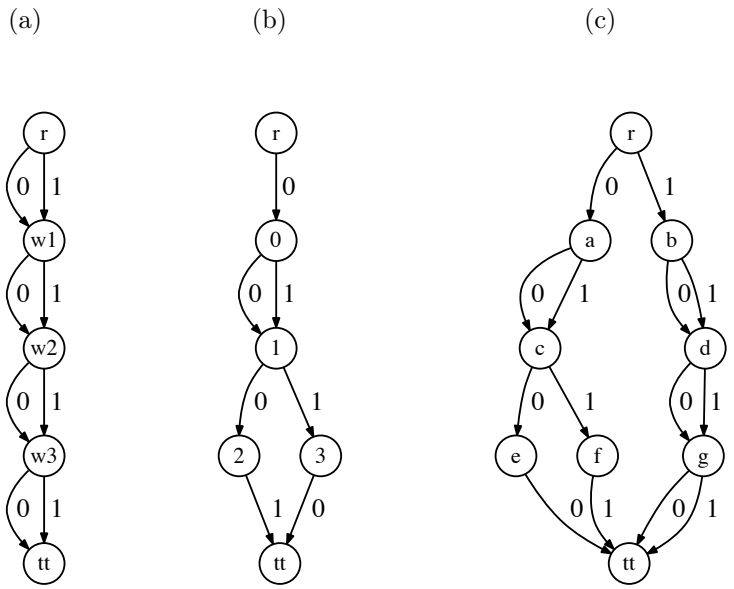


Figure 5.7: The MDD (a) T in this figure represents the GCS $g = \{\{0,1\},\{0,1\},\{0,1\},\{0,1\}\}$. The MDD (b) M is the MDD from Figure 5.5 which is the union of the two MDD from Figure 5.4. The MDD (c) is the complementary of the middle MDD. It has been constructed by applying the $T - M$ operation.

pair from layer i is processed, no more pair for layer i is generated. What is proposed here is to generate the pairs for layer i, and before processing the layer i, to merge all the equivalent pairs (i.e. pairs defined by the same two nodes).

This operation can be efficiently done, with a linear memory and space complexity. Let TM_1 be an array of sets whose size is the number of nodes from the first MDD and TM_2 be an array of set size of the second MDD. The sets can easily be implemented using list. The algorithm first indexes the pairs in the array TM_1 using the first node index, and then for each set of nodes, it indexes again the pair in the array TM_2 using the second node index. Algorithm 9 is a possible implementation of this method.

Avoiding the H The H hash table was used during the search in order to build a reduced MDD during the construction. The proposed algorithm does not reduce the MDD on the fly but at the end of the algorithm. This choice is made because it does not increase the memory complexity, see equation (5.3).

Algorithm 8 Generic Apply Function.

```
APPLY(mdd_1, mdd_2, op, V): MDD   // L[i] is the set of nodes in layer i.
  root ← CREATENODE(root(mdd_1), root(mdd_2))
  L[0] ← {root}
  for each i ∈ 1..r do
    L[i] ← ∅
    for each node x ∈ L[i − 1] do
      get x_1 and x_2 from x = (x_1, x_2)
      if V[i] = nil then
        V[i] ← VALUES(ω+(x_1) ∪ ω+(x_2))
      for each v ∈ V[i] do
        if ∄(x_1, v, y_1) ∈ ω+(x_1) then
          if ∄(x_2, v, y_2) ∈ ω+(x_2) ∧ op[0] then
            ADDARCANDNODE(L, i, x, v, ff)
          if ∃(x_2, v, y_2) ∈ ω+(x_2) ∧ op[1] then
            ADDARCANDNODE(L, i, x, v, ff, y_2)
        else
          if ∄(x_2, v, y_2) ∈ ω+(x_2) ∧ op[2] then
            ADDARCANDNODE(L, i, x, v, y_1, ff)
          if ∃(x_2, v, y_2) ∈ ω+(x_2) ∧ op[3] then
            ADDARCANDNODE(L, i, x, v, y_1, y_2)
  merge all nodes of L[r] into t
  PREDUCE(L)
  return root
ADDARCANDNODE(L, i, x, y_1, v, y_2)
  if ∄y ∈ L[i] s.t. y = (y_1, y_2) then
    y ← CREATENODE(y_1, y_2)
    add y to L[i]
  CREATEARC(L, i, x, v, y)
```

Example:

Consider the two MDDs on the left of Figure 5.8. We process the intersection of these MDD, that result to the MDD on the right.

First we build the pair (*r1*,*r2*), we consider here the smallest arcs, thus the arcs (*r1*,0,*a*) and (*r2*,0,*i*), the intersection build an arc if and only if both of the nodes have the arc, which is true here. We build the

Algorithm 9 Merge of equivalent pairs.

```
pairsMerge(L, mdd_1, mdd_2): L
    TM_1 = array of set (|mdd_1|)
    TM_2 = array of set (|mdd_2|)
    Q_1 = empty queue
    Q_2 = empty queue
    for each c = (u, v) ∈ L do
        if TM_1(u) = ∅ then
            Q_1.push(u)
        TM_1(u).add(c)
    for each u ∈ Q_1 do
        for each c = (u, v) ∈ TM_1(u) do
            if TM_1(v) = ∅ then
                Q_2.push(v)
            TM_2(v).add(c)
        for each v ∈ Q_2 do
            Merge all pairs from TM_2(v)
        Q_2.clear()
```

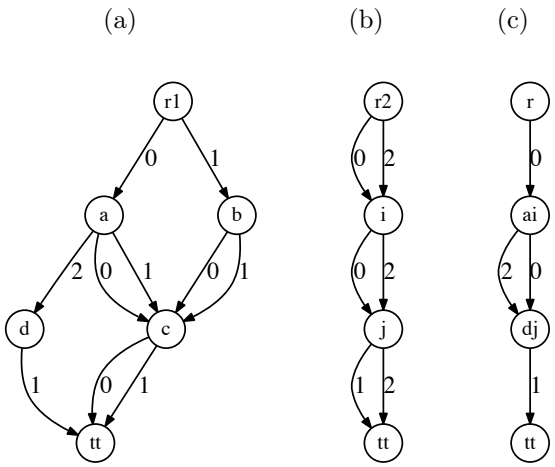


Figure 5.8: The MDD (c) represents the intersection of the two left MDDs. The tuples on the right MDD are present in both of the left MDDs.

arc (r,0,ai) and add the pair (a,i) to the next layer. Then the algorithm process the arc ($r1$,1,b) which is alone thus no arc creation for r, and same for ($r2$,2,i). We merge all the equivalent pairs, here their is only one pair thus nothing to do. We process the pair (a,i), both a and i have an arc labeled by 0, thus we can create the arc (ai,0,cj) and add the pair (c,j) to the next layer. Only a has an arc labeled by 1, thus we do not create the arc, and both have an arc labeled by 2, thus we create the arc (ai,2,dj) and add the pair (d,j) to the next layer. We have processed all the node of this layer thus we go to the next layer.

We first look for equivalent pairs, thus apply the Algorithm 9. This algorithm put the pair (d,j) in one cell and the pair (c,j) in another cell. They are both singleton, thus cannot be merged. The algorithm pick a node in the layer, the pair (c,j). Only node c has an arc labeled by 0, thus we cannot create this arc, both of them have an arc labeled by 1, thus we create the arc (cj,1,`tt`). Only j has an arc labeled by 2, thus we do not create such an arc. pair (d,j) is process in an equivalent way.

Finally, we apply the reduction algorithm from chapter 3 which merges the nodes of the pairs (c,j) and (d,j).

Complexity Let $M_1 = (N_1, E_1)$ and $M_2 = (N_2, E_2)$ be two MDDs, let $M = (N, E)$ be the resulting MDD from an operation between M_1 and M_2. First the memory complexity of the representation of M is $O(|N| + |E|)$, which is linear. The worst-case complexity of the operation is well now and is $O(|M_1| * |M_2|)$.

Special case: Multiple terminal values Several MDD constructors build MDDs having multiple terminal values, like the sum MDD [Trick 2003]. This kind of MDD can be transformed into a classical `tt ff` terminals MDD by adding a layer where arcs are leaving the terminal states a labeled by a and directed to `tt`. Such a transformation is done in Chapter 12 and in Chapters application 19 and 14.

Advantages Thanks to this new scheme of operations, efficient operators that can be processed in parallel, for relaxed or even non-deterministic MDDs will be designed in the next chapters.

5.3 In-place Operations

In this section, we define in-place algorithms for the addition/deletion of tuples from an MDD. The operator algorithms presented until here build a resulting

Operation	Expression	Bit	ITE	Graph-Based
0	0	0000	0	0000 | 0000
AND	$F.G$	0001	ite(F,G,0)	0001 | 0001
F > G	$F.\overline{G}$	0010	ite(F,$\overline{G}$,0)	0011 | 0010
F	F	0011	F	0011 | 0011
F < G	$\overline{F}.G$	0100	ite(F,0,G)	0101 | 0100
G	G	0101	G	0101 | 0101
XOR	$F \oplus G$	0110	ite(F,$\overline{G}$,G)	0111 | 0110
OR	$F + G$	0111	ite(F,1,G)	0111 | 0111
NOR	$\overline{F + G}$	1000	ite(F,0,$\overline{G}$)	1111 | 1000
XNOR	$F \oplus G$	1001	ite(F,G,$\overline{G}$)	1111 | 1001
NOT(G)	$\overline{G}$	1010	ite(G,0,1)	1111 | 1010
F ≥ G	$F + \overline{G}$	1011	ite(F,1,$\overline{G}$)	1111 | 1011
NOT(F)	$\overline{F}$	1100	ite(F,0,1)	1111 | 1100
F ≤ G	$\overline{F} + G$	1101	ite(F,G,1)	1111 | 1101
NAND	$\overline{F.G}$	1110	ite(F,$\overline{G}$,1)	1111 | 1110
1	1	1111	1	1111 | 1111

Figure 5.9: The proposed algorithm can perform at least of the operations done by classical algorithms. In addition, for example, we can do the intersection of the k first variables and then the union of the others, while it is impossible with existing methods.

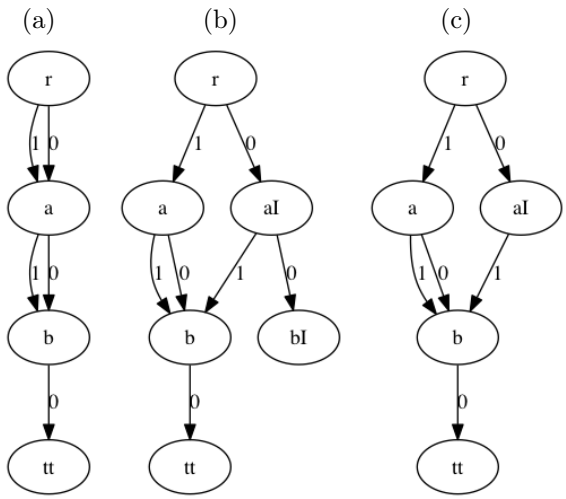


Figure 5.10: Tuple {0,0,0} is removed from the MDD (a). The isolation of the path corresponding to the tuple is performed (MDD (b)) and then the reduction is applied (MDD (c)). Nodes aI and bI are created from nodes a and b during the path isolation.

MDD.

The problem of removing compressed information inside of an MDD have been already studied in [Ciré 2014a]. The problem was to remove a partial assignment from an MDD. This problem can be seen as removing the GCS composed of the values of the partial assignment and a wild-card value for the other.

This section proposes to explore the insights of the modification possible for an MDD. Since for the operator presented before considers an MDD as a compressed tuple store, the addition or deletion of tuples in an MDD are studied.

5.3.1 Deletion of tuples from an MDD

First, an algorithm for deleting one tuple from an MDD is given. Then, the method is generalized for a set of tuples.

The deletion of a tuple τ from an MDD is based on an operation named path isolation, which is a kind of local decompression. The idea is to build a specific path whose arcs are labeled by the values of τ. Furthermore, arcs equivalent to the ones of the isolated path are deleted from the MDD. After the isolation process, the MDD is reduced. Let $\tau[i]$ be the value of the variable $x[i]$. The isolation is performed in 3 steps:

Step 1. Identify $a_1 = (root, \tau[1], n_2)$, the arc of the first layer labeled by $\tau[1]$ the first value of the tuple. Create the node ne_2, the arc $(root, \tau[1], ne_2)$ and delete the arc a_1. Set x_{mdd} (a node of the MDD) to n_2 and x_{path} (an isolated node) to ne_2.

Step 2. For each layer i from 2 to $r-1$ repeat the following operation. Identify $a_i = (x_{mdd}, \tau[i], n_{i+1})$ the outgoing arc from the x_{mdd} labeled with $\tau[i]$. Create the node ne_{i+1} and the arc $(x_{path}, \tau[i], ne_{i+1})$. For each arc (x_{mdd}, w, y) such that $w \neq \tau[i]$ create the arc (x_{path}, w, y). Set x_{mdd} to n_{i+1} and x_{path} to ne_{i+1}.

Step 3. For each arc (x_{mdd}, w, tt) such that $w \neq \tau[r]$ create the arc (x_{path}, w, tt).

If at any moment, the algorithm cannot identify an arc then it means that τ does not belong to the MDD. Fig. 5.10 shows the application of this algorithm. The complexity of the deletion of a tuple is bounded by $O(rd)$ because for each isolated node we need to recreate its arcs. However, in practice it is often close to $O(r)$.

5.3.1.1 Deletion of a set of tuples.

This section proposes a better method than repeating the previous algorithm for each tuple. Transform the set of tuples into an MDD and subtract this new MDD from the initial one by following the same steps of the previous algorithm. Isolate nodes having a common path in both MDDs, then remove the common arcs to the isolated nodes of the second last layer. At last, call the incremental reduction algorithm (see 3). Algorithm 10 is a possible implementation of this method.

> **Example:**
> Fig. 5.11 shows the subtraction of the GCS {1,{0,1,2,3},1} from the MDD representing all the tuples possible for the values {0,1,2,3}. The GCS is isolated from the MDD. Then, the deletion of the arc labeled 1 of node d corresponds to the deletion of only the tuples contained in the GCS.

It is difficult to bound the complexity of the deletion of T tuples, because the MDD created from them may compress the information.

5.3.2 Addition of tuples to an MDD

The addition of tuples into MDD follows the same principles as for the deletion. In this case, the isolated path contains arcs labeled by the values of the

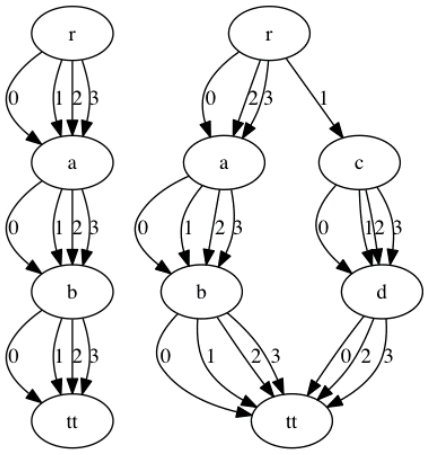


Figure 5.11: The left MDD represents all the possible tuples for the values {0,1,2,3}. The right MDD represents the deletion of the GCS {1,{0,1,2,3},1} from the left MDD.

tuple that must be added. It is performed by applying the same steps as for the deletion.

First, consider the addition of one tuple τ. The two first steps are very similar to the ones of the deletion. Except that at a point, there will be no more path in the MDD having the same subpath as τ. Otherwise, it would mean that τ is already in the MDD. Thus, at a certain moment we will not be able to identify any arc $(x_{mdd}, \tau[i], n_{i+1})$ as in step 2 in the deletion algorithm. When this case arises the algorithm can stop the step 2 and directly create the path from the current isolated node to the terminal node. This path will be labeled by the values of τ for the remaining layers. Step 3 can be skipped. At last, call the incremental reduction algorithm. The complexity of the addition of a tuple is in $O(rd)$ because for each isolated node we need to recreate its arcs.

5.3.2.1 Addition of a set of tuples.

Let mdd_1 be the initial MDD. Transform the set of tuples into an MDD, named mdd_2. Add mdd_2 to mdd_1 by following the same steps as for the previous algorithm. Isolate nodes having a common path in both MDDs. When an arc belongs to mdd_2, create an isolated node and create an arc from the current isolated node to it. When an arc belongs only to mdd_1, create an arc from the current isolated node to the node in mdd_1.

Fig. 5.12 shows the effect of the addition of the tuple {1,2,1} in the MDD given in Fig. 5.11. We can see the usefulness of the path isolation for avoiding

Algorithm 10 In-place Deletion Algorithm

```
DELETION(mdd_1, mdd_2)
  L ← empty Array of set of nodes
  for each (root(mdd_1), v, y_1) ∈ ω^+(root(mdd_1)) do
    if ∃(root(mdd_2), v, y_2) ∈ ω^+(root(mdd_2)) then
      ADDARCANDNODE(L, 1, root(mdd_1), v, y_1, y_2)
      DELETEARC(root(mdd_1), v, y_1)
  for each i ∈ 1..r − 2 do
    L[i] ← ∅
    for each node x ∈ L[i − 1] do
      get x_1 and x_2 from x = (x_1, x_2)
      for each (x_1, v, y_1) ∈ ω^+(x_1) do
        if ∃(x_2, v, y_2) ∈ ω^+(x_2) then
          ADDARCANDNODE(L, i, x, v, y_1, y_2)
        else CREATEARC(x, v, y_1)
  for each node x ∈ L[r − 1] do
    get x_1 and x_2 from x = (x_1, x_2)
    for each (x_1, v, tt) ∈ ω^+(x_1) do
      if ∄(x_2, v, y_2) ∈ ω^+(x_2) then
        CREATEARC(x, v, tt)
  IPREDUCE(L)
ADDARCANDNODE(L, i, x, y_1, v, y_2)
  if ∄y ∈ L[i] s.t. y = (y_1, y_2) then
    y ← CREATENODE(y_1, y_2)   add y to L[i]
  CREATEARC(x, v, y)
```

the addition of the tuples {1,{0,1,3},1}. The right MDD shows the impact of the reduction on the MDD: nodes e and b are merged because they have the same outgoing arcs. It is difficult to bound the complexity of the addition of T tuples, because the MDD created from them may compress the information. Algorithm 11 is a possible implementation of the in-place addition operations.

Duality The proximity of these algorithms is due to the duality of the problems: adding a tuple set T to an MDD M is equivalent to delete T from the complementary MDD of M.

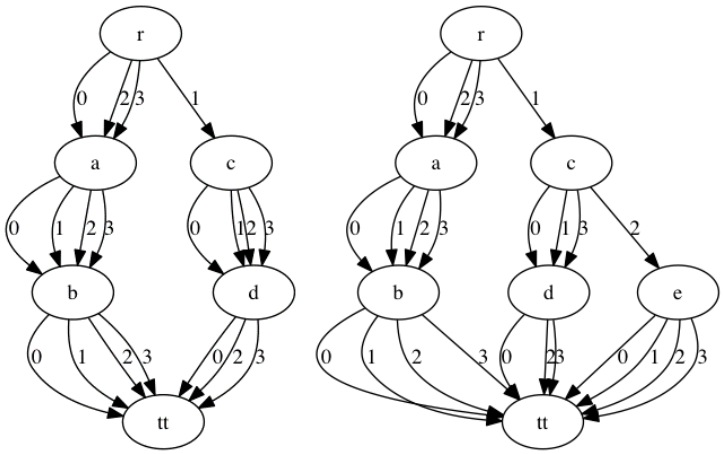


Figure 5.12: The right MDD represents the addition of the tuple {1,2,1} to the left MDD, before the reduction.

Algorithm 11 In-place Addition Algorithm

```
ADDITION(L, mdd_1, mdd_2)
for each v ∈ ω+(root(mdd_1)) ∪ ω+(root(mdd_2)) do
    if ∃ (root(mdd_1), v, y_1) ∈ ω+(root(mdd_1)) then
        if ∃ (root(mdd_2), v, y_2) ∈ ω+(root(mdd_2)) then
            ADDARCANDNODE(L, 1, root(mdd_1), v, y_1, y_2)
            DELETEARC(L, i, root(mdd_1), v, y_1)
    else ADDARCANDNODE(L, 1, root(mdd_1), v, nil, y_2)
for each i ∈ 1..r − 2 do
    L[i] ← ∅
    for each node x ∈ L[i − 1] do
        get x_1 and x_2 from x = (x_1, x_2)
        // If x_1 is nil then ω+(x_1) is empty
        for each v ∈ ω+(x_1) ∪ ω+(x_2) do
            if ∃ (x_1, v, y_1)  ω+(x_1) then
                if ∃ (x_2, v, y_2) ∈ ω+(x_2) then
                    ADDARCANDNODE(L, i, x, v, y_1, y_2)
                else CREATEARC(L, i, x, v, y_1)
            else ADDARCANDNODE(L, i, x, v, nil, y_2)
for each node x ∈ L[r − 1] do
    // If x_1 is nil then ω+(x_1) is empty
    for each v ∈ ω+(x_1) ∪ ω+(x_2) do
        CREATEARC(L, i, x, v, tt)
IPREDUCE(L)
```

	Classic		In-place	
instances	deletion	reduction	deletion	reduction
30*300K-300K	35,4	4.2	24.8	1.8
300K - 1K	5.3	0.7	1.2	0.6
90K-30K	2.1	0.2	1.6	0.2
300K-10	4.7	0.6	0.002	0.2

Table 5.2: Arity 12, domain size 10. Average deletion time (s) for random instances.

5.4 Experiments

All the applications of this dissertation use the Apply operation define here. Some results presented here are directly coming from these chapters. Chapter 17 contains many experiments using these algorithm, so do the two others application chapters.

MaxOrder The MaxOrder problem, is a sequence generation problem, based on corpus and avoiding plagiarism. This problem is one of the motivation for building new algorithms, since it contains thousands of different values by variables. This implies that each node can potentially have thousands of outgoing arcs. Using classical approaches, looking for each values, none of the existing algorithms were able to solve the problem for just 6 variables. It needed close to 64 GB of memory and the time needed was more than one hour.

Our apply algorithm solve the problem for 20 variables in 143.5 seconds. 20 variables was the goal at first. Moreover, the required memory is close to 8GB. For more details see chapter 17.

Random instances We propose to compare the performance of the classical and the in-place algorithms and the performance of the classical and the incremental reduction algorithms. We use random instances obtained using a uniform distribution.

On the instances column of TableThe 5.2, the first number corresponds to the number of the tuples represented by the MDD whereas the second number is the number of tuple that are removed from the MDD. This table shows that using in-place algorithm while remove small part of an MDD can drastically improve the processing time.

We also propose a table summarizing the advantages of the different algorithms. We add results for the BDD and MDD packages proposed in

#tuples	#deleted	Fct based	P&R15	in-place
20000	1000	159	11.5	6
40000	2000	291	40	21
40000	20000	663	51	33
80000	40000	2643	174	114
40000	10	466	185	19

Table 5.3: Arity 12, domain size 10. Average deletion time (s) for random instances.

[Srinivasan 1990, Brace 1991] (See column "Fct based"). "P&R15" represents the results we previously obtained and "in-place" column corresponds to the new algorithms. Table 5.3 gives some results for MDD representing 10,000s tuples. With such an amount of tuple and domain size, this is one of the worst case for classical algorithm. Moreover, the MDDs are often sparse, thus building an array for each column often lead to huge memory requirement.

Part II

MDDs: Advanced Algorithms

Chapter 6

Parallel Computing

Contents

6.1 Introduction

Parallelism is a classical approach for improving the efficiency of algorithms consisting in running an algorithm on several machines/cores/workers. Parallel version of many algorithms have already been proposed, but finding good parallel algorithm is often challenging. For example, many different works exist for parallelizing the search in CP solvers [Régin 2013, Hamadi 2002].

As shows in the application part of this thesis and in many existing applications, the computing time for operations and reductions of MDDs can be huge. For example, in chapter 17, up to 143.5 seconds are needed to process several intersections, and in chapter 19, more than 2,000 seconds are needed.

Thus parallel version of such algorithms can drastically improve the building of model. Note that the building of a model is an operation which is rarely parallelized.

This chapter first recall several existing works about parallelism and MDDs, then it proposes a parallel version of both the reduction and the apply operation.

6.1.1 Related Work

Related works The processing in parallel of MDDs or automata has been well studied and is still a hot topic Bergman et al. introduced a parallel B&B search that branches on nodes in the MDDs instead of branching on variable-value pairs as it is done in conventional search methods [?]. Other works focus on minimizing an automaton in parallel, using specific algorithms [Ravikumar 1996, Tewari 2002] or by using the well-known map-reduce principles [Hedayati Somarin 2016, Dean 2008]. Finally for Binary Decision Diagrams, parallel creation and manipulation algorithms [Kimura 1990, Stornetta 1996] have been designed. These algorithms use global hash-tables and they are organized so that locks are only needed for these global hash tables and the global tree nodes. In addition, a thread safe unordered queue using asynchronous messages is required. These algorithms and their implementation are quite complex, define complex dedicated data structures and use locks. The hash-table manipulation is describe in chapter 3.

This chapter proposes a new method for applying operations and minimizing MDDs or acyclic deterministic automaton in parallel without any complex parallel data structure, which is easy to implements and most important, without any lock.

6.2 Background

6.2.1 Parallelism

When a parallel program is correct, that is when race condition and deadlock issues[1] have been resolved, several other aspects must be taken into account to reach a good scalability. At least four difficulties that may impact that scalability can be identified:

- *Data dependencies*: it is a situation in which an instruction refers to the data of a preceding instruction. Thus, no program can run more quickly than the longest chain of dependent calculations (known as the

[1] Race conditions depends on the sequence or timing of processes or threads for it to operate properly. A deadlock is a state in which each member of a group is waiting for some other member to take action [Coulouris 2005].

critical path), since calculations that depend upon prior calculations in the chain must be executed in order.

- *Software lock-out*: it is the issue of performance degradation due to the idle wait times spent by the CPUs in kernel-level critical sections.

- *Resource contention* and particularly *false sharing*: it is a conflict over access to a shared resource such as random access memory, disk storage, cache memory, internal buses or external network devices. False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line.

- *Load balancing*: it refers to the distribution of workloads across multiple computing resources, such as cores.

For convenience we will use the word *worker* to represent an entity performing computation. Usually it corresponds to a core.

6.3 Parallel Reduction

The reduction of an MDD consists in removing nodes that have no successor and merging equivalent nodes, i.e. nodes having the same set of neighbors associated with the same labels. This means that only nodes of the same layer can be merged. In addition, two nodes can be merged iff they have the same signature. Figure 1 gives an example of reduction.

The main difficulty is to identify nodes having the same signature. The pReduce algorithm (Chap 3) improved previous algorithms that checked for each node whether it can be merged with another node or not. It works per layer and groups the nodes having the same suffix of signatures into packs[2]. Nodes that remain in the same pack with their entire signature, and not only a suffix, can be merged together. The worst-case time complexity is bounded by the sum of the size of common suffix of the nodes.

Efficient parallelization of this algorithm is not trivial, thus a simplified version of the sequential algorithm is considered first. Then, a more complex algorithm is presented to fit the best complexity of the sequential algorithm.

[2]The algorithm normally works with prefixes, but it can be straightforwardly adapted to deal with suffixes.

6.3.1 Parallel Sort

The identification of nodes having the same signature can be simply performed by sorting the node according to their signature. Since nodes and labels are integers, a linear sort algorithm can be used. We propose to consider a radix sort.

We reproduce the presentation in [**?**]. Consider that each element in the q-element array A has δ digits, where digit 1 (resp. δ) is the least (resp. most) significant digit. The radix sort algorithm consists of calling for $r = 1..\delta$ a stable sort to sort array A on digit r. The counting sort can be used as a stable sort. Counting sort assumes that each of the q input elements is an integer in $[0, k]$, for some integer k. It determines, for each input element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array. For example, if there are 17 elements less than x, then x belongs in output position 18. When several elements have the same value we distinguish them by their order of appearance in order to have a stable sort. Thus, the time complexity of the radix sort is $\delta O(q + k)$.

The parallel radix sort with w workers [Zagha 1991]. It uses a parallel counting sort as stable sort. Let V be a vector of q elements. The parallel counting sort splits V into w subvectors, one for each worker. Then each worker applies a counting sort on its subvector. Finally, the workers put the nodes in their new position.

Example. We propose to detail the parallel radix sort for a vector of nodes of the MDD. We assume that there each node has only one neighbors and one label. For the sake of clarity we represent an ordered pair (l_i, u_j) by ij, i.e. (l_0, u_1) is written 01.

Consider the following vector of nodes, the second line gives the index of the node, the third line is associated with the signatures.

a					b					
0	1	2	3	4	5	6	7	8	9	V
10	**00**	**11**	**11**	**00**	**10**	**10**	**01**	**11**	**01**	*sig*

Using two workers (a and b), we can split the vector into two independent parts and apply a counting sort. The two parts are $[0, 4]$ and $[5, 9]$, b is always after a in order to avoid collision. The first step of the radix sort considers the rightmost digit. Let $a\#_i$ (resp. $b\#_i$) be the number of i counted by worker a (resp. b). These numbers are computed by traversing the values. This is the counting step of the counting sort. We obtain:

$a\#_0 = 3$	$a\#_1 = 2$	$b\#_0 = 2$	$b\#_1 = 3$

Then we determine the global indices of each digit by workers, let ia_r (resp. ib_r) be the position in the resulting vector of the first value of the elements of part a (resp. b) whose current digit is r. We have $ia_0 = 0$; $ib_0 = ia_0 + a\#_0 = 3$; $ia_1 = ib_0 + b\#_0 = 3 + 2 = 5$ and $ib_1 = ia_1 + a\#_1 = 7$. When there are more than two workers the same principle applies: the information of the previous worker is used for the current worker. We have:

$ia_0 = 0$	$ia_1 = 5$	$ib_0 = 3$	$ib_1 = 7$

This step, denoted by the cumulative step, can also be performed in parallel. Each worker receives a set of values ($[0..k]$ is divided into w subranges) and performs the computations for its set of values. Each worker j computes the number of time each of its value is taken and $cs(j)$ the cumulative sum of these numbers. Then, we compute for each worker j the sum of the cumulative sum of the previous workers: $scs(j) = \sum_{i=1}^{j-1} cs(i)$. This can be globally done in $O(w)$. From these scs values, each worker computes the global indices of its values.

Using these positions, the workers a and b can independently build the global vector without any collision and thus without lock. So this last step, named the position step, can also be performed in parallel by assigning to each vector its initial subvector of elements. For example, a puts the value 10 of node 0 in position $ia_0 = 0$ then, it increments ia_0, so $ia_0 = 1$, then it puts the value 00 of node 1 in position $ia_0 = 1$ and increments it again, etc. The global vector is:

0	1	4	5	6	2	3	7	8	9
10	**00**	0**0**	**10**	**10**	**11**	**11**	0**1**	**11**	0**1**

The same process has to be applied to the second digit in order to sort the nodes. We finally obtain:

a					b				
1	4	7	9	0	5	6	2	3	8
00	00	01	01	10	10	10	11	11	11

Complexity. For each worker, the counting step[3] is in $O(q/w)$, the cumulative step is in $O(k/w)$ and the position step is in $O(q/w)$. Thus, the overall time complexity of the parallel counting sort is in $O(q/w + k/w)$. The parallel

[3]The count array can be initialized by traversing the elements to set the count of their values to 0 after the different steps.

radix sort considers all digits of the signatures, thus its time complexity is in $\delta \times O(q/w + k/w)$, which is more than the sum of common suffix of the pReduce algorithm. For instance, consider the following signatures of nodes: 0010, 1101, 1012, 0113, 1004, 0015, 0116. The algorithm will process for each node all four digits while the sequential algorithm will process only the least significant one.

In order to remedy this issue, the pReduce algorithm works with packs, which are common suffixes. Two nodes belong to the same pack when their signature have the same suffix defined by the pack. Only nodes in the same pack can be merged. So, if at a moment, the signature of a node u has a different digit value for a given position than any other node of its current pack, then u cannot be merged with any node and it can be ignored. Nodes that remain in the same pack will be merged at the end.

Working by digits such as the radix sort is similar as working by common suffixes. Thus, we propose to apply the same mechanism for the parallel algorithm by introducing the pack notion.

6.3.2 Parallel pReduce

Packs and leaders. For the sake of clarity, node at position p in the layer is denoted by u_p. Consider the iteration r of the radix sort. A pack is a set of nodes having signatures sharing the same digits from 1 to r (1 being the least significant digit). This means that for any iteration all nodes of a pack are consecutive. A particular node of a pack can be identified: its leader. The leader is the node of the pack having the smallest position. Then, for any node u_p with $p > 0$, either u_p is a leader and u_{p-1} belongs to another pack than u_p, or u_p is not a leader and u_{p-1} belongs to the same pack as u_p. Therefore, packs can be deduced from leader nodes.

At the beginning of the algorithm, all nodes are in the same pack, and the pack leader is the first node. For the previous example, all nodes have 0 as leader (first line):

0	0	0	0	0	0	0	0	0	0	ldr
0	1	2	3	4	5	6	7	8	9	V
10	0**0**	1**1**	1**1**	0**0**	1**0**	1**0**	0**1**	1**1**	0**1**	*sig*

Then, packs are refined depending on the current digit values, because nodes in a pack have signatures having the same suffix. When a new pack is created, its leader is the node having the smallest index of the nodes in the pack. The identification of packs and leaders is performed after the position step of the counting sort.

For the previous example if we have one worker then the initial pack is split into two packs when considering the least significant digit. Node 2 is the leader of the new pack.

0	0	0	0	0	2	2	2	2	2	ldr
0	1	4	5	6	2	3	7	8	9	V
10	**00**	**00**	**10**	**10**	**11**	**11**	**01**	**11**	**01**	*sig*

We precisely define when a pack is created for the iteration r. Pack are defined by their leader. Node u_0 is always a leader. Consider $p > 0$. First, we assume that u_{p-1} has already been set. Node u_p is a leader iff u_{p-1} was in the same pack at the previous iteration and the r^{th} digit of the signature of u_p and u_{p-1} are different, or u_{p-1} was not in the same pack at the previous iteration.

For the second iteration of the previous example we obtain the following result:

1	1	7	7	0	0	0	2	2	2	ldr
1	4	7	9	0	5	6	2	3	8	V
00	00	01	01	10	10	10	11	11	11	*sig*

Nodes having the same leader at the end must be merged.

Parallel computation of packs and leaders. When workers are introduced, a problem arises because when u_p is set, it is possible that u_{p-1} has not been set. After the positioning step of the counting sort, each worker will define the leader of its part by checking whether each element is a leader or not. Then, a problem arises for consecutive nodes that are handled by different workers, known as junction.

For instance in the previous example, after the first counting sort, nodes are ordered as follows: $0, 1, 4, 5, 6, 2, 3, 7, 8, 9$. Then, worker a considers the five first nodes, and worker b the five last nodes. That is, a deals with $0, 1, 4, 5, 6$. Since all signatures have the same first digit (0), then 0 is the leader of this group. In addition node at position 0 is always a leader so 0 is a global leader. Worker b deals with $2, 3, 7, 8, 9$. Since all signatures have the same first digit (1), then 2 is the leader of this group. However, there is not enough information to deduce that 2 is a global leader. The leadership of node 2, at position 5, will be deduced by comparing the previous pack of 2 and the current digit of its signature of 2 with the data of node at position $5 - 1 = 4$, which is not necessary available. The $w - 1$ junctions will be studied when all the workers have finished their work. The relation between nodes and leader can be maintained by using a union find data structure [Tarjan 1975]. Each

Algorithm 12 parallel reduce of an MDD.

```
paraReduce(mdd, W)
    // W is the set of workers
    for each i from n − 1 to 0 do
        V ← L[i], the set of nodes in layer i.
        δ ← max_{u∈V}(size of sig(u))
        for each r from 1 to δ while V ≠ ∅ do
            Partition V into |W| parts: V_1, ..., V_{|W|}
            parallel for w_j ∈ W do
                w_j.initUnionFind(V_j, r, V)
            parallel for w_j ∈ W do
                w_j.countingSort(V_j, r, V)
            parallel for w_j ∈ W do
                w_j.computeLeaders(V_j, r, V)
            Define leaders for junctions
            Remove from V nodes in singleton packs
        in parallel Merge in L[i] nodes u ∈ V with its leader
```

tree represents nodes of a pack. By performing merge according to decreasing indices, the depth of the tree can never be more than 2, so the time complexity of these operations is globally linear, that is in $O(1)$ per node.

At last and for reducing the time complexity in practice, if at any moment a pack contains only one element, then it is removed from the vector of nodes and ranges of indices of workers are accordingly redefined.

Algorithm. Algorithm 12 is a possible implementation. It works by layer. For a layer, the successive digits of the signatures are considered from the least significant digit. The vector of nodes is partitioned into as many parts as workers, each part having the same number of elements. Each worker w_j performs a counting sort on this set of nodes V_j and puts the result in V (Function countingSort(V_j, r, V)). Then, each worker computes the leader of its part of the nodes (Function computeLeaders(V_j, r, V)). The junctions are managed and nodes that cannot be merged are removed from V. Note that the internal data structures are managed at the beginning of each loop. Finally, nodes that remain in V are merged in $L[i]$ in parallel by partitioning V and by keeping only the leaders. If the leader of a node in V_j is not managed by w_j, then the node in V_j with leftmost index becomes a local leader and nodes in V_j are merged with it instead of the global leader. Then,

local leaders are merged to global leaders. Note that there is at most one local leader per worker, so it does not impact the time complexity, which remains the same as the one of the pREDUCE algorithms.

Parallelization difficulties. This algorithm overcomes the four difficulties of the parallelization of a sequential algorithm. The data dependencies are controlled by working by layer. There is no software lock. Workers get the information from different places, i.e. the V_i parts, and write the result in different positions and these two actions are performed separately and so the chance of false sharing is reduced. At last, the vector of nodes is always split in equal parts, thus the workload is well balanced.

6.3.3 Discussion

The time complexity of the parallel reduction algorithm is the same as the parallel radix sort involving w workers. It is in $O(\delta(q/w + k/w))$ (1) where q is the number of elements and k the greatest possible value and δ the number of digits. If $q = O(k)$ or if k is clearly smaller than q then the complexity is in $O(\delta q/w)$ (2).

If k is clearly greater than q, then there are two ways to reduce the complexity. First, a different algorithm [Perez 2015a] can be used. Indeed, the counting sort can be relaxed because the reduction algorithm does not require to sort the elements. Instead it searches to identify elements having the same digits. Thus, we can use an algorithm similar to the counting sort whose complexity is based only on the number of values, and not on their range, i.e. $[0..k]$. This algorithm uses A, an array of values ranging from $[0..k]$ initially at zero. Like the counting sort, it traverses the elements to count the number of times a value appears. However at the same time it builds the list of taken values and it uses this list to define the position instead of the range $[0..k]$. In this way, the ordering between the values is lost, but the algorithm still groups together the elements having the same value, without traversing the range $[0..k]$. So it can be used for our purpose. Unfortunately this algorithm is quite complex to parallelize because it requires to use a local queue per worker and the compare-and-swap instruction to be correct and efficient.

Second, the same algorithm is used but the number of digits of the signature is increased. If k is greater than q then we can reduce its size by splitting the number k into several digits. The number k can be written with $\log_m(k)$ digits in base m. So, we can express the overall time complexity (1) by $\log_m(k)O(\delta(q/w + m/w))$. By using $m = 256$ we have $\log_{256}(k)O(\delta(q/w + 256/w)) \leq 4O(\delta(q/w + 256/w))$ for $k \leq 2^{32}$. This complexity is equals to $4O(\delta q/w)$ if $q \geq 256$. So, in practice (i.e. $k \leq 2^{32}$), by

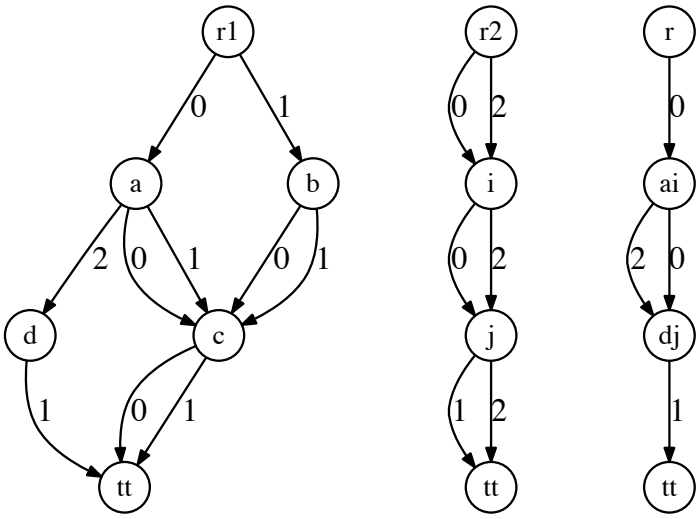


Figure 6.1: Intersection of two MDDs.

writing k in base 256 we multiply the initial complexity (2) by at most a factor of 4.

6.4 Parallel Apply

In chapter 5, we have introduced an efficient APPLY algorithm in order to define operations between MDDs. From the MDDs mdd_1 and mdd_2 it computes $mdd_r = mdd_1 \oplus mdd_2$, where $\oplus$ is union, intersection, difference, symmetric difference, complementary of union and complementary of intersection.

The algorithm proceeds by associating nodes of the two MDDs. Each node u of the resulting MDD is associated with a node u_1 of the first MDD and a node u_2 of the second MDD. So, each node of the resulting MDD can be represented either by an index, or by a pair (u_1, u_2). First, the root is created from the two roots. Then, layers are successively built. From the nodes of layer $i-1$, nodes of layer i are built as follows. For each node $u = (u_1, u_2)$ of layer $i-1$, arcs outgoing from nodes u_1 and u_2 and labeled by the same value l are considered. Note that there is only one arc leaving a node u with a given label. Thus, there are four possibilities depending on whether there are v_1 and v_2 such that (u_1, l, v_1) and (u_2, l, v_2) exist or not. The action that is performed for each of these possibilities defines the operation that is performed for the given layer. For instance, a union is defined by creating a node $v = (v_1, v_2)$ and an arc (u, l, v) each time one of the arcs (u_1, l, v_1) or (u_2, l, v_2) exists. An

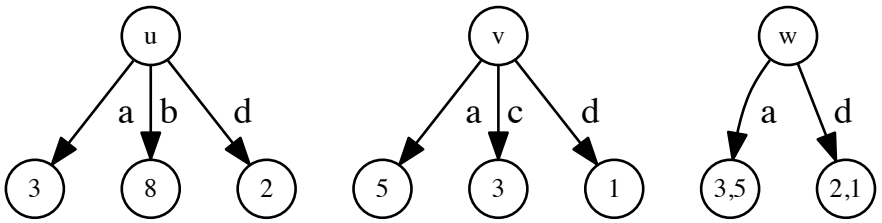


Figure 6.2: Intersection of two nodes

intersection is defined by creating a node $v = (v_1, v_2)$ and an arc (u, l, v) when both arcs (u_1, l, v_1) and (u_2, l, v_2) exist. Figure 6.2 gives an example of the intersection of two nodes. Thus, these operations can be simply defined by expressing the condition for creating a node and an arc. . We assume that Function BUILDARCS&NODES implements this mechanism and returns the array of created nodes with its length.

After each layer, the algorithm merges equivalent ordered pairs (x_1, x_2), because a lot of them can be created. For instance, consider a node u_1 of the first MDD at layer i with an arc (u_1, l, v_1) and v_2 a node of the second MDD at layer $i + 1$. Then, every arc of the second MDD labeled by l and reaching v_2 will provoke the creation of the ordered pair (v_1, v_2). Function MERGEORDEREDPAIRS is in charge of this task. At last, the computed MDD is reduced.

Parallelization. Function BUILDARCS&NODES can be easily parallelized by splitting the nodes of the layer according to the number of workers that are involved. The returned arrays must be merged into one an array of created nodes. This can be done in parallel, by using the length of each array to distribute the workload among the workers. Since, an ordered pair of node can be seen as a signature containing two digits, algorithm PARAREDUCE can be used for implementing Function MERGEORDEREDPAIRS.

Algorithm. Algorithm 13 is a possible implementation of the parallel version of APPLY.

Complexity. Let mdd_1 be the first MDD, mdd_2 be the second, n_1 (resp. n_2) be the number of nodes of mdd_1 (resp. mdd_2), and d be the maximum

Algorithm 13 Parallel Apply.

```
APPLY(mdd_1, mdd_2, op, W): MDD
    // W is the set of workers
    // We assume that each node can access its signature
    Define mdd s.t. L[i] is the set of nodes in layer i.
    root ← CREATENODE(root(mdd_1), root(mdd_2))
    L[0] ← {root}
    for each i ∈ 1..n do
        Partition L[i − 1] into |W| parts: V_1, ..., V_|W|
        L[i] ← ∅
        parallel for w_j ∈ W do
            arr_j ← w_j.BUILDARCS&NODES(V_j, op)
        Build in parallel L[i] from arr_1, ..., arr_|W|
        parallel for w_j ∈ W do
            w_j.MERGEORDEREDPAIRS(L[i])
    PARAREDUCE(mdd, W)
    return mdd
```

number of labels of a layer. For any layer, for each node of mdd_1 of this layer and for each node of mdd_2 of this layer, a node may be built. In addition, this created node may have d outgoing arcs. Thus, the complexity of the sequential APPLY algorithm is in $O(n_1 n_2 d)$. The time complexity of the parallel version of APPLY can be divided by the number of workers.

Parallelization difficulties. The data dependencies are controlled by working by layer. There is no explicit software lock, but objects are created therefore it is important to manage the memory per worker and independently from the others. Some false sharings have been observed (See Experiments) because the array of nodes per layer is shared. It was solved by postponing as much as possible the access to common cache [**?**].

6.5 Experiments

Global settings All the experiments have been made on a Dell machine having four E7- 4870 Intel processors, each having 10 cores with 256 GB of memory, 16 memory channels and running under Scientific Linux. Each tested combination is followed by a reduction.

Real instances First, we have run the instances from the application part. The first one named MaxOrder consists of intersecting and applying a symetric difference between two very large MDDs, the result contains more than 1 million of nodes and 200 millions of arcs. The second one is named Dispersion, consists of the intersection of three MDDs representing different sum functions. Curves from Figure 6.3 show the speed-up, i.e. ratio between parallel and sequential runtimes, for these two problems. As we can see, for these instances, the speed-up is significant. Thanks to parallel algorithms, we are able to close previously hard problems in seconds.

Random instances We have also tested many random MDDs generated by fixing a lower and upper bound on the number of nodes on each layer and then building the outgoing arcs with respect to some probabilities. This allow to handle the different sizes of MDDs, like small ones with high density, big ones with low density, etc.

We have run instances with a high arc density (> 0.60). The number of variables does not have an impact on the results. Figure 6.3 shows the application of the parallel intersection between two MDDs. The number of nodes by layer varies from 20k to 100k and three main curves are presented:

- For 50k-90k and 75k-100k, the speed-up is a straight line from 1 to 16 cores with 1 as coefficient, which corresponds to the number of available memory channels. Then, it is a line with 0.65 as coefficient. Note that this comes from the fact that the resulting MDDs are bigger than 70 GB and so the memory channels are saturated.

- For 20k-50k, the speed-up ratio is lower, because the amount of work is low and thus the time is hardly reducible.

Thanks to these speed-ups, we are able to build MDDs representing very large constraints in seconds or minutes while hours was required before. This implies that we can reinforce our model by intersecting constraints as a preprocess before running the search for solutions.

Remarks. The sequential algorithm is less than 10% more efficient than the parallel version running with one worker. When some operations are done in less than one second, our parallel algorithms are slower. An explanation can be the time needed to create the required memory.

False Sharing. The blue line shows the False sharing problem we had, as we can see, the algorithm scales until eight workers, then fails to keep a good speed-up, and finally loses efficiency with the growing number of workers.

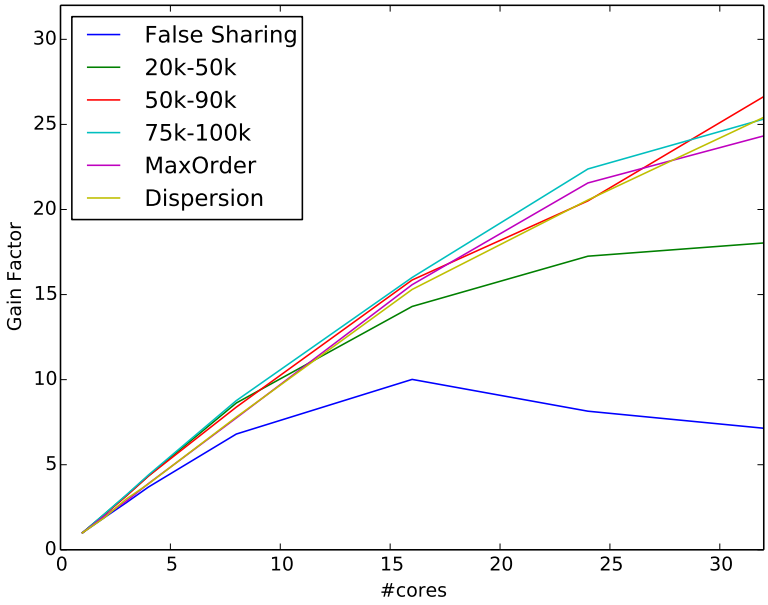


Figure 6.3: Gain factor value compared to the number of workers.

Dedicated algorithm or more digits? Figure 6.4 compares the complex algorithm (See Discussion in Section Parallel Reduction) and the method of the augmentation of digits for the parallel reduction when the values are very large. The dedicated algorithm outperforms the second method by at most a factor of 2.5.

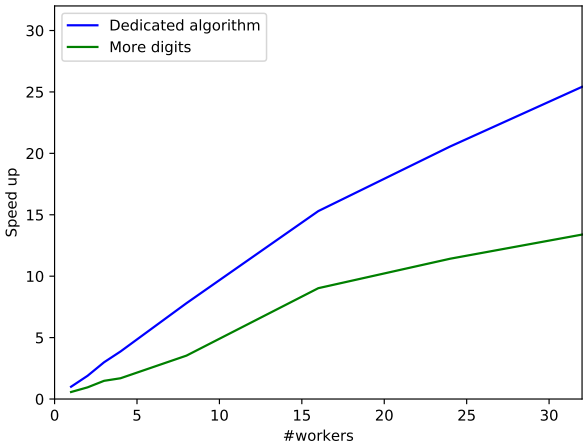


Figure 6.4: Very large values management.

Laptop On a simple Macbook pro 2013 using four cores, we observe the following speed-up:

40 variables	20 values	40 values
5k-100k nodes/layer	3.78	3.37
10k-200k nodes/layer	3.67	3.84

Remark: With some operation that are done in less than one second, our parallel algorithms are slower. An explanation can be the time needed to create the required memory.

6.6 Conclusion

A parallel version of the reduction and the apply algorithms for MDDs have been presented. These algorithms do not need any complex data structure and are simple to implement. They overcome the common difficulties encountered when parallelizing a sequential algorithm. Experimental results show that they accelerate the sequential algorithms by a linear factor according to the number of involved workers.

Chapter 7

Non-deterministic operation

Contents

7.1 Introduction

Non deterministic Decision Diagrams are studied since the 90s, and in several ways. From the simplest form, the NOBDD for Non deterministic Ordered Binary Decision Diagrams [Finkbeiner 2001], to several restricted versions exists (k-OBDD, k-IBDD, k-PBDD...) [Bollig 1999].

We consider here Non deterministic Ordered Multi-valued Decision Diagram, the notation NMDD will be used. An NMDD is a rooted, labeled, directed, acyclic graph, where each node can have an arbitrary number of outgoing arcs. The main differences with classic MDDs is that NMDDs can have several outgoing arcs emanating from a node and labeled by the same value.

A tuple is valid in a NMDD if it exists at least one valid path in the NMDD, from the root to the valid *tt* terminal node, labeled by the tuple. Figure **??** gives an example of NMDD.

The choice of this definition is simple, in constraint programming, propagators like MDD4R or MDD_w can already handle such NMDDs without modification. Thus we can directl apply all the existing CP solvers containing one of these propagators.

Moreover, NMDD can be smaller than MDD for representing the same tuples. Consider the MDD from Figure 7.2, this MDD has 7 arcs, but an NMDD having 6 arcs represents the same tuples.

Automata Finding the minimal NFA (Non deterministic Finite Automaton) starting from a DFA is PSPACE-complete [Jiang 1993], the minimum

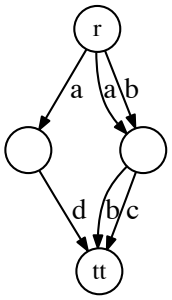


Figure 7.1: A NMDD. The root node has two outgoing arc labeled by a.

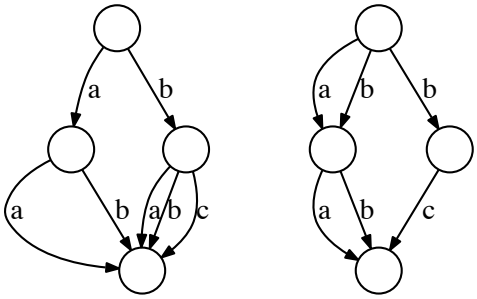


Figure 7.2: On the left, an MDD having 7 arcs. On the right, an NMDD having 6 arcs and representing the same tuples as the MDD on the left.

union or intersection of two NFA is NP-complete [Jiang 1993]. Thus existing algorithms focus on restricted version of the determinization, for example the negative-normal RONBDDs accepts only one arc labeled by 0 by node in order to be able to perform operations.

Some works focus on defining "good" NFA for a given set of words [Sgarbas 2001], by successively add the words into the NFA and randomly creating new paths instead of following the existing ones. Such work can directly be applied to MDD and thus integrated into CP solvers.

Motivations Motivations of this chapter are twofold. First a propagator defined in chapter 11 needs to intersect an MDD with a NMDD defined by mapping a cost function over the arcs of an MDD.

Second, NFAs can gain a exponential factor in size over DFAs. Using MDDs, union of GCSs is exponential on the number of GCS. Using NMDDs, by simply applying non determinism to the root, then the space required becomes linear. Figure 7.3 shows an example. This implies that NMDDs can gain an exponential factor over MDDs too. That's some of the reason of

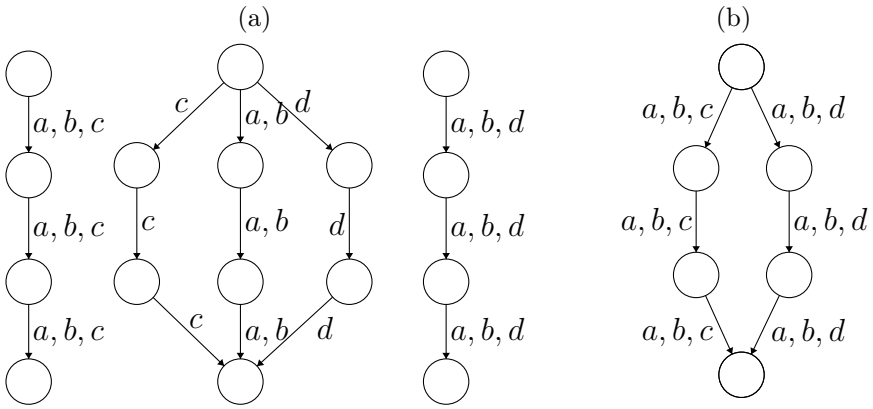


Figure 7.3: Union of the two GCSs $(\{a,b,c\},\{a,b,c\},\{a,b,c\})$ and $(\{a,b,d\},\{a,b,d\},\{a,b,d\})$. (a) the result is an MDD. (b) the result is a NMDD.

why researchers, even in CP [Cheng 2012], want to be able to deal with such MDDs.

Finally, even if NMDDs have many advantages, they also have same drawbacks. The negation of an NMDD is an NP-Hard task, while it is trivial for classical MDD.

Plan This section proposes to work directly with non deterministic MDDs. Thus an apply operator for non deterministic MDDs that output a non deterministic MDD is proposed. Then an apply operator for non deterministic MDDs that output a deterministic MDD is proposed. Both of these operation came with some theoretical insight.

7.2 Apply for Non Deterministic

We want to build a NMDD resulting from an operation between two NMDDs. An easy method is to consider all the combination of arcs with same label. If a node has 3 arcs labeled by a and the other has 2 arcs labeled by a, then we are going to try 3*2=6 combinations. Starting from the Apply defined in chapter 5, we can define an operator for the non deterministic MDDs.

This algorithm build the couple associated to all the combination of arcs having the same label. For each arc labeled by a value in the first node, the algorithm need to check each arc labeled by the value in the other node. Consider a couple of node $w = (u, v)$, with u a node from the first MDD and v a node from the second MDD. If u has 3 arcs labeled by a and v has 2 arc

labeled by a, then w is going to build $2 * 3 = 6$ arcs labeled by a.

The Algorithm 14 is a possible implementation of this method.

Remark: Non deterministic MDDs are not that easy to handle, thus this operator can only do operations that do not contain complementation, like the difference [Amilhastre 2014]. This can be easily seen when we have two equivalent sub-paths leading to two distinct nodes. One can create an arc directed to the accepting path while the label of this arc can be used in the other path.

Complexity The number of couple of nodes that can be processed is bound by the product of the nodes from both MDDs. A finer grain complexity can be bound by the sum of the product of the nodes of the same layer in both MDDs. Let M_1 and M_2 be two MDDs, the complexity of this operation is $O(|M_1| * |M_2| * N)$ with N the complexity of processing a couple of nodes. The complexity of processing a node is bound by the product of the outgoing arcs. Let $|\omega_1|$ (resp. $|\omega_2|$) be the maximum number of outgoing arcs from M_1 (resp M_2), $N = |\omega_1| * |\omega_2|$.

Reduction The reduction of Non deterministic MDDs differs since several arc out of a node can be labeled by the same value. Thus before comparing the nodes we need to ensure that their signature is the same. This can be easily done by first sorting by the label and then by the terminating node. Note that the reduction operation does not output a canonical representation anymore.

7.3 Apply for Deterministic

This section proposes to directly determinize the MDD while building it. To do so, we will group all the terminating extremities of the arcs of a node labeled by the same value, during the algorithm. Thus the apply operator consider now couple of set of nodes instead of couple of nodes.

Outgoing arcs In order to build the outgoing arcs, we can iterate over the arcs of the nodes of each set using the same method as for the classical apply. Thanks to that the complexity over each node of the set is bound by its number of arc.

Complexity The maximum number of couple of set of nodes possible is a product of power set from the nodes of both MDDs. Given two NMDDs M_1 and M_2, the complexity is bounded by $O(2^{M_1} * 2^{M_2} * d * (|M_1| + |M_2|)$. This

power set can be define by a sum for each layer since a set of nodes contains only nodes of a same layer. Chapter 8 proposes a method for handling this complexity by relaxing the operation.

Remark The power set complexity arises the fact that the determinization of an NFA contains a power set [Rabin 1959].

Algorithm 14 Generic Non Deterministe All to All Apply Function.

```
NDApply(mdd_1, mdd_2, op, V): MDD   // L[i] is the set of nodes in layer i.
  root ← createNode(root(mdd_1), root(mdd_2))
  L[0] ← {root}
  for each i ∈ 1..r do
      L[i] ← ∅
      for each node x ∈ L[i − 1] do
          get x_1 and x_2 from x = (x_1, x_2)
          if V[i] = nil then
              V[i] ← values(ω+(x_1) ∪ ω+(x_2))
          for each v ∈ V[i] do
              if ∄(x_1, v, y_1) ∈ ω+(x_1) then
                  if ∄(x_2, v, y_2) ∈ ω+(x_2) ∧ op[0] then
                      addArcAndNode(L, i, x, v, ff)
                  else
                      for each e_2 = (x_2, v, y_2) ∈ ω+(x_2) do
                          if op[1] then addArcAndNode(L, i, x, v, ff, y_2)
              else
                  for each e_1 = (x_1, v, y_1) ∈ ω+(x_1) do
                      if ∄(x_2, v, y_2) ∈ ω+(x_2) ∧ op[2] then
                          addArcAndNode(L, i, x, v, y_1, ff)
                      else
                          for each e_2 = (x_2, v, y_2) ∈ ω+(x_2) do
                              if op[3] then addArcAndNode(L, i, x, v, y_1, y_2)
  merge all nodes of L[r] into t
  pReduce(L)
  return root
```

Algorithm 15 Generic Non Deterministic All to One Apply.

```
NDApply(mdd_1, mdd_2, op, V, Width): MDD   // L[i] is the set of nodes in layer i.
  root ← createNode({root(mdd_1)}, {root(mdd_2)})
  L[0] ← {root}
  for each i ∈ 1..r do
    L[i] ← ∅
    for each node x ∈ L[i − 1] do
      get s_1 and s_2 from x = (s_1, s_2)
      E_1 ← ∪_{x_i ∈ s_1} ω^+(x_1)
      E_2 ← ∪_{x_2 ∈ s_2} ω^+(x_2)
      if V[i] = nil then
        V[i] ← values(ω^+(X_1) ∪ ω^+(X_2))
      for each v ∈ V[i] do
        E_1^v ← |(∗, v, ∗) ∈ E_1|
        E_2^v ← |(∗, v, ∗) ∈ E_2|
        X_1^v ← |y ∈ (∗, v, y) ∈ E_1|
        X_2^v ← |y ∈ (∗, v, y) ∈ E_2|
        if E_1^v = ∅ then
          if E_2^v = ∅ ∧ op[0] then
            manageWildcardPath(i, w)
            createArc(L, i, x, v, w[i])
          if E_2^v ≠ ∅ ∧ op[1] then
            addArcAndNodeSet(L, i, x, v, {}, X_2^v)
        else
          if E_2^v = ∅ ∧ op[2] then
            addArcAndNodeSet(L, i, x, v, X_1^v, {})
          if E_2^v ≠ ∅ ∧ op[3] then
            addArcAndNodeSet(L, i, x, v, X_1^v, X_2^v)
  merge all nodes of L[r] into t
  pReduce(L)
  return root
addArcAndNodeSet(L, i, x, s_1, v, s_2)
  if ∄y ∈ L[i] s.t. y = (s_1, s_2) then
    y ← createNode(s_1, s_2)
    add y to L[i]
  createArc(L, i, x, v, y)
```

Chapter 8

Relaxations

Contents

8.1 Introduction

MDDs are more and more used in optimization [Andersen 2007, Hadzic 2008]. One of the fundamental reasons is their compression efficiency. For example in a musical scheduling problem (chapter 18), an MDD having 14.000 nodes and 600.000 arcs represents 10^{90} combinations of more than 100 variables.

An advantage of MDDs is that their creation does not necessarily need the enumeration of all their solutions. In contrary with trees, where each leaf represent a solution, an MDD can have much less nodes and arcs than solutions because the paths store the combinations. Usually MDDs can be built from dynamic programming models and often allow the resolution of pseudo-polynomial problems.

The three important operations for MDDs are the creation, the reduction, which ensures that the MDD is as small as possible and canonical, and the

combination of MDDs. Several works focus on these three operations (see Part I), because they are fundamental for building advanced MDD-based models.

Unfortunately sometimes MDDs are too big to fit in memory, a solution is to relax these MDDs [Hadzic 2008, Bergman 2011]. A relaxed MDD is an MDD representing a super set of the solution of the exact (non relaxed) MDD. The requirement is that these relaxed MDDs have to be smaller than their exact version. Others objectives of a good relaxation are the quality of the obtained MDD, the amount of non-solution or in optimization the distance to the optimal solution.

In this chapter, we study the 3 fundamental operations on relaxed MDDs:

Relaxed Creation. Existing works focus on trying to relax the creation of the MDD, as said in the Cire's PhD thesis [Cire 2014b], the relaxed DDs can be associated to state-space relaxation [Christofides 1981]. These methods build an MDD from specific problem by relaxing the problem during the creation of the MDD. In this paper, we provide an improvement of the existing creation methods. Also, while the relaxation of the MDD creation need the definition of problem specific functions, we provide in this chapter several generic selection functions that can be used for many different problems.

Relaxed Reduction. We introduce the relaxation of the reduction operation, by relaxing the equivalence function of the reduction and giving a generic method for merging nodes in existing MDDs.

Relaxed Combination. We study the relaxation of the combination of MDDs and we give two algorithms with different complexities and properties. One with a quadratic complexity, and another one with a $k*(m_1+m_2)$ complexity.

Finally, the creation of an MDD-based model for solving a problem is rarely easy, the user has to make choices during this process, and these choices can drastically change the memory requirement and the quality of the obtained model. We describe at the end of this chapter different schemes for MDD-based models building.

Relaxed MDDs definition The MDD of a constraint can be too large to be represented in computer memory. In order to remedy to this drawback, several works [Hadzic 2008, Bergman 2011] proposed to use a relaxation of the MDD. The condition of this relaxation is that the set of solutions of this relaxed MDD must be a super set $\hat{S}$ of the solutions S of the exact MDD, a super set implies that $S \subseteq \hat{S}$. Thus this MDD must contain all the solutions of S.

8.2 Relaxed Creation : Existing Works

Node splitting One of the first algorithm building relaxed MDDs [Hadzic 2008] was define to compile a CSP into relaxed MDDs. This Algorithm, builds an MDD representing the compilation of the constraint in a top-bottom manner. During this process, it considers nodes as equivalent by measuring the distance between their partial assignment and allowing a given threshold. This distance depends on the constraint type.

The algorithm proceed as follow, first an MDD allowing a big threshold in the distance is built, then this allowed distance is reduced. This algorithm is a refinement algorithm, and proceeds a Breadth-First Search (BFS) inside the MDD and splits nodes having incoming paths that are no longer equivalent, with respect to the allowed distance. If this algorithm proceed to a refinement with a threshold of zero, then an exact MDD is obtained. Generally the algorithm stops when a maximum size is reached.

Dynamic Programming relaxation Many works relaxing the creation operation often consists of relaxing the Dynamic Programming (DP) model generating the MDD [Bergman 2011]. A dynamic programming model can be seen as a set of states and a transition function. Relaxed creation methods using DP start by trying to build the exact MDD for the DP model. In a top-bottom BFS building layer by layer the MDD, when a given maximum amount of nodes is reached, then the algorithm merges nodes of a layer by merging their states from the DP model. This merging function has to ensure that the newly created nodes generates at least all the combination starting from the merged nodes states. The state of this new node is a super-state of the states of the nodes it contains.

This method needs the definition of at least two functions in order to be used: *dp_select* and *dp_merge*. The *dp_select* function is a heuristic function who takes as argument the current layer of nodes of an MDD and return the nodes that have to be merged. The *dp_merge* heuristic function defines the procedure of state merging of the node, this heuristic has to ensure that the sub-MDD of the resulting node contains all the paths from the sub-MDD of the merged nodes. This function depends on the state representation of the problem.

Max width Several works propose to bound the width of a layer (i.e. the number of nodes of a layer). The main idea behind this method is to have upper-bounds on the maximum memory and time used during the process. This Maximum width constraint can be ensured by both the node splitting and dynamic programming creation methods.

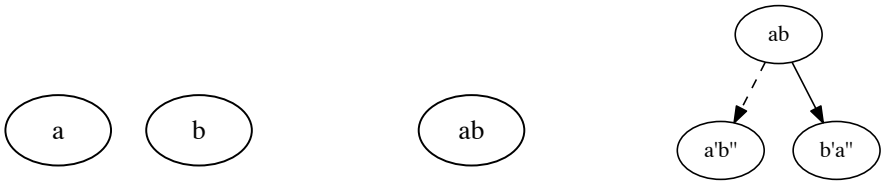


Figure 8.1: This is the classical method, the two states in left are merged, leading to the middle nodes, and then the outgoing edges are created.

8.3 Relaxed Creation : New Method

8.3.1 Delayed Relax Creation

Main idea As we mention, the Dynamic Programming relaxation performs two steps. The first step is to find nodes in the current layer using the *select* function and then merge these nodes using the *merge* function. The second step is to build the arcs directed to the next layer. See Figure 8.1.

However, in MDD theory, the merge of two nodes is performed by the reduction operation and has a different meaning. The reduction operator merges equivalent nodes by considering their outgoing arcs. This kind of algorithms perform either a BFS in a bottom-up fashion, or a DFS and merge nodes during the post-visit. In these algorithms, the outgoing arcs (label, terminating node) are considered. Two nodes are equivalent iff they have the same list of outgoing arcs.

The main idea of this new method is to consider the outgoing arcs for the *dp_select* and *dp_merge* in addition to the node state. To do so, we have to reverse the process of relaxation.

Delayed Relax Creation (DRC) During the process of a layer, the creation of the outgoing arcs is firstly performed, then if in the current layer there is more than the maximum allowed number of nodes, then the *dp_select* function chooses nodes. These nodes are merged and the *dp_merge* function is applied to their child nodes.

Advantages This delayed reduction, has an access to more information when the *dp_select* and *dp_merge* function are called. Thanks to these information, we can perform better relaxation. These advantages are:

- The *dp_select* function has access to much more information. The outgoing arcs, their labels and the state on wich they are directed. Using these information can lead to better heuristics.
- The *dp_merge* function is applied to the child nodes, this lead to a better state relaxation (See Figure 8.2).

The differences of this method is presented in Figure 8.2. We can see that the states of the child nodes are different from the previous method Figure 8.1.

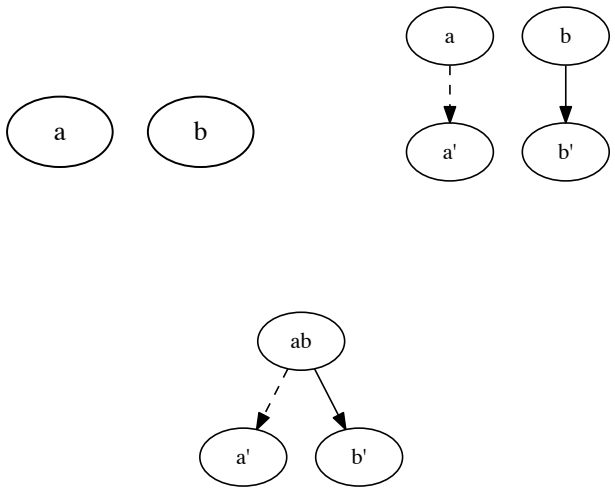


Figure 8.2: Delayed Relax Reduction method: from left to right, first the arcs are created, then the nodes are merged. We can see that no residual states are added to the nodes.

Definition 3 *A residual state is the part of a state of a node due to the merge of the parent node, which would not exists if there is no merge.*

In Fig. 8.1, the state b" is a residual state of the node with state a'.

8.3.2 Generalization

The idea of the Delayed Relax Creation is to build the layer i and then merge the nodes of the layer i-1. We can unfold this process until the wanted depth.

Definition 4 *$DRC(k)$ unfold k layers before merging the nodes.*

If k is greater or equal to the number of variables, then the merges of the nodes leads to an exact union.

8.3.3 Generic merging heuristic

While specific *dp_ select* and *dp_ merge* function are required in general for each specific constraint, some general scheme for state representation, selection and merging heuristic can be presented.

Set of values A simple state representation is often a set of values. A set of values can represent the state of many problem, for example Independent set or allDifferent etc. For this set of value, we can define as *dp_ merge* function the classic union operation for set. For the *dp_ select* function, we can use the minimum number of difference between the set.

8.3.4 States relaxation

The existing relaxation of Dynamic Programming model merge node states during the building. another method is to build an exact MDD by using a relaxed state function. For example on a relaxed version of a graph problem, or even using methods like bit scaling.

8.4 Relaxed Reduction

The reduction operation of MDDs is based on the equivalence function. This operation merges equivalent nodes until a fix point is reached. After the application of a reduction, an MDD is said to be reduced.

Let $\omega^+(u)$ be the list of outgoing edges of node u. Let $SG(n)$ be the subgraph starting from the node n. Let $|G|_p$ be the number of paths of the MDD G from the root node to the true terminal node.

Definition 5 *Two nodes u and v are equivalent, denoted by $u \equiv v$ iff:*

$$|\omega^+(u)| = |\omega^+(v)| \wedge \forall (u, w, a) \in \omega^+(u), \exists (v, w, a) \in \omega^+(v) \tag{8.1}$$

Let u, v be nodes. Let n be the node resulting from the merge of u and v:

$$u \equiv v \implies SG(u) = SG(v) = SG(n)$$

Union of nodes: We define the union of two nodes $\cup$ as the application of the union operator on the MDDs starting from these two nodes. Let u and v be nodes and n be the resulting node from the union of u and v:

$$SG(u \cup v) = SG(n)$$

Property 2 *The union of two nodes from layer i in an MDD cannot increase the number of nodes in the layers 1..i. But it can increase or decrease the number of nodes of the layer i+1..n.*

Property 3 *The union of two nodes u and v from layer i in an MDD can increase the number of paths to* `tt` *starting from the ancestor nodes of u and v.*

Relaxed equivalence $\approx$: We introduce a relaxed version of the equivalence, this relation does not necessarily come from Definition 5 anymore, and can be defined by the user. An MDD is relax reduced iff a fix point is reached for the relaxed equivalence ($\approx$).

While it was easy to merge equivalent nodes, mergin relax equivalent nodes is more complicated, because they can have arcs labeled by the same value and directed to different nodes. So we need to use the union of nodes ($\cup$) as merging method.

Relaxing an existing MDD The definition of the relaxed equivalence function will define the order of application of the union. Consider that the user want to reduce the number of nodes in each layer in a top-bottom fashion. We can apply the union operation to the nodes from layer 1 to n while the number of nodes of the considered layer is greater than the wanted amount. Since property 2 ensures that the union of nodes from layer i cannot increase the number of nodes of the layer from 1 to i, then the algorithm is valid.

We now have a simple method allowing the relaxation of exact MDDs. The relaxation of an exact MDD is an interesting question and allow us to build different kind of relaxed MDD-based models. The section 8.6 gives an application for such relaxation.

8.5 Relaxed Combination

The combination of MDDs is mainly done by the Apply operator. This operator can perform the intersection, the union, the symmetric difference of MDDs and many other operations. Several implementations exist for this operator

(see chapter 5), they are based either on a Boolean function or on the graph structure.

The main idea of these algorithms is to build the resulting MDD by building each node of the resulting MDD in function of nodes from the operand MDDs. Precisely while processing $m = m_1 \cap m_2$, the algorithm associates to each node from m a couple of nodes from m_1 and m_2.

Simple idea A simple relaxed version of the Apply algorithm is to build the resulting MDD using a DFS. When the number of nodes of the MDD is too big, then we apply a relaxed reduction to the current MDD.

The main advantages of a DFS algorithm against a BFS for this relaxation is that we can have a look at a real part of the MDD. Since, until the first relaxed merge, the information is exact, and the MDD a sub-graph of the exact one. Thanks to that we have more information about the general form of the MDD. One of the drawback of this simple relaxed Apply is that the time complexity is the same as the exact intersection.

Since this idea has a bad complexity, the next section will introduce another method with a better complexity.

8.5.1 Relax Apply

The relaxation of an MDD is based on the union operation, so we will make the union of the operands during the operation.

The classical Apply operator has usually two nodes as operands, one node from the first MDD and another one from the second MDD. The Apply operator then build the arc function of these two nodes. The goal of this relaxation is to be able to merge nodes during the process of the apply.

To do so, the relaxed Apply algorithm does not consider couple of nodes from both MDDs, but couple of sets of nodes. When to nodes of the Apply process have to be merged, then we can build a first set containing both nodes from the first MDD, and a set containing both nodes from the second MDD. When the nodes to merge are already couple of set of nodes, we make the union of these set from both MDDs.

The Apply algorithm (section 5.2, graph based apply) can be modified in this way. During the Processing, if to many couples of sets of nodes are created in the currently processed layer, then using an equivalence function (`select`), we choose couples and merge them by making the union of their corresponding sets.

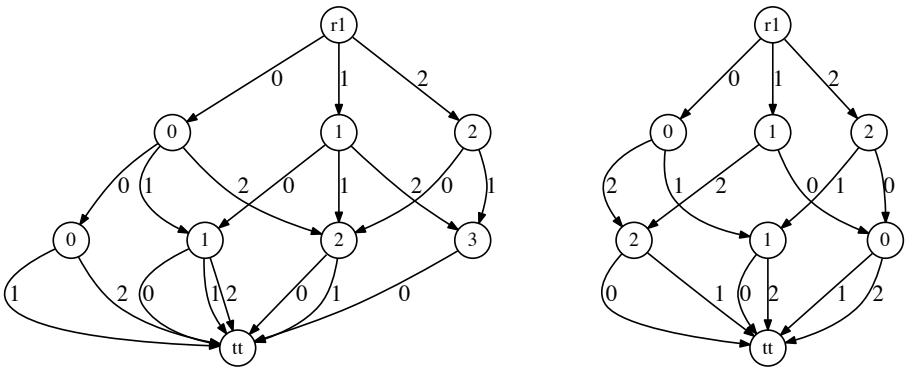


Figure 8.3: On the left an MDD representing the sum between [1,3]. On the right, an MDD representing the no two consecutive same values constraint.

Example:

Consider the two MDDs from Figure 8.3. The exact intersection is the MDD on the left of Figure 8.4.

We process the relax intersection of these two MDDs by bounding the maximum number of nodes at a layer to be lower or equal to 4. In order to perform the intersection, we build the couple (*r1*,*r2*) of nodes. Using this couples we can generate the couples (0,0) using the arcs 0, (1,1) using the arcs labeled by 1 and (2,2) using the arcs labeled by 2. We have less than four nodes, thus we can continue. From the couple (0,0), we build the couple (1:1) by following the arcs labeled by 1 and (2:2) by following the arcs labeled by two. Note that the couples (1,1) and (2,2) are at a different layer as the one previsouly defined. Using the other couples we build (1,0), (3,1), (3,2) and (2,0). We have 6 couples, thus we have to merge some of them.

The choice of the nodes to merge is very important. First, as shown in Figure 8.4, the couples (3,2) and (3,1) are equivalent, but we cannot know until we build the outgoing arcs. Moreover, the choice is often made using the cost of the current state, but we do not have any cost in this example.

Consider that you want to enforce *more* the non consecutive constraint than the sum constraint. Thus we allow to merge randomly couples having the same second nodes. Applied to our couples (1,0), (1,1), (2,0), (2,2), (3,1) and (3,2), this can lead to merging the couples (1,0) and (2,0) into

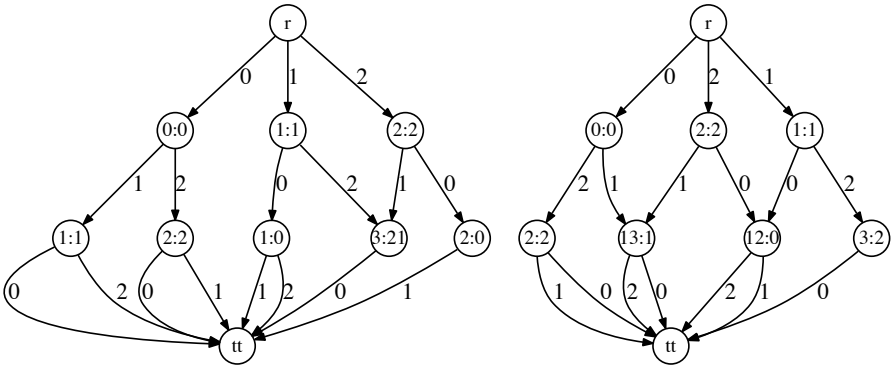


Figure 8.4: On the left, the exact intersection of the two MDDs from Figure 8.3. On the right, a relax version of the intersection.

({1,2},0). We have still 5 couples, thus we select again and choose (1,1) and (3,1), giving ({1,3},1). Our couples are now ({1,2},0), ({1,3},1), (2,2) and (3,2).

Starting from couples ({1,2},0), the possible values outgoing from nodes in {1,2} are {0,1,2} from 1 and {0,1} from 2. We perform the union of the possible label to obtain {0,1,2}. Intersecting with the node 0 from the second MDD build the arcs labeled by 1 and 2 directed to `tt`.

The global result is the right MDD in Figure 8.4. As we can see, in this MDD, no two consecutive values are the same, but the sum are no longer between 1 and 3.

The main advantage of this method is that a relaxed MDD is build on the fly, and the time complexity is less than the exact computation.Let m_1 and m_2 be the operands of the operation, if the maximum width is one, then the complexity is linear on the operands $O(|m_1| + |m_2|)$, if there is no maximum number of nodes, then time complexity is quadratic over the operands $O(|m_1| * |m_2|)$, finally if the maximum size allowed in a layer is k, then the complexity is : $O(k(|m_1| + |m_2|))$.

See Algorithm 16 for a possible implementation of this algorithm.

Remark Since the number of created nodes at layer i+1, $|L[i+1]| \leq |L[i]| * d$, then we can use the same delayed fusion for the addArcAndNode as for the usual Apply. We first build all the nodes at layer i, then merge the equivalent ones. If needed, after the exact merge, we merge relax equivalent nodes.

Another remark We can efficiently merge equivalent nodes by sorting them in the same manner as the reduction algorithm. By first indexing the couple of sets by the first node of the first set, then the second one etc... when a node is alone in an index, it cannot be merged with any other node. When two nodes are in the same cell and all the nodes of their sets have been visited, then we can merge them. This algorithm is linear.

Select The relax version of the Apply needs the definition of a relax equivalent function (`select`) which chooses the non equivalent nodes to merge. Note that this function can be used either in the fly as defined in the algorithm, or after building the whole layer and so it has all the nodes as argument and return a set of set of node where each set of nodes have to be merge.

Last `select` function. The last relax equivalent function return always the k^{th} nodes of the layer to be merge with the $> k$ nodes.

Rand`select` function. The rand function choose uniformly between the k first nodes.

Remark Obtaining a relax MDD from an existing one can be also seen as relax intersecting an MDD with itself.

Non deterministic The Apply for non deterministic MDDs presented in chapter 7 is close to this algorithm. Moreover, this algorithm can be applied to non deterministic MDDs allowing to deal with their exponential complexity. This can be done by either merging nodes, or by splitting the set into smaller one. For example, do not allowing more than k nodes in a set.

8.5.2 Experiments

Consider the following problem, we have 2 MDDs and we aim at building the relaxation of the intersection of these two MDDs.

We have tried to change both the maximum layer size and the `select` function. The Fig. 15.3 shows the results.

In the left table, the first MDD is fully included in the second one, the exact intersection has 2775 nodes and 3773 arcs. In the right table, the first MDD is not at all included in the second one, the exact intersection has 0 node and 0 arc.

First, this example shows that the way we merge nodes has a strong impact on the number of created solution.

The *last* `select` shows that we should avoid merging too much nodes on the same node, otherwise we are going to accept everything. Thus we completely lost our information.

Also, thank to this relaxed apply algorithm, we are able to build relaxed version of MDDs defined by combination of others.

Width	last	Rand
90	157515761	11822467
200	6102219	186531
400	2456238	15497
exact	1000	1000

Width	last	Rand
100	851331	0
200	630	0
400	0	0
exact	0	0

Figure 8.5: Number of solutions for the Last and Random heuristics of selection of node on small instances.

8.6 Relaxed MDDs : Use

While the previous sections presented methods for building relaxed MDDs, the final user will have to choose between several possibilities while building its own model. This section describes the different opportunities for the final user to build its own MDD-based models.

Example For modeling purpose, consider the need of performing the intersection of two MDDs. Formally we want to define:

$$M_r = M_1 \cap M_2 \tag{8.2}$$

But while performing these intersection, either the time is too long or the memory is overloaded. That's why the modeler has to relax this intersection.

From this intersection, we can decide to either relaxing the resulting MDD during its creation, to build a relaxed version of M_1 and/or M_2 and then perform an intersection while having an upper-bound of the size of the resulting MDD etc.

When to relax The relaxation of an MDD can be done at different moment, during its creation, after its creation, during the combination. When it is possible, we can build an MDD and relaxing part that are less useful than other, for example in optimization, we can relax part of the MDD where the cost is bad.

Remark: Relaxing an MDD after its creation generally implies that we want to use it for another combination.

Who to relax Choosing the MDDs to relax is important, for example in the MaxOrder problem, a problem of text generation from a corpus avoiding plagiarism, two MDDs are involves, one containing all the Markov sequences from a corpus, and the other containing all the plagiarism sequences. Relaxing the Markov MDD will allow the generation of sequences which are not define from the Markov matrix of the corpus, but this relaxation can be considered as better than generating plagiarism sequences. Such choices are made by the authors of the models.

Another reason of relaxing an operand MDD is that this MDD is too huge to fit in memory. When performing an intersection, we can either relaxing the MDD which is currently building, or we can relax one of several operands.

When and who Using the previous example, $M_r = M_1 \cap M_2$, we have 4 general choices:

- Relax M_r during its construction (relax combination).
- Relax M_r after its construction (relax reduction).
- Relax M_1 (and/or M_2) during the construction (relax construction).
- Relax M_1 (and/or M_2) after the construction (relax reduction).

The choice between these options depend on the possibility of representing these MDDs in memory. But any choice will impact on the resulting MDD.

Algorithm 16 Generic Relax Apply.

```
RelaxApply(mdd_1, mdd_2, op, V, Width): MDD   // L[i] is the set of nodes in layer i.
root ← createNode({root(mdd_1)}, {root(mdd_2)})
L[0] ← {root}
for each i ∈ 1..r do
    L[i] ← ∅
    for each node x ∈ L[i − 1] do
        get s_1 and s_2 from x = (s_1, s_2)
        E_1 ← ∪_{x_i ∈ s_1} ω^+(x_1)
        E_2 ← ∪_{x_2 ∈ s_2} ω^+(x_2)
        if V[i] = nil then
            V[i] ← values(ω^+(X_1) ∪ ω^+(X_2))
        for each v ∈ V[i] do
            E_1^v ← |(∗, v, ∗) ∈ E_1|
            E_2^v ← |(∗, v, ∗) ∈ E_2|
            X_1^v ← |y ∈ (∗, v, y) ∈ E_1|
            X_2^v ← |y ∈ (∗, v, y) ∈ E_2|
            if E_1^v = ∅ then
                if E_2^v = ∅ ∧ op[0] then
                    manageWildcardPath(i, w)
                    createArc(L, i, x, v, w[i])
                if E_2^v ≠ ∅ ∧ op[1] then
                    addArcAndNodeSet(L, i, x, v, {}, X_2^v)
            else
                if E_2^v = ∅ ∧ op[2] then
                    addArcAndNodeSet(L, i, x, v, X_1^v, {})
                if E_2^v ≠ ∅ ∧ op[3] then
                    addArcAndNodeSet(L, i, x, v, X_1^v, X_2^v)
    merge all nodes of L[r] into t
    pReduce(L)
    return root
addArcAndNodeSet(L, i, x, s_1, v, s_2)
    if ∄y ∈ L[i] s.t. y = (s_1, s_2) then
        if Width > |L[i]| then
            y ← createNode(s_1, s_2)
            add y to L[i]
        else
            y ← select(L[i])
            y.set(y.s_1 ∪ s_1, y.s_2 ∪ s_2)
    createArc(L, i, x, v, y)
```

Chapter 9

Sampling

Contents

9.1 Introduction

For solving some automatic generation problems, sampling from a knowledge data set is used to generate new data. Often, some additional control constraints must be satisfied. One approach is to generate a vast amount of sequences for little cost, and keep the satisfactory ones. However, this does not work well when constraints are complex and difficult to satisfy. Thus, some works have investigated to integrate the control constraints into the stochastic process.

For instance, in text generation, a Markov chain, which is a random process with a probability depending only on the last state (or a fixed number of them), is defined from a corpus [Jurafsky 2014, Papadopoulos 2014, Papadopoulos 2015]. In this case, a state can represent a word, and such a

process will generate sequences of words, or phrases. It can be modeled as a directed graph, encoding the dependency between the previous state and the next state. Then, a random walk, i.e. a walk in this graph where the probability for choosing each successor has been given by the Markov model, will correspond to a new phrase. Such a walk corresponds to a sampling of the solution set while respecting the probabilities given by the Markov chain. This process generates sequences imitating the statistical properties of the corpus. Then, the goal is to be able to incorporate some side constraints defining the type of phrase we would like to obtain. For example, we may want to only produce sequences of words that contains no subsequence belonging to the corpus, longer than a given threshold, in order to limit plagiarism [Papadopoulos 2014].

Such Markov models have long been used to generate music in the style of a composer [Brooks 1957, Papadopoulos 2014]. The techniques of Markov constraints have been introduced to deal precisely with the issue of generating sequences from a Markov model estimated from a corpus, that also satisfy non Markovian, user defined properties [Pachet 2011, Barbieri 2012, Roy 2013].

Hence, there is a real need for being able to sample some solutions while satisfying some other constraints.

The idea of this chapter is to represent the corpus dependencies and the additional constraints by an MDD and develop sampling algorithms dealing with the solution set represented by this MDD.

Papadopoulos *et al.* have designed an algorithm which can be applied to a `regular` constraint [Papadopoulos 2015]. However, the paper is complex because it is a direct adaption of the powerful and general belief propagation algorithm and requires the definition of a regular constraint.

This chapter proposes a simpler solution defined on a more general data structure (the MDD), which may represent any regular constraint, but also different constraints (see chapter 4. In addition, it shows how to apply it for any kind of samplings and not only on Markov samplings. Thus, instead of developing ad-hoc algorithms or forcing the use of regular constraints, we propose a more general approach that could be used for a large range of problems provided that we have enough memory for representing the MDD.

However, combining samplings and MDDs is not an easy task. Consider, for instance, that we have a very simple MDD involving only two variables x_1 and x_2 whose values are a and b and that it represents the three solutions $S = \{((x_1, a), (x_2, a)),\ ((x_1, a), (x_2, b)), ((x_1, b), (x_2, b))\}$ (Fig. 9.1). Assume that we want to sample uniformly the solution set. In other words, we want to randomly select one solution with an equal probability for each solution. This can easily be done by randomly selecting a solution in S. Since there are 3 solutions, any solution has a probability of $1/3$ to be selected. The issue

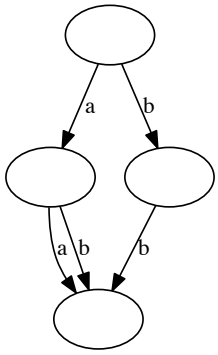


Figure 9.1: A simple MDD.

with MDDs is that they compress the solution set, so picking a solution with a uniform probability is not straightforward. For instance, if we randomly select the first value of the first variable and if we randomly select the value of the second variable then the selection is not uniform, because we are going to select more often the solution $((x_1, b), (x_2, b))$ than the others. This problem can be solved by computing the local probabilities of selecting a value according to the probabilities of the solutions containing that value.

Furthermore, this chapter studies the case where the probabilities of values are not the same and considers Markov sampling, that is sampling where instead of considering the probability of selecting one value, we consider the probability of selecting a sequence of values.

Plan This chapter is organized as follows. First it recalls some definitions about probability distribution and Markov chain. Then, it proposes some algorithms for sampling the solution set of an MDD while respecting the probabilities given by a distribution, that can be a probability mass function or a Markov chain. Finally, it presents some experiments on the geomodelling of a petroleum reservoir, on the generation of French alexandrines based on the famous La Fontaine's fables and on the generation of huge random number.

9.2 Definitions

9.2.1 Probability distribution

We consider that the probability distribution is given by a probability mass function (PMF), which is a probability density function for a discrete random variable. The PMF gives for each value v, the probability $P(v)$ that v is taken:

Given a discrete random variable Y taking values in $Y = \{v_1, ...v_m\}$ its probability mass function P: $Y \rightarrow [0, 1]$ is defined as $P(v_i) = Pr[Y = v_i]$ and satisfies the following condition: $P(v_i) \geq 0$ and $\sum_{i=1}^{m} P(v_i) = 1$.

Property 4 *Let f_P be a PMF and consider $\{x_i\}$ a set of n discrete integer variables independent from a probabilistic point of view and associated with f_P that specifies probabilities for their values. Then, the probability of an assignment of all the variables (i.e. a tuple) is equal to the product of the probabilities of the assigned values. That is $\forall i = 1..n$, $\forall a_i \in D(x_i)$ $P(a_1, a_2, ..., a_n) = P(a_1)P(a_2)...P(a_n)$.*

9.2.2 Markov chain

A Markov chain[1] is a stochastic process, where the probability for state X_i, a random variable, depends only on the last state X_{i-1}. A Markov chain produces sequence $X_1, ..., X_n$ with a probability $P(X_1)P(X_2|X_1)...P(X_n|X_{n-1})$.

Property 5 *Let P_M be a Markov chain and consider a set of n discrete integer variables associated with P_M that specifies probabilities for their values. Then, $\forall i = 1..n$, $\forall a_i \in D(x_i)$ $P(a_1, a_2, ..., a_n) = P(a_1)P(a_2|a_1)...P(a_n|a_{n-1})$.*

Several methods can be used to estimate the Markov chain from a corpus, like the maximum likehood estimation [Jurafsky 2014]. This work is independent from such methods and considers that the Markov chain is given.

Sampling a Markov chain can be simply and efficiently done by a random walk (i.e. a path consisting of a succession of random steps) driven by the distribution of the Markov chain. If we need to build a finite sequence of length k, then we perform a random walk of k iterations using the given distribution.

Example:

Consider M, the Markov chain in Fig. 9.2 and an initial probability of 0.6 for a and 0.4 for b. If we apply M on two variables x_1 and x_2, then the probability of the tuple (a, a) is $P(x_1, a)P((x_2, a)|(x_1, a)) = 0.6 \times 0.9 = 0.54$. The probabilities of the four possible tuples are given in Fig. 9.2. The sum of the probabilities is equal to 1.

[1] Order k Markov chains have a longer memory: the Markov property states that $P(X_i|X_1, ..., X_{i-1}) = P(X_i|X_{i-k}, ..., X_{i-1})$. They are equivalent to order 1 Markov chains on an alphabet composed of k-grams, and therefore we assume only order 1 Markov chains.[Papadopoulos 2015]

\	a	b
a	0.9	0.1
b	0.1	0.9

Tuple	Probability
aa	0.54
ab	0.06
ba	0.04
bb	0.36

Figure 9.2: Markov chain for two variables. The starting probabilities are 0.6 for a and 0.4 for b.

9.3 Sampling and MDD

This section aims at sampling the solution set of an MDD while respecting the probabilities given by a distribution, that can be a PMF or a Markov chain.

Let M be an MDD whose n variables are associated with a distribution that specifies the probabilities of their values. For sampling the solutions of M, we associate with each arc a probability, such that a simple random walk from the root node to tt according to these probabilities will sample the solution set of M while respecting the probabilities of the distribution of M.

First, we consider that the distribution of M is given by a PMF and that the variables of M are independent from a statistical point of view. Then, we will consider that we have a Markov chain for determining the probability of a value to be selected.

9.3.1 PMF and Independent variables

If the distribution associated with M is defined by a PMF f_P and if the variables of M are independent from a statistical point of view, then we associate with each arc e a probability $P(e)$. From Property 12 we know that the probability of a solution $(a_1, ..., a_n)$ must be equal to $\Pi_{i=1}^{n} P(a_i)$.

We could be tempted to define $P(e)$ as the value of $f_P(label(e))$ where $label(e)$ is the label (i.e. value) associated with e. However, this is not exact because the MDD usually does not contain all possible combinations of values as solutions. For instance, consider the example of Fig. 9.1 with a uniform distribution. If all probabilities are equivalent then each solution must be able to be selected with the same probability, which is 1/3 since there are three solutions (a, a), (a, b) and (b, b). Now, if we do a random walk considering that the probability of each arc is 1/2 then we will choose with a probability 1/2 the solution (b, b) which is incorrect. The problem stems from the fact that the probabilities of the higher layers are not determined according to the probabilities of solutions that they can reach while it should be the case. The

choice (x_1, a) allows to reach 2 solutions and (x_1, b) one. So, with a uniform distribution the probability of choosing a for x_1 should be $2/3$ while that of choosing b should be $1/3$.

Definition 6 *The partial solutions that can be reached from a node n in an MDD are defined by the paths from n to tt.*

In order to compute the correct values, we compute for each node n the sum of the original probabilities of the partial solutions that we can reach from n. Then, we renormalize these values in order to have these sums equal to 1 for each node. For instance, for the node reached by traversing the first arc labeled by a in Fig. 9.1, the sum of the original probabilities is $1/2 + 1/2 = 1$, so the original probabilities are still valid. However, for the node reached by traversing the arc from the root and labeled by b, the sum of the original probabilities is $1/2$, so half of the combinations are lost. This probability is no longer valid and new values must be computed.

The sum of the original probabilities of the partial solutions that can be reached from a node is defined as follows:

Property 6 *Let M be an MDD defined on X and f_P a PMF associated with M. Let n be any node of the MDD and A be any partial instantiation of X reaching node n. The sum of the original probabilities of the partial solutions that can be reached from n is $v(n) = \sum_{s \in S(n)} P(s|A)$, where $S(n)$ is the set of partial solutions that we can reach from n and $P(s|A)$ is the probability of s under condition A. The probability of any arc $e = (n', n, a)$ is defined by $P(e) = f_P(a) \times v(n)$.*

proof: By induction from tt. Assume this is true at layer $i+1$. Let n' be a node of layer i, n a node in layer $i+1$ and $e = (n', n, a)$ an arc. We have $P(e) = f_P(a) \times v(n)$, that is $P(e) = f_P(a) \times \sum_{s \in S(n)} P(s|A)$, where A is any partial instantiation reaching node n. So for node n' we have:
$v(n') = \sum_{e \in \omega^+(n')} P(e)$, where $\omega^+(n')$ is the set of outgoing arcs of n'
$v(n') = \sum_{e \in \omega^+(n')} f_P(label(e)) \times \sum_{s \in S(n)} P(s|A)$. Note that A is any partial instantiation reaching node n, so it can go through e. So we have
$v(n') = \sum_{s \in S(n')} P(s|A')$ where A' is any partial instantiation reaching node n'. □

The correct probabilities can be computed by a bottom-up algorithm and a top-down algorithm. First, we consider the second to last layer and we define the probability P of an arc labeled by a as $f_P(a)$. Then, we directly apply Property 6 from the bottom of the MDD to the top: once the layer $i+1$ is determined, we compute for each node n' of the layer i the value

$v(n') = \sum_{e \in \omega^+(n')} P(e) = \sum_{e \in \omega^+(n')} f_P(label(e)) \times v(n)$. Once the bottom-up part is finished, we normalize the computed values P in order to have $v(n) = 1$ for each node n. We use a simple top-down procedure for computing these values.

Example:

Fig. 9.3 details this process. The left graph simply contains the probability of the arc labels. The middle graph shows the bottom-up procedure. For instance, we can see that the right arc outgoing from the source has a probability equal to $1/2 \times 1/2 = 1/4$. Thus a normalization is needed for the root because the sum of the probabilities of the outgoing arcs is $1/2 + 1/4 = 3/4 < 1$. The right graph is obtained after normalization.

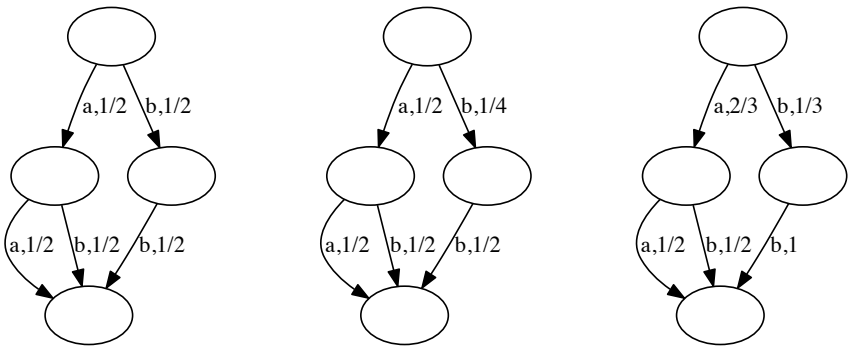


Figure 9.3: Sampling from a simple MDD. The probability of a and b are $1/2$.

Note that the normalization consists of computing the probability according to the sum of the probabilities. If $P(e)$ is the current value for the arc $e = (u, v, a)$ and T is the sum of the probability of the outgoing arcs from u, then the probability of e becomes $P(e)/T$.

This step can be avoided in practice by computing such normalized values only when needed.

Algorithm computeMDDProbabilities can be described as follows:

1. Set $v(tt) = 1$; For each node $v \neq tt$, in a Breadth First Search (BFS) in bottom-up fashion:
   (a) Compute $v(n)$ the sum of the original probabilities of the outgoing arcs of n.
   (b) Define the probability of each incoming arc e of n labeled by a as $P(e) = f_P(a) \times v(n)$

2. For each node in a BFS top-down fashion, normalize the probabilities of the outgoing arcs.

During this algorithm, each sum is calculated once for each node during the bottom up processing, and the normalization is performed once for each arc. The final complexity is $O(|E| + |V|)$.

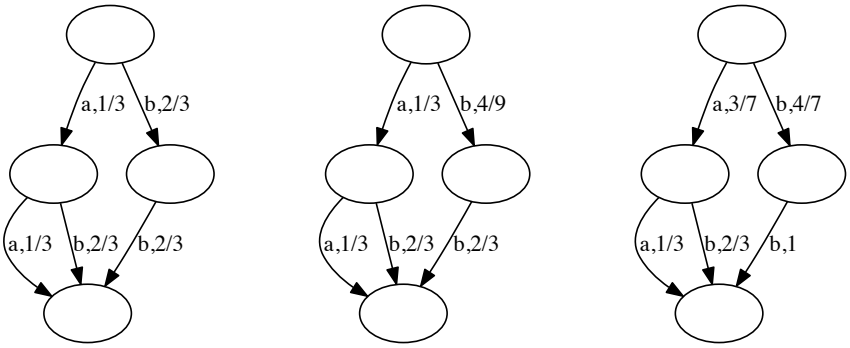


Figure 9.4: Sampling from a simple MDD. The probability of a is $1/3$ and it is $2/3$ for b.

Example:

Fig. 9.4 gives an example of the running of this algorithm when the probabilities are not uniform.

9.3.2 Markov chain

As in the previous section, our goal is to associate each arc with a probability and then sample the solution set by running a simple random walk according to these probabilities. The second method we obtain is equivalent to the one proposed by Papadopoulos *et al.* for the `regular` constraint [Papadopoulos 2015]. However, their method is complex and the propagation of the regular constraint costs more memory than the one of an MDD (see chapter 10).[2]

It is more difficult to apply a Markov chain than a PMF because in a Markov chain the probability of selecting a value depends on the previous

[2]In addition, our methods are quite simpler to understand since they do not require the use of belief propagation.

selected value, that is, probabilities must be defined in order to satisfy Property 5. More precisely, in an MDD, a node can have many incoming arcs, and these different incoming arcs can have different labels. Since the Markov probability depends on the previous value, the outgoing arcs of that node may have different probabilities depending on which was the incoming arc label. Thus, for an arc e, we need to have several probability values depending on the previous arc that has been used.

There are two possible ways to deal with a Markov chain. Either we transform the MDD by duplicating nodes in order to be able to apply an algorithm similar as COMPUTEMDDPROBABILITIES or we directly deal with the original MDD and we design a new algorithm.

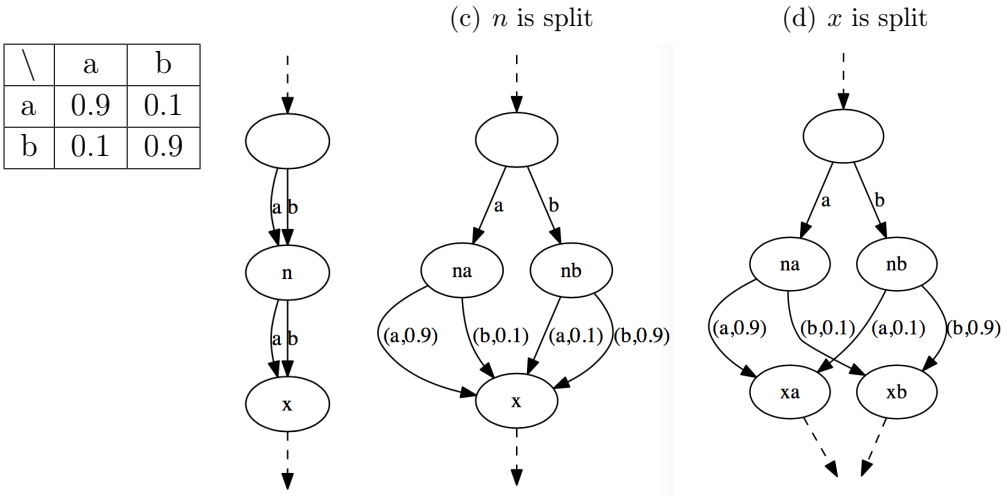


Figure 9.5: Duplication of a node and computation of probabilities.

9.3.2.1 Duplication of nodes

We can note that the matrix of the Markov chain represents a compression of nodes. Thus, if we duplicate each node according to its incoming arcs then we obtain a new MDD for which the probabilities become independent. More precisely, for each node n we split the node n in as many nodes as there are different values incoming. This means that each node n has only incoming arcs having the same label, and so only one value a incoming. Thus, the probability of each outgoing arc of the duplicated nodes of n can be determined directly by the Markov matrix.

Example:

For instance, consider the probabilities of Fig 9.2 and that we have a

node n with two incoming arcs: one labeled by a an the other labeled by b; and with two outgoing arcs: one labeled by a an the other labeled by b (Fig. 9.5). The node n is split into two nodes n_a and n_b. Node n_a has only incoming arcs labeled by a, and n_b has only incoming arcs labeled by b ((c) in Fig 9.5). In this case, we can define the probabilities as if we had independent variables. The probability of the arc (n_a, x, a), is defined by $P(a|a) = 0.9$, the probability of the arc (n_a, x, b) is $P(b|a) = 0.1$, the probability of the arc (n_b, x, a) is $P(a|b) = 0.1$, the probability of the arc (n_b, x, b) is $P(b|b) = 0.9$. Fig. 9.5 shows the duplication of a node. Note that when the node x will be split into two nodes x_a and x_b, then each of them will have two incoming arcs having the same label, a for x_a and b for x_b ((d) in Fig. 9.5).

Let $P_C(e)$ be the computed probability of any edge e computed by the duplication process. We can establish a Property similar as Property 6

Property 7 *Let M be an MDD defined on X and P_C a probability associated with each arc. Let n be any of node of the MDD and A be any partial instantiation of X reaching node n. The sum of the original probabilities of the partial solutions that can be reached from n is $v(n) = \sum_{s \in S(n)} P(s|A)$, where $S(n)$ is the set of partial solutions that we can reach from n and $P(s|A)$ is the probability of s under condition A. The probability of any arc $e = (n', n, a)$ is defined by $P(e) = P_C(e) \times v(n)$.*

proof: Similar as for Property 6. □

From this property we can design an algorithm similar as COMPUTEMDDPROBABILITIES by using $P_C(e)$ instead of $f_P(label(e))$ for each arc e. The drawback of this method is that it can multiply the number of nodes by at most d, the greatest cardinality domain of variables and also increases the number of edges which slowdowns the propagators. The next section presents another method avoiding this duplication.

9.3.2.2 A new algorithm

In order to deal with the fact that the probability of an outgoing arc depends on the label of the incoming arc without duplicating nodes, we associate each node with a probability matrix whose row depends on the incoming arc label. We denote these matrices by P_M^n for the node n. For efficiency, we only have one vector by incoming value instead of the full matrix, and each vector contains only the probability of the possible outgoing arcs labels. Then, the same reasoning as previously can be applied. We just need to adapt the previous algorithm by using matrices instead of duplicating nodes:

Algorithm computeMDDMarkovProbabilities can be described as follows:

1. For each node n, build the P_M^n matrix by copying the initial Markov probabilities.
2. For each node n, in BFS in bottom-up fashion:
   (a) Build the vector $vv(n)$ whose size is equal to the number of different incoming labels[3]. Each cell contains the sum of the probabilities of the row of the corresponding label in the P_M^n matrix.
   (b) Multiply each incoming arc probability by the cell of $vv(n)$ corresponding to its label.
3. For each node in a BFS top-bottom fashion, normalize the probability of the outgoing arcs.

Example:

Consider the MDD of Fig. 9.6.a, if we reuse the Markov distribution of Fig. 9.2 and apply the step 1 of the method, we obtain the MDD in Fig. 9.6.b.

Now from the MDD in Fig. 9.6.b, we perform the step 2, first (step 2.a) we process the sum of the outgoing probabilities for each node. For example for node 5 its probability is $0.1 + 0.9 = 1$ and for node 3 the sum is 0.9. For these two nodes the sum does not depend on the incoming arc label because there is only one. This is not the case for node 4 which has a sum of 0.1 for the incoming arc labeled by a and 0.9 for the incoming arc labeled by b. Now we apply step 2.b: we multiply the probability of the incoming arcs by the sum associated to their label in their destination node. Consider the arc from node 1 to node 3 and labeled by a, its probability was 0.9 and the sum of probabilities in its destination node is 0.9, then its new probability is 0.81. The arc from node 1 to node 4 is labeled by b; its probability was 0.1. For node 4, the sum is 0.9 for the incoming arc labeled by b, so the new probability of the $(1, 4, b)$ is $0.1 \times 0.9 = 0.09$. The MDD in Fig. 9.7.a is labeled with the resulting global probabilities.

Finally, from the MDD in Fig. 9.7.a, we normalize the outgoing arc probability of each node (step 3). For the root node 0, the outgoing probabilities sum is $0.54 + 0.364 = 0.904$. For its arc labeled by a and directed to node 1, the probability become $0.54/0.904 = 0.597$, this value

[3] $vv(n)$ represents a vector of $v(n)$.

has been rounded to 3 digits for readability. For its arc labeled by b and directed to node 2, the probability becomes $0.364/0.904 = 0.403$ (rounded). Thus, the outgoing sum of probabilities emanating from node 0 becomes $0.597 + 0.403 = 1$. The MDD from Figure 9.7.b shows the normalized probabilities.

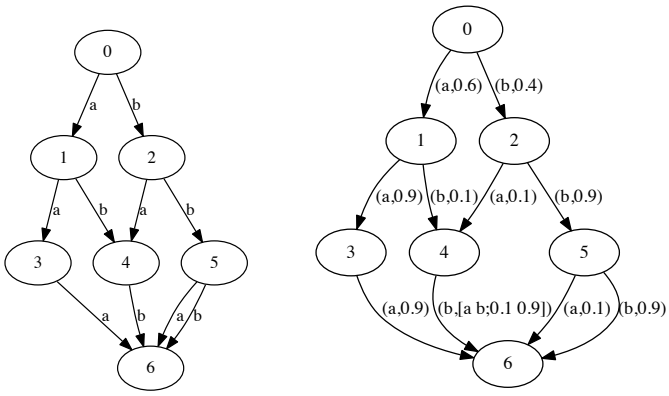


Figure 9.6: (a) left: an MDD. (b) right: the MDD whose arcs have their probability set thanks to the Markov distribution.

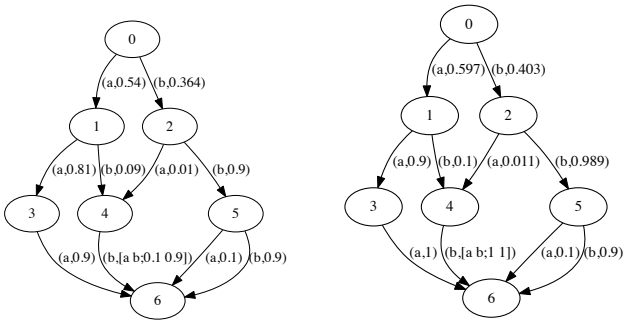


Figure 9.7: (a) left: MDD from Fig. 9.6.b whose arcs probability has been multiplied by the sum of the probabilities of the outgoing arcs from their destination node. (b) right: the MDD with renormalized probabilities.

Complexities. The complexities of COMPUTEMDDMARKOVPROBABILITIES algorithm are the following. The number of matrices is $|V|$, in the worst case the number of columns and rows is d, so the global memory complexity

is $O(|V| \times d^2)$. The complexity of each of the operations of this method are all linear over the matrices, so the overall time complexity is $O(|V| \times d^2)$. Since the number of columns of the matrix of a node is equal to the number of outgoing arcs of this node, a more realistic complexity for space and time is $O(|V| + |E| \times d)$, knowing that in a MDD, $|E| \leq |V| \times d$. Note that, for a given layer, nodes can be processed in parallel.

9.3.3 Incremental modifications.

If some modifications occur in the MDD, then instead of reprocessing all the probabilities we can have an incremental approach. From Step 2 of algorithms computeMDDProbabilities or computeMDDMarkovProbabilities, which performs a BFS in bottom-up, we perform the BFS only from the modified nodes since they are the only ones that can trigger modifications of the probabilities.

The reset principle used in MDD4R (chapter 10) can also be applied in this case. In other words, when there are fewer remaining arcs than deleted arcs, it is worthwhile to recompute from scratch the values.

9.4 Experiments

This chapter is mainly about modeling and the advantage of having general methods for dealing with different kinds of problems occurring in Artificial Intelligence. As an example of this advantage, we apply these methods to the transformation of classical texts written in French into alexandrine texts.This means that we try to express the same idea as the original text with the same style but by using only sentences having twelve syllables. The generation of the text in the same style as an author uses Markov chain that are extracted from the corpus. An MDD is defined from the corpus and ensures that each sentence will have exactly twelve syllables. Then, a random walk procedure is used for sampling the solutions. Thus, the model of this problem is simple and easy to implement.

We also test these methods on a real world application mainly involving convolutions which are expressed by knapsack constraints (i.e. $\sum \alpha_i x_i$). In addition, the probability of a value to be taken by a variable is defined by a probability mass function and outliers are not allowed.

Finally, the problem of generating huge number (number with many digits) without repetition is considered. This problem is also solved by encoding the number using MDDs.

The experiments were run on a macbook pro (2013) Intel core i7 2.3GHz

with 8 GB of memory. The constraint solver used is or-tools. MDD4R [Perez 2014] is used as MDD propagator and cost-MDD4R as cost-MDD propagator [Perez 2017c].

9.4.1 PMF constraint and sampling

The data come from a real life application: the geomodeling of a petroleum reservoir [Pennington 2001]. The problem is quite complex and we consider here only a subpart. The chapter 19 contains a clear definition of the problem and its models, a small part is given here.

Given a seismic image we want to find the velocities. Velocities values are represented by a probability mass function (PMF) on the model space. Velocities are discrete values of variables. For each cell c_{ij} of the reservoir, the seismic image gives a value s_{ij} and from the given seismic wavelet (α_k) we define a sum constraint $C_{ij} : \sum_{k=1}^{22} \alpha_k log(x_{i-11+k-1}j) = s_{ij} \pm \varepsilon$. Locally, that is, for each sum, we have to avoid outliers w.r.t. the PMF for the velocities. The problem is huge (millions of variables) so we consider here only a very small part.

We recall that the MDD of the constraint $\sum_{x_i \in X} f(x_i)$ is defined as follows. For the layer i, there are as many nodes as there are values of $\sum_{k=1}^{i} f(x_k)$. Each node is associated with such a value. A node n_p at layer i associated with value v_p is linked to a node n_q at layer $i+1$ associated with value v_q if and only if $v_q = v_p + f(a_i)$ with $a_i \in D(x_i)$. Then, only values v of the layer $|X|$ with $a \leq v \leq b$ are linked to tt. The reduction operation is applied after the definition and delete invalid nodes. The construction can be accelerated by removing states that are greater than b or that will not permit to reach a during the construction.

Each constraint C_{ij} is represented by MDD($\Sigma_{a_i,I}(X)$) where $a_i(x_i) = \alpha_i x_i$ and I is the tight interval representing $[s_{ij} - \varepsilon, s_{ij} + \varepsilon]$. Outliers are avoided thanks to an `MDDProbability` constraint defined from the PMF for the velocities. P_{min} is defined by selecting only values having the 10% smaller probabilities, P_{max} is defined by selecting only values having the 10% greater probabilities. This constraint is represented by a cost-MDD constraint (see chapter 11). Then, we intersect it with MDD($\Sigma_{a_i,I}(X)$).

We consider 20 C_{ij}. We repeat the experiments 20 times and take the mean of the results.

For each constraint C_{ij}, the resulting MDD has in average 116,848 nodes and 1,239,220 edges. More than 320s are needed to compute it. Only 8 ms are required by COMPUTEMDDPROBABILITIES algorithm in average. When a modification occurs the time to recompute the values are between a negligible value when the modifications are close to the root of the MDD and 8 ms when

another part is modified.

For sampling 100,000 solutions we need 169 ms with the old C rand() function and 207 ms with the Mersenne-Twister random engine in conjunction with the uniform generator of the C++ standard library. Note that the time spends within the rand() function is 15 ms, whereas it is 82 ms with the second function. Therefore, the sampling procedures require less than 3 times the time spent in the random function.

9.4.2 Markov chain and sampling

We evaluate the method by generating French alexandrines. That is, sentences containing exactly twelve syllables. The goal is to transform an existing text into a text having the same meaning but using only alexandrines. Thus, the corpus defines a Markov chain and the MDD defines the sentences having the right number of syllables. The sampling procedure we define generates solutions of the MDD associated with the Markov chain, that is, sentences hopefully resembling those of the corpus and having exactly 12 syllables. This model is simple and easy to implement. Note that we are not able to model this problem with any other technique, even the one proposed by Papadopoulos *et al*, because we need to deal only with sentences having 12 syllables and we do not know how to integrate this constraint into their model.

First, we use a corpus defined by one of the famous La Fontaine's fables. Here is the result we obtain for the fable: La grenouille qui veut se faire aussi grosse que le boeuf (The Frog and the Ox). We have underlined the syllables that must be pronounced when it is unclear:

La grenouille veut se faire aussi grosse que le bœuf

Grands seigneurs Tout bourgeois veut bâtir comme un Bœuf
Plus sages Tout marquis veut bâtir comme un œuf
Pour égaler l'animal en tout M'y voila
Voici donc Point du tout comme les grands seigneurs
Chétive Pécore S'enfla si bien qu'elle creva
Seigneurs Tout petit prince a des ambassadeurs

The generation of the MDD with the correct probabilities, that is just before the random walk, can be performed in negligible computational time.

We also considered a larger corpus: "A la recherche du temps perdu" of Proust, which contains more than 10,000 words. In this case, the results are less pertinent and some more work must be done about the meaning of the sentences. However, the method is efficient in term of computing performance

because only 2 seconds are needed to create the MDD with the correct probabilities.

9.4.3 Big Number generation

This third experiment considers the generation of huge numbers (number with many digits) without repetition.

Let M be the biggest number we want to represent and m be the smallest. Let b be a base used for representing the number, and B be the number of digit in base b for representing M.

Let $X = (x_1, ..., x_B)$ be the set of variables having the values $[0, b-1]$ as domain (i.e. the digits in base b). It exists an assignment of the variable in X such that:

$$x_1 * b^{B-1} + x_2 * b^{B-2} + ... + x_B * b^0 = M \tag{9.1}$$

Using the construction of an MDD using a sequence of tuples (chapter 4), we can easily build an MDD representing the number between $[m, M]$ over the variables of X.

Sampling By applying a PMF law of $1/b$ on the arc and using the sampling method for PMF, then we can extract a tuple representing a number $n \in [m, M]$. If we repeat this process then we obtain a generation of huge numbers with repetition.

In order to have generation without repetition, we need to enforce that once a number n has been chosen, it cannot be chosen again. To do so, we can simply remove this number from the MDD by using the in-place tuple deletion (chapter 5). Then we reapply the sampling algorithm. Thus once a number n has be chosen and deleted, the next tuple is in the interval $[m, n[\cup]n, M]$.

For example, the MDD from Figure 9.8 contains the numbers in the interval [1352,6293].

9.5 Conclusion

This chapter has presented two methods for sampling MDDs, one using a probability mass function and another one using the Markov distribution. These methods require the definition of probabilities for each arc and we have given algorithms for performing this task.

Thanks to these algorithms we can easily model and implement complex problems of automatic music or text generations having good performances

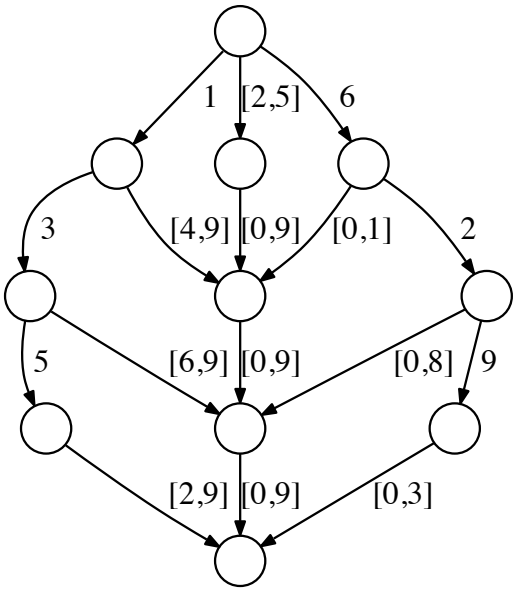


Figure 9.8: The MDD representing the numbers in the interval [1352,6293].

in practice. Moreover, this chapter shows how it is easy to define the model and to generate solutions.

Part III

MDDs: Constraints and Propagators

Chapter 10

Table & MDD-based Constraints

Contents

10.1 Introduction

In constraint programming, the notion of defining a constraint by the set of allowed tuples is common and very useful. Those constraints, often named extensional constraints or table constraints, allow to model any other constraints and is often defined directly by the users or synthesized from constraints of sub-problems [Lhomme 2012].

Efficient filtering algorithms for the table constraint are the kernel of constraint programming solvers and the topic of many research since several years [Régin 2011, Mohr 1988, Lecoutre 2012a, Lecoutre 2011, Lecoutre 2015, Bessiere 1997, Demeulenaere 2016, Wang 2016, Verhaeghe 2017,

x_1	x_2	x_3
1	1	1
1	2	1
2	2	3
3	3	3

Table 10.1: A table containing 4 tuples of 3 values.

Bessière 2005, Lhomme 2005, Xia 2013, Mairy 2012, Régin 2005]. Improving table constraints filtering algorithms is very challenging and still studied.

Consider an extensional constraint C. Arc-consistency algorithms for C operate as follows: for each value a in the domain of a variable x, they search for a combination of values in the current domains of the other variables in the scope of C that contains (x, a) and satisfies C.

A tuple of C is a combination of values in the domain of the variables in the scope of C. We say that the tuple is *allowed*, or a support, when it appears in the constraint definition. We say that the tuple is *valid* if and only if its values appear in the *current* domains of the respective variables. Note that a valid tuple is not necessarily allowed. Arc-consistency algorithms can be distinguished depending on how they manage allowed and valid tuples [Lhomme 2005]. While the allowed tuples do not change during the search because they are listed in the constraint definition, their validity is determined by the current domains.

Example:

Let T be the table from Table 10.1. A constraint C using T as allowed tuples and applied to the variables x_1, x_2 and x_3 allows only the following assignment: $\{(x_1{=}1,\ x_2{=}1,\ x_3{=}1),(x_1{=}1,\ x_2{=}2,\ x_3{=}1),(x_1{=}2,\ x_2{=}2,\ x_3{=}3),(x_1{=}3,\ x_2{=}3,\ x_3{=}3)\}$

The arc consistency algorithms for table constraints mainly differ by how they operate when a value is deleted. Some algorithms are lazy (e.g., GAC-Schema [Bessiere 1997] or STR-3 [Lecoutre 2012a]). They try to reduce the operations executed at each modification (i.e., deletion of value of a domain) at the cost of increasing the complexity of the implementation. Others, such as GAC-4 [Mohr 1988] or STR-2 [Lecoutre 2011], operate more systematically, thus keeping simple the implementation. Moreover some arc-consistency algorithms operate on allowed tuples to check their validity, while others first consider the current domains and look for a combination satisfying the constraint. The related work section of this chapter gives more insights about

table constraint algorithms.

Motivations A key idea for improving the performance of arc-consistency algorithms for table constraints is to reduce the size of the representation of the tuples because the complexity depends on it. Several algorithms for compressing the allowed tuples of a constraint have been proposed, and arc consistency algorithms adapted for dealing with them [Katsirelos 2007, Gent 2007, Régin 2011, Mairy 2015, Demeulenaere 2016, Verhaeghe 2017, Wang 2016, Cheng 2008, Cheng 2010].

Some compressions are based on trees [Gent 2007], global cut seeds [Katsirelos 2007], sequences of tuples [Régin 2011] or smart tuples [Mairy 2015]. As presented in chapter 4, the global cut seed and the sequences of tuples data structures may gain an exponential factor in representation. They are also efficiently transformed into MDDs.

Several works focus on regular expressions and automatons [Pesant 2004, Beldiceanu 2004a]. An automaton or a regular expression can be seen as an intensional way for expressing a table constraint [Cook 2009].

More recently several works used a bit-set representation of the table [Demeulenaere 2016, Wang 2016]. They project the tuples over a bit-set by associating a bit to each tuple, representing the validity of the tuple. Even if the maximum gain factor is bound by the number of bits of the machine, the efficiency in practice is strong. In order to improve this theoretical gain factor, some works focus on representing smart tuples using a bit-set representation [Verhaeghe 2017].

Multi-valued Decision Diagrams are one of the most advanced and powerful compressed representations. As presented in Chapter 4 MDDs can be used for storing tuples, many data structures, problem definitions and even other constraints. Thus an efficient filtering algorithm is required. In response to this need, several MDD filtering algorithms have been proposed [Cheng 2008, Cheng 2010, Gange 2011].

> **Example:**
> Consider the MDD from Figure 10.1. This MDD represents the set of 12 tuples of the table (a). While the table needs 12 * 3 = 36 cells in order to represent the tuples, the MDD need 11 arcs and 6 nodes.

This chapter presents GAC-4R and MDD4R, two improved versions of the classical GAC-4 algorithm for tables and MDDs. They are already implemented in several constraint programming solvers [OscaR Team 2012, Perron 2013] and have a linear complexity. They are very efficient in practice. The plan is organized as follows. First we recall some state of the art

(a) A Table

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

(b) An MDD

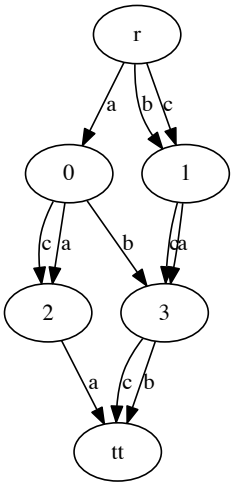


Figure 10.1: On the left, a table containing 12 tuples of 3 values. On the right an MDD representing the table on the left.

algorithms for table and MDD constraints. Then GAC-4 and GAC-4R are described. Finally, MDD4R the MDD propagator is described.

10.2 Related Work

10.2.1 Table Constraint propagators

Table constraint A table constraint is a constraint defined directly by its set of allowed or prohibited tuples. Filtering algorithms enforcing arc consistency on table constraints have to ensure that for each value of each variable, there exists a valid combination of the current domain of the variables that is allowed by the table.

Pending values The pending values [Régin 2005] are the values whose validity is uncertain and have to be checked by the propagators.

We can describe the filtering algorithms of table constraints by three main parts:

- How the algorithm reacts when a value is removed?
- Which are the pending values?

- How these pending values are validated?

For the several table constraints filtering algorithms, a simple description of these operations is given below.

GAC-3 GAC-3 is one of the simplest constraint propagators [Mackworth 1977]. For each value of each variable, GAC-3 tests all the tuples until it found a valid tuple and supports the value.

- Removing a value: Nothing to do.
- Pending values: All the values in the current domain of the variables.
- Validation of the pending values: For each pending value, search in the table for the first valid tuple supporting the value.

GAC-4 GAC-4 is the first optimal GAC algorithm for table constraints [Mohr 1988]. It maintains for each value of each variable the set of valid tuples containing the value. This algorithm is going to be explained in the next section.

- Removing a value: For each tuple t in the support of the value, remove t from all the sets of support from the values of t.
- Pending values: All the values whose set of support has been modified.
- Validation of the pending values: Is their set of support non-empty?

GAC-Schema Or GAC-6 is the "lazy" algorithm between GAC-4 and GAC-3 [Bessiere 1997]. Instead of testing all the tuples each time like GAC-3 or maintaining all the valid tuples like GAC-4, GAC-Schema gives a schema of propagator maintaining at least one support by value. Thus GAC-6 does not maintain all the valid support. Moreover GAC-Schema can be combined with many data structures like trees, etc [Gent 2007].

- Removing a value: Mark all the values sharing a support with one of the removed values.
- Pending values: All the marked values.
- Validation of the pending values: For each pending value, starting from the last valid support, search on the data structure of possible supports if one of them is valid.

STR The STR algorithm [Ullmann 2007, Lecoutre 2011] proposes the idea searching over the valid tuples and validates the supported values instead of searching for each particular value a support. STR projects the tuples into a reversible set S of possible valid tuples. When a modification occurs, STR iterates over the S set of tuples. For each tuple, if the tuple is valid, then it validates all the values of the tuples. Otherwise it removes the tuple from S. The algorithm can stop the iteration when all the values are supported.

- Removing a value: Nothing to do.
- Pending values: All the values in the current domain of the variables.
- Validation of the pending values: Search over the set of possibly valid tuples and check if the tuple is valid. If the tuple is valid then validate the values of the tuples, otherwise remove it. Stop when all the values are supported.

STR2. STR2 [Lecoutre 2011] is an improvement of the STR algorithm. But instead of maintaining a set of possible valid tuples, it maintains the set of valid tuples. Moreover it improves the test of the validity of a tuple by considering only the variables whose domain changed since the last call. Then in order to support the pending values, it simply iterates over the valid tuples and support all the values belonging to one of them.

- Removing a value: Iterate over the set of valid tuples and remove all the invalid tuples by considering only the modified variables.
- Pending values: All the values in the current domain of the variables.
- Validation of the pending values : Iterate over the valid tuples and validate the values that appear on them. Stop when all the values are supported.

STR3. STR3 has been defined in [Lecoutre 2012a]. This algorithm combines the principle of both GAC-4 and GAC-schema, while having an optimality property. Like GAC-4, when a value is removed then all the tuples involving the value are marked invalid and so removed. Then like the GAC-schema behavior, it iterates over the possible supports of the pending values, starting from the last valid, and seeks for a new support.

- Removing a value: For all the tuples in the list of support of the value, mark them as invalid.
- Pending values: All the values that belong to a removed tuple.

- Validation of the pending values: For each pending value, starting from the last valid tuple of the value, search on the list of possible support if one of them is valid.

STR-Bit This algorithm is one of the first global table propagator using bit-set [Wang 2016]. It consists of an efficient transformation of the STR3 propagator using a bit-set for the IsValid function of the tuples. But since the bit-set representation does not allow to easily know the pending values, like STR3 can do, all the values are considered pending.

- Removing a value: For all the tuples in the list of support of the value, mark them invalid.
- Pending values: All the values in the current domain of the variables.
- Validation of the pending values: For each pending value, starting from the last valid tuple, search on the list of possible support if one of them is valid.

Compact Table Compact-Table [Demeulenaere 2016] has been recently introduced, while it is implemented since several years in CP solvers. In the same way as STR-Bit is a bit-set transformation of STR3, Compact Table can be seen as a bit-set transformation of STR2 using the bit-set as the IsValid function of a tuple.

- Removing a value: Iterates over the set of valid tuples and removes all the invalid tuples by considering only the modified variable.
- Pending values: All the values in the current domain of the variables.
- Validation of pending values : For each pending value, test the last support. If the last support is invalid then search in the table for the first valid tuple supporting the value.

Bit-wise based propagators Both STR-Bit and Compact Table use bit-wise operations in order to maintain the validity function of a tuple. For each tuple, a bit is associated with its state, 1 if the tuple is valid and 0 otherwise. Both of the algorithm maintained and backtrack this information during the search. Thanks to this bit-set representation, the invalidation of several tuples involve in the same word can be efficiently done using a bit-wise operation.

10.2.2 MDD Constraint Propagators

The MDD associated with a constraint C is an MDD which models the set of tuples satisfying C. An MDD propagator of C is an algorithm which removes some inconsistent values of $X(C)$, the variables on which C is defined.

10.2.2.1 MDD of a constraint.

Let C be a constraint defined on $X(C)$. The MDD associated with C, denoted by MDD(C), is an MDD which models the set of tuples satisfying C. More precisely, MDD(C) is defined on $X(C)$, such that the labels of arcs of the layer of the variable x correspond to values of x, and a path of MDD(C) where a_i is the label of layer i corresponds to a tuple $(a_1, ..., a_n)$ on $X(C)$.

10.2.2.2 Consistency with MDD(C).

A value a of the variable x is valid iff $a \in D(x)$. An arc (u, v, a) at layer i is valid iff $a \in D(x_i)$. A path is valid iff all its arcs are valid.
Let $path^r_{tt}$(MDD(C)) be the set of paths from root r to tt in MDD(C). The value $a \in D(x_i)$ is consistent with MDD(C) iff there is a valid path in $path^r_{tt}$(MDD(C)) which contains an arc at layer i labeled by a.

10.2.2.3 MDD propagator.

An MDD propagator associated with a constraint C is an algorithm which removes some inconsistent values of $X(C)$.

The MDD propagator establishes arc consistency of C if and only if it removes all inconsistent values with MDD(C). This means that it ensures that there is a valid path from the root to the true terminal node in MDD(C) if and only if the corresponding tuple is allowed by C and valid.

The MDD propagator enforces MDD consistency if it removes all the arcs that do not belong to a valid path with respect to the constraint [Andersen 2007, Hoda 2010]. MDD consistency is a stronger notion and arc consistency is inherited from MDD consistency.

10.2.2.4 `mddc` propagator

Cheng and Yap [Cheng 2008, Cheng 2010] provide `mddc`, an algorithm maintaining arc consistency for an MDD constraint. This algorithm searches on the MDD, using a depth first search, by considering only arcs whose label is still a valid value from the domain of its corresponding variable. When this search reaches the final tt nodes, a valid tuple has been found and we we can make valid all the values of this tuple in their associated variable domains.

When the depth first search ends, the values that have not been validated can be safely removed because there is no longer a path from the root to the positive terminal node tt, which involves an arc corresponding to this value.

This propagator uses the array representation of an MDD (see Appendix A). This implies that each node contains an array of d values and so random access for the cells of values that belong to the current domain of the variable is easy.

In order to improve the algorithm, several improvements are made. The first one is to record during the search, in a backtracking data structure, the nodes that do not belong to a valid path from the root node to the `tt` node. Then during the search, when such a node is reached, the algorithm does not need to process it. This can be efficiently done using a reversible sparse set. A second improvement is during the depth first search on the MDD, the algorithm keeps the nodes for which a valid path from the root node to the tt node has been found. When one of these nodes is reached again by another incoming path, the algorithm can support the values of this incoming path since the values belong to at least one valid path from the root node to the `tt` node.

Example:

Consider M, the MDD from Figure 10.1. Let the constraint C defined on the variables x_1, x_2 and x_3 and using the MDD M. Consider that, for external causes, the second variable x_2 is set to b. The algorithm is going to perform the following steps: Starting from the root node r, the arc labeled by a is used to reach the node 0. Since the second variable contains only the value b in its current domain, the arc labeled by b is used to reach node 3. Starting from node 3 the arc labeled by a reaching ff is used because the algorithm considers the current domain. The algorithm returns to node 3 and the arc labeled by b is used to reach tt. A valid path is found, b is supported in x_3, then the node 3 is set valid. The algorithm used the arc labeled by c starting at 3 and valid c from x_3. The algorithm returns to node 0, marks 0 as valid and supports $x_2 = b$ and returns to r and supports $x_1 = a$. The algorithm follows the arc labeled by b and reaches the node 1. From node 1 the algorithm follows the arc labeled by b and reaches `ff`. The algorithm returns to 1 and marks it as invalid, it returns to r, follows the arc labeled by c, reaches 1 and returns to r since 1 is marked as invalid. There is no more arc to follow, the algorithm is over and the values b and c are removed from the current domain of x_1 and the value a is removed from the current domain of x_3.

The arcs processed by *mddc* are dashed in Figure 10.2.

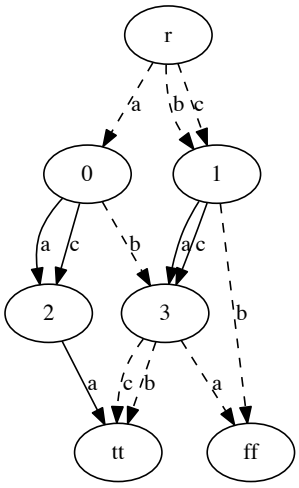


Figure 10.2: Application of the *mddc* algorithm for processing the decision $x_2 = b$. The dashed arcs are the arcs touched by the propagator.

In `mddc`, it is important to note that any arc is traversed at most once, because a depth first search is used and because the MDD is not changed during the search by the algorithm. The way the MDD is traversed only depends on the current domains.

Moreover it is not straightforward to find an arc corresponding to a value belonging to the current domain and this task is even more difficult when the domain size is reduced. But when the MDD is small, or when the domain size is small, then `mddc` can be very efficient.

Remark: `mddc` can be seen as a tuple based algorithm like the STR algorithm. It iterates over the tuples, with a compression mechanism (MDD + validation/negation of the nodes), and supports all the values occurring in one of the discovered tuples.

10.2.2.5 Regular Constraint Propagators

The regular or automaton constraints [Pesant 2004, Beldiceanu 2004a] are constraints that can be defined using a regular expression or an automaton. The relation between MDDs and automaton is strong, and the propagators close. In contrast to the extensional constraints, regular expressions or automaton can be seen as intensional constraints. In constraint programming, the number of variable (i.e. the arity) of a constraint is given by r. This implies that the valid tuples of such a constraint C are the subset $T(C)$ of

tuples of size r that respect the intensional definition.

Even if the intensional representation can be very small, the set of tuples $T(C)$ can be exponential. In order to propagate such intensional constraint, the algorithm has the two following points:

- Before removing a value a from a variable x, be sure that there is no combination of the values of the current domains of the variable of C that is in $T(C)$.
- Before not removing a value a from a variable x, be sure that there is at least one combination of the values of the current domains of the variable of C that is in $T(C)$.

Since the simplest way to do it would be to extract the table from the intensional definition, this table could be exponential in size. Better algorithms have been found, and they can be seen with a common data structure, the unfolded graph representation.

Unfolded Graph Representation Consider an automaton defined by the transition table τ. The unfolded version over the r variables of this automaton represents the set of tuples $T(C)$. To do so the automaton is unfolded r times. Each node u from a layer is associated to a state $s(u)$ from the automaton. There is an arc between a node u from layer i and v from layer $i+1$ labeled by a if and only if there is a transition $(s(u), s(v), a) \in \tau$.

> **Example:**
> Consider the automaton from Figure 10.3. The automaton (a) is used to prevent a machine of working more than twice without a break, for example for cooling the machine, but prevents the machine to have two consecutive breaks. The unfolded graph (b) is the unfolded version of the automaton. As we can see each node corresponds to a pair (state - layer).

This unfolded representation can be seen as a possible non reduced MDD. This unfolded graph representation is used to propagate the constraint. The next two propagators are mainly suited for regular or automaton constraints but can easily be adapted to MDDs.

Ternary Decomposition The ternary decomposition of regular and automaton constraints is often considered as a good option [Beldiceanu 2004a, Quimper 2006]. This decomposition consists of associating to each layer of nodes from the unfolded graph representation an *intermediate* variable. The domain of these variables are the *indexes* of the nodes of the associated layer.

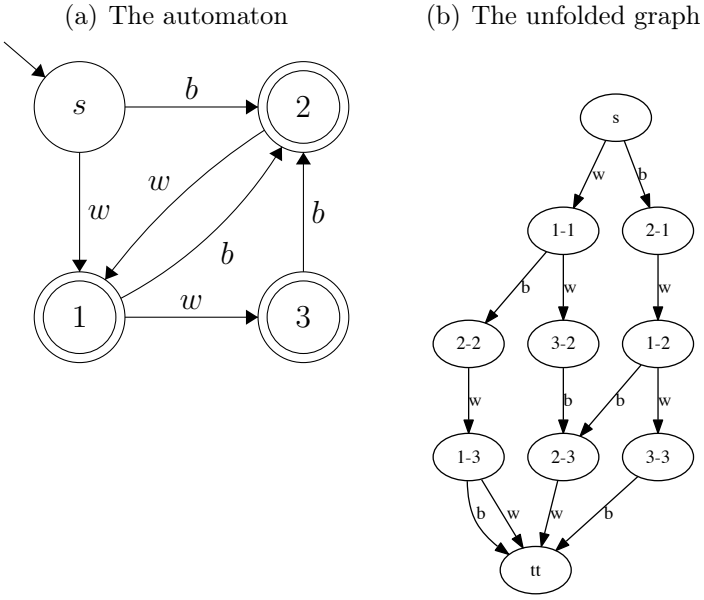


Figure 10.3: On the left, an automaton that prevents a machine of working (w) more than twice without a break (b) and prevents the machine to take two consecutive breaks. The right graph represents the unfolded version over 4 variables.

This allows to control the nodes of the graph by controlling the domains of the intermediate variables.

The arcs are associated with the original variables from the constraint since the label of the arc are associated with the values of the domains of the variables.

Then for each layer $i \in 1..r$, the nodes of layer i and $i+1$ are extracted and the arcs between these nodes are used for building a table constraint. For each arc between the node u_i at layer i to node u_{i+1} at layer $i+1$ labeled by a, the tuple (u_i,a,u_{i+1}) is created. Each of these table constraints involve the two intermediate variables and the variable associated with the arcs.

Once these table constraints defined, we can use any existing table constraint algorithms to propagate them. This decomposition is Berge acyclic, this implies that enforcing arc consistency on the table constraints allows to enforce arc consistency on the decomposed constraint.

Graph-Based Propagator An algorithm using the unfolded graph representation directly in order to propagate the constraint has been proposed [Pesant 2004]. This implies no more intermediate variables due to the ternary decomposition.

This algorithm is close to GAC-4 since it maintains for each value a of each variable x a set $Q_{x,a}$ of all the nodes that contain an arc labeled by a in the layer of x. The algorithm also maintains for each node of the graph the list of outgoing arc and the list of incoming arcs.

The algorithm processes as follows: For each node u stored in the support $Q_{x,a}$ of the pair variable x value a that has to be propagated, it removes the arc labeled by a starting at u from both its extremities. Then it propagates the modification in the unfolded graph in a recursive way. When an arc has to be removed, it is also removed from the Q set associated with the variable/value that it supports.

When the Q set of a value becomes empty, this implies that there is no arc in the unfolded graph supporting the value anymore. The value is invalid and can be safely removed.

For the Q sets of the values, storing the node is enough since using the node and the value, the transition function gives the other extremity node. Maintaining the arcs of the nodes is made using two double linked lists of arcs. Finally, an arc contains a pointer to both of its positions in the lists of its extremities, allowing efficient removing and backtracking.

Example:

Consider M, the MDD from Figure 10.1. Let the constraint C defined

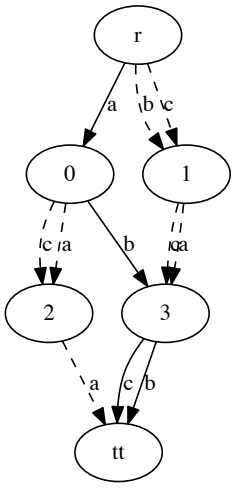


Figure 10.4: Application of the graph-based propagator for processing the decision $x_2 = b$. The dashed arcs are the arcs touched by the propagator.

on the variables x_1, x_2 and x_3 and using the MDD M. Consider that, for external causes, the second variable x_2 is set to b. The algorithm is going to perform the following steps: First it considers $Q_{x_2,a}$, it removes the arc $(0, 2, a)$, checks if 0 has to be deleted, checks if 2 has to be deleted. Then it removes $(1, 3, a)$, checks if 1 has to be deleted, checks if 3 has to be deleted. After the $Q_{x_2,a}$ list, the $Q_{x_2,c}$ list is processed. The arc $(0, 2, c)$ is removed, the node 0 is checked for deletion, the node 2 is checked and has to be removed. The algorithm removes the node 2, removes the arc $(2, tt, a)$ and remove 2 from the $Q_{x_3,a}$ list. Since the list $Q_{x_3,a}$ is now empty the value a can be safely removed from the current domain of x_3. The algorithm removes the arc $(1, 3, c)$, checks if the node 1 has to be removed, removes the node 1 and its incoming arcs $(r, 1, b)$ and $(r, 1, c)$. modify the $Q_{x_1,b}$ and $Q_{x_1,c}$ lists and removes the values b and c from the current domain of x_1.

The arcs processed by the *graph-based propagator* are dashed in Figure 10.5.

Remark: This algorithm is close to the one from GAC-4 and so it will be close to MDD4. It maintains for each value of each variable the list of valid support and so for nodes. When a modification occurs, it removes the invalid supports and removes the supports that are related to them.

10.2.2.6 mdd_w propagator

mdd_w is the propagator defined in [Gange 2011] for MDDs, I have chosen the name mdd_w for MDD propagator with watched literals. In the same way as other lazy algorithms [Bessiere 1997], the algorithm tries to do as less operation as possible. This implies trying not to maintain the whole graph like [Pesant 2004].

To do so, the algorithm maintains only one support by value or node at a moment. But just like the differences between GAC-4 and GAC-schema, lazy algorithms are often more complex.

Let x be a variable and a a value in the domain of the variable x, $arcs(x, a)$ is the list of arcs emanating from a node at the layer of x and labeled by a. Let u be a node, the list $arcs^+(u)$ and $arcs^-(u)$ contains respectively the emanating and terminating arcs for the node u.

The algorithm maintains for each arc (u, v, a), with u and v being the two extremities of the arc and a being the label, the following information:

- Is the arc valid?
- Is the arc the support of the node above (u watcher)?
- Is the arc the support of the node beyond (v watcher)?
- Is the arc the support of the pair variable value ($x = Var(u)$,a) (value watcher)?

This algorithm also maintains for each value a of each variable x a valid arc $valid_{(x,a)}$ inside the MDD.

When a value a of a variable x is removed, all the valid arcs for the pair (x, a) are removed from the MDD. To do so, the algorithm searches over the list $arcs(x, a)$, starting from the last valid arc $valid_{(x,a)}$, and for each arc $e = (u, v, a)$ of these arcs, if the arc is valid, then:

1. The arc is set to invalid.
2. If the arc was the support of u, then u is added to the set Up_i of the possible inconsistent nodes.
3. If the arc was the support of v, then v is added to the set $Down_i$ of the possible inconsistent nodes.

Once all the arcs in $arcs(x, a)$ have been removed, the algorithm starts a second step which consists on propagating the modification on the MDD. To do so, for each of the node u in $Down_i$, a new valid arc is searched in $arcs^-(u)$. If such an arc is found, then this arc become the new support of u. Otherwise for all the arc $e = (u, v, a)$ in $arcs^+(u)$, if the arc is valid:

1. The arc is set to invalid.
2. If the arc was the support of the pair variable value ($x = Var(u)$,a) then add the pair (x, a) to the set Var_i of possible inconsistent values.
3. If the arc was the support of v, then v is added in the set $Down_i$ of the possible inconsistent nodes.

The same process is then applied to the set Up_i. Finally, for each pair (x, a) in Var_i, the algorithm searches over the list $arcs(x, a)$ for a valid arc. If such an arc is not found, this implies that there is no valid arc in the MDD for the variable x and labeled by a, thus the value a can be safely removed from the domain of x.

Example:

Consider M, the MDD from Figure 10.1. Let the constraint C defined on the variables x_1, x_2 and x_3 and using the MDD M. Consider that, for external causes, the second variable x_2 is set to b. The algorithm is going to perform the following steps: First it considers the list $arcs(x_2, a)$, it invalidates the arc $e_1 = (0, 2, a)$, checks if e_1 is the watch of node 0, let consider it is since it's the first one, and puts 0 in the Up_i set. Then it checks if e_1 is the watch of node 2, let consider it is since it's the first one again, and puts 2 in the $Down_i$ set. The second arc $e_2 = (1, 3, a)$ is invalidated, mdd_w checks if e_2 is the watch of node 1, let consider it is since it's the first one, and puts 1 in the Up_i set. Then it checks if e_2 is the watch of node 3, let consider it is since it's the first one again, and puts 3 in the $Down_i$ set. Second, it considers the list $arcs(x_2, b)$, it invalidates the arc $e_3 = (0, 2, b)$, checks if e_3 is the watch of node 0. Then it checks if e_3 is the watch of node 2. The second arc $e_4 = (1, 3, b)$ is invalidated, mdd_w checks if e_4 is the watch of node 1. Then it checks if e_4 is the watch of node 3. Now that all the arcs from the $arcs$ list of the values have been removed, the algorithm has to propagate these modifications. Starting with the Up_i set, the node 0 is pop, the algorithm iterates over the $arcs^+(0)$ and finds the valid arc b which is marked as the new watch of 0. Then the node 1 is pop from Up_i, the algorithm searches over the $arcs^+(1)$ list of arcs and tests both the outgoing arcs of 1 but they both are invalid, so the algorithm has to remove the incoming arcs of 1, the arcs in $arcs^-(1)$. The arc $e_5 = (r, 1, b)$ is marked invalid, the algorithm checks if e_5 is the support of r, it is not here, then checks if e_5 was the support of the pair (x_1, b), it is so (x_1, b) is pushed into Var_i. The arc $e_6 = (r, 1, c)$ is marked invalid, the algorithm checks if e_6 is the support of r, it is not here, it checks if e_6 was the support of the pair (x_1, c), it

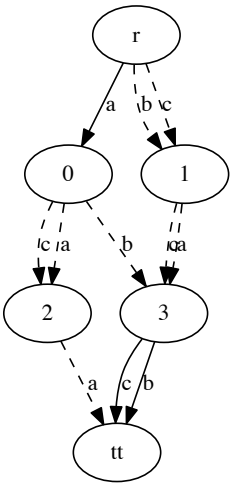


Figure 10.5: Application of the mdd_w propagator for processing the decision $x_2 = b$. The dashed arcs are the arcs touched by the propagator.

is so (x_1, c) is pushed into Var_i. The algorithm now focuses on the set $Down_i$, the node 2 is pop, the algorithm searches over the $arcs^-(2)$ list of arcs in order to determine if the node 2 has to be removed or not. It tests both the incoming arcs of 2 but they both are invalid, so the algorithm has to remove the outgoing arcs of 2. The arc $e_7(2, tt, a)$ from $arcs^+(2)$ is marked as invalid, then the algorithm checks if e_7 is the watch of node tt, let consider it is not, then it checks if e_7 is the watch of the pair (x_3, a), it is so (x_3, a) is pushed into Var_i. Finally, once all the propagation of the modifications is over, the algorithm has to ensure that the pending values, the values inside the Var_i set, still have a support. The pair (x_1, b) is extracted from Var_i, the algorithm iterates over the list $arcs(x_1, b)$, the only arc is invalid, so we can remove the value b from the current domain of x_1. The same process is made for the two other pairs (x_1, c) and (x_3, a).

The arcs processed by mdd_w propagator are dashed in Figure 10.5.

The complexity over a branch tree of the mdd_w algorithm can be amortized to linear on the number of arcs if for each of the searches onto the $arcs*$ lists, the algorithm start from the last watch. While this information has to be backtracked in order to keep this complexity, a problem often occurs in these kinds of algorithms: The *trashing*.

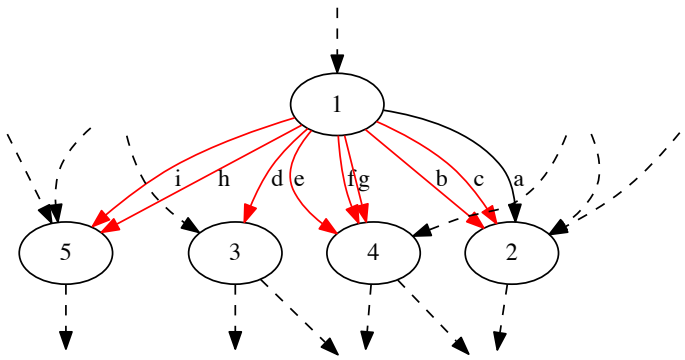


Figure 10.6: A trashing example. When the variables associated with node 1 is set to a, if the arc labeled by a is already the support of node 1, then no work has to be done for node 1. But later, when the arc $(1, 2, a)$ is removed, for example after the removing of node 2, the algorithm has to check all the outgoing arcs of 1. Since this work is delayed, several repetitions are going to be processed.

Remark This algorithm is close to STR3. When a value has to be removed, all the valid supports and possible supports are removed, and then the algorithm looks for at least one support by value or node.

Trashing One of the main goals of lazy algorithms (STR3, GAC-schema or mdd_w) is trying to avoid work that will never be required. But such a lazy behavior can do the contrary.

Consider for example a list of values, and an algorithm that ensures during the search that at least one of these values is valid from a given criteria. Lazy algorithms are going to keep the first valid value from this criteria and while it is not required, these lazy algorithms are not going to check or so to remove the invalid values from the list. In this case, if the first value of the list is the only valid value during the whole search and is removed only on the leaves, then these lazy algorithms are going to perform an exponential number of times the delayed work.

Consider now the mdd_w algorithm, in the bottom of the search tree the algorithm often has a lot of work to do that has been delayed because of the

Algorithm 17 Functions for manipulating a sparse set. k is an element. S is a sparse set with two arrays $S.\texttt{dense}$ and $S.\texttt{sparse}$ and scalar $S.\texttt{members}$.

```
MEMBER(k, S)
    return S.sparse[k] < S.members and S.dense[S.sparse[k]] = k
ADD(k, S) // assume k is not a member
    S.sparse[k] ← S.members
    S.dense[S.members] ← k
    S.members← S.members +1
DELETE(k, S) // assume k is a member
    ik ← S.sparse[k];
    ie ← S.members −1;
    e ← S.dense[ie]
    S.sparse[e] ← ik
    S.dense[ik] ← e
    S.sparse[k] ← ie
    S.dense[ie] ← k
    S.members← S.members −1
```

lazy behavior. For example, consider a value a of a variable x that is supported by it first arc in $arcs(x, a)$. If the first decision invalidates all the other arcs in $arcs(x, a)$, then the lazy behavior of the algorithm will not process this deletion until it needs to search for a new support. While processing this deletion costs D, if the deletion is processed each time at worst on the leaves, if i variables are still not assigned to a single value, the possible worst case complexity of this deletion is going to be $O(d^i D)$. The same reasoning can be applied to several other lazy algorithms.

Example of trashing The MDD given in Figure 10.6 presents one of the possible cases of trashing. Here it is the one associated with the delayed deletion of the arcs.

10.2.3 Sparse Set

Sparse sets are an efficient data structure for manipulating sets with a fixed size universe U [Briggs 1993]. they have been successfully used in CP for representing sets or lists [Cheng 2008, Cheng 2010, Lecoutre 2011, Lecoutre 2012a]. We are going to use and modify the behaviour of these sets for designing efficient propagators.

For convenience, the elements in U are mapped to integers 0 through $|U| - 1$. The Sparse Set representation has three components: two vectors (named `dense` and `sparse`), each $|U|$ elements long and a scalar (named `members`) that records the number of members in the set. The values in the

array `dense` from 0 to `members` - 1 corresponds to the elements in the set. The array `sparse` contains indices of the array `dense`. If a number k is a member of the set, it must satisfy two conditions $0 \leq$`sparse`$[k] <$ `members` and `dense`[`sparse`$[k]] = k$. It means that `sparse`$[k]$ is the index i in the array `dense` of the value k, that is, we have `dense`$[i] = k$. Here is a sparse set:

sparse	5	2	-	0	-	1	-	3	4	-
dense	3	5	1	7	8	0				
members	6									

The membership, addition and deletion functions are defined in Algorithm 17. Function DELETE has been modified from its original definition in [Briggs 1993] in order to be able to restore the sparse set easily after some deletions. Consider the sparse set previously defined. Suppose that member 7 is deleted. Before the deletion the scalar `members` is equal to 6 and after the deletion, we have the new sparse set:

sparse	3	2	-	0	-	1	-	5	4	-
dense	3	5	1	0	8	7				
members	5									

When 7 has been deleted, the members 7 and 0 (i.e. the last value of the set) have been exchanged. Precisely, we swap `dense[sparse[7]]` and `dense[sparse[0]]` and we swap `sparse[7]` and `sparse[0]`. Thanks to these swaps, we can easily restore the sparse set simply by setting the members value to 6. The sparse set contains the same elements but not in the same order.

10.3 GAC-4R: Table Propagator

In order to present the GAC-4R algorithm, this section first recalls the GAC-4 filtering algorithm.

10.3.1 GAC-4

GAC-4 is a fully incremental algorithm associating to each variable-value pair (x, a), the list $S(x, a)$ of valid tuples involving (x, a) that satisfy C. When a value b is deleted from the domain of a variable y, the tuples associated with (y, b) are no longer valid and must be removed. Consequently, for each tuple $t \in S(y, b)$ and for each variable-value pair (z, c) in t, we remove t from $S(z, c)$. If $S(z, c)$ becomes empty, then no valid tuple involving (z, c) and satisfying C exists. Thus, we can safely remove c from $D(z)$.

The algorithm can be described as follows:

Algorithm 18 REVISEGAC-4

```
REVISEGAC-4(C: constraint; deletionSet: list): Boolean
    for each (x, a) ∈ deletionSet do
        for each t ∈ S(x, a) do
            for each (z, c) ∈ t do remove t from S(z, c)
            if S(z, c) = ∅ then
                remove c from D(z) ;
                add (z, c) to deletionSet
            if D(z) = ∅ then
                return False;
    return True
```

- Initialization: for each tuple t and for each value (x, a) belonging to t we add t to $S(x, a)$.
- Invariant: $\forall x \in X(C)$, $\forall a \in D(x)$: $S(x, a)$ *contains the valid tuples* $t \in T(C)$ *with* $t[x] = a$.

When modifications occur in the current domain of the variable, the function REVISEGAC-4 (See Algorithm 18) is called in order to propagate the consequences of these modifications and to close the invariant.

10.3.2 GAC-4R

10.3.2.1 Motivation

GAC-4 is a simple and an easy algorithm to implement. Its worst case complexity is optimal. However it is mainly focused on the study of the consequences of the deletions of values. GAC-4 is efficient when there are only few tuples for each value, which typically occurs at deeper levels of the search tree. However, at shallower levels its performance is qualitatively different in that maintaining the internal data structures is costly.

While GAC-4 focuses on the deletions, it could be worthwhile to recompute some data structures instead of maintaining them incrementally. In other words, the performance of GAC-4 can be improved by rebuilding, from scratch, the data structures of GAC-4 when the modifications have reached a given threshold.

Example Consider a table constraint with k tuples and involving a variable x having 10 values in its domain (the arity is not important here). Assume

that the tuples are homogeneously distributed among the values of x. In other words, every value of x appears in about $\frac{k}{10}$ tuples. Now, assume that a is assigned to x. Thus, only about $\frac{k}{10}$ tuples remain valid. GAC-4 will consider and propagate deletions of $\frac{9k}{10}$ tuples although only about $\frac{k}{10}$ tuples remain. Thus, it is more effective to reset the constraint with the elements of $S(x, a)$, in other words, to rebuild the constraint from scratch. In this situation, we can restart from a tuple set of only $\frac{k}{10}$ tuples and save a factor of 9.

We can determine exactly when it is worthwhile to apply such an operation. When the sum of the sizes of S lists of the deleted values of x is larger than the sum of the sizes of S lists of the remaining values in the current domain of x.

The Reset The idea of reset had already been applied to define which algorithm should be preferred between AC-2001 and AC-6 [Bessière 2001] or to design an adaptive algorithm [Régin 2005]. Combining GAC-4 with this idea will save a lot of computations for a shallow depth of the tree search. Such a combination requires to answer two questions:

1. How can we know whether a reset is better or not?

2. How can we perform this reset and the restoration of the previous set efficiently?

We can simply answer the first question. Consider a variable x and $\Delta(x)$ the set of values of $D(x)$ that have been deleted and not yet considered by GAC-4 algorithm (i.e. they belong to deletionSet). The number of tuples that are no longer valid, denoted by $\#T\Delta(x)$, is given by the sum of the size of the S lists of the values in $\Delta(x)$, and the number of remaining tuples is the difference between the total number of tuples and $\#T\Delta(x)$ because a tuple contains only one value per variable. So we have:

Property 8 *Let x be a variable, $\Delta(x)$ be the values of x that have been deleted and that must be propagated, $\#T\Delta(x) = \sum_{a \in \Delta(x)} |S(x, a)|$ be the number of tuples that are no longer valid and T be the current number of tuples.*
If $\#T\Delta(x) > \frac{T}{2}$ then a reset operation will consider less tuples than the application of Function REVISEGAC-4.

This property is useful only if we can answer the second question. A good answer to this second question is the use of sparse sets for efficiently computing a reset operation in such a way that the restoration is easy.

The question can be reformulated as follows. Consider S a set with two lists of elements R and Q. The lists are disjoint and their union contains exactly

Algorithm 19 Function re-add of a sparse set S. k is an element.

RE-ADD(k, S) // We assume that $S.\texttt{dense}[S.\texttt{sparse}[k]] = k \quad ik \leftarrow S.\texttt{sparse}[k]$;
$e \leftarrow S.\texttt{dense}[S.\texttt{members}]$
$S.\texttt{sparse}[k] \leftarrow S.\texttt{members}$
$S.\texttt{sparse}[e] \leftarrow ik$
$S.\texttt{dense}[S.\texttt{members}] \leftarrow k$
$S.\texttt{dense}[ik] \leftarrow e$
$S.\texttt{members} \leftarrow S.\texttt{members} + 1$

all the elements of S. The sets R and Q are not explicitly given, that is, we do not have a set representing them but we can traverse them (in terms of programming language, they are given by an iterator) and their size is known. We want to modify S by removing the set of elements $R \subseteq S$ in order to obtain a set containing only the elements of Q. However, instead of performing $|R|$ operations for this task, we want to have a number of operations bounded by $\min(|R|, |Q|)$. In addition, we have to be able to restore the set S after performing the modifications with a similar complexity (or less).

Let S be represented by a sparse set. If $|R| \leq |Q|$, then we delete the elements of R from S as it is explained in the sparse set section. The restoration of S consists of modifying the scalar `members` of S. Assume that $|Q| < |R|$. S is recomputed as follows. First, we set $S.\texttt{members}$ to 0. Then, we traverse Q and for each element $a \in Q$ we add a to S by calling Function RE-ADD (See Algorithm 19) which is a modified version of Function ADD of the sparse set. It exploits the fact that the value which is added was previously in the set. Thus, it proceeds to a swap in a way similar as the one used by Function DELETE in order to be able to restore the set in the future. More precisely, when an element i is re-added to the sparse set, we swap i and the value j at the index defined by `members`. That is, we exchange the value of i and the value of j in the `sparse` array and we exchange i and j in the `dense` array. For instance, consider the left sparse set in Figure 10.7. The set contains the values 3 and 5. If we re-add the value 8 then we will exchange the value of $\texttt{dense}[\texttt{members}]$, i.e. 1, with 8. So we will have $\texttt{dense}[2] = 8$; $\texttt{dense}[4] = 1$; $\texttt{sparse}[8] = 2$; $\texttt{sparse}[1] = 4$. We obtain the right sparse set.

The advantage of this method is that the restoration of the scalar `members` is enough for restoring the sparse set. Function RE-ADD has a complexity of $O(1)$ per call. Thus, we can re-add $|Q|$ elements in $O(|Q|)$.

A possible implementation of GAC-4R is given by Algorithm 20. Each list S is represented by a sparse set with a fixed size universe equal to $|T(C)|$. For convenience, we will consider that t is a tuple and also the index of the tuple in the table of tuples.

sparse	3	2	-	0	-	1	-	5	4	-
dense	3	5	1	0	8	7				
members	2									

sparse	3	4	-	0	-	1	-	5	2	-
dense	3	5	8	0	1	7				
members	3									

Figure 10.7: The first sparse set contains the values 3 and 5, while the second one is the re-add of the value 8.

The complexity of GAC-4R remains the same as the complexity of GAC-4, because the deletion of a tuple or the re-addition of a tuple have the same complexity which corresponds to the arity of the constraint. In addition traversing the valid tuples costs at least the cost of traversing all the domains of the variables involved in the constraint and since we do this only when there are less valid tuples than non-valid tuples, the traversal of all the domains does not impact the complexity.

10.4 MDD4R: MDD Propagator

This section proposes to adapt the principles of GAC-4R to be able to deal with an MDD instead of a table. While the graph-based propagator for regular constraints was already close to GAC-4, the MDD-4 algorithm presented here is another way of implementing an MDD version of GAC-4 for MDD. Thanks to this modification, the algorithm MDD4R will be designed by incorporating the efficient idea of *reset*.

10.4.1 MDD4 Algorithm

The algorithm MDD-4 is a modification of GAC-4 for dealing with MDDs. This algorithm maintains the whole MDD during the search. This implies that for each node n, MDD4 maintains the $\omega^+(n)$ and $\omega^-(n)$ lists of outgoing and incoming arcs. Moreover MDD4 maintains for each value a of each variable x the list $S(x, a)$ of valid arcs labeled by a at the layer of x.

In GAC-4 the maintenance of the list of valid tuples is made by managing the S lists. With an MDD this is more complex, because the tuples are not explicitly represented in an MDD. The representation is implicit: a valid tuple corresponds to a path from the root node to the `tt` node, traversing only arcs

Algorithm 20 GAC-4R. T is the current number of tuples

```
REVISEGAC-4R(C: constraint; deletionSet: list, T: number of tuples): Boolean
    for each x ∈ X(C) do #TΔ(x) ← 0
    for each (x, a) ∈ deletionSet do #TΔ(x) ← #TΔ(x) + |S(x, a)|
    #TΔmax ← max_{x∈X(C)}(#TΔ(x))
    if #TΔmax > T/2 then
        // we reset the data structures
        pick a variable x with #TΔ(x) = #TΔmax
        Tset ← ∅; T ← 0
        for each a ∈ D(x) do add S(x, a) in Tset
        for each y ∈ X(C) do
            for each b ∈ D(y) do S(y, b).members← 0
        // we re-add valid tuples into the S lists
        for each t ∈ Tset do
            if t is valid then
                for each i = 1...n do RE-ADD(t, S(x_i, t[i]))
                T ← T + 1
        // We remove values having an empty S list.
        for each y ∈ X(C) do
            for each b ∈ D(y) do
                if S(y, b) = ∅ then remove b from D(y)
                if D(y) = ∅ then return False;
    else
        // classical GAC-4 deletion process
        for each (x, a) ∈ deletionSet do
            for each t ∈ S(x, a) do
                for each i = 1...n do
                    DELETE(t[i],S(x_i, t[i]))
                    T ← T − 1
                    if S(x_i, t[i]) = ∅ then remove t[i] from D(x_i)
                    if D(x_i) = ∅ then return False;
    return True
```

corresponding to valid values. That is why we keep the list of valid arcs since a valid arc belongs to at least one valid tuple.

Since MDD4 is a GAC-4 like algorithm, most of the work occurs when

values are removed. Consider a variable x whose Δ set of values has been removed and has to be propagated. The MDD4 algorithm removes all the arcs from layer x labeled by the values in Δ. Then all the arcs that do not belong to a valid path anymore are removed from the ω and S lists.

In order to do that efficiently, the algorithm use two queues $Q\uparrow$ and $Q\downarrow$. They contain the nodes that do not belong to a valid path anymore. Nodes for which all the outgoing or incoming arcs have been removed. Each time an arc (i, j) is removed, if it was the last outgoing arc of node i then i is pushed into $Q\uparrow$, if it was the last incoming arc of node j then j is pushed into $Q\downarrow$.

The algorithm processes as follows: First all the arcs of the values in the Δ set are removed from the MDD. Then, while $Q\uparrow$ is not empty, all the arcs from the nodes of $Q\uparrow$ are removed. Finally, while $Q\downarrow$ is not empty, all the arcs from the nodes of $Q\downarrow$ are removed. The algorithm 21 is a possible implementation of the MDD4 algorithm.

Implementation Both the S list and the ω lists are implemented using sparse sets. This allows an efficient backtracking, and in an analogous way with GAC-4R, it will allow us to apply the reset idea.

Differences The MDD4 algorithm is close to the graph-based propagator used for filtering the regular constraint. They differ by the use of a BFS instead of a DFS. Furthermore the S list contains the valid arcs and no longer the nodes, this implies that MDD4 (and so MDD4R) is able to deal with non-deterministic MDDs. Finally, the MDD4 algorithm propagates the deletion into the MDD when all the deletions have been made and not at each arc deletion. This last property will allow us to reset.

Example:

Consider M, the MDD from Figure 10.1. Let the constraint C defined on the variables x_1, x_2 and x_3 and using the MDD M. Consider that, for external causes, the second variable x_2 is set to b. The algorithm is going to perform the following steps: First for variable x_2 MDD4 removes the arcs from the list $S(x_2, a)$, The arc $(0, 2, a)$ is removed. The arc $(1, 3, a)$ is removed. Then for the list $S(x_2, c)$, the arc $(0, 2, c)$ is removed and 2 is pushed into $Q\downarrow$. The arc $(1, 3, c)$ is removed and 1 is pushed into $Q\uparrow$. The algorithm has removed all the arcs of the deleted values, now the algorithm is going to propagate the Q queues. First the node 1 is pop from $Q\uparrow$. The arc $(r, 1, b)$ is removed from r, removed from $S(x_1, b)$ which becomes empty and thus b is removed from x_1. The arc $(r, 1, c)$ is removed from r, removed from $S(x_1, c)$ which becomes empty and thus

Algorithm 21 MDD-4.

```
removeArc(MDD, Q↓, Q↑, (i, j)):Boolean
    delete the arc (i, j) from the MDD
    if ω⁺(i) = ∅ then push node i into Q↑
    if ω⁻(j) = ∅ then push node j into Q↓
    (y, b) ← pair (variable, value) of the arc (i, j);
    remove the arc (i, j) from the S(y, b)
    if S(y, b) = ∅ then remove b from D(y)
    return (D(y) ≠ ∅)
reviseMDD-4(C: constraint; Δ: list of values; x : variable): Boolean
    Q↓← ∅; Q↑← ∅
1   for each (x, a) ∈ deletionSet do
        for each arc (i, j) ∈ S(x, a) do
            if ¬ removeArc(MDD, Q↓, Q↑, (i, j)) then return False
2   while Q↑≠ ∅ do
        pick the node i ∈ Q↑ with the lowest layer
        for each arc (j, i) do
            if ¬ removeArc(MDD, Q↓, Q↑, (j, i)) then return False
        remove i from Q↑
3   while Q↓≠ ∅ do
        pick the node j ∈ Q↓ with the highest layer
        if there is no incoming arc to j then
            for each arc (j, i) do
                if ¬ removeArc(MDD, Q↓, Q↑, (j, i)) then return False
        remove j from Q↓
return True
```

c is removed from x_1. The node 2 is pop from $Q\downarrow$. The arc $(1, tt, a)$ is removed from tt, removed from $S(x_3, a)$ which becomes empty and thus a is removed from x_3.

The arcs processed by MDD4 are the dashed arcs of the left MDD from Figure 10.8.

10.4.2 MDD-4R

MDD-4 can be improved by integrating the idea of resetting the data structures instead of being focused only on the deletions. MDD-4 works by layer in the MDD, that is, variable per variable. Let $\#A(x)$ be the total number of arcs associated with a variable x. This number is stored and maintained for each variable.

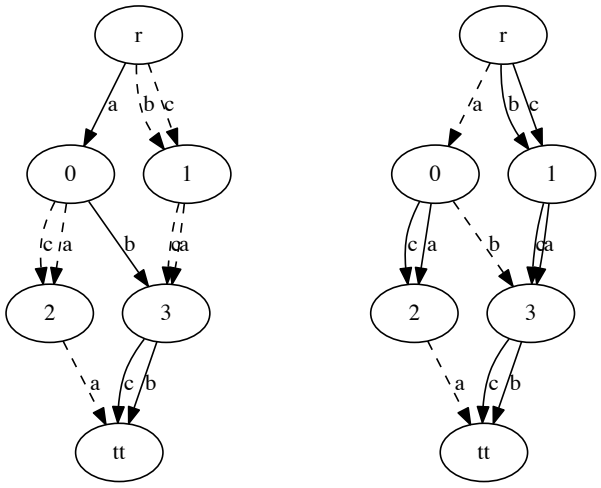


Figure 10.8: On the left, the application of the MDD4 propagator for processing the decision $x_2 = b$. On the right, the application of the MDD4R algorithm. The dashed arcs are the arcs touched by the propagator.

For a given layer corresponding to the variable x, we have to compute $\#DA(x)$ the number of arcs that will be deleted for this layer. By comparing this number to $\#A(x)$ we will know whether it is better to reset the layer or not. Resetting the layer means that we rebuild the layer of the graph from the remaining nodes by adding their remaining arcs instead of deleting the arcs and nodes of the layer.

The computation of $\#DA(x)$ depends on the type of modification occurring in the MDD. There are two kinds of modifications:

- First: the arcs corresponding to the deletions of values of a variable x are removed. In this case, we have $\#DA(x) = \sum_{a \in \Delta(x)} |S(x, a)|$. This happens only once.

- Second: the consequences of the deletion of arcs and nodes in the MDD are propagated, in other words, the MDD is maintained. MDD-4R proceeds by layer. For a given variable y, MDD-4 has the list $Q(y)$ of nodes that must be deleted, which is for the given layer the content of the queue $Q\uparrow$ or $Q\downarrow$ depending on the sense of propagation. We have $\#DA(y) = \sum_{u \in Q(y)} |\omega(u)|$, where $\omega(u)$ is the list of arcs associated with y having u as extremity.

Property 9 *Let x be any variable, If $\#DA(x) > \frac{\#A(x)}{2}$ then a reset operation for the layer of x will consider less arcs than the application of MDD-4 for this layer.*

Note that this property computes exactly whether it is better to reset or not the data structure.

Example:

Consider M, the MDD from Figure 10.1. Let the constraint C defined on the variables x_1, x_2 and x_3 and using the MDD M. Consider that, for external causes, the second variable x_2 is set to b. The algorithm is going to perform the following steps: The algorithm sums the number of arcs to delete which is 4, 4 is greater than 5/2=2.5 so MDD4R is going to apply a reset. The set of nodes of the layer 1 and 2 are cleared. The arcs $(0, 3, b)$ put back the node 0 in the layer 1 and node 3 in layer 2 and clear their arcs list. $(0, 3, b)$ is put back into $\omega^+(0)$ and $\omega^-(3)$. For layer 1, the number of arcs to restore is 1 and the number of arcs to delete is 2 so the algorithm chooses to reset. The layer 0 is cleared and the S lists of the values of x_1 are cleared. The arc $(r, 0, a)$ puts back r in layer 0 and is put back into the $S(x_1, a)$ list. The algorithm iterates over the S lists, since the lists $S(x_1, b)$ and $S(x_1, c)$ are empty, the values b and c are removed from x_1. Then the algorithm considers the layer 2 in down propagation, there are 2 arcs to keep and 1 arc to delete: the algorithm chooses to delete the arc. The arc $(2, tt, a)$ is removed from both $\omega^-(tt)$ and $S(x_3, a)$ which becomes empty so a is removed from x_3.

The arcs processed by MDD4R are the dashed arcs of the right MDD from Figure 10.8. The figure 10.9 shows the processed arcs for all the MDD propagators.

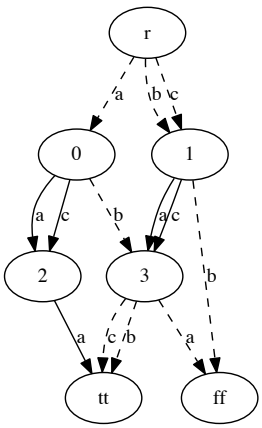
10.4.3 Improvements

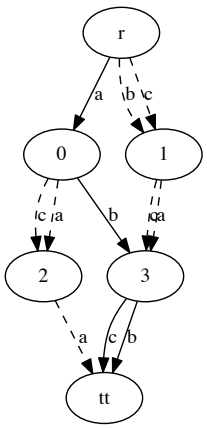
Counting on the fly The processing of the sum $\#DA(y) = \sum_{u \in Q(y)} |\omega(u)|$ can be made while putting the nodes on the set Q, and during a reset, by considering the re-added nodes.

Reverse counting When a variable is set to a single value a, or even in the general case of modification, it can be worthwhile to count the number of arcs to remove by considering the smallest set of values.

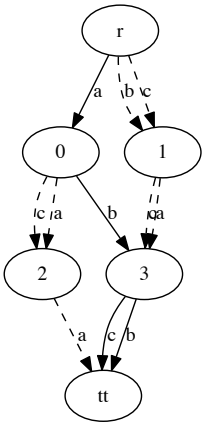
One side deletion The deletion of an arc can be triggered by two actions: The deletion of a value or the Deletion of a node. When an arc has to be

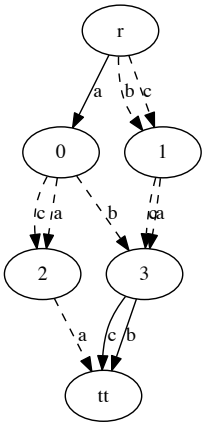
(a) mddc (b) graph-based





(c) MDD4 (d) MDD_w (e) MDD4R





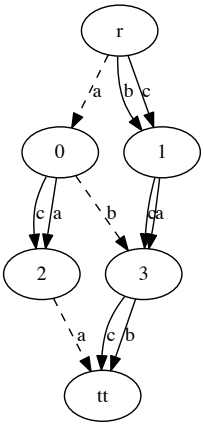


Figure 10.9: The touched (dashed) arcs during the processing of the decision $x_2 = b$ for all the MDD propagators.

removed because the node of one of its extremities has to be removed, then the algorithm can avoid removing the arc from the deleted node.

MDD4R-Last When an arc has to be removed from a node and this arc was the last support of this node, then the algorithm can prevent this deletion. In order to do that efficiently, when the algorithm has to delete an arc, the algorithm checks if it is the last arc of the extremity, in this case it adds the arc into the Q set, otherwise it removes the arc.

Some of these optimization are already implemented into the CP solvers [OscaR Team 2012, Perron 2013] having the MDD4R propagator.

10.5 Experiments

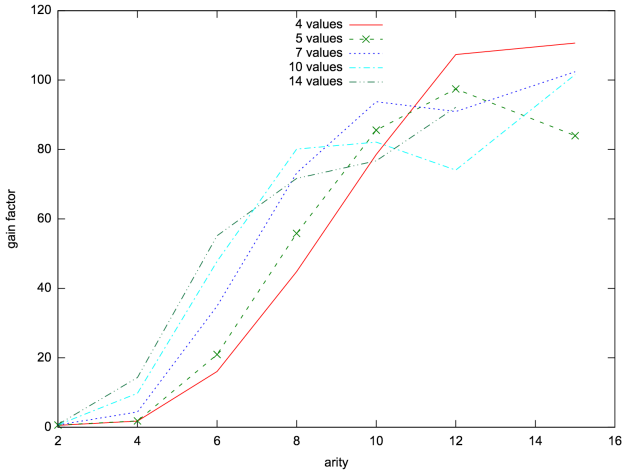
Maxorder This experiment comes from chapter 17. The problems is composed of two constraints, an allDifferent constraint and an MDD constraint associated with an MDD with more than 1 millions of nodes and close to 200 millions of arcs for the size 20. The following table gives the time for finding the 50 first solutions.

size	MDD4R	*mddc*	mdd_w
9	6	3021	T-O
20	26.8	T-O	T-O

This tables shows that MDD4R is robust to huge MDDs.

Ternary decomposition It is often considered (See [Beldiceanu 2004a, Quimper 2006]) that the best way for maintaining arc consistency for regular constraint is to decompose the constraint into a set of ternary transition constraints and to directly deal with them. We propose to compare this model with the explicit use of the MDD corresponding to the automaton of the regular constraint in conjunction with MDD4R algorithm. The MDD is reduced.

We use constraints defined by transition constraints involving 8,000 tuples. The following figure gives the factor of gains of the use of a MDD + MDDR4 in comparison with transition constraints + GAC4R and clearly shows the advantage of our approach.



We also compare the two approaches on a problem with 5 random constraints and one knapsack constraint imposing that the sum of all variables must be greater than a value k (usually defined as the mean of the domains). The results given in the following table should a gain factor of 1.4:

Arity	dom size	1 sol		all sol	
		Ternary	MDD4R	Ternary	MDD4R
8	6	0.7	0.3	18.4	12.2
8	8	0.8	0.5	25.1	17.2
10	8	0.9	0.4	44.3	31.8
10	10	1.3	0.7	58.4	41.3
12	10	2.3	1.5	89.2	66.4
12	12	4.2	2.8	109.6	82.5

10.5.1 CP14 experiments

The following experiments are the ones made in 2014, at the moment of the publication. Since, several modification of algorithms have been made, for example the *allowed* is now the *CT* algorithm published in 2016.

Machine: Dell server having four E7- 4870 Intel processors, each having 10 cores with 256 GB of memory and running under Scientific Linux.

Solver: or-tools 3158.

Selected Benchmarks: all problems can be downloaded from the Solver Competition archive [Lecoutre 2009]. We selected problems having only positive table constraints and at least one variable whose domain is not Boolean. We do not include BDD-based instances and instances involving only binary constraints because we are interested in large arity constraints. rand-8-20-5-18-800 is abbreviated by rand-1 and rand-10-20-10-5-10000 is abbreviated by rand-2. half-n25-d5-e56-r7-1 is abbreviated by half-1 and contains the problems of half-n25-d5-e56-r7 that are solved by `mddc` in less than 1800s and half-n25-d5-e56-r7-2 is abbreviated by half-2 and contains the problems of half-n25-d5-e56-r7 that are not solved by `mddc` in less than 1800s.

We also used random problems that we defined. One of the most difficult parameter to define is the tightness of the constraint that is, the ratio between the number of tuples allowed by the constraint and the total number of tuples. We use ratio from 0.00004% to 1%. For a comparison, we can note that an alldiff constraint defined on a set of k variables sharing the same k values is equal to $\frac{k!}{k^k}$ which is 0.6% for $k = 7$; 0.034% for $k = 10$ and 0.0003% for $k = 15$.

Search Strategy: we select the variable that appears in most constraints and its smallest value as proposed for testing `mddc` [Cheng 2010]. Our algorithm is more robust than the Cheng's one for the strategy, because we maintain the MDD whereas they traverse the initial MDD according to the current domains. Thus `mddc` algorithm can lose time for finding an arc corresponding to a value belonging to the current domain of the associated variable. This problem does not arise in neither MDD-4 nor MDD-4R.

Table 10.2: Geometric means of the time needed to solve some categories of problems.

benchmark	MDD-4	MDD-4R	`mddc`	GAC-4	GAC-4R	STR2	STR3	allowed
nonograms	0,33	**0,27**	0,8	4,3	3,18	2,77	1,52	1,03
cw-m1c-ogd	3,07	**1,72**	32,69	4,03	2,74	13,21	2,69	3,09
cw-m1c-uk	3,96	1,91	21,14	3,24	2,27	8,32	**1,89**	2,22
rand-1	6,06	2,93	13,71	2,90	1,38	1,56	1,93	**1,09**
rand-2	192,47	**50,51**	186,35	241,56	170,36	141,03	T-O	T-O
half-1	975,49	**471,25**	1438,20	T-O	T-O	T-O	T-O	T-O
half-2	1720	**778,28**	T-O	T-O	T-O	T-O	T-O	T-O

Results: times are expressed in seconds. Time Out (T-O) is set at 1800s. All means are geometric.

General comparison Table 10.2 gives results for the selected benchmarks. MDD-based algorithms perform well in general. Algorithm `mddc` is clearly improved by MDD-4 and MDD-4R algorithms. GAC-4R outperforms GAC-4 and is competitive with other GAC algorithms for table constraints. The reset strategy is quite interesting and MDD-4R clearly outperforms all the other algorithms.

We propose to study in detail the behavior of these algorithms according to the tightness and the domain size. For each graph we select randomly 10 problems and run them 30 times and take the mean.

10.6 Conclusion

This chapter presents several new algorithms, for handling both table constraints and MDD constraints. These algorithms have good results in practice. With the recent development of algorithm like CT [Demeulenaere 2016] or STRbit [Wang 2016], more experiments should be done. Usually, when the problem can be compressed, using an MDD, then using an MDD instead of a regular table constraint can improve the running time. Otherwise, the choice of the table algorithms depends on the table structure (number of variables, of values). Many problem, they can't even be defined by a table (see Application part of this thesis). For this kind of problem, the choice depend once again of the structure of the MDD, if the MDD is not too big and contains few values, then `mddc` can be really effective. When it starts being bigger, with many

values etc, then using MDD4R can lead to better performances.

Chapter 11

Cost-MDD constraint

Contents

11.1 Introduction

In constraint programming, cost constraints are constraints associating a cost to each combination satisfying the constraint. This cost can be processed in several ways and mainly depends on the constraint.

Consider the cost version of the Global Cardinality Constraint (GCC) [Régin 2002]. In this constraint the cost is given by a function associating a cost to each pair variable value. Thus the cost of an assignment is the sum of the cost of the pair variable value of the assignment. The cost can be handled by giving a cost-variable to the constraint in addition to its main variables, but also by giving a maximum or/and a minimum cost to the constraint like the knapsack constraint [Trick 2003].

In constraint programming, cost version of constraints are widely used, for example the cost versions of the global constraint of difference [Sellmann 2002]

or the regular constraint are used in time-tabling problems [Demassey 2006]. Moreover, cost constraints are often used in scheduling or vehicle routing problems [Quimper 2010, Malapert 2008, Houndji 2014, Dejemeppe 2015, Vilím 2004].

One of the main advantages of having efficient cost-MDD constraints filtering algorithms is that cost-MDDs offer cost versions for table constraints and any constraints which can be represented by an MDD (regular, slide, knapsack...).

Furthermore, the main motivation in this thesis for building cost version of MDDs is an application consisting on the generation of text and music from a corpus while avoiding plagiarism [Papadopoulos 2014, Perez 2015a]. The goal of this problem, named `maxOrder`, is to generate sequences of words, where for example, each subsequence of size two belongs to the corpus (Markovian transition) and no subsequence of size 4 belongs to the corpus. Here 4 denotes the maximum plagiarism size. While the chapter 17 is dedicated to this problem, the experimental section of this chapter contains some of the experiments using the cost-MDD propagator proposed here.

The cost version of an MDD is an MDD whose arcs have an additional information: the cost of the arc. Several state of the art papers about decision diagram [Lai 1992, Gange 2013] use the name *value* instead of the *cost* of an arc, but in constraint programming, the term *value* is ambiguous and so replaced by *cost*, which is widely used for these kinds of constraints.

In a cost-MDD, each path from the top layer to the bottom layer has a cost, which is the sum of the cost of its arcs, and a cost-MDD propagator aims at bounding the maximum or minimum path cost passing through an arc. Cost-MDDs are useful to model optimization problems [Bergman 2016a, Bergman 2011] or dynamic programming problems [Hooker 2013].

Example:

Consider the cost-MDD from Figure 11.1. The arcs of this MDD can be defined by a 4-uplet (origin, destination, label, cost). For example, the arc (r,1,*a*,1) is the arc emanating from the root node, directed to node 1, labeled by *a* and with a cost of 1. The cost of the path (*b*,*b*,a), composed of the arcs (r,2,*b*,2), (2,5,*b*,2) and (5,`tt`, *a*, 2) is $2+2+2=6$.

Several cost-MDD propagators already exist [Demassey 2006, Gange 2013, Andersen 2007, Hoda 2010], they consist in the modification of existing MDD filtering algorithms allowing to handle in addition to the propagation of the deletion, the cost propagation inside the MDD. They will be presented in the next section.

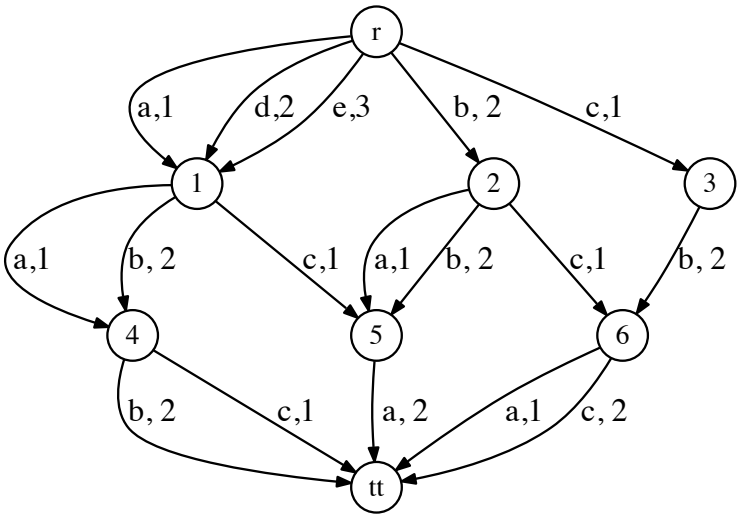


Figure 11.1: Cost-MDD defined over three variables. Each arc is associated with both a label and a cost.

As presented in chapter 10, MDD4R is a powerful propagator for MDD constraints. For some industrial instances, MDD4R improves on previous propagators by a speed factor of up to 500. Hence, this chapter proposes an adaptation of the propagator MDD4R to process cost-MDDs and shows that good speed up are also observed for cost-MDD4R compared to existing methods.

11.2 Cost-MDD

11.2.1 Definition

A cost-MDD is an MDD whose arcs have additional information: the cost c of the arc. That is, an arc is a 4-tuplet $e = (u, v, a, c)$, where u is the emanating node, v the terminating node, a the label and c the cost. Let M be a cost-MDD and p be a path of M. The cost of p is denoted by $\gamma(p)$ and is equal to the sum of the costs of the arcs it contains. A shortest path of M is a path of M whose cost is minimum. A shortest path of an arc e, denoted by $p_{min(e)}$, is a path such that there is no path of M containing e having a smaller cost.

11.2.1.1 Cost-MDD of a constraint

For a definition of the MDD of a constraint, see chapter 10. Let C be a constraint and f_C be a function associating a cost with each value of each variable of $X(C)$. The cost-MDD of C and f_C is denoted by cost-MDD(C, f_C) and is the MDD(C) in which the cost of an arc labeled by a at layer i is $f_C(x_i, a)$.

Remark Cost-MDDs are not only built from existing constraints, and that, in the general case, the cost of two arcs from the same layer and with the same label can be different.

11.2.1.2 Cost-MDD propagator

A cost-MDD propagator for the constraint C associated with the cost-MDD M, a value H, and a symbol $\prec$ (which can be $\leq$ or $\geq$) is a propagator for C which ensures that it exists a path p such that $a \in p$ and $\gamma(p) \prec H$.

A cost-MDD propagator establishes arc consistency of C iff for each value of each variable it exists an arc labeled by the value at the layer of the variable in cost-MDD(C) that belongs to p a valid path of $path^r_{tt}$(cost-MDD(C)) with $\gamma(p) \prec H$[1]. For the sake of clarity, only the case $\leq$ is considered here, the other case is equivalent.

11.2.2 Related Work

The first cost-MDD propagator came from the regular constraint, so it has been named the *cost-regular* constraint propagator [Demassey 2006] and is the adaptation of the graph-based MDD propagator used for classical regular constraints [Pesant 2004] (see chapter 10). Then two filtering algorithms for the cost-MDD constraint have been provided [Andersen 2007, Gange 2013], consider a cost-MDD propagator named here $cost{-}mdd_w$ since it is the adaptation of the mdd_w MDD constraint propagator.

All of these propagators are based on the same idea of processing and/or maintaining `up[u]`, the shortest path cost between the *root* node and every node u, and `dn[u]`, the shortest path cost between each node u and the terminal *tt* node. Thanks to this information, an arc $e = (u, v, a, c)$ is safely deleted when $up[u] + dn[v] + c > H$.

These algorithms use modified versions of their own MDD/regular propagators to handle this new deletion and propagation on the cost-MDD. Basi-

[1]Note that the definition given in [Perez 2017c] is wrong. Also the definition saying for establishing arc consistency the algorithm needs to ensure that for each arc it exists a path whose cost is lower or equal to H is stronger than the arc consistency for the constraint but ensures the MDD consistency [Hoda 2010] since each arc belongs to a feasible solution.

cally, for a cost-MDD constraint C, when a variable of $X(C)$ is modified, the arcs labeled by the deleted values have to be removed, and, if a node lost all its incoming or outgoing arcs, it has to be removed. When a node is removed, all its remaining arcs have to be removed. This behavior is classical for MDD propagators, but in addition, if a value `up[u]` or `dn[u]` of a node u is modified, because of the arcs deletion, then the new value has to be propagated.

An algorithm maintaining during the search the additional information which is the costs (`up` and `down`) of a node during the search can propagate the costs modification by performing a DFS or a BFS starting at the modification and updating the `up` and `down` values of the impacted nodes. This is what the existing algorithms do.

Example:

Consider the MDD from Figure 11.1. Let the maximum cost H be 5. The first step is to build the shortest path cost of each node u (`up[u]` and `dn[u]`).

Node	r	1	2	3	4	5	6	tt
up	0	1	2	1	2	2	3	3
dn	3	2	2	3	1	2	1	0

Then for each node, we can determine the cost of the shortest path passing through each arc $e = (u, v, a, c)$ by processing $\texttt{up}[u] + c + \texttt{dn}[v]$. Thus the arc $(1, 4, b, 2)$ as a minimal cost of $\texttt{up}[1]+2+\texttt{dn}[4] = 1+2+1 = 4$, which is lower than 5. But the arc $(2, 5, b, 2)$ has a cost of $\texttt{up}[2]+2+\texttt{dn}[5] = 2+2+2 = 6$, which is greater than 6, Thus we can safely remove this arc since no solution with an acceptable cost pass through it.

Two-sided inequalities Another algorithm has been provided for handling the two-sided inequalities of the cost of an MDD [Hoda 2010]. The main difference between this algorithm and the others is that it stores in each node all the possible path costs. Let u be a node of the MDD, let $Up[u]$ be the set of different costs of paths reaching u from the root, let $Dn[u]$ be the set of different costs of paths reaching tt from u. An arc $e = (u, v, a, c)$ can be safely removed when:

$$\forall up \in Up[u], \forall dn \in Dn[v], up + c + dn > H \tag{11.1}$$

While this chapter first considers the one-sided inequality, it then describes a method enforcing arc consistency for the cost equal to a set of distinct values and thus will be compared to this algorithm.

11.3 Cost-MDD4R

This section proposes Cost-MDD4R, a modified version of the MDD4R algorithm for establishing the arc consistency of a constraint C represented by cost-MDD(C). But, in order to deal with costs, in the same ways as the existing cost-MDD propagators, cost-MDD4R adds and maintains for each node u, the value `up[u]` and `dn[u]`.

While a propagator dealing with classical MDD constraints only has to manage variables modifications, a propagator dealing with cost-MDD constraints also needs to handle cost modifications. The description of the cost-MDD4R is split into two parts, one for each of these modifications.

11.3.1 Variable Modification

When the domain of a variable is modified, cost-MDD4R performs the same work as MDD4R, which is maintaining for each value of each variable its set of valid arcs and for each node its set of valid incoming and outgoing arcs. But in addition it maintains for each node u the values `up[u]` and `dn[u]`.

To do so, the algorithm marks the modified nodes and, between the work made layer by layer, it updates the cost of the modified nodes, and deletes the arcs that just became invalid. Then, it continues this cost propagation, layer by layer, even if no more arcs have to be deleted by MDD4R.

To avoid unnecessary work, the algorithm only marks nodes whose value is equal to the value brought by the deleted arc. For example, if we remove the arc $e = (u, v, a, c)$, we mark u only if `dn[u]` $= c +$ `dn[v]` and we mark v only if `up[v]` $= c +$ `up[u]`.

Remark: The dn value of the root and the up value of the tt node contain the lower bound of the cost of the constraint, which is the shortest path cost of the cost-MDD constraint considering the current domain of the variables.

Reset While the previous definition and behavior can already be applied to existing algorithms, the main advantage of MDD4R is the efficient use of a reset. If a reset is performed (i.e. fewer valid arcs than invalid arcs), then the algorithm is able to reset the values `up` and `dn` while putting back the arcs.

An interesting remark is that the arcs propagating a cost propagation will always be processed during the propagation. This means that if the algorithm performs a reset or a deletion, in both cases the arcs will be processed, thus the choice of the reset of the arcs has to avoid these arcs. We can consider that the nodes are partitioned into three partition:*unchanged, modified, removed.*

Algorithm 22 cost-MDD4R Propagate.

```
PROPAGATE(mdd, x : variable, δ : values)
    for each value a ∈ δ do
        for each e = (u, v, a, c) ∈ S(x, a) do
            if dn[v] + c = dn[u] then
                put u in Up_c
            if dr[u] + c = dr[v] then
                put v in Down_c
            remove(e)
    for each Layer i ∈ layer(x)..0 if Q↑ ∪ Up_c ≠ ∅ do
4       Up_c ← checkModified↑(Up_c)
        if Should Reset then
            Put back all the arcs not in Q↑
        else
            Remove all the arcs from Q↑
5           Propagate the arcs from Up_c
    for each Layer i ∈ layer(x) + 1..r if Q↓ ∪ Down_c ≠ ∅ do
        Down_c ← checkModified↓(Down_c)
        if Should Reset then
            Put back all the arcs not in Q↓
        else
            Remove all the arcs from Q↓
            Propagate the arcs from Down_c
```

Algorithm Let Up_c be the set of nodes whose `dn` may have changed and let $Down_c$ be the set of nodes whose `up` may have changed. Just like MDD4R, the algorithm decides for each layer to reset by considering the arcs to remove and the remaining arcs. When an arc in the *modified* partition is put in the *removed* partition, this implies that the node is removed from the set Up_c (or $Down_c$) and put into $Q\uparrow$ (or $Q\downarrow$). The Algorithm 22 is a possible implementation of this propagation algorithm. The line 4 checks for each node which has lost one of its incoming arcs bringing the best `dn` value if it was the last one, and so modify its `dn`, otherwise it removes the node from the set Up_c. The algorithm 23 is a possible implementation of the update of the Up_c set (checkModified↑).

The line 5 from Algorithm 22 first tests for each arc which is propagating

Algorithm 23 cost-MDD4R nodes cost update.

```
checkModified↑(Up_c : setofnodes)
    for each node u ∈ Up_c do
        dn_tmp ← inf
        for each arc e = (u, v, a, c) ∈ ω⁺(u) do
            dn_tmp ← max(dn[v]+c, dn_tmp)
            if dn_tmp = dn[u] then break
        if dn_tmp ≠ dn[u] then
            dn[u] ← dn_tmp
        else
            Remove u from Up_c
```

a new cost if this new cost is lower than H. If the cost is strictly greater, then the arc is removed, otherwise, if the emanating node has a `dn` value equal to the previous `dn` value of the terminating node plus the cost of the arc, then the emanating node is added to the Up_c set for the next layer. This part is close to the first lines of the algorithm.

Example:

Consider the cost-MDD from Figure 11.1. Consider a cost-MDD constraint defined for the three variables (x_1, x_2, x_3), with a $H = 5$. At first, as shown in the previous example, the arc $(2, 5, b, 2)$ is removed. We obtain the MDD from Figure 11.2 and the `up` and `dn` values:

Node	r	1	2	3	4	5	6	tt
`up`	0	1	2	1	2	2	3	3
`dn`	3	2	2	3	1	2	1	0

Consider the assignment of the third variable x_3 (the variable on the last layer) to the value b. cost-MDD4R first count the number of arc to remove and to keep at the layer of x_3. We have four arcs to remove and only one to keep, thus cost-MDD4R perform of reset. It consider all the nodes a the layer of its emanating nodes as potentially removed (nodes 4,5 and 6). Then it puts back the arc (4,`tt`, b, 2). This arc set the node 4 as valid. since the node 5 and 6 have not been set as valid, they are considered as removed. The previous `dn` value of 4 was 1, but the only puted back arc have a cost of two so the `dn` cost of 4 is now 2. For `tt`, the value `up` was 3, but its only value is now 4.

Both the deletion and the cost modification of 4 have to be propagated.

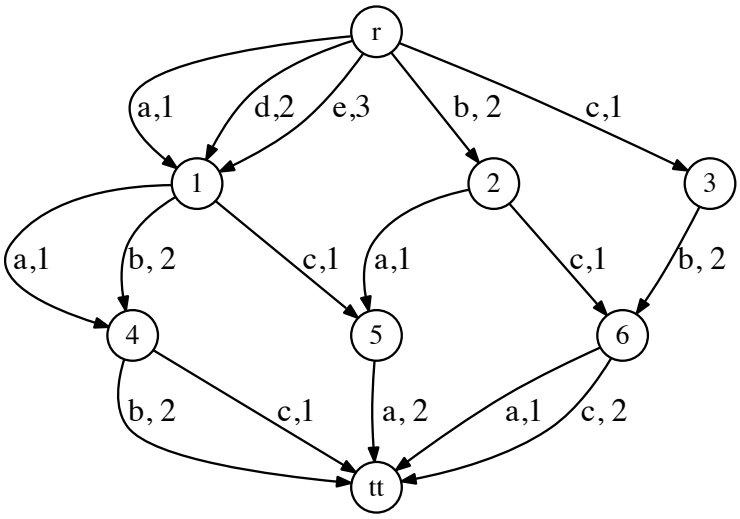


Figure 11.2: Cost-MDD defined over three variables. Removing of the arc $(2, 5, b, 2)$ from the cost-MDD of figure 11.1.

there are four arcs to remove and 2 arc to keep, the algorithm reset again. It puts back the arcs (1,4,a,1) and (1,4,b,2), since the cose of 4 has changed, it impact the cost of 1, which is set to 3 while putting back the arcs. 1 is put in the modified nodes set Up_c.

For the layer of variable X_1, there are 2 arcs to remove and three to keep, thus the algorithm choose to remove the arcs. The arc (r,2,b,2) is removed, it was not the support of the `dn` value of r since `dn[2]` + 2 = 4 and `dn[r]` = 3, so their is no need of putting r in the modified nodes set. Same for the arc (r,3,c,1). Now we consider the modified nodes inside of UP_c, there is node 1, whose cost changed from 2 to 3. We consider all the incoming arcs of 1, first (r,1,a,1), whose minimal cost is now $\texttt{dn[1]} + 1 + \texttt{up[r]} = 3 + 1 + 0 = 4 \leq 5$ thus this arc is still valid, but it was a probable support of the `dn[r]` value, thus r is pending for a new support of its `dn` value. Arc (r,1,d,2) is still valid since its minimal cost is 5, but the arc (r,1,e,3) has a cost of $\texttt{dn[1]} + 3 + \texttt{up[r]} = 3 + 3 + 0 = 6 > 5$, thus we have to remove this arc. Finally, r update its new `dn` value to 4. The resulting MDD is given in Figure 11.3.

Complexity The complexity of MDD4R is linear amortized over a branch tree. This implies that while processing all the events of a branch, then the

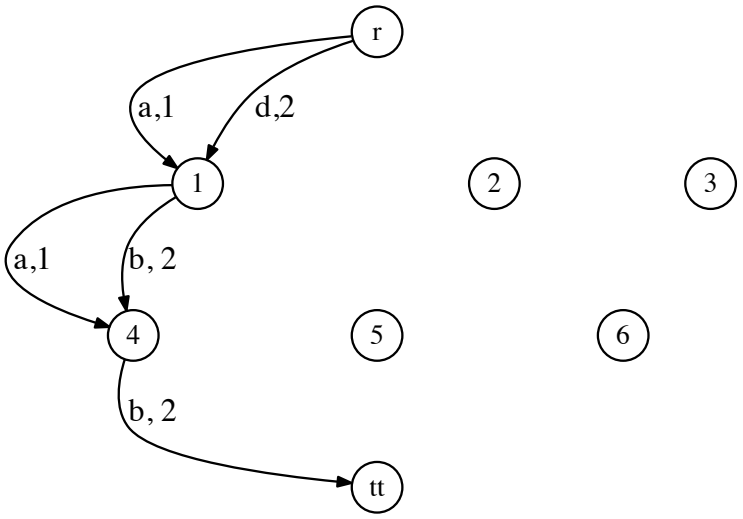


Figure 11.3: Propagation of $x_3 = b$ with $H = 5$ in the cost MDD of Figure 11.2.

total amount of work is linear on the size of the MDD. The cost-MDD4R algorithm is mainly composed of MDD4R and the propagation of the cost. Since the propagation of the cost on the MDD can lead to search over the whole MDD, the complexity of a cost-MDD4R propagation is bound by the size of the MDD at each call.

11.3.2 Modification of the cost value.

While the first part describes how the algorithm reacts when the variables of the constraint are modified, the algorithm has to face another modification, the modification of the H value.

During the search, constraint solver can modify this H value, consider for example constraint optimization solvers, when a solution with best value H is found then the solvers often search with $H' = H - 1$ in order to prove that no solution exists. In order to efficiently handle this modification, consider the following property.

Let $e = (u, v, a, c)$ be an arc, the Minimal Path Cost of e denoted by $MPC(e)$ is $MPC(e) = c + \texttt{up[u]} + \texttt{dn[v]} = \gamma(p_{min(e)})$.

Proposition 1 *Let C be an arc consistent constraint with a cost value $H = k + i$. If H is reduced to k, then removing all arcs of cost-MDD (C) such that $MPC(e) > k$ is sufficient for C to be arc consistent.*

Algorithm 24 cost-MDD4R BoundUpdate.

```
boundUpdate(mdd)
    for each layer l ∈ mdd.Layers do
        #e_t_remove ← Σ_{c=Nmax+1}^{Pmax} |E_l[c]|
        #e_t_keep ← #edges[l] − #e_t_remove
        if #e_t_remove < #e_t_keep then
            for each c ∈ [Nmax + 1, Pmax] do
                for each e ∈ E_l[c] do
                    remove(e)
        else
            for each c ∈ [0, Nmax] do
                for each e ∈ S_l[c] do
                    restore(e)
```

Proof: Any arc ε with $MPC(\varepsilon) > k$ is not consistent by definition, and so can be safely deleted. Let e be any arc with $MPC(e) \leq k$. Then, for every arc $e' \in p_{min(e)}$ there is p' a path such that $\gamma(p') \leq \gamma(p_{min(e)}) = MPC(e)$, so $MPC(e') \leq MPC(e) \leq k$. Thus, no arc of $p_{min(e)}$ has been deleted and $p_{min(e)}$ is a valid path of $path^r_{tt}(M)$. Hence, e is consistent with C. □

Thanks to this property, we can efficiently propagate the modification of the cost H of a cost-MDD constraint. When the value H is reduced from $k+i$ to k, cost-MDD4R establishes arc consistency by performing a BFS, and for each layer, it simply removes the arcs such that $MPC > k$. This can be efficiently done by maintaining the arcs sorted by their MPC.

Reset Here again, the reset idea can be applied. When there are fewer arcs with $MPC \leq k$ than arcs with $MPC > k$, then cost-MDD4R will choose to clear the data structures and put back the arcs with $MPC \leq k$. This is an important part of the algorithm because bound propagation can be costly. This idea avoids deleting almost all the MDD when only few arcs are still valid.

The algorithm Let E_i be the backtracking data structure storing the arcs of a layer and ordering them by their MPC value. Since this value is bound between the shortest path cost from the original MDD and the maximum cost H, a linear sort can be used. The Algorithm 24 is a possible implementation

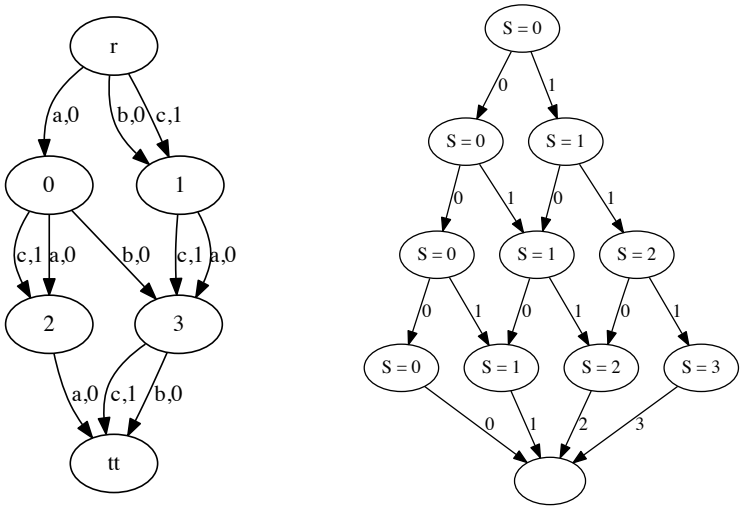


Figure 11.4: On the left, a cost-MDD assigning a cost of 1 for the value c and 0 otherwise. On the right, $MDD_{\Sigma\{0,1\}}$ on 3 variables

of the bound modification algorithm.

Complexity Consider that the algorithm does not perform any reset, then the number of arc processed along a branch tree is bound by the number of arcs of the MDD, since each time an arc is removed, it can't be removed again. Consider now that the algorithm performs a reset only when the number of remaining arcs is lower than the number of arcs to remove, then since the overall number of arcs is lower than the number of arcs of the MDD, the complexity remains the same.

Remark: Since the cost-MDD4R can be seen as an add-on of the MDD4R propagator, we can apply several cost constraint to the same MDD [Demassey 2006]. This implies a stronger propagation than propagating several distinct cost-MDD constraints.

11.4 Cost Intersection Method

The cost-MDD4R filtering algorithm is able to handle cost-MDD constraints. But the complexity of the cost-MDD4R propagator is worse than the complexity of the MDD4R propagator. This section proposes a transformation of the cost-MDD into a simple MDD that can be handled by an MDD propagator

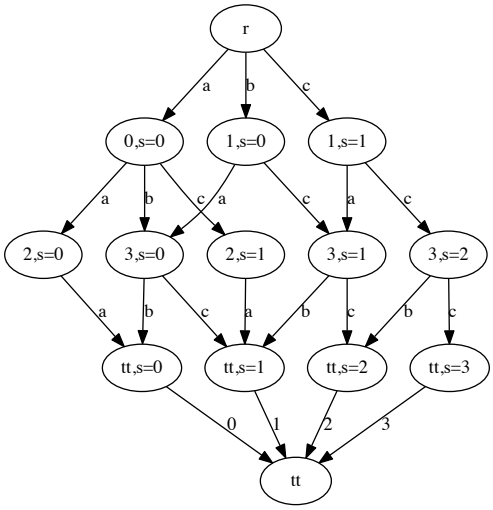


Figure 11.5: The resulting MDD after the cost intersection between the MDDs from Figure 11.4

and enforcing arc consistency on the cost-variable. Having a strong consistency for cost-variable usually allow to remove sooner inconsistent values and thus having a smaller search tree. This transformation will be done by adding a layer to the MDD corresponding to the cost.

Intersection of MDDs As shown in chapter 5, efficient operators between MDDs allow us to perform several operations on MDDs. In general these operations aim at combining MDDs according to the label of their arcs. In this section, we use these operations in another way. Instead of applying operators on the label of the arcs, we apply them on their cost.

0-1 cost intersection Let $MDD_{\Sigma\{0,1\}}$ (right on Figure 11.4) be the MDD representing the sum of n variables with the set $\{0, 1\}$ as a domain. The last variable represents the possible values of this sum. Consider an MDD mdd_c where the cost of an arc is either O or 1. If we perform the intersection between mdd_c and $MDD_{\Sigma\{0,1\}}$, then we obtain an MDD whose last layer corresponds to the cost of the assignment. Thus, any MDD constraint propagator can be used and the last variable represents the cost-variable.

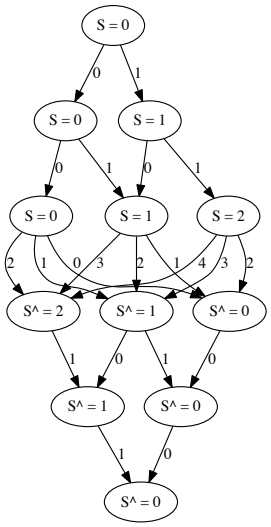


Figure 11.6: $MDD_{\Sigma\{0,1\}}$ on 4 variables

Example:

Consider the left MDD from Figure 11.4, the cost of its arcs are either 0 or 1, thus we can intersect this MDD with the MDD on the right. We obtain the MDD of Figure 11.5, in this MDD all the nodes have been duplicated in as many different incoming sums, for example consider the node 3 from the original MDD, since 3 different sums reach the node, the node has been split 3 times.

Complexity We can compute the size of this newly created MDD. Let M be any cost-MDD, we denote by $|M.layer(i)|$ the number of nodes at the layer i and by $|M|$ the total number of nodes of M. The maximum number of nodes at a layer i of the intersection of M and $MDD_{\Sigma\{0,1\}}$ is bounded by $|M.layer(i)| * |MDD_{\Sigma\{0,1\}}.layer(i)|$. The sum of all the layers is bounded by $|M| * \max_i(|MDD_{\Sigma\{0,1\}}.layer(i)|)$. In our case, the maximum for a layer of $MDD_{\Sigma\{0,1\}}$ is the number of variables plus one, that is $n+1$. So the resulting MDD has a maximum size of $(n+1)|M|$

Note that we can reduce the size of $|MDD_{\Sigma\{0,1\}}|$, by modeling $\sum_i x_i = S$, the sum constraint, in a different way. Instead of ordering the variables from x_1 to x_n and to S the final sum variable,we can order the variable from x_1 to $x_{\frac{n}{2}}$ then S and then from $x_{\frac{n}{2}+1}$ to x_n. This leads to an MDD having half as many nodes for its largest layer (Figure 11.6).

Any cost intersection This intersection method can be applied to any constraint represented by a cost-MDD. However, in general the cost of an arc is not only 0 or 1 and so this sum is not bounded by a small number.

Let $MDD_{\Sigma[0,k]}$ be the MDD representing the sum of n variables with the set $\{0, 1, ..., k\}$ as a domain. In the best representation of $MDD_{\Sigma[0,k]}$, the number of nodes of the middle layer is the number of different values reachable by summing the numbers between 0 and k, which is $nk/2$.

The intersection of a cost-MDD M with $MDD_{\Sigma[0,k]}$ leads to an MDD having at most $nk|M|/2$ nodes. Note that this number is an upper bound, because the number of times a node is duplicated in the intersection is equal to the number of different values of the sums reaching this node. This number of different values is the same as the considered values of the algorithm presented in [Hoda 2010] for the two-sided inequality. If this number is not too large, then applying this transformation is a good idea because MDD propagators are faster than cost-MDD propagators.

11.4.1 Discussion

Cost of a value Some search heuristic or algorithms require, a lower bound for the cost of a pair variable value (x, a). This value can be extracted from the constraint by looking for the minimum MPC value of the arcs in the $S(x, a)$ list. Thus a linear algorithm can be define to extract such value.

Remark Consider two distinct cost-MDD constraints, mdd_1 associated with a cost C1 and mdd_2 associated with a cost C2. Enforcing arc consistency on $MDD_i = mdd_1 \cap mdd_2$, having on the same MDD both of the cost function is stronger than propagating the two constraints independently.

11.5 Experiments

11.5.1 MaxOrder

The main motivation and experiments are the one made on chapter 17. The results are the one for the soft constraint handle with a cost-MDD constraint.

11.5.2 Random instances

We test the propagators on several random instances, in order to detect the location of the best performances of each propagator. We select a certain number randomly of tuples and build an MDD from this tuple set. We associate each arc with a random cost between 0 and 10. This implies the use of $MDD_{\Sigma[0,10]}$ instead of $MDD_{\Sigma\{0,1\}}$ for the intersection method. We have a constraint of arity 18 and an allDifferent constraint. Table 11.1 shows that the intersection method can be very efficient in practice. Intuitively, the intersection method "precomputes" some operations that are recomputed each time by the cost-MDD propagator. However, when the MDD grows up to reach our memory limit (around 1.7GB), then cost-MDD4R become faster than the intersection method.

#tuples	cost-MDD4R	ev-mdd	inter
50	35,89	59,23	**2,55**
150	19,15	33,98	**1,97**
500	19,61	35,38	**2,77**
1k	19,97	37,37	**4,15**
2k	32,04	66,22	**8,23**
5k	32,27	71,43	**14,22**
10k	44,26	83,58	**19,12**
25k	101,57	189,30	**49,84**
50k	201,94	378,533	**150,316**
100k	**478,296**	755,570	1668,508

Table 11.1: Time (s) for finding the best solution. Construction time is included, arity 18, domain size 18.

Chapter 12

Soft-MDD constraint

Contents

12.1 Introduction

Many real-world problem do not contain any solution: we call these kinds of problems over constrained problems.

Consider for example a seating plan for a wedding. Suppose we have 2 tables of 8 people, 15 different people and 4 people that do not want to be on the same table. In this problem, each person is represented by a variable, the domain of these variables are the tables to which people can be assigned, either table a of b. For each table, a constraint prevents that more than 8 people to be seated on it. For the four incompatible people, a constraint of difference (all-different) prevents them from being assigned to the same table.

This problem is unsatisfiable. But since a wedding planner cannot answer that there is no solution, because this implies that the wedding is canceled, a good non-solution has to be found.

Consider that the number of tables and the size of the table are given by the restaurant, so they cannot be changed, thus the first constraints cannot be modified. We usually call these constraints *hard* constraints.

The constraint preventing the four people of being on the same table is usually a preference constraint, we would like to satisfy it but it is not necessarily possible. We call this kind of constraint *soft* constraints, the constraints that can be violated, but with respect to a certain violation measurement.

Several soft versions of constraints exist, like the soft all-different constraint [Van Hoeve 2004] or the soft table constraint [Lecoutre 2012b]. The violation measurement of a constraint mainly depends on the constraint.

For example, consider a soft all-different applied to 4 variables x_1 to x_4 having as domain the set a, b. should the solutions (a, a, a, b) and (a, a, b, b) have the same violation cost?

Several criteria for the violation of a constraint have been given [Beldiceanu 2004b, Petit 2001]. One for example counts the number of variables to remove or change in order to satisfy the constraint, and is named the *variable based violation cost*. Another counts the number of violated constraints in the binary decomposition of the constraint, and is named the *decomposition based violation cost*.

In the previous example, applying a soft all-different gives for the assignment (a, a, b, b), a variable based cost of 2 and a decomposition based cost of 2. Now considering the assignment (a, a, a, b), the variable based cost is 2 but the decomposition based cost is 3. This implies that for our seating plan, the solution putting in each table two incompatible people seems to be better considering the decomposition based violation cost than putting three of them on the same table.

In the context of MDDs, no work has been done for soft-MDD, but in the context of the regular constraint, an algorithm has been proposed [Van Hoeve 2006]. For a regular constraint, two main violation costs can be extracted, one of them is the classical variable based violation cost, and the second one is based on the language accepted by the automaton. This second violation criteria, named *edit based violation cost*, considers the minimum amount of modification, that can be an insertion, a deletion or a substitution required to get an accepted word.

An MDD represents a set of tuples, thus a simple violation criteria can be defined by the minimum distance between a given solution and a tuple in the MDD. Using a Hamming distance, we obtain the variable based modification cost.

12.2 Soft-MDD Propagator

A soft constraint is a constraint which allows some violations. In this section, we consider only the variable based violation cost [Petit 2001]. Precisely, for a given assignment A of valid values of a constraint C, the cost of the violation of C by A is defined as the minimal number of values of A that should be changed in order to satisfy C. In other words, it corresponds to the minimum of the

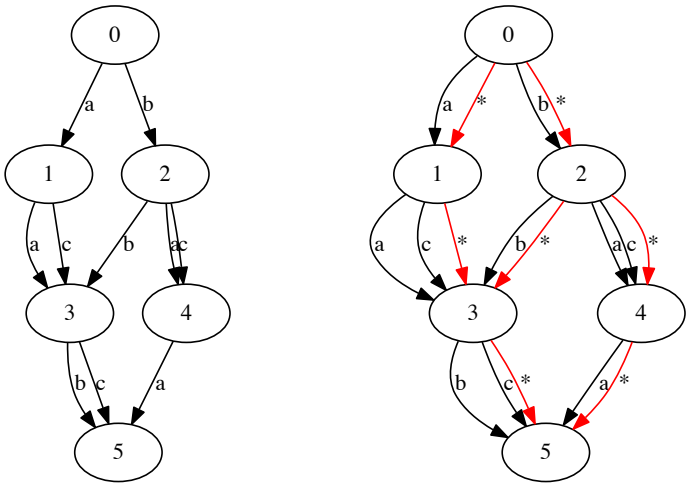


Figure 12.1: Soft MDD and $scMDD$

Hamming distance between A and any tuple of C. We denote this distance by Hamming(A, C). We consider that a value $a \in D(x)$ is consistent with a soft constraint C associated with an integer H if and only if there exists A, an assignment of valid values involving (x, a), such that Hamming$(A, C) \leq H$.

A soft-MDD propagator of a soft constraint C associated with an integer H is an algorithm which removes some values inconsistent with C and H.

> **Example:**
> For example, consider the top left MDD of Figure 12.1. If all the arcs of any path are valid, then the constraint is not violated. But if values a and b are deleted from the domain of the first variable (i.e. the variable of the first layer) then the propagation of this deletion will remove all the nodes and arcs of the MDD. This shows the need for new propagators in order to deal with the amount of violation.

This section presents three methods to propagate a soft constraint C represented by MDD(C). The first one is a simple propagator, which does not modify MDD(C) and uses some properties on the shortest path to establish arc consistency of C. The second one uses the idea from the soft regular in order to transform MDD(C) into a cost-MDD and uses a cost-MDD propagator on it. The last one builds an MDD explicitly dealing with the violation cost variable and intersects it with MDD(C) and applies on the resulting MDD an MDD propagator.

12.2.1 Dedicated Propagator

The first propagator does not modify MDD(C) and is easy to implement. Consider $\mu(p)$ the function which counts the number of arcs of the path p that are not valid. During the search, only assignments involving values in the current domains of the variables are considered, so the Hamming distance between any assignment and a tuple of the MDD corresponding to the path p is at least $\mu(p)$.

Let M be an MDD, let e be an arc of M. If e is not involved in a path p of $path^r_{tt}(M)$ with $\mu(p) \leq H$ then it means that no path containing e may support a value, so e can be safely deleted.

Property 10 *Let p be a path of $path^r_{tt}(M)$. If $\mu(p) = H$, then it means that p supports any value of a variable corresponding to the layer of non-valid arcs of p.*

Proof: The $\mu(p) = H$ implies that $r - H$ values of the path belong to the current domain of their respective variables. Let $X(H)$ be the variables at the layers of the invalid arcs. Let $X\bar{(}H)$ be the set of variables at the layers of the valid arcs. The partial instantiation of the $X\bar{(}H)$ variables to the label of their valid arc has a violation cost of 0. Thus using this partial instantiation, the instantiation of the H remaining variables to any values of their current domain cannot have a violation cost greater than H. □

Property 11 *Let p be a path of $path^r_{tt}(M)$. If $\mu(p) < H$, then it means that p supports any value of any variable.*

Proof: Let $h < H$ be the number of invalid arcs of p. Let $X(h)$ be the variables at the layers of the invalid arcs. Let $X\bar{(}h)$ be the set of variables at the layers of the valid arcs. For any variable $x \in X\bar{(}h)$, the partial instantiation of the $X\bar{(}h)\backslash\{x\}$ variables to the label of their valid arc has a violation cost of 0. Thus using this partial instantiation, the instantiation of the $h + 1 \leq H$ remaining variables to any values of their current domain cannot have a violation cost greater than H. □

Algorithm Using these two properties about paths, we design a simple propagator for the soft-MDD constraint. First, we search for the shortest path of the MDD according to function μ. If this path has a cost strictly lower than the maximum cost H, then all the values are supported (Property 11). Otherwise, we delete all arcs e with a shortest path cost according to μ greater than H.

The resulting MDD can be handled by any MDD propagator. Using 2 BFS, we can determine the shortest path cost of all arcs and remove all impossible paths. This method establishes arc consistency of the soft constraint.

Note that a classical cost-MDD cannot handle this constraint by considering μ as cost function because the cost of an arc depends on the domain of the variables.

12.2.2 Transformation into a cost-MDD

The second method for handling soft-MDD constraints is an adaptation of the method initially created for the regular constraint [Van Hoeve 2006].

The idea is to add, for each two nodes which have at least one arc between them, an additional arc labeled by *, with a cost of 1. The cost of all the other arcs is set to 0. An arc labeled by * is an arc which supports any value of the variable. We call *-arcs such arcs and we denote by $scMDD$ the resulting cost-MDD and f_{SC} the cost function we have defined. Then, we define a cost-MDD propagator on $scMDD$ with f_{SC}, $H+1$ and $<$.

Example:

For instance, in Figure 12.1 the *-arcs (in red) are created only between connected nodes. Nodes 2 and 3 are connected by an arc labeled by b, so we create the *-arc, but 1 and 4 are not connected, so we do not create the *-arc between them. Now, assume that the values a and b are deleted from the domain of the variable x1. The resulting MDD is the MDD in Figure 12.2. We can see that only 2 arcs have been deleted, and, unlike in MDD(C), the nodes are not deleted, thanks to the *-arcs. It is easy to see that the shortest path cost in this MDD is 1 because all paths contain at least one *-arc.

12.2.3 Intersection of MDDs

Chapter 11 proposed a method allowing to handle a cost-MDD constraint with an MDD propagator. To do so, the cost-MDD is intersected with an MDD representing the cost function.

For soft MDDs, the cost function is a 0-1 function. Thus the intersection method can intersect the MDD given by the transformation of section 12.2.2 and the MDD $|MDD_{\Sigma\{0,1\}}|$. This intersection leads to duplicating each node at worst $min(r/2, H)$ times.

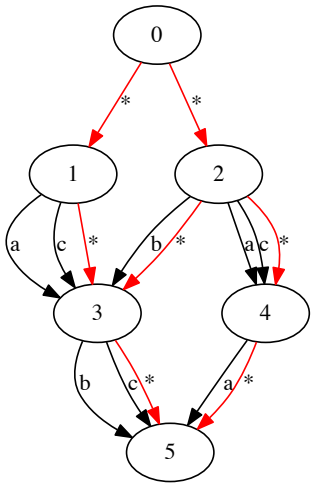


Figure 12.2: Soft MDD propagation

Example:

For example, if we take the left MDD from Figure 12.3, and we intersect it with the one on the right, then we obtain the MDD of Figure 12.4. In the resulting MDD, the node `(1 S=0)` represents the copy of the node `1` from the first MDD having an incoming cost of 0. We can observe that the outgoing arcs of node `(1 S=X)` are still directed to a node labeled by `(3 S=X)`.

Thanks to this intersection transformation, any MDD propagator can handle and enforce arc consistency for a soft-MDD constraint, by trading some memory, but allowing to use more efficient propagators and enforcing arc consistency on the violation variable.

12.3 Discussion

Choice The choice between the possible implementations of the soft constraint depends first on the algorithms already present in the used solver. Many solvers have MDD constraint propagators, while very few of them have cost-MDD constraint propagators. The second point to care about is the transformation size, while the first one can add at worst as many arcs as the number of existing arcs, the second transformation can use $r/2$ times more

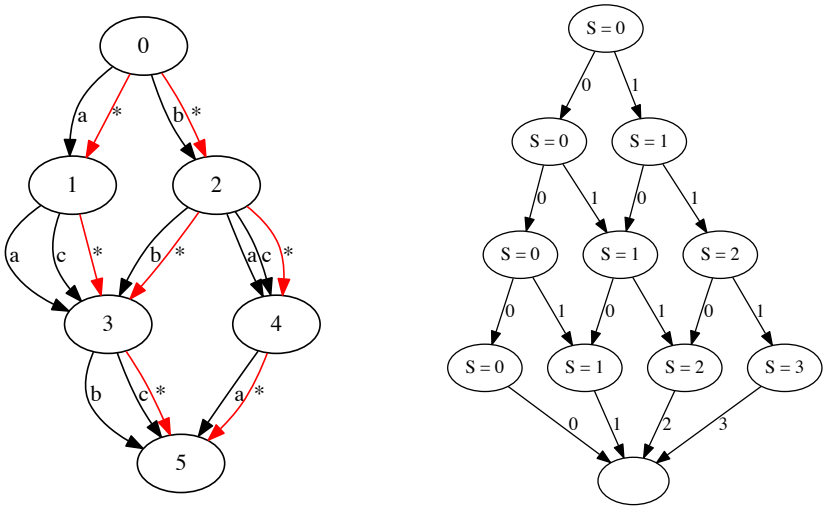


Figure 12.3: $scMDD$ and the $|MDD_{\Sigma\{0,1\}}|$ for 3 variables

memory. Thus the choice between the methods has to consider all these points. Considering the tests made, when the intersection method can be used, the results are often faster.

Remark MDDs are often used in order to combine constraints. The use of a soft-MDD constraint for handling such combination has to be done carefully. An MDD M which is the intersection of the two MDDs M_1 and M_2 contains the tuples that appears on both M_1 and M_2. Thus applying a soft-MDD constraint using M is not the same as applying two soft-MDD constraints using M_1 and M_2. The intersection loose the information of the original MDDs, when the information is not shared by both of the MDDs. The experimental section gives an example of such behavior.

Relax As discussed in chapter 8, several works focus on relaxed MDDs [Bergman 2016a, Bergman 2011]. They aim at obtaining an MDD representing a superset $\hat{S}$ of the solutions S of the exact MDD, a superset implies that $S \subseteq \hat{S}$. They generally bound the number of nodes in a layer. Applying the softening method proposed in this paper to a relax MDD will give a soft version of the $\hat{S}$ set of solutions.

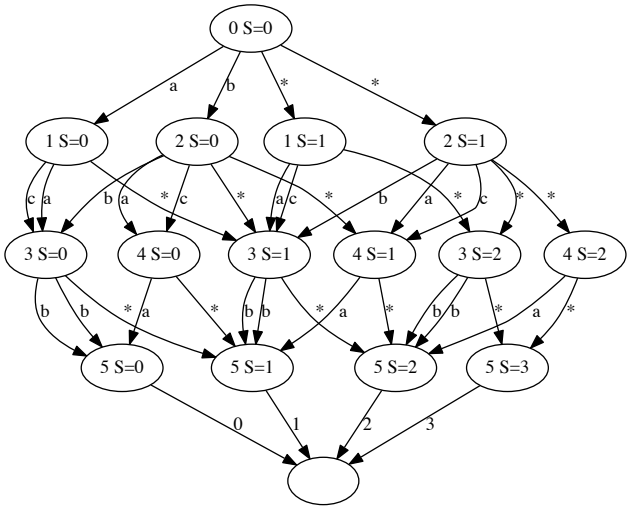


Figure 12.4: MDD resulting from the intersection of the two MDDs of Figure 12.3.

12.4 Experiments

We compare cost-MDD4R with *ev-mdd*, the incremental algorithm presented in [Gange 2013] and with *inter*, the intersection method we proposed.

Sequence generation The problem presented here partially comes from chapter 17.

We consider the problem detailed in Section Motivation. We have tested both ways of softening the constraint and they both give pertinent results. For the experimentation, we used `"The fables of Jean de La Fontaine"` because they contain several sentences, not too many words and often produce funny results.

An important remark is that, if the corpus size grows, then the `maxOrder` constraint becomes satisfiable. If it grows again, then it becomes useless to apply a `maxOrder` constraint because it becomes exponentially improbable to build a sequence containing plagiarism. That's why we focus on corpus like fables and short texts.

Table 12.1 gives the time results (in seconds) and Table 12.2 gives the size of the MDDs. Note that the model also contains an alldifferent constraint. `Markov` means that we apply the soft constraint on the Markovian transition, `Plagiarism` is for the plagiarism part. The creation time is similar for both

Algo	Markov			Plagiarism		
size	18	20	22	18	20	22
inter	5,5	104,8	111,7	**4,7**	**8,1**	**9,3**
cost-MDD4R	**5,3**	**86,5**	**94,9**	23,7	44,6	67,9
ev-mdd	11,1	361,9	355,5	26,2	58,5	78,0

Table 12.1: Times needed to build the sequences with minimum of violations (Time out 1800s).

	Markov		Plagiarism	
	#nodes	#arcs	#nodes	#arcs
original	73	168	261	21.5k
arc *	73	380	261	22.1k
intersection	147	590	783	54.6k

Table 12.2: Size of the MDDs. 60 different words.

MDDs, and insignificant compared to the search time.

These tables show that both methods are useful, and that our algorithms clearly outperform the existing methods. First, the intersection method, seems to be very efficient, by being either the best one, or close to the best one. The using a cost-MDD constraint for handling soft-MDD is possible as shows this experiments. Moreover, our cost-MDD4R seems to performe better than ev-mdd in this examples.

Chapter 13

Channeling Constraints and MDDs

Contents

13.1 Introduction

Desiging efficient algorithms enforcing constraints on MDDs often lead to solve hard combinatorial problems [Andersen 2007, Hoda 2010, Bergman 2014b, Cheng 2005, Hadzic 2008]. For example, several algorithms exist enforcing the allDifferent constraint, inequality constraints, among constraints or even MDD constraints onto MDDs.

The simplest way for constraining MDDs is to intersect them with other constraints. While such a method can easily be implemented and often gives good results, the size of the resulting MDD can become a problem. Thus several works focus on applying algorithms on the MDD in order to apply other constraints to the MDD, by modifying or by storing information on the nodes and removing arcs using this information. A simple example is the cost-MDD, which stores in nodes the minimal path cost and uses it for removing arcs whose cost is greater than a maximum cost.

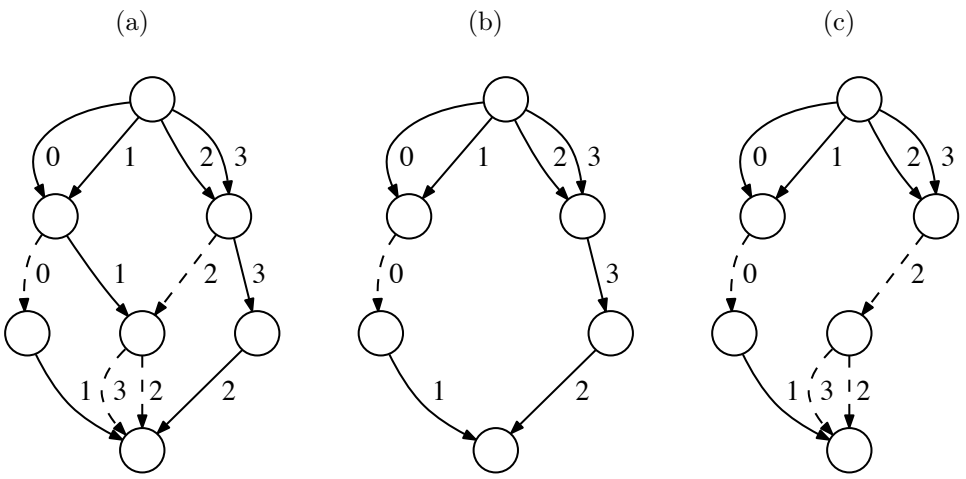


Figure 13.1: An MDD (a) having 4 marked arcs (dashed). This marking implies that the variables involved are x_2 and x_3 and the values are $\{0, 2, 3\}$. (b) the prohibition of variable 3 for the marked arcs of the MDD (a). (c), the mandatory of variable 2 for the marked arcs of MDD (a).

What is proposed here is to constrain a sub-part of an MDD. This is done in two steps. First, we mark a subset of arcs of the MDD using a *marking* function. A marking function selects a sub-set of arcs depending on a criteria which depends on the problem. Then we manage these arcs by constraining their values or the index of their variables. The channeling constraint [Cheng 1999] is well suited for this problem.

Example:

Consider for example the MDD (a) of Figure 13.1, applied to the three variables x_1, x_2 and x_3. The dashed arcs are the arcs marked by a given marking function. The set of possible values for these arcs is $\{0, 2, 3\}$ and the set of possible variables are $\{2, 3\}$. If we enforce that the variable x_3 cannot have marked arcs, then we obtain the MDD (b) of Figure 13.1. Furthermore, if instead we enforce that x_2 must have a marked arc, then we obtain the MDD (c) of Figure 13.1.

Managing the arcs must allow to select the possible, mandatory or prohibited values and the possible, mandatory or prohibited variables for these arcs. Thus the use of set variables for managing the arcs seems to be a good option.

This chapter first describes the definition of the MDD-Arc constraint, then it proposes several algorithms managing the constraint.

13.2 MDD Channeling Constraint

13.2.1 Set Variables

Set variables [Gervet 1993, Puget 1993] are variables that can be assigned to several values at the same time, a set of values. In a CP solver, a set-variable can be defined using a set of Boolean variables, by introducing a Boolean variable for each value. The state of a value a is defined as follows depending on the current domain of its associated Boolean variable b_a:

- $D_c(b_a) = \{0\}$ implies that the value does not belong to the Set.
- $D_c(b_a) = \{1\}$ implies that the value is mandatory.
- $D_c(b_a) = \{0, 1\}$ implies that the value is possible but not mandatory.

Constraint programming solvers have several constraints for set variables, like the cardinality, the intersection or partition [Bessiere 2004, Yip 2010]. Set variables can be used for representing problems like the social golfer [Harvey] or music titles selection [Pachet 1999].

13.2.2 Definition

Let C be an MDD constraint associated with the MDD M. Let f be a marking function, let I and V be set variables. The $mddChannel(C, X, f, I, V)$ constraints ensure that the marked arcs by the function have their values in V and their variable indexes in I. Thus I and V manage a subset of the arcs of the MDD used by the constraint C.

Definition 7 *An index i is possible for I if there exists a valid marked arc at layer i.*

Definition 8 *An index i is mandatory for I if there does not exist a valid non-marked arc at layer i.*

Definition 9 *An index i is prohibited (removed) for I if it does not exist a valid marked arc at layer i.*

Definition 10 *A value a is possible for V if there exists a valid marked arc labeled by a.*

Definition 11 *A value a is mandatory for V if it does not exist a valid path that does not contain a marked arc labeled by a.*

Definition 12 *A value a is prohibited (removed) for V if it does not exist a valid marked arc labeled by a.*

The next section presents how the modification of the set variables I and V allow to manage the marked arc of the MDD.

Remark: The constraint C can be any MDD constraint thus it can be an *mddChannel* constraint. This implies that an MDD can be constrained by several different *mddChannel* constraints, in the same ways as an MDD can be constrained by several cost-MDD constraints or several distinct constraints [Andersen 2007, Hoda 2010].

13.3 Propagation

We can distinguish two modifications of a set variable for a given value, the value becomes mandatory or is removed. Thus we have to define the impact of such modifications for the two set variables I and V. Moreover, since the MDD used by the constraint can be modified for external reasons, like another mddArcs constraint modifying the MDD, we have to handle the impact of these modifications to the set variables I and V.

13.3.1 Modification of I

13.3.1.1 Removing an index i in I

Consider that the value i is removed from the set variable I. Its associated Boolean variable is set to 0. This implies that none of the marked arcs at the layer i can belong to a solution anymore. Thus we have to enforce that such a solution does not belong to the MDD.

This can be simply done by removing all the arcs marked at the layer i and then propagating these modifications inside the MDD. This propagation is close to the one of MDD4R and thus the complexity of applying this algorithm is linear. The MDD (b) of Figure 13.1 shows an example of removing an index of I.

13.3.1.2 Mandatory of an Index i in I

Consider that the index i becomes mandatory in the set variable I. Its associated Boolean variable is set to 1. This implies that at least one of the marked arcs at the layer i must belong to a solution. Thus we have to enforce that all the solutions of the MDD contain a marked arc at layer i.

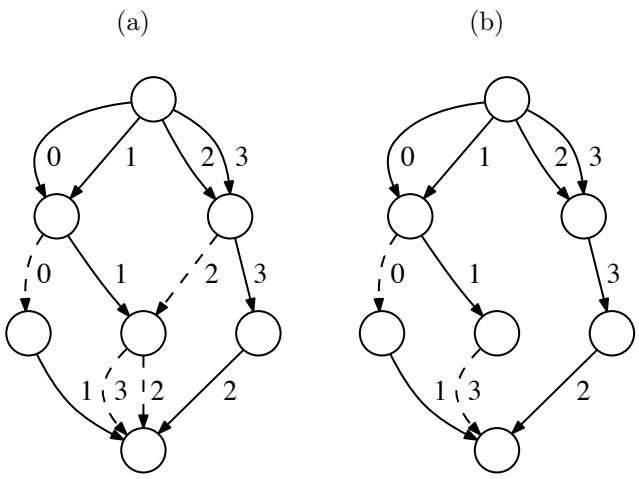


Figure 13.2: (a) an MDD having 4 marked arcs (dashed). (b) the propagation of 2 being removed from V.

As for the removing, a layer becoming mandatory can efficiently be enforced by removing all the non-marked arcs of the layer i. Thus all the solutions of the MDD contain at least one arc at the layer i and this arc is marked otherwise it would have been deleted. The MDD (c) of Figure 13.1 shows an example of an index of I becoming mandatory.

13.3.2 Modification of V

13.3.2.1 Value a Removing in V

Consider the removing a value a from the set variable V. This implies that, just like the removing of an index in I, none of the marked arcs labeled by a can belong to a solution. Thus we have to enforce that none of the solutions of the MDD contains a marked arc labeled by a anymore.

We can enforce that efficiently by removing all the marked arcs labeled by a in the MDD and then propagating these modifications. Once all these prohibited arcs removed, they cannot be part of a solution anymore.

Example:

Consider the MDD from Figure 13.2, if the value 2 becomes prohibited for the marked arcs, its Boolean variable is set to 0, then all the marked arc labeled by 2 are removed. The result is the MDD (b).

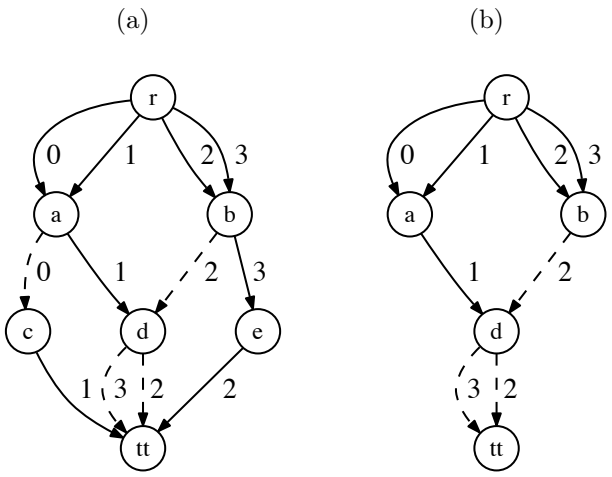


Figure 13.3: (a) an MDD having 4 marked arcs (dashed). This marking implies that the variables involved are x_2 and x_3 and the values are $\{0, 2, 3\}$.

13.3.2.2 Mandatory of a Value a in V

Consider that the value a becomes mandatory in the set variable V. This implies that at least one of the marked arcs labeled by a must belong to a solution. Thus we have to enforce that all the solutions of the MDD contain such an arc.

Remark: In contrary to the previous modification, the MDD containing only solution is not a sub-MDD of the current MDD. This implies that we cannot only remove a set of arcs for enforcing the constraint.

Example:

Consider MDD (a) of Figure 13.3, if the value 2 becomes mandatory in V then all the solutions have to contain a marked arc labeled by 2. Thus the marked arc $(a, c, 0)$ cannot be part of a solution anymore since it does not belong to any path from r to tt containing a marked arc labeled by 2. The same reasoning allows to remove the arcs $(c, tt, 1)$, $(b, t, 3)$ and $(e, tt, 2)$. We obtain the MDD (b).

In this MDD, all the arcs belong to at least one path containing a marked arc labeled by a. But not all the paths of the MDD contain an arc labeled by 2, for example the path of the tuple $(0, 1, 3)$ does not contain any marked arc labeled by 2. Moreover, if the arc $(d, tt, 2)$ is removed, because of the current domain of x_3 for example, then there is no more valid path for the arcs $(r, a, 0)$, $(r, a, 1)$ and $(a, d, 1)$.

First method As the previous example shows, a value becoming mandatory for the set variable V is not trivial to handle since a simple deletion is not enough.

The first method for handling such a modification is to add for each value a becoming mandatory a cost constraint. The cost function of this constraint is to set a cost of 1 for all the marked arcs labeled by the mandatory value, and a cost of 0 otherwise. Then the constraint ensures that the solutions have a cost greater or equal to 1.

> **Example:**
>
> Considering the MDD on the right of figure 13.3. If the value 2 becomes mandatory for V, the application of this first method gives a cost of 1 to the arcs $(b, d, 2)$ and $(d, tt, 2)$ and a cost of 0 for all the others. The cost-MDD constraint ensures that the cost of a path is greater or equal to 1.
>
> The longest path of the arcs $(c, tt, 1)$, $(b, t, 3)$ and $(e, tt, 2)$ is 0 so the algorithm can remove these arcs. Now if the arc $(d, tt, 2)$ is removed, because of the current domain of x_3 for example, then the longest path cost of the arcs $(r, a, 0)$, $(r, a, 1)$ and $(a, d, 1)$ becomes 0 and these arcs are removed too.

Second method The first method needs a cost-MDD propagators for handling the values becoming mandatory. As shown in chapter 11, the difference in efficiency between MDD and cost-MDD propagator can be huge.

This second method proposes to ensure that all the paths of the MDD contain a marked arc labeled by the mandatory value. To do so, we are going to use the following remark:

Remark: The MDD containing all the good paths is not a sub-graph of the original MDD, but the MDD containing all the prohibited paths is a sub-graph of the original MDD.

Algorithm Consider an MDD M and a value a becoming mandatory in V. The algorithm for enforcing that all the solutions of an MDD M_a contain a marked arc labeled by a first extracts the sub-graph $M_{\bar{a}}$. In order to extract $M_{\bar{a}}$, we remove all the marked arcs labeled by a and propagate these deletion in the MDD. Then we can build $M_a = M - M_{\bar{a}}$ using any operators for MDDs.

By construction, M_a contains all the solutions of M minus the solutions of M that do not contain a marked arc labeled by a. Moreover, since the modification of the MDD can be small, the use of the in-place operator proposed in chapter 5 is well suited for them.

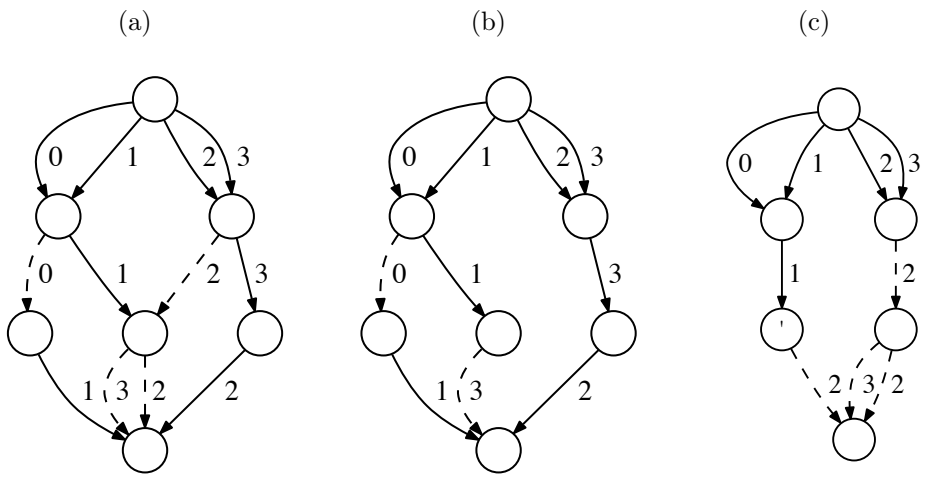


Figure 13.4: (a) an MDD having 4 marked arcs (dashed). This marking implies that the variables involved are x_2 and x_3 and the values are $\{0, 2, 3\}$. In (b) the modification of (a) enforcing that no solution contain a marked arc labeled by 2. MDD (c) is MDD (a) minus MDD (c), thus the MDD enforcing that all the solutions contain a marked arc labeled by 2.

Example:

Consider the MDD (c) of the figure 13.4 and that the value 2 becomes mandatory for V. For applying this second method we first extract from the original MDD, the MDD that does not contain any solution having a marked arc labeled by 2. We obtain the MDD (b) of Figure 13.4. Then we remove from the original MDD this new MDD and obtain the MDD containing all the solutions of the original MDD without solutions that do not contain a marked arc labeled by 2, the MDD (c).

13.3.3 Modification of the MDD

Modifying the MDD can lead to modification of the set variables I and V. For example, consider MDD (a) of Figure 13.5, if we set the variable of the layer 3 (x_3) to the value 3, we obtain the MDD on the right. These modifications enforce that index 3 is mandatory in I, that value 0 is removed from V and that value 3 is mandatory for V.

We need to define algorithms enforcing that the modifications made to the MDD are propagated to the set variables I and V.

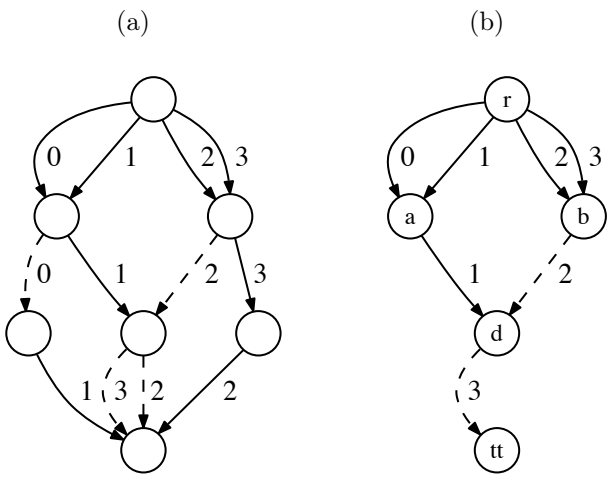


Figure 13.5: (a) an MDD having 4 marked arcs (dashed). This marking implies that the variables involved are x_2 and x_3 and the values are $\{0, 2, 3\}$.

13.3.3.1 Propagation for I

As shown in Figure 13.5, the modification can make indexes to be mandatory. But the MDD modification can also remove all the marked arcs of a layer and thus the index of this layer has to be removed from the set variable I.

Removing an index Using property 9, we can remove an index i of the set variable I when no more valid marked arc exists at layer I. This can be done by maintaining for each index $i \in I$ the set of valid marked arcs at layer i, then when this set is empty, the value i can be safely removed from I. Note that this set can also be used for propagating the deletion of the index of the set variable into the MDD.

Index becoming mandatory Using property 8, we know that an index i is mandatory if all the valid arcs of the layer i are marked. Thus we can use the set used for the removing of an index and compare its cardinality with the number of arcs of the layer. If they are equal, then the index is mandatory.

13.3.3.2 Propagation for V

Just like for the index, the Figure 13.5 shows that the modification of the MDD can trigger modification of the set variable V. Some values may have to be removed or may become mandatory. Thus we need to be able to propagate these modifications to the set variable V.

Removing a value Using property 12, we can easily know if a value a has to be removed from V. In the same manner as for the index, we can maintain, for each value a in V, the set of valid marked arcs labeled by a. Thus a value has to be removed when this set becomes empty. This set can also be used for propagating the deletion of a value of the set variable V in the MDD.

Value becoming mandatory The property 11 says that a value is mandatory if all the valid paths of the MDD contain at least one marked arc labeled by the value. For this one, it is not trivial to test if the value is mandatory or not.

First method A simple method is to perform a BFS on the MDD for extracting the mandatory values. This BFS stores in the node if all its incoming paths use a marked arc labeled by a given value.

During the BFS, when a valid marked arc labeled by the value is used, then it propagates to its destination node that all the paths passing through it use a marked arc labeled by value. Then for each node, the algorithm checks if all its incoming arcs are marked by the value or if they are coming from a node whose all the incoming paths use a marked arc labeled by the value. If it is true, then the node keeps the information that all the paths passing through it use a marked arc labeled by the value. Finally, if the tt node has kept this information for a given value, the value is mandatory.

We can perform all the values at the same time by keeping at each node the set of mandatory values and performing the intersection at each node of the incoming sets.

Cost method While the first method is quite simple, it can be costly to perform a search over the MDD at each modification and it implies to write an *ad hoc* code. Thus we are going to use an existing algorithm for handling this information.

In the same manner as we deal with the values becoming mandatory in the set variable V, we can use the cost-MDD constraint for having information about mandatory values.

Let C be a cost-MDD which, for a value a in a set V, gives a cost of 1 for all the marked arcs labeled by a and a cost of 0 otherwise. Let c_p be the shortest path cost of the cost-MDD, if c_p is greater or equal to 1, then the value is mandatory, otherwise it exists a path whose cost is 0, which implies that it does not contain any marked arc labeled by a.

13.4 Conclusion

This chapter has presented a method for defining channeling constraint for MDDs. This new method allows to constrain sub-parts of the MDD in a finer fashion compared to existing methods. Moreover, we propose several propagator enforcing different level of consistency. This method is used in chapter 14 for building the Allen constraint and helps to solve the problem of music synchronization of chapter 18. Thus the experiments are presented in these sections.

Part IV

MDDs: Constraints Modeling

Chapter 14

Allen constraint

Contents

14.1 Introduction and Related Works

Many difficult combinatorial problems consist in arranging sequences of events in time, subject to *horizontal* and *vertical* constraints, they are often called matrix models [Flener 2001]. These constraints are expressed on the *temporal position* of events. Horizontal constraints relate events in the same sequence, but occurring at different positions. Vertical constraints relate events occurring simultaneously, *i.e.,* at the same position in different sequences. This is similar to scheduling problems, such as job-shop scheduling, in which tasks are performed on machines according to sequential and resource constraints. The combination of horizontal and vertical constraints make these problems extremely difficult to solve: the job-shop scheduling problem is notorious among the hardest combinatorial problems.

A typical constraint programming approach to generating such sequences is to define a variable for each item of the sequence, and to post constraints on these variables. Temporal sequences challenge this model, since the position of an event is determined by the duration of all the preceding events, and so is

only weakly dependent on its index. It is therefore difficult, if not impossible, to express temporal properties using constraints on item variables.

This problem appears naturally in application domains related to entertainment [Derrien 2015, Galvane 2015a, Galvane 2015b, Berrani 2013]. Structural properties usually involve long-range dependencies between events. Deep learning approaches attempt precisely at capturing these dependencies in a statistical model, to reproduce them during classification or sampling. However, the representation of structure in statistical models is not explicit, making them inappropriate for specifying hard constraints on sequences.

Generally, in many interactive or content generation applications, we need to specify sequences with structural properties that cannot be inferred using statistical models. Constraint programming provides an ideal way of enforcing structure on sequences. However, as highlighted earlier, we cannot state structural constraints on events based on their index alone.

Related Work Adopting a position-based model, in which variables represent events of smaller, *atomic duration* whereby longer objects are made up of several consecutive variables, solves this issue. For a given total duration, a fixed number of variables are defined and therefore indexes correspond to temporal positions. This requires discretizing time into a grid of equal-duration slices, small enough so that all events are aligned with the grid. In this model, the number of variables is considerably larger than the number of events in the generated sequences: if durations are expressed as fractions of the longest event, the atomic duration decreases with the *least common multiple* of the denominators, whose growth is exponential [Nair 1982]. Hence, the size of the grid may be exponentially smaller than the event lengths, creating an intractable number of variables. Moreover, the position-based model requires additional horizontal constraints to aggregate atomic events to form longer objects. These constraints are not easy to specify in general. This approach is therefore not applicable in many real problems.

Several frameworks using constraint propagation make inferences about temporal relations from a qualitative [Allen 1983] or quantitative standpoint [Dechter 1991]. The computational efficiency of these approaches is very limited in the general case, but they offer a precise and powerful representation of relations between times events.

Allen Allen [Allen 1983] introduced an algebra with 13 binary relations between time intervals for temporal reasoning. A constraint based on Allen's algebra [Derrien 2015] has already been defined and takes a set of tasks, a set of Allen relations, a set of intervals, and checks that every task satisfies at least one relation for one interval. They apply this work to the generation of

video summaries. In their approach, the checks for every task are independent from one another.

The ALLEN constraint proposed here uses the Allen algebra as a language to express temporal positions. It defines variables corresponding to a given Allen relation. Technically, for a given time interval t and a given Allen relation $\mathcal{R}$, ALLEN maintains two set variables: the set of events and the set of variable indexes satisfying $\mathcal{R}$ for t. Then, temporal properties of the sequence can be represented by constraints defined on these set variables.

One of the simplest example is the following. Given an time interval $t = [2, 5]$, I want to enforce that at least one of my variables will take the value c. This will be define by first defining an ALLEN constraint, and then by constraining my set variable defined for the values to contain a c.

Plan This chapter first presents the definition of the ALLEN global constraint. Then two models implementing ALLEN are presented: the first model is based on a classical scheduling approach and the second model uses Multi-valued Decision Diagrams (MDDs). Finally, the experimental section shows that the MDD models seem to be well suited for the constraint and performs well in practice.

14.2 Constraining Contiguous Temporal Sequences

A *temporal event* e is a symbol with a duration $d(e)$. A *contiguous temporal sequence*, *CTS* for short, is a finite sequence of temporal events $(e_1, \ldots, e_n)$. A CTS is basically a concatenation of events: two consecutive events in a CTS are considered contiguous. Therefore, for a CTS $S = (e_1, \ldots, e_n)$, the *duration* $d(S)$ of S is the sum of the duration of the events contained in S, defined by $d(S) = \sum_{i=1}^{n} d(e_i)$. The *absolute temporal position*, or *starting time* of an event e_p in S is defined by $s(e_p) = \sum_{i=1}^{p-1} d(e_i)$. Note that the absolute temporal position is not an intrinsic property of a temporal event, it is a property of a temporal event with respect to a CTS. A same temporal event may appear several times in a same CTS at different starting times.

Here, we consider only temporal events with integer duration and, therefore, we address CTS in which all events have integer temporal positions.

Given a set E of temporal events, a model for the generation of CTS is to represent a CTS containing n temporal events of E as a sequence of n constrained variables $(X_1, \ldots, X_n)$, each with domain $dom(X_i) = E$. With this model, it is easy to state constraints relating events based on their index in the sequence, such as $X_1 = X_n$, or $X_i \neq X_{i+1}$. However, the absolute

temporal position of an event in a CTS is not directly related to its index as it depends on the duration of all preceding events. There is therefore no straightforward way of constraining the elements of the sequence based on their absolute temporal position.

14.2.1 Definition of the Allen Constraint

The idea behind the ALLEN constraint is to use Allen relations between temporal intervals to specify some temporal element(s) of a CTS (the 13 atomic relations of Allen are given on Table 14.1). Let S be a CTS $(e_1, \dots e_n)$. An Allen relation $\mathcal{R}$ and a temporal interval t specify a subsequence of S. For instance, if $\mathcal{R}$ is **d**, *i.e.,* the relation "during", and $t = [a, b]$, then $\mathcal{R}$ and t specify the subsequence of S containing the events which start after a and end before b.

Let E be a set of temporal events and let $X_1, \dots, X_n$ be n constrained variables, each with domain E. The X_is are the *sequence* variables. Let t be a temporal interval and let $\mathcal{R}$ be a relation of Allen between temporal intervals (see Table 14.1). Let $\mathcal{I}$ be a set variable, with domain $\{1, \dots, n\}$ and $\mathcal{E}$ be a set variable with domain E. The ALLEN constraint

$$\text{ALLEN}_{\mathcal{R},t}(X_1, \dots, X_n, \mathcal{I}, \mathcal{E}) \tag{14.1}$$

ensures that $\mathcal{I}$ contains the *indexes* of all sequence variables X_i belonging to the subsequence of $(X_1, \dots, X_n)$ specified by Allen relation $\mathcal{R}$ and temporal interval t. Similarly, the constraint (14.1) ensures that $\mathcal{E}$ contains the *values* of all sequence variables X_i belonging to the subsequence of $(X_1, \dots, X_n)$ specified by $\mathcal{R}$ and t.

The ALLEN constraint defined above is satisfied if and only if

$$\mathcal{I} = \{i \in \{1, \dots, n\} \mid [s(X_i), s(X_{i+1})] \; \mathcal{R} \, t\} \text{ and } \mathcal{E} = \{X_i \mid i \in \mathcal{I}\}$$

14.3 Implementing the Allen Constraint

This section describes two implementations of the ALLEN constraint. The first one is a simple model, based on scheduling, and performing only local propagations. The second one uses an MDD to represent the sequences explicitly, which makes it possible to prune more values during the search. In both models, the sequence variables $X_1, \dots, X_n$ take temporal event values.

Relation	Symbol	Example	Semantics	Inverse
t_1 before t_2	$<$	t_1 t_2	$t_1+ < t_2-$	$>$
t_1 equal t_2	**eq**	$t_1 t_2$	$t_1- = t_2-$ and $t_1+ = t_2+$	**eq**
t_1 meets t_2	**m**	t_1 t_2	$t_1+ = t_2-$	**mi**
t_1 overlaps t_2	**o**	t_1 t_2	$t_1- < t_2-$ and $t_2- < t_1+ < t_2+$	**oi**
t_1 during t_2	**d**	t_1 t_2	$t_1- > t_2-$ and $t_1+ < t_2+$	**di**
t_1 starts t_2	**s**	t_1 t_2	$t_1- = t_2-$ and $t_1+ < t_2+$	**si**
t_1 finishes t_2	**f**	t_2 t_1	$t_1- > t_2-$ and $t_1+ = t_2+$	**fi**

Table 14.1: The 13 atomic relations of Allen. The lower bound of a time interval t_i is denoted by t_i- and the upper bound by t_i+.

14.3.1 A First Model

The ALLEN constraint can be seen as a non-preemptive scheduling problem with unary resources where variables correspond to activities having a variable duration. In this model, each variable X_i is associated with two variables S_i and D_i. Variable S_i represents the absolute temporal position of X_i in the CTS, and D_i represents the duration of X_i. The start and duration variables are related via a set of constraints

$$S_{i+1} = S_i + D_i, \forall i = 1, \ldots, n-1 \tag{14.2}$$

with $S_1 = 0$.

In order to define the propagation rules, we will use the following five predicates:

- $\text{HOLDS}_{\mathcal{R},t}(s,d) \overset{def}{\iff} [s, s+d]\,\mathcal{R}\,t$, where s is a start time (i.e., absolute temporal position) and d a duration
- $\text{POSSIBLE}_{\mathcal{R},t}(i,e) \overset{def}{\iff} e \in dom(X_i)$ and $\exists s \in dom(S_i)$, $\text{HOLDS}_{\mathcal{R},t}(s, d(e))$
- $\text{POSSIBLE}_{\mathcal{R},t}(i) \overset{def}{\iff} \exists e \in dom(X_i)$ such that $\text{POSSIBLE}_{\mathcal{R},t}(i,e)$
- $\text{REQUIRED}_{\mathcal{R},t}(i,e) \overset{def}{\iff} X_i = e$ and $\forall s \in dom(S_i)$, $\text{HOLDS}_{\mathcal{R},t}(s, d(e))$
- $\text{REQUIRED}_{\mathcal{R},t}(i) \overset{def}{\iff} \forall e \in dom(X_i), \forall s \in dom(S_i)$, $\text{HOLD}_{\mathcal{R},t}(s, d(e))$

Variables $\mathcal{I}$ and $\mathcal{E}$ are set variables. We will use the notation $lb(.)$ for the lower-bound of a set-variable domain and $ub(.)$ for its upper-bound. Intuitively, during the filtering procedure, the lower-bound $lb(\mathcal{I})$ (resp., $lb(\mathcal{E})$) is the set of required values for $\mathcal{I}$ (resp., $\mathcal{E}$). Similarly, the upper-bound $ub(\mathcal{I})$ (resp., $ub(\mathcal{E})$) is the set of possible values for $\mathcal{I}$ (resp., $\mathcal{E}$). The filtering rules presented below rely on the equivalences:

$$i \in ub(\mathcal{I}) \iff \textsc{Possible}_{\mathcal{R},t}(i) \tag{14.3}$$
$$i \in lb(\mathcal{I}) \iff \textsc{Required}_{\mathcal{R},t}(i) \tag{14.4}$$
$$e \in ub(\mathcal{E}) \iff \exists i, \textsc{Possible}_{\mathcal{R},t}(i, e) \tag{14.5}$$

Note that $e \in lb(\mathcal{E})$ is more difficult to express in terms of the predicates, which is why Rule (14.15) is more complex. In fact, reasoning on $lb(\mathcal{E})$ is the most complex operation for maintaining the consistency between the sequence variables and the set variables. In the next section, we use an MDD model, which is sufficiently rich to infer the exact lower-bound $lb(\mathcal{E})$.

The consistency between the event, start, and duration variables, and the lower and upper bounds of the Allen set variables, is maintained with a set of filtering rules.

When S_i is modified, *i.e.,* a value was removed from its domain, the following rules may apply:

$$i \in ub(\mathcal{I}) : \neg\textsc{Possible}_{\mathcal{R},t}(i) \Rightarrow i \notin ub(\mathcal{I})$$
$$\textsc{Required}_{\mathcal{R},t}(i) \Rightarrow i \in lb(\mathcal{I}) \tag{14.6}$$
$$i \in lb(\mathcal{I}) : e \in dom(X_i) \wedge (\forall s \in dom(S_i), \neg\textsc{Holds}_{\mathcal{R},t}(s, d(e)))$$
$$\Rightarrow e \notin dom(X_i) \tag{14.7}$$
$$i \notin ub(\mathcal{I}) : e \in dom(X_i) \wedge (\forall s \in dom(S_i), \textsc{Holds}_{\mathcal{R},t}(s, d(e)))$$
$$\Rightarrow e \notin dom(X_i) \tag{14.8}$$
$$e \in ub(\mathcal{E}) : \nexists j, \textsc{Possible}_{\mathcal{R},t}(j, e) \Rightarrow e \notin ub(\mathcal{E}) \tag{14.9}$$
$$e \notin ub(\mathcal{E}) : (\forall s \in dom(S_i), \textsc{Holds}_{\mathcal{R},t}(s, d(e)))$$
$$\Rightarrow e \notin dom(X_i) \tag{14.10}$$

Rule (14.6) in detail Rule (14.6) is applied when a value is removed from the domain of S_i and if $i \in ub(\mathcal{I})$. The predicate $\textsc{Possible}_{\mathcal{R},t}(i)$ is evaluated, and if it does not hold true, index i is removed from $ub(\mathcal{I})$. The predicate $\textsc{Required}_{\mathcal{R},t}(i)$ is also evaluated, and if it holds true, index i is added to $lb(\mathcal{I})$. The variable S_i represents the starting times of the i-th event in the CTS. The property $i \in ub(\mathcal{I})$ means exactly that predicate $\textsc{Possible}_{\mathcal{R},t}(i)$ holds true (by Equivalence (14.3)). A possible consequence of removing a value from the domain of S_i is that there may be no more starting time s in S_i such that

$[s, s + d(e)]\,\mathcal{R}\,t$. Therefore, we reevaluate $\textsc{Possible}_{\mathcal{R},t}(i)$, and if it does not hold true anymore, we remove i from $ub(\mathcal{I})$. Another possible consequence of removing a value from S_i is that all remaining values $s \in dom(S_i)$ are such that $[s, s+d(e)]\,\mathcal{R}\,t$ for any event $e \in dom(X_i)$, which means that $\textsc{Required}_{\mathcal{R},t}(i)$ holds true. Equivalence (14.4), we add index i to $lb(\mathcal{I})$.

When X_i is modified, we apply the following rules:

$$
\begin{aligned}
i \in ub(\mathcal{I}) : {} & \neg\textsc{Possible}_{\mathcal{R},t}(i) \Rightarrow i \notin ub(\mathcal{I}) \\
& \textsc{Required}_{\mathcal{R},t}(i) \Rightarrow i \in lb(\mathcal{I}) && (14.11)\\
i \in lb(\mathcal{I}) : {} & dom(X_i) = \{e\} \Rightarrow e \in lb(\mathcal{E}) \\
& s \in dom(S_i) \wedge (\forall e \in dom(X_i), \neg\textsc{Holds}_{\mathcal{R},t}(s, d(e))) \\
& \Rightarrow s \notin dom(S_i) && (14.12)\\
i \notin ub(\mathcal{I}) : {} & s \in dom(X_i) \wedge (\forall e \in dom(X_i), \textsc{Holds}_{\mathcal{R},t}(s, d(e))) \\
& \Rightarrow s \notin dom(S_i) && (14.13)\\
e \in ub(\mathcal{E}) : {} & \nexists j, \textsc{Possible}_{\mathcal{R},t}(j, e) \Rightarrow e \notin ub(\mathcal{E}) && (14.14)\\
e \in lb(\mathcal{E}) : {} & \exists i \in ub(\mathcal{I}) \text{ s.t.} \\
& \quad \textsc{Possible}_{\mathcal{R},t}(i, e) \wedge \\
& \quad \forall j \in ub(\mathcal{I}) \text{ s.t. } j \neq i, (e \notin dom(X_j) \vee \neg\textsc{Possible}_{\mathcal{R},t}(j, e)) \\
& \Rightarrow dom(X_i) = \{e\} \wedge i \in lb(\mathcal{I}) && (14.15)
\end{aligned}
$$

When $\mathcal{I}$ is modified: if $i \in lb(\mathcal{I})$, apply Rule (14.7) and Rule (14.12); if $i \notin ub(\mathcal{I})$ apply Rule (14.8) and Rule (14.13). When $\mathcal{E}$ is modified: if $e \in lb(\mathcal{E})$ apply Rule (14.15); if $e \notin ub(\mathcal{E})$, $\forall i \in ub(\mathcal{I})$, apply Rule (14.10).

Most of those rules are straightforward implications of the predicate definitions, except rule (14.15). The first line of Rule (14.15) says that it is possible to have value e in the sequence. The following lines express the fact that if a only one variable X_i may take value e, we perform the assignment $X_i \leftarrow e$. We can easily verify that no rule removes any consistent value, *i.e.,* the rules are sound. However, this model does not remove all inconsistent values, *i.e.,* it does not achieve arc-consistency for Allen.

14.3.2 MDD-Based Model

This model uses an MDD constraint to represent the extension of the Allen constraint. By using propagators for MDDs, we can therefore achieve arc consistency of the whole Allen constraint. In fact, we can even combine several Allen constraints into a single MDD and thus achieve arc-consistency for a set of Allen constraints.

Defining the Allen constraint with MDDs can be decomposed into two steps:

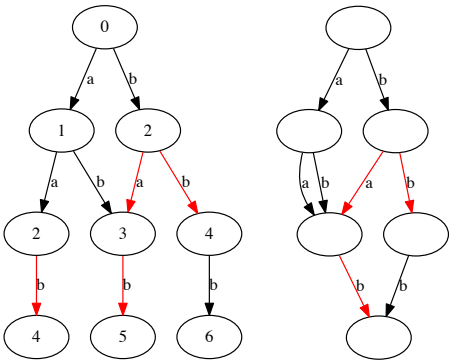


Figure 14.1: The graph (left) and MDD (right) representations of the constraint $\text{ALLEN}_{\mathbf{d}\vee\mathbf{s}\vee\mathbf{fi}\vee\mathbf{eq}[2,5]}$ (see Table 14.2). Red labels correspond to values satisfying the constraint. Numbers in the graph on the left represent the temporal position.

- We first represent the temporal constraint by a transition function computing the set of all temporal positions reachable from a given temporal position. In this MDD, each arc correspond to a temporal intervall associated to an event. The MDD is defined as follow: We create a root node associated to position 0. Then, we successively apply the time cumulative transition function to determine all reachable temporal positions. An arc is associated to a duration, *i.e.,* time difference between the temporal position of its ending node and its origin node, *i.e.,* for and arc $a = (i, j)$, we have $t(j) = t(i) + d(a)$, where $t(.)$ denotes the temporal position of a node. The MDD constructed this way simply represents a sum function. Events are introduced in the MDD as follows: for each arc associated to a duration, we create as many arcs as there are events with this duration. Each arc in the resulting MDD is therefore labeled with a couple (event, duration).

- Then, for a given Allen relation, we identify all the arcs in the MDD that satisfy this relation. We can do this by noting that an arc $a = (i, j)$ occupies the temporal interval $[t(i), t(j)]$. These are the red arcs in Figure 14.1. Using this ALLEN relation as a marking function, we can use the *mddChannel* constraint (chapter 13).

Let M be the MDD defined in the first step, A be the marking function of the MDD using the Allen relation as define for the second step. Let $\mathcal{I}$ and $\mathcal{E}$ be the set variables of the constraint. The constraint $mddChannel(M, A, I, E)$ enforce consistency for the Allen constraint.

X_1	X_2	X_3	$\mathcal{I}$	$\mathcal{E}$
a	a	b	$\{3\}$	$\{b\}$
a	b	b	$\{3\}$	$\{b\}$
b	a	b	$\{2,3\}$	$\{a,b\}$
b	b	b	$\{2\}$	$\{b\}$

Table 14.2: The extension of $\text{ALLEN}_{R[2,5]}$ for the example. Events that are, not strictly, during $[2,5]$ are in red.

Example Consider two events a and b with $d(a) = 1$ and $d(b) = 2$ and a sequence of three variables X_1, X_2, X_3 with domains $dom(X_1) = dom(X_2) = \{a, b\}$ and $dom(X_3) = \{b\}$. Let R denote the relation $\mathbf{d}\vee\mathbf{s}\vee\mathbf{f}\vee\mathbf{eq}$, which is similar to $\mathbf{d}$ except it is not strict. The extension of $\text{ALLEN}_{R[2,5]}([X_1, X_2, X_3], \mathcal{I}, \mathcal{E})$ is shown in Table 14.2, where events that occur, not strictly, during [2,5] are in red.

The list of valid sequences of the constraint may be represented by the graph in Figure 14.1 (left). Each layer represents one sequence variable (X_1 is the top layer, X_2 is the middle layer, and X_3 the bottom layer). Node labels represent start times and edge labels are events. Edges corresponding to events satisfying $\text{ALLEN}_{R[2,5]}$ are in red.

Note that the Allen relation does not change during search. As a consequence, one can ignore the temporal information in the nodes and apply the MDD reduction operation to the graph. This yields the reduced MDD in Figure 14.1 (right). Note that the reduction distinguishes between black and red labels.

The MDD4R algorithm is used to filter the domains of the sequence variables in the constraint represented by the MDD. The set variables $\mathcal{I}$ and $\mathcal{E}$ must satisfy the following properties:

1. if $\forall a \in A_i$, a is red, then $i \in lb(\mathcal{I})$;

2. if $\exists a \in A_i$ such that a is red, then $i \in ub(\mathcal{I})$ and $label(a) \in ub(\mathcal{E})$.

where A_i denotes the i-th layer of the MDD, and a denotes an arc. The MDDArc constraint ensures by definition these two properties.

In practice, the MDD representation is efficient because the bottom layers are compressed. This approach solves problems with up to 150 variables by MDDs, which is enough for the targeted applications (see chapter 18).

An important aspect of this approach it that we represent *several* ALLEN constraints stated on the same sequence in a *single* MDD. Then, we implement

channeling relations between the MDD and the set variables for each relation. This allows us to achieve arc-consistent of the conjunction of all the Allen constraints. Note that integrating the set variables in the MDD would require the definition of one MDD per Allen relation, and would sacrifice compression without improving filtering.

14.4 Experiments

The main problem of the experiments and motivation for the Allen constraint is a music synchronization problem. This is a lead sheet generation problem containing 3 different tracks, one for each instrument. In this problem, the sequences generated for each instrument must respect a given transition constraint, but more important, we need to impose several synchronization points between the instruments.

Thus the problem contains three transition constraints, one for each instrument, which are implemented using MDDs in both of the models. And then, for each synchronization point, three Allen constraints are added to the model, one for each instrument. Moreover, in order to constrain the possible values at the synchronization points, dedicated table constraints are imposed to the set variables of the Allen constraints. The full definition of the experiments is given in chapter 18.

The experiments were run on a MacBook pro late 2013, having a I7 2.3Ghz and 8 Go of rams. The code is implemented using the OR Tools solver [Perron 2013].

14.4.1 Evaluation of the First Model

We evaluate two implementations of the scheduling model, depending on how we implement the constraint which links start times and duration (defined in Section 14.3.1). First, we enforce arc-consistency on this ternary sum constraint. The model solves the problem for two bars in 8.4 seconds. It does not solve the problem for more than two bars in less than 30 minutes, which we consider a timeout.

A lighter version has also been implemented where the ternary sums constraints only perform bound-consistency, based on the intuition that propagating information about the bounds of event duration offers a good trade off between simplicity and pruning. This model solves the problem for two bars in 5.4 seconds, but does not scale either to larger instances.

14.4.2 Evaluation of the MDD-Based Model

Note that, as said in Section 14.3.2, all the Allen constraints on a same track are represented by a single MDD.

n	MDD size (#Vertices, #Edges)						Time
	Guitar		Bass		Drum		(ms)
6	2382	41k	848	13667	1864	73k	2301
8	4199	74k	1493	24k	3817	156k	7219
10	6530	117k	2388	39k	6513	275k	23k
12	9374	169k	3623	61k	9957	429k	57k
14	12k	231k	5085	87k	14k	617k	112k

Table 14.3: The size of the MDDs and the execution time to find 5 solutions for various multitrack lengths

.

The comparison with the performance of the simple model for Allen is clearly in favor of the MDD approach (see Table 14.3). The simple model does not solve problems longer than two bars in less than 30 minutes. In contrast, the MDD-based model solves the 14-bar problem in less than 2 minutes. The extra cost of performing the MDD construction and operations is more than compensated for by the higher pruning offered by this model, especially regarding the treatment of the set variable $\mathcal{E}$.

14.5 Conclusion

This chapter has presented the Allen global constraint. Allen maintains set variables representing events in a temporal sequence in two ways: one variable is the set of events occurring at a given position, defined by an Allen relation with a reference time interval; the other variable is the set of indexes of these events. In practice, Allen offers the possibility to control the generation of temporal sequences by constraining events defined by their index *and* temporal position. Allen makes it possible to model and solve new types of problems involving structural constraints on patterns, represented by sub-sequences.

Two models for Allen have been proposed: a simple model using local propagation and a model based on MDDs. The experimental section has shown that the MDD representation, which achieves the global AC of the constraint, performs much better than the simple model on a temporal sequence synchronization problem.

Chapter 15

Markov and Statistical Constraints

Contents

15.1 Introduction

Several constraints, like `spread` [Pesant 2005], `deviation` [Schaus 2007a, Schaus 2007b, Schaus 2007c, Schaus 2014], `balance` [Beldiceanu 2007, Bessiere 2014] and `dispersion` [Pesant 2015], have mainly been defined to balance certain features of a solution. For example, the balanced academic curriculum problem [bac] involves courses that have to be assigned to periods so as to balance the academic load between periods. Most of the time the mean of the variables is fixed and the goal is to minimize the standard deviation, the distance or the norm.

The `dispersion` constraint is a generalization of the `deviation` and `spread` constraints. It ensures that X, a set of variables, has a mean (i.e. $\mu = \sum_{x \in X} x$) belonging to a given interval and Δ a norm (i.e. $\sum_{x \in X} (x - \mu)^p$) belonging to another given interval. If $p = 1$ then it is a `deviation` constraint and $p = 2$ defines a `spread` constraint. Usually, the goal is to minimize the value of Δ or find a value below a given threshold.

In some problems, variables are independent from a probabilistic point of view and are associated with a distribution (e.g. a normal distribution) that specifies probabilities for their values. Thus, globally the values taken by the variables have to respect that law and we can define a constraint ensuring this property, either by using a `spread`, a `dispersion`, a `KolmogorovSmirnov` or a `Student's t-test` constraint[Rossi 2014]. However, if only a subset of variables is involved in a constraint, then the values taken by these variables should be compatible with the distribution (e.g. the normal law), but we cannot impose the distribution for a subset of values because this is a too strong constraint.

Therefore, we need to consider the interval of values for μ and Δ. The definition of an interval for μ can be done intuitively. For instance we can consider an error rate of 10%. Unfortunately, this is not the case for Δ. It is hard to control the relation between two values of Δ, because data are coming from measures and there are some errors and because it is difficult to apply a continuous law on a finite set of values. Since we use constraint programming solvers we have to make sure that we do not forbid tuples that could be acceptable. This is why, in practice, the problem is not defined in term of μ and Δ but by the probability mass function (PMF).

The probability mass function gives the probability that X_r, a discrete random variable, is exactly equal to some value. In other words, if X_r takes its values in V, then the PMF gives the probability of each value of V. The PMF is usually obtained from the histogram of the values. From f_P, a PMF, we can compute the probability of any tuple by multiplying the probability of the values it contains, because variables are independent. Then, we can avoid outliers of the statistical law but imposing that the probability of a tuple belongs to a given interval $[P_{min}, P_{max}]$. With such a constraint we can select a subset of values from a large set having a mean in a given interval while avoiding outliers of the statistical law. Roughly, the minimum probability avoids having tuples with values having only very little chance to be selected and the maximum probability avoids having tuples whose values have only the strongest probability to be selected.

It is sometimes interesting to define some constraints on the sampling. For instance, the problem of generating the sequences with the maximum probability in the Markov chain estimated from the corpus satisfying other

constraints has been studied by Pachet and Roy [Pachet 2011, Pachet 2001]. Hence this chapter focus on constraints imposing that the probability of the solutions belongs to a given range of probabilities.

The motivation of this chapter is mainly a real world application involving convolutions which are expressed by knapsack constraints (i.e. $\sum \alpha_i x_i$). The experimental section proposes several models for solving the problem.

The chapter is organized as follows. First, it recalls some basics about the dispersion constraint, PMF and Markov processes. Then, it introduce simple models using MDDs for modelling the `dispersion` constraint with a fixed or a variable mean, and shows how we can combine them in order to obtain only one MDD. Next, it presents the `PMF` and markov constraints and show how they can be represented by an MDD and filtered by MDD propagators.

15.2 Definition

15.2.1 Probability distribution

We consider that the probability distribution is given by a probability mass function (PMF), which is a probability density function for a discrete random variable. The PMF gives for each value v, the probability $P(v)$ that v is taken:

Given a discrete random variable Y taking values in $Y = \{v_1, ...v_m\}$ its probability mass function P: $Y \rightarrow [0, 1]$ is defined as $P(v_i) = Pr[Y = v_i]$ and satisfies the following condition: $P(v_i) \geq 0$ and $\sum_{i=1}^{m} P(v_i) = 1$.

Property 12 *Let f_P be a PMF and consider $\{x_i\}$ a set of n discrete integer variables independent from a probabilistic point of view and associated with f_P that specifies probabilities for their values. Then, the probability of an assignment of all the variables (i.e. a tuple) is equal to the product of the probabilities of the assigned values. That is $\forall i = 1..n$, $\forall a_i \in D(x_i)$ $P(a_1, a_2, ..., a_n) = P(a_1)P(a_2)...P(a_n)$.*

15.2.2 Markov chain

A Markov chain[1] is a stochastic process, where the probability for state X_i, a random variable, depends only on the last state X_{i-1}. A Markov chain produces sequence $X_1, ..., X_n$ with a probability $P(X_1)P(X_2|X_1)...P(X_n|X_{n-1})$.

[1]Order k Markov chains have a longer memory: the Markov property states that $P(X_i|X_1, ..., X_{i-1}) = P(X_i|X_{i-k}, ..., X_{i-1})$. They are equivalent to order 1 Markov chains on an alphabet composed of k-grams, and therefore we assume only order 1 Markov chains.[Papadopoulos 2015]

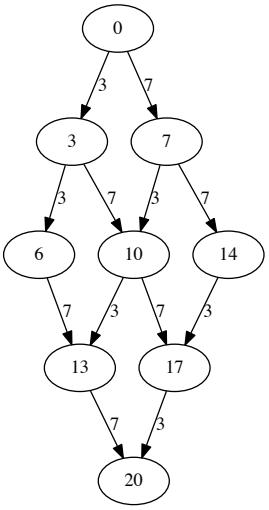


Figure 15.1: MDD of the $\sum x_i = n\mu$ constraint

Property 13 *Let P_M be a Markov chain and consider a set of n discrete integer variables associated with P_M that specifies probabilities for their values. Then, $\forall i = 1..n$, $\forall a_i \in D(x_i)$ $P(a_1, a_2, ..., a_n) = P(a_1)P(a_2|a_1)...P(a_n|a_{n-1})$.*

Several methods can be used to estimate the Markov chain from a corpus, like the maximum likehood estimation [Jurafsky 2014]. Consider that the Markov process is given, see chapter 9 for more about Markov.

15.2.3 MDD of a Generic Sum Constraint

We define the generic sum constraint [Trick 2003]. $\Sigma_{f,[a,b]}(X)$ which is equivalent to $a \leq \sum_{x_i \in X} f(x_i) \leq b$, where f is a non negative function. The MDD of the constraint $\sum_{x_i \in X} f(x_i)$ is defined as follows. For the layer i, there are as many nodes as there are values of $\sum_{k=1}^{i} f(x_k)$. Each node is associated with such a value. A node n_p at layer i associated with value v_p is linked to a node n_q at layer $i+1$ associated with value v_q if and only if $v_q = v_p + f(a_i)$ with $a_i \in D(x_i)$. Then, only values v of the layer $|X|$ with $a \leq v \leq b$ are linked to tt. The reduction operation is applied after the definition and delete invalid nodes (see chapter 3). The construction can be accelerated by removing states that are greater than b or that will not permit to reach a. For convenience, $\Sigma_{id,[\alpha,\alpha]}(X)$ is denoted by $\Sigma_\alpha(X)$.

Example:
Figure 15.1 is an example of MDD($\Sigma_{20}(X)$) with $\{3, 7\}$ as domains.

Since f_C is non negative, the number of nodes at each layer of MDD($\Sigma_{f,[a,b]}(X)$) is bounded by b.

15.2.4 Dispersion Constraint

Given $X = \{x_1, ..., x_n\}$, a set of finite-domain integer variables, μ and Δ, bounded-domain variables and p a natural number. The constraint DISPERSION(X, μ, Δ, p) states that the collection of values taken by the variables of X exhibits an arithmetic mean $\mu = \sum_{i=1}^{n} x_i$ and a deviation $\Delta = \sum_{i=1}^{n} |x_i - \mu|^p$ [Pesant 2015]..

The `deviation` constraint is a `dispersion` constraint with $p = 1$ and a `spread` constraint is a `dispersion` constraint with $p = 2$.

The main complexity of the `dispersion` constraint is the relation between μ and Δ variables, because μ is defined from the X variables, and Δ is defined from X and from μ. So, some information is lost when these two definitions are considered separately. However, when μ is assigned, the problem becomes simpler because we can independently consider the definitions of μ and Δ. Therefore, this chapter studies some models depending on the fact that μ is fixed or not.

15.3 Dispersion Constraint

15.3.1 Dispersion Constraint with fixed mean

Arc consistency for the X variables has been established by Pesant [Pesant 2015], who proposed an ad-hoc dynamic programming propagator for this constraint. This algorithm builds an acyclic graph representing the sum for the μ and the cost function then the cost-regular propagator handle it.

However, it exists a simpler method avoiding problems of ad-hoc algorithms: we define a cost-MDD from μ and Δ and obtain a propagator having the same complexity.

15.3.1.1 MDD on μ and Δ as cost

The mean μ is defined as a sum constraint. Since μ is fixed, we propose to use the cost-MDD of the constraint $\sum x_i = n\mu$ and the cost function defined by $\Delta = \sum_{i=1}^{n} |x_i - \mu|^p$.

The constraint $\sum x_i = n\mu$ can be represented by MDD($\Sigma_{n\mu}(X)$).

Δ as cost. We represent the dispersion constraint by cost-MDD$(\Sigma_\mu(X), \Delta)$. There are two possible ways to deal with the boundaries of Δ. Either we define two cost-MDD propagators on cost-MDD$(\Sigma_\mu(X), \Delta)$, one with a and $\geq$, and one with b and $\leq$; or we define only one cost-MDD propagator on cost-MDD$(\Sigma_\mu(X), \Delta)$ which integrates the costs at the same time as proposed by Hoda et al [Hoda 2010].

These methods are simpler than Pesant's algorithm because they do not require to develop any new algorithm. If we use an efficient algorithm (chapter 11) for maintaining arc consistency for cost-MDDs then we obtain the same worst case complexity as Pesant's algorithm but better result in practice.

15.3.1.2 MDD on μ intersected with MDD on Δ

Since μ is fixed, the definition of Δ corresponds to a generic sum as previously defined. Thus, the dispersion constraint can be model by defining the MDD of $\Sigma_\mu(X)$ and the MDD of $\Sigma_{\Delta,[\underline{\Delta},\overline{\Delta}]}(X)$ and then by intersecting them. Replacing a cost-MDD by the intersection of two MDDs may strongly improve the computational results (chapter 11). In addition, we can intersect the resulting MDD with some other MDDs in order to combine more the constraints. This method is the first method establishing arc consistency for both μ and Δ. The drawback is the possible size of the intersection.

With similar models we can also give an efficient implementation of the `Student's t-test` constraint and can be used to close the open question of Rossi et al.[Rossi 2014].

15.3.2 Dispersion Constraint with variable mean

In order to deal with a variable mean, we can consider all acceptable values for $n\mu$, that is the integers in $[n\lfloor\underline{\mu}\rfloor, n\lceil\overline{\mu}\rceil]$, and for each value we separately apply the previous models for the fixed mean. Unfortunately, this often leads to a large number of constraints. Therefore it is difficult to use this approach in practice. In addition, note that there is no advantage in making the union of these constraints because they are independent.

Thus, the next section proposes another model using the probability mass function.

15.4 Probabilities Based Constraint

15.4.1 MDDs and Probabilities based constraints

For some reasons, like security or for avoiding outliers of the statistical laws, some paths of MDDs can be unwanted, because they have only very little chance to be selected or because they contain almost only values having the strongest probability to be selected. In other words, we accept only paths whose probability is in a certain interval.

We define constraints for this purpose. One, named the `MDDProbability`, considered that the MDD is associated with a PMF and independent variables and the other, named `MDDMarkovProcess`, that the MDD is associated with a Markov chain.

Definition 13 *Given M an MDD defined on $X = \{x_1, x_2, ..., x_n\}$ that are independent from a probabilistic point of view and associated with f_P a probability mass function , P_{min} a minimum probability and P_{max} a maximum probability. The constraint `MDDProbability`$(X, f_P, M, P_{min}, P_{max})$ ensures that every allowed tuple $(a_1, a_2, ...a_n)$ is a solution of the MDD and satisfies $P_{min} \leq \Pi_{i=1}^{n} f_P(a_1) \leq P_{max}$.*

This constraint can be easily transformed into cost-MDD constraints. The cost associated with an arc labeled by a is $\log(f_P(a))$, and the logarithms of P_{min} and P_{max} are considered for dealing with a sum instead of a product[2]. Thus, any cost-MDD propagator can be used (see chapter 11).

Definition 14 *Given M an MDD defined on $X = \{x_1, x_2, ..., x_n\}$ and associated with P a Markov chain, P_{min} a minimum probability and P_{max} a maximum probability. The constraint `MDDMarkovProcess`$(X, P, M, P_{min}, P_{max})$ ensures that every allowed tuple $(a_1, a_2, ...a_n)$ is a solution of the MDD and satisfies*
$P_{min} \leq P(a_1)P(a_2|a_1)...P(a_n|a_{n-1}) \leq P_{max}$.

As we have seen, with a Markov chain, the probability for selecting an arc depends on the previous selected arc. Thus, each arc of the MDD is associated with several probabilities. So we cannot directly use a cost-MDD propagator as for the `MDDProbability` constraint. However, if we accept to duplicate the nodes as proposed in the previous section then we can immediately transforms the constraint into a simple cost-MDD constraint by considering logarithms of probabilities and any cost-MDD propagator can be used. Since the number of time a node can be duplicated is bounded by d, the overall complexity of this transformation is $O(d \times (|V| + |E|))$.

[2]We can also directly deal with products if we modify the costMDD propagator accordingly.

15.4.2 Probabilities and Means

In this section we define the `PMF` constraint which aims at respecting a variable mean and avoiding outliers according to a statistical law given by a probability mass function.

The PMF gives for each value v, $P(v)$ the probability that v is taken. Let f_P be a PMF and consider a set of variables independent from a probabilistic point of view and associated with f_P that specifies probabilities for their values. Since the variables are independent, we can define the probability of an assignment of all the variables (i.e. a tuple) as the product of the probabilities of the assigned values. Then, in order to avoid outliers we can constrain this probability to be in a given interval.

Definition 15 *Given a set of finite-domain integer variables* $X = \{x_1, x_2, ..., x_n\}$ *that are independent from a probabilistic point of view, a probability mass function* f_P*, a bounded variable* μ *(not necessarily fixed), a minimum probability* P_{min} *and a maximum probability* P_{max}*. The constraint* $\text{PMF}(X, f_P, \mu, P_{min}, P_{max})$ *states that the probabilities of the values taken by the variables of* X *is specified by* f_P*, the collection of values taken by the variables of* X *exhibits an arithmetic mean* μ *and that* $\Pi_{x_i \in X} x_i$ *the probability of any allowed tuple satisfies* $P_{min} \leq \Pi_{x_i \in X} f_P(x_i) \leq P_{max}$*.*

This constraint can be represented by $\texttt{MDDProbability}(X, f_P, \Sigma_{id,[\underline{\mu},\overline{\mu}]}(X), P_{min}, P_{max})$, note that arc consistency can be enforce on the variable μ using the $\Sigma_{id,[\underline{\mu},\overline{\mu}]}$ mdd.

15.5 Experiments

The experiments were run on a macbook pro (2013) Intel core i7 2.3GHz with 8 GB. The constraint solver used is or-tools. MDD4R is used as MDD propagator and cost-MDD4R as cost-MDD propagator.

The data come from a real life application: the geomodeling of a petroleum reservoir [Pennington 2001]. The problem is quite complex and we consider here only a subpart. Given a seismic image we want to find the velocities. Velocities values are represented by a probability mass function (PMF) on the model space. Velocities are discrete values of variables. For each cell c_{ij} of the reservoir, the seismic image gives a value s_{ij} and the from the given seismic wavelet (α_k) we define a sum constraint $\sum_{k=1}^{22} \alpha_k log(x_{i-11+k-1}j) = s_{ij}$. Locally, that is for each sum, we have to avoid outliers w.r.t. the PMF for the velocities. Globally we can use the classical dispersion constraint. The problem is huge (millions of variables) so we consider here only a very small part.

The models used a briefly given here, a full description is given in chapter 19.

The first experiment involves 22 variables and a constraint C_α: $\sum_{i=1}^{n} \alpha_i x_i = I$, where I is an tight interval (i.e. a value with an error variation). C_α is represented by $mdd_\alpha = \text{MDD}(\Sigma_{a_i,I}(X))$ where $a_i(x_i) = \alpha_i x_i$.

First, we impose that the variables have to be distributed with respect to a normal distribution with μ, a fixed mean.

- $M_{\sigma<,\sigma>}$ represents the model of Section 3.2: one cost-MDD propagator on $mdd_\mu = \text{cost-MDD}(\Sigma_{n\mu}(X), \sigma)$ with σ and $\leq$ and one with σ and $\geq$. This model is similar to Pesant's model.

- M_{GCC} involves a GCC constraint [Régin 1996] where the cardinalities are extracted from the probability mass function.

- $M_{\mu\cap\sigma}$ represents the mean constraint by $mdd_\sigma = \text{MDD}(\Sigma_{n\mu}(X))$. It represents the sigma constraint by the $\text{MDD}(\Sigma_\sigma(X))$. Then the two MDDs are intersected. An MDD propagator is used on this MDD, named $mdd_{\mu\sigma}$. See Section 3.3.

- $M_{\mu\cap\sigma\cap\alpha}$ intersects mdd_α, the MDD of the constraint C_α, with $mdd_{\mu\sigma}$ the previous MDD to obtain mdd_{sol}. In this case, all constraints are combined.

- $M_{\mu\cap\sigma\cap\alpha}$ intersects mdd_α, the MDD of the constraint C_α, with $mdd_{\mu\sigma}$ the previous MDD to obtain mdd_{sol}. In this case, all constraints are combined.

		Fixed μ				Variable μ	
Sat?	#sol	$M_{\sigma<,\sigma>}$	M_{GCC}	$M_{\mu\cap\sigma}$	$M_{\mu\cap\sigma\cap\alpha}$	M_{log}	$M_{log\cap\alpha}$
Sat	Build	50	31	138	2,203	34	317,921
	10 sol	14	T-O	16	0	14	0
	All sol	T-O	T-O	T-O	0	T-O	0
UnSat	Build	55	28	121	151	37	133,752
	10 sol	T-O	T-O	T-O	0	T-O	0
	All sol	T-O	T-O	T-O	0	T-O	0

Figure 15.2: Comparison solving times (in ms) of models. 0 means that their is no need of solver. T-O indicates a time-out of 500s.

		Fixed μ					Variable μ		
Sat?	N/A	mdd_α	mdd_μ	mdd_σ	$mdd_{\mu\sigma}$	mdd_{sol}	mdd_{I_μ}	$mdd_{I_{log}}$	$mdd_{log\alpha}$
Sat	#nodes	3	3	5	67	521	2	18	24,062
Sat	#arcs	44	27	55	660	4,364	30	268	341,555
UnSat	#nodes	3	2	5	67	0	2	18	0
UnSat	#arcs	46	27	55	660	0	30	268	0

Figure 15.3: Comparison of MDD sizes (in thousands) of different models. 0 means that their is no need of solver.

Then, we consider a `PMF` constraint and that μ is variable:

- M_{log}. We define a cost-MDD propagator on mdd_{I_μ} = cost-MDD$(\Sigma_{id,[\underline{\mu},\overline{\mu}]}(X), logP)$ with $\log(P_{min})$ and $\geq$ and with $\log(P_{max})$ and $\leq$. See Section 5.
- $M_{log\cap\alpha}$. We define $mdd_{I_{log}}$ = MDD$(\Sigma_{logP,I_{log}}(X))$ and we intersect it with mdd_{I_μ}. Then, we intersect it with mdd_α, the MDD of C_α, to obtain $mdd_{log\alpha}$.

Table 15.2 shows the result of these experiments. As we can see when the problem involves many solutions, all the methods perform well (excepted M_{GCC}). We can see that an advantage of the intersection methods is that they contain all the solutions of problem. Table 15.3 shows the different sizes of the MDDs.

Random instances. The intersection methods $M_{\mu\cap\sigma}$ and $M_{\mu\cap\sigma\cap\alpha}$ have been tested on random bigger instances. Table 15.4 and 15.5 gives some results showing how this method scales with the number of variables. In the first line, the couple is #var/#val. Times are in ms. Experiments of Table 15.4 set $0 < \sigma < 4n$ for having a delta depending on the number of variables like

$0 < \sigma < 4n$									
Method	n/d	20/20	30/20	40/30	40/40	50/40	50/50	100/40	100/100
$M_{\mu\cap\sigma}$	T(ms)	26	132	391	401	848	875	12,780	14,582
	#nodes	18	63	153	1578	306	311	2,285	2,532
	#arcs	198	808	2,308	2,427	5,196	5,354	53,757	62,057
$M_{\mu\cap\sigma\cap\alpha}$	T(ms)	561	3,084	11,864	10,789	58,092	60,513	M-O	M-O
	#nodes	163	764	0	0	0	0	M-O	M-O
	#arcs	1,788	8,416	0	0	0	0	M-O	M-O

Figure 15.4: Time (in ms) and size (in thousands) of the MDDs of models $M_{\mu\cap\sigma}$ and $M_{\mu\cap\sigma\cap\alpha}$. 0 means that the MDD is empty. M-O means memory-out.

$100 < \sigma < 400$					
Method	n/d	20/20	30/20	40/30	40/40
$M_{\mu\cap\sigma}$	T(ms)	162	333	586	602
	#nodes	81	184	326	338
	#arcs	823	1,865	3,329	3,479
$M_{\mu\cap\sigma\cap\alpha}$	T(ms)	2,663	10,379	21,063	26,393
	#nodes	1,098	2,555	35	0
	#arcs	11,166	23,764	151	0

Figure 15.5: Time (in ms) and size (in thousands) of the MDDs of models $M_{\mu\cap\sigma}$ and $M_{\mu\cap\sigma\cap\alpha}$. $M_{\mu\cap\sigma\cap\alpha}$ is empty because there is no solution.

in [Pesant 2015], whereas experiments of Table 15.5 impose $100 < \sigma < 400$, these numbers come from our real world problem.

These experiments show that the $M_{\mu\cap\sigma}$ model can often be a good trade-off between space and time. Using the lower bound of the expected size of the MDD (chapter 11), we can estimate and decide if it is possible to process $M_{\mu\cap\sigma\cap\alpha}$. The last two columns of Table 15.4 show that it is not always possible to build such an intersection.

15.6 Conclusion

This chapter has shown that modeling constraints by MDDs has several advantages in practice. It avoids to develop ad-hoc algorithms, gives competitive results and leads to efficient combination of constraints outperforming the other approaches.

Moreover, we are now able to constrain the probability of solution considering statistical distribution during the search.

Finally, the experimental section has shown the advantage of such method for solving the geomodeling of a petroleum reservoir.

Chapter 16

Unefficient MDDs

Contents

16.1 Introduction

This chapter is an outsider considering problems where building an MDD is not really a good idea. But more efficient algorithms must be defined.

We propose here to focus on building the MDD of the alldifferent constraint, to integrate set variables in MDDs and to use MDDs for handling the Pareto constraint.

16.2 AllDifferent

Building MDDs can be done using state machine [Hooker 2013]. The *state* of an node in a MDD representing the allDifferent constraint containing n variables and n values can be given by a Boolean vector having n digits. If the digit at cell i is 0, then the value can be taken, otherwise the value cannot be taken. The number of different state for this MDD is 2^n, but it represents $n!$ solutions, thus even if the MDD is huge, its compression ratio is exponential. An example over 4 variables is given in Figure 16.1.

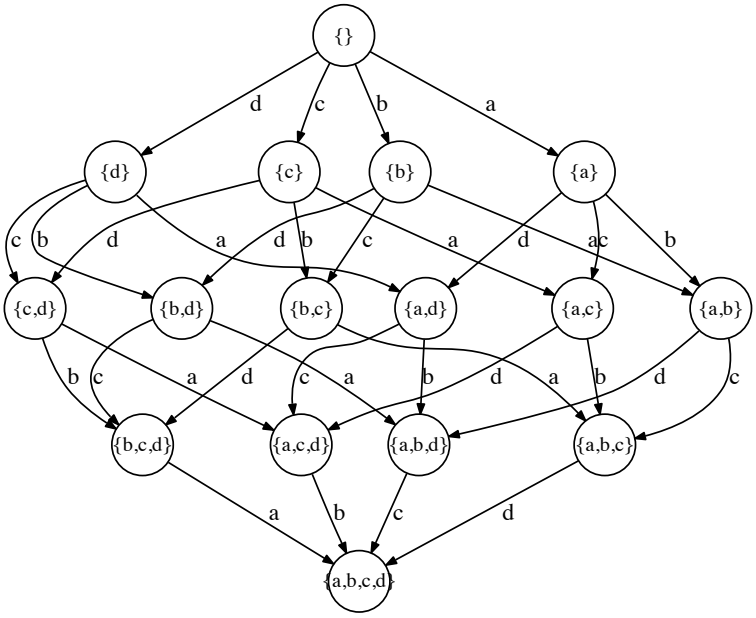


Figure 16.1: MDD representing the solutions of an allDifferent constraint over four variables having the set $\{a, b, c, d\}$ as domain.

Given an allDifferent constraint having n distinct variables and d values, with $d \geq n$, the number of possible states is given by:

$$\sum_{i=1}^{n} \binom{n}{i} \tag{16.1}$$

Directly defining an MDD for this constraint can be costly. That's why efficient algorithms handling MDD has been defined [Andersen 2007, Cire 2013]. Thanks to them, we are able to incorporate the allDifferent information into MDD and to efficiently remove incosistent arcs.

16.3 Set Variables

Instead of using the constraining method proposed in chapter 13, we could be interesting in integrating the set variables into the MDD. Given a set variable containing d values in its set, the number of possible assignment is 2^d, while classical variables with d values have d possible assignment. Thus since an MDD has an arc for each value, possibly by node at the layer of the variable, having set variables inside the MDD instead of using at the top as proposed in this thesis can lead to an exponential decompression.

16.4 Pareto

Multi-objective combinatorial optimization (MOCO) problems are present in many industrial applications. A lot of works has been done to handle this kind of problem, one of the principal problem is the set of Pareto solutions [Finkel 1974, Gavanelli 2002, Hartert 2014, Mostaghim 2002, Schaus 2013].

In a MOCO, variables has a discrete domain, and the problem has several incomparable objective function $p \geq 2$. A good solution of this problem is a non-dominated solution.

Definition 16 *Let $\vec{u}$ and $\vec{v}$ be solutions of a MOCO. $\vec{v}$ is weakly-dominated $\vec{u}$ ($\vec{u} \preceq \vec{v}$) iff:*

$$\vec{u} \preceq \vec{v} \iff \forall i \in [1, p], \vec{u}[i] \leq \vec{v}[i] \tag{16.2}$$

Definition 17 *Let $\vec{u}$ and $\vec{v}$ be solutions of a MOCO. $\vec{v}$ is dominated $\vec{u}$ ($\vec{u} \prec \vec{v}$) iff:*

$$\vec{u} \prec \vec{v} \iff \vec{u} \preceq \vec{v} \wedge \exists i \in [1, p], \vec{u}[i] < \vec{v}[i] \tag{16.3}$$

Definition 18 *A set P of solutions is pareto optimal iff:*

$$\forall \vec{v} \in P, \nexists \vec{u} \in P, \vec{u} \prec \vec{v} \tag{16.4}$$

In MOCO, solvers aim at generating all the pareto solutions. One of the most important part is the data structure used to store all these pareto solutions. Existing methods are described in the section 16.4.1. In the section 16.4.3 is describe another efficient method to store the pareto set using MDDs.

16.4.1 Storing the Pareto solutions

Several data structures exist for storing the set of pareto solutions. In this section, we recall the state of the art of data structures used to store the pareto front.

List: One of the simplest and often used is the list. The list contains all the pareto solution vectors.

Quad-Trie: The Quad-trie are one of the most efficient data structure used to store the pareto set of solution. A Quad-Trie is a trie where each node contains 2^p childs.

Several others methods exist, but these two ones are the most used in practice. The complexity of these two data structures for storing and checking if a solution is or is not dominated by another solution is given in table 16.4.3.

16.4.2 Pareto Constraint

The Pareto constraint well defined in [Schaus 2013], is used to allow the CP solvers to deals with MOCO. This global constraint is define on the objectives variables and ensures that all the next solution is non-dominated.

Definition 19 *Let S be a storing data structure for the current pareto set. The Pareto global constraint pareto($obj_1, obj_2...obj_p, A$) ensures that it exists a affectation of the objectives variables $\vec{v} = (obj_1, obj_2...obj_p)$ such that:*

$$\nexists \vec{u} \in S, \vec{u} \prec \vec{v} \tag{16.5}$$

They generally performs Bound Consistency (BC) on the objective variables by checking for each if the tuple composed of the lower bound for all the objective variables except one using its upper bound is possible.

Definition 20 *Let S be a storing data structure for the current Pareto set. The Pareto global constraint pareto($obj_1, obj_2...obj_n, A$) is Bound Consistant (BC) iff:*

$$\forall i \in [1, p] \ \nexists \vec{u} \in S, \vec{u} \prec \{\underline{obj_1}, \underline{obj_2}, ..., \overline{obj_i}, ..., \underline{obj_p}\} \tag{16.6}$$

Current implementation of this constraint use list or quad-tree and a specialized algorithm. In this section we present another method using MDDs as a storage for the Pareto set. This allows any CP solver containing a Multi-valued Decision Diagrams package to deal with MOCO.

16.4.3 MDD as a store for the Pareto set

In this section we describe how we can use an MDD as a storing method for the Pareto set.

A data structure used to represent the Pareto set has to be able to deal with two requests:

1. *Check*: Given a solution, the data structure has to answer if the solution is dominated or not by a solution inside the data structure.

2. *Insertion*: Given a solution, the data structure have to introduce the solution is the solution is not dominated, and remove all the solution already inside the data structures dominated by the new solution.

We want the MDD to contains all the solutions non-dominated by any solution of the pareto set. using this MDD, the *Check* request is really fast because we only need to know if a given path exist in the MDD. For the

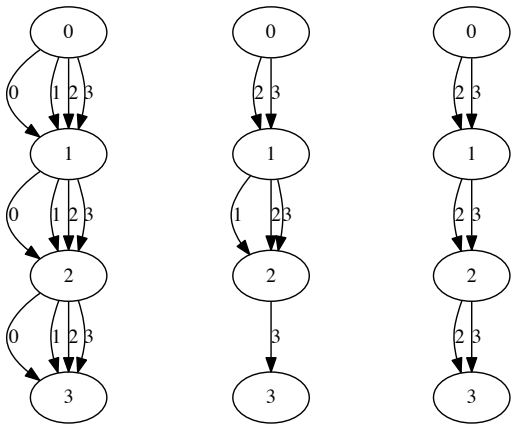


Figure 16.2: The Left MDD is the initial MDD with domain = {0,1,2,3}, the middle MDD represents the GCS $\{\{2,3\}\{1,2,3\},\{3\}\}$ from the solution $\{2,1,3\}$. The right MDD represents the GCS $\{\{2,3\}\{2,3\},\{2,3\}\}$ from the solution $\{2,2,2\}$

second request, we will perform modification in the MDD in order to update its solutions.

Let the solution $(v_1, v_2, ..., v_p)$ be discovered, then all the dominated solutions inside the MDD have to be removed. Since a solution is dominated iff all the p values are greater or equal to the dominating solution. We can represent the set of solution by the following GCS: $\{\{v_1, v_1+1, ..., d\}\{v_2, v_2+1, ..., d\}...\{v_p, v_p+1, ..., d\}\}$.

Example:
An example of such a GCS is given in Figure 16.2 for the solutions {2,1,3} and {2,2,2}.

Thanks to the In-Place deletion algorithm (Chapter 5), we can simply remove from our MDD representing all the previously non-dominated solutions, the GCS containing all the newly dominated solution.

Initialisation: At first, no solution are dominated, then building an MDD representing all the possible solutions is enough. The MDD representing the GCS $\{\{1,2,...,d\}\{1,2,...,d\}...\{1,2,...,d\}\}$ represents all the solutions. An example of initial MDD is given in Figure 16.2 for three variables with the set {0,1,2,3} as domain.

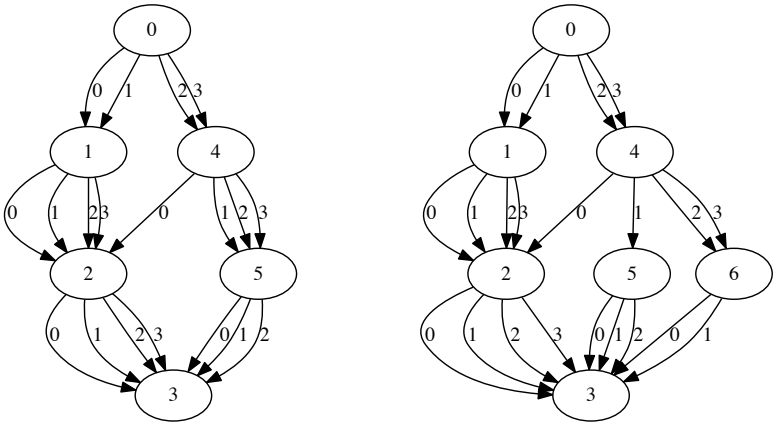


Figure 16.3: The left MDD is the result of the deletion from the initial MDD (Left MDD in Figure 16.2) of all the dominated solutions from the solution $\{2, 1, 3\}$ (MDD in the middle of Figure 16.2). The right MDD is the result of the deletion from the right MDD of all the dominated solutions from the solution $\{2, 2, 2\}$ (Right MDD in Figure 16.2

Example:

An example is given in Figure 16.3. It shows how the MDD handle the deletion of a set of solutions. The first MDD (in the left) is the result of deleting the GCS $\{\{2, 3\}\{1, 2, 3\}, \{3\}\}$ (MDD in the middle of Figure 16.2) from the solution $\{2, 1, 3\}$. As we can see the Resulting MDD represents all the non-dominated solution. This implies that each path in the MDD lead to an affectation of the objective variables that is non-dominated by the solution $\{2, 1, 3\}$.

The right MDD from Figure 16.3 represent the deletion from the previous MDD of all the solutions dominated by the solution $\{2, 2, 2\}$, to do so, the GCS has been created (right MDD from Figure 16.2) and then deleted from this MDD. The resulting MDD is an MDD containing all the solution non-dominated by the 2 previous solutions.

Complexity: The table 16.4.3 gives the different complexities of the two requests, for the MDDs and for the existing methods. As we can see, the MDDs are really efficient at checking if a solution is dominated or not. But the deletion of a solution can need the evalution of the whole MDD.

Operation	MDDs	List	Quad-trie
Check	O(p)	O(|P|*p)	O(|P|*p)
Insertion	O(MDD)	O(|P|*p)	O(|P|*p)

Table 16.1: Complexity of check and insertion for the pareto storing data structure, |P| denotes the number of exising pareto solutions, p denote the number of objectives.

16.4.4 Why does this Fail?

The problem came from the insertion operation. The complexity is in $O(MDD)$, there is a factor of two. Thus, each time a solution is inserted, the size of the MDD can be twice more than its previous size.

This implies that if we insert k Pareto solution, the size of the MDD can growth up to 2^k. Thus using MDD to directly store the non dominated solution is not really a good idea.

Note that we also investigate how to use MDD for storing the pareto front, instead of the non dominated solution. This lead to a more complex algorithm, which seems to be slower than the existing methods when the number of objectives grows.

Related Even if we should not directly use MDDs for storing the pareto front, some works between MDDs and Pareto have been done. For example in [**?**], the authors process Pareto shortest paths in MDDs.

16.5 Conclusion

This chapter has presented several problems where MDDs fail to solve the problem. It is interesting to detect efficiently such problem.

Most of the time, the problem come from the number of *states* of the DP represented by the MDD. But as showed for the Pareto constraint, the complexity is sometimes hidden.

Part V

Applications

Chapter 17

MaxOrder

Contents

17.1 Introduction

The MaxOrder problem [Papadopoulos 2014] consists on, starting from a corpus, generating sequences using a transition function, but preventing the subsequences of a certain length of coming from the corpus.

For example a goal can be, using a corpus of books, to generate sequences of words, where each subsequence of size 2 belongs to the corpus (Markovian transition) and no subsequence of size 4 belongs to the corpus. Here 4 denotes the maximum plagiarism size. suppose that the corpus is the word *electricity*. we can generate the word *citri* or *lecit* but not *lecity* because *city* is a subsequence of length 4 from the corpus

Plan This chapter is split as follows, first it recalls the model proposed by the original authors, then several models are presented for CP solvers. Finally, a soft version of the problem is considered, which aims to minimize the number of plagiarisms or the number of invalid transitions.

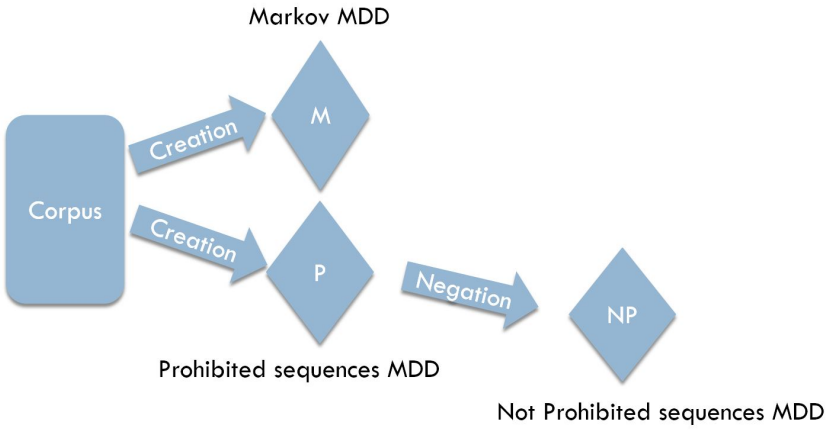


Figure 17.1: MaxOrder model 1, extraction of the constraints from the corpus.

17.2 Models

Let $X = x_1, x_2, ...x_r$ be the variables, let B be the corpus, let M be the transition between the words extracted from the corpus. Let p be the minimum plagiarism size for the subsequences.

The original model [Papadopoulos 2014] first builds the automaton from the transition function. Then, it builds the trie containing all the no-goods (plagiarism subsequences). Finally, it removes from the automaton the trie of no-goods using automaton algorithms. The resulting automaton can then be handled by any CP solver.

MDD models This problem can be solved using several distinct models. This section presents 3 different MDD models for solving the problem, each with different efficiency.

17.2.1 Model 1

The first model is twofold. For the transition constraint we can either use binary constraints, since the value of x_i depends on the value of x_{i-1}, or we can use the transition function for building an MDD.

For the plagiarism constraint, we build mdd_P, the MDD containing all the prohibited sequences contained in the corpus. Thus all the sequences of size p in B. Then we can build $mdd_{\bar{P}}$, the complementary of mdd_P, thus the MDD containing all the sequences of length p that do not belong to the corpus. Note that the sequences of $mdd_{\bar{P}}$ may not respect the transition function M. The Figure 17.1 shows the extraction of the model from the corpus.

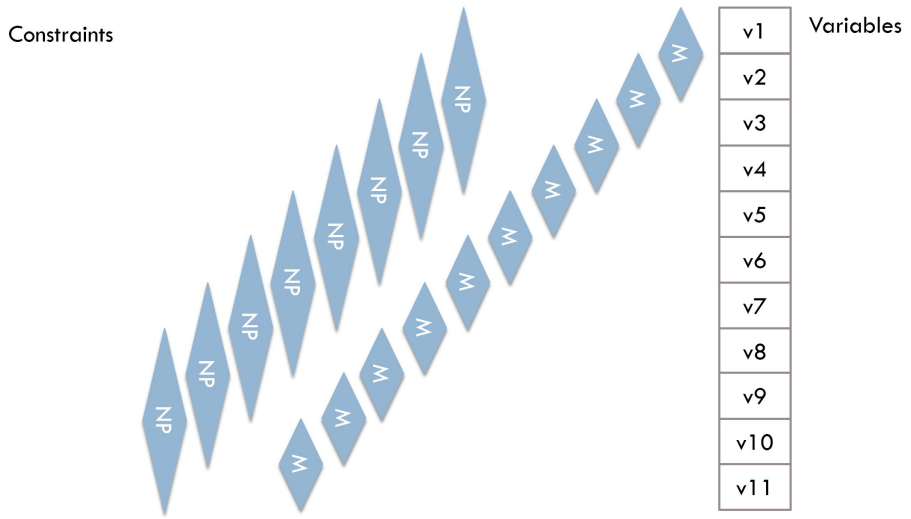


Figure 17.2: MaxOrder model 1, 11 variables, $p = 3$.

Using this MDD, we can build the model for constraining the sequences. For each 2 consecutive variables put a binary transition constraint. For each p consecutive variables, put an MDD constraint using $mdd_{\bar{P}}$. The solutions of this model are the sequences that respect the transition constraint and that do not contain any plagiarism subsequences.

Example Figure 17.2 shows the model applied to 11 variables and with $p = 3$. As we can see, all 3 consecutive variables are constrained by an MDD constraint using the MDD $mdd_{\bar{P}}$ (NP in the Figure). M is the binary transition function.

Remark: This model is weaker than the automaton model, but simpler, smaller in space and faster to build.

17.2.2 Model 2

The second model is close to the idea of [Papadopoulos 2014]. The first step is to build mdd_T, the MDD defined for p variables containing all the sequences composable using the transition function.

The second step is to build mdd_P, the MDD containing all the prohibited sequences contained in the corpus. Then using the operator for MDD, we can build $mdd_A = mdd_T - mdd_P$, the MDD composed of all the sequences composable using the transition function minus all the prohibited sequences. Thus this MDD contains all the solutions for a set of p variable. Figure 17.3 shows the extraction of the model from the corpus.

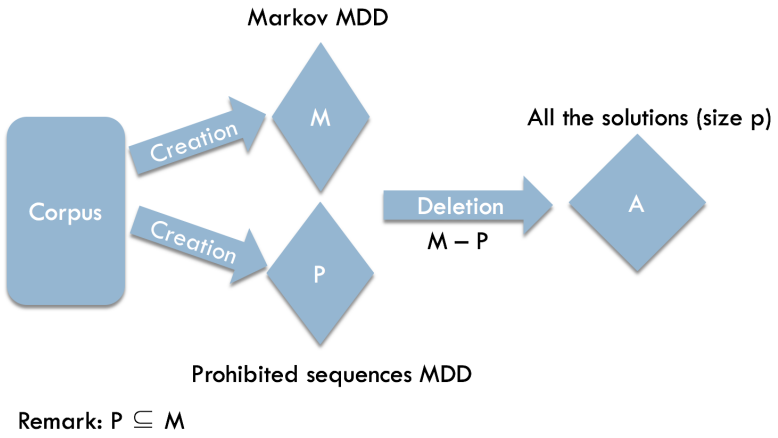


Figure 17.3: MaxOrder model 2, extraction of the constraints from the corpus.

Using mdd_A, we can build a model for the problem by putting an MDD constraint using mdd_A for all the p consecutive variables.

Example Figure 17.4 shows the model applied to 11 variables and with $p = 3$. As we can see, all the 3 consecutive variables are constrained by an MDD constraint using mdd_A.

Remark: This model is still weaker than the automaton model, since the automaton model contains a kind of intersection of these constraints. But still simpler, smaller in space and faster to build.

17.2.3 Model 3

The third model uses the mdd_A defined in the second model, the MDD containing all the solutions for p variables. The problem here is to constrain the r variable and not p variables. The search, in addition to the constraint, gives us the solution that respect the conjunction of the constraints, thus we are going to process directly the intersection of the constraints.

Model 2 defines an MDD constraint for all the p consecutive variables. This implies $r - p + 1$ constraints. The ith constraint is defined over the variables $(x_i,x_{i+1},x_{i+2},...,x_{i+p})$ and the $i + 1$th constraint is defined over the variables $(x_{i+1},x_{i+2},x_{i+3},...,x_{i+p+1})$. If we intersect these two MDD constraints, we obtain an MDD constraint defined over the variables $(x_i,x_{i+1},x_{i+2},...,x_{i+p+1})$. Note that the intersection has to take care of the variables of the MDD.

Finally, if we intersect all these MDD constraints we obtain mdd_G, an MDD defined over the variables $(x_i,x_{i+1},x_{i+2},...,x_{i+r})$ and that contains all the solution of the problem. Figure 17.5 shows how to build this model.

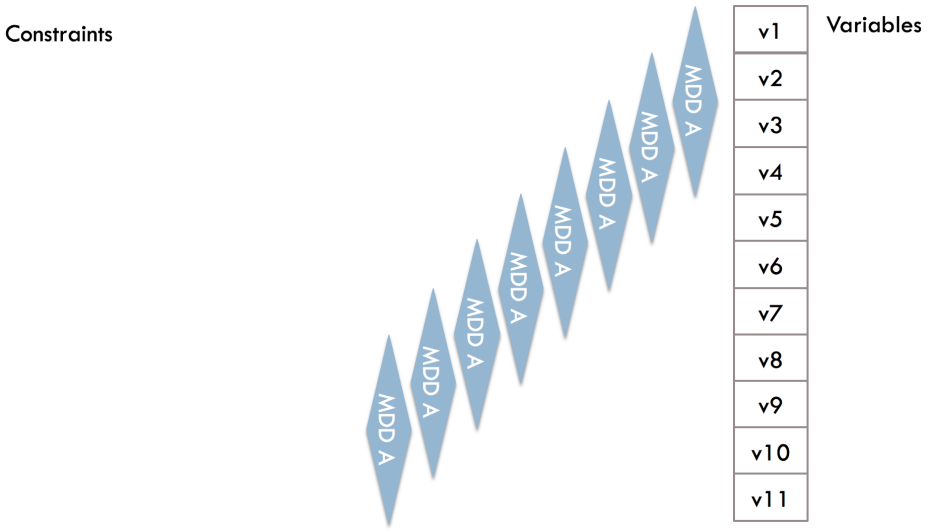


Figure 17.4: MaxOrder model 2, 11 variables, $p = 3$.

mdd_G corresponds exactly to the reduced version of the unfolded automaton from the original model.

Intersection The choice of the order in which the intersections are made has a strong impact on the construction time. We focus here on 3 different ordering for the intersections.

Intersection order (a) The first one is to intersect the MDD constraint from the variable 1 to r. Thus this implies intersecting the constraint 1 and 2 defined over the $p+1$ first variables. Then intersecting the result of the first intersection with the constraint 3, thus obtaining a constraint over the $p+2$ first variables. Then continuing until all the constraints have been intersected. See ordering (a) in Figure 17.6.

Intersection order (b) The second ordering is simply the reverse of the first one. Instead of starting at the constraint 1, we start at the constraint $r-p+1$ and intersect it with the constraint $r-p$. See ordering (b) in Figure 17.6.

Intersection order (c) This third ordering uses the following remark: intersecting the MDD constraint defined over the variables $(x_i,x_{i+1},x_{i+2},...,x_{i+p})$ with the MDD constraint defined over the variables

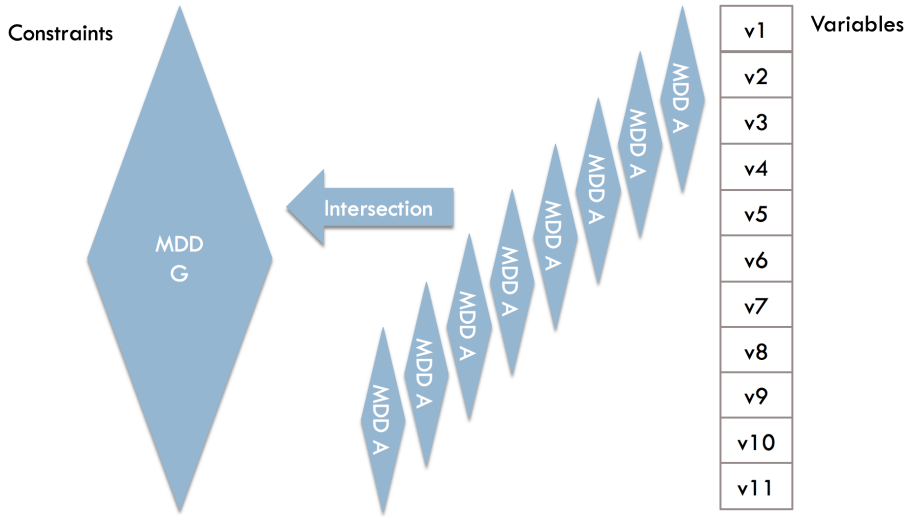


Figure 17.5: MaxOrder model 3, 11 variables, $p = 3$.

$(x_{i+p+1},x_{i+p+2},x_{i+p+3},...,x_{i+2p+1})$ is simple. It consists on the concatenation of the two MDDs. See ordering (c) in Figure 17.6.

Using this remark, we can intersect first all the contiguous MDD constraints. We obtain p MDD constraints and we can intersect them. Note that we perform $\max(p, r-p)$ intersections instead of $r-p$.

17.2.4 Experiments

The main instance of the problem deals with Markov Sequence Generation on corpus having more than 10,000 different words, thus the domain of the variables is greater than 10,000. The goal is to generate phrases having 20 words where all successions of 2 words come from the corpus and where there is no sequence of more than 4 words coming from the corpus.

The main issue with this approach is the size of the MDDs. The corpus has 10,785 words. mdd_T has 15,950 nodes and 129,465 arcs, mdd_P has 56,225 nodes and 127,786 arcs, mdd_G has 1,208,219 nodes and 188,035,203 arcs. Note that the reduction is efficient, for instance for mdd_P the number of nodes go from 123,025 to 56,225 and mdd_G has more than 2.2 times fewer nodes than the unfolded automaton.

Building times The next table gives the times for building the different MDDs from the different models.

MDD	mdd_T	mdd_P	mdd_A
Time (ms)	733	77	1248

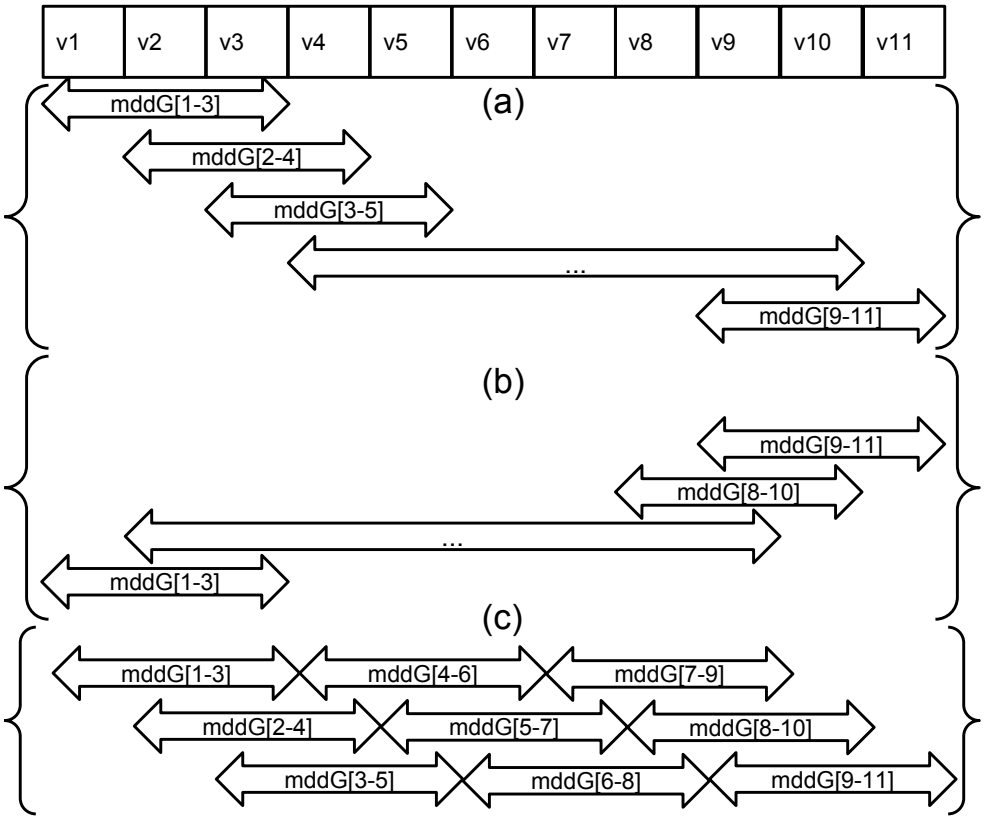


Figure 17.6: The three ordering for the intersection of the constraints.

Thus building either the models 1 or 2 is simple and fast. The next table gives the times for the intersections depending on the ordering for the model 3.

Method	(a)	(b)	(c)
Time (s)	390.0	271,9	143.5

The ordering (c) gives the best results for this benchmark. It is interesting to remark that the ordering (b) which consist on the reverse ordering of ordering (a) strongly improves the construction time. The original time published was 425 seconds [Perez 2015a] since it proposed only the ordering (a). Authors of the original problems indicate that their time was approximately the same. The time is now 3 times faster using the ordering (c).

Resolution Times In addition to the constraints for the maxorder, the model contains an alldifferent constraint. Propagators like `mddc` [Cheng 2010] cannot represent the MDD for 20 variables since it requires an array of size 10,000 by node. The memory consumption exceeded 64 GB quickly whereas it is kept below 10 GB with our algorithms. Restricted to a set of 9 variables

(instead of 20) and using the model 2, since the model 3 cannot be handled. It takes more than 50 min with `mddc` to find the first 50 solutions whereas it took 6 seconds for MDD4R with mdd_A. It is a gain factor of 500.

Using MDD4R, for the whole 20 variables, the model 3 takes 26.8 seconds for finding the first 50 solutions. The resolution time of the original model is the same.

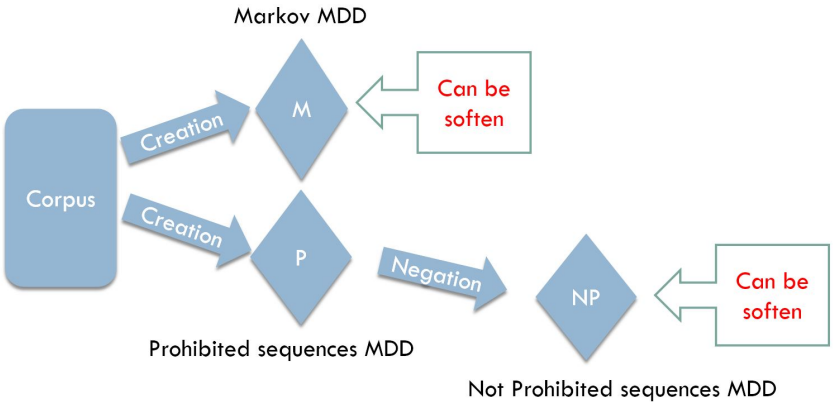


Figure 17.7: MaxOrder: soft version

17.3 Soft Version

17.3.1 Introduction and Model

Several corpus do not have any solution. For example, when the corpus is too small, or too complex (too many different words compared to the total number of words), there is no solution. In this case, we would like to provide solutions with respect to an amount of violation.

The problem is that in this problem, we would like to soften either the transition function, or the plagiarism. Using model 3 or the original model, it is not possible to soften either one of then. Moreover, the meaning of softening the mdd_G or the automaton of the original model is not trivial.

Consider model 1. If we use a soft MDD constraint for the MDD_T MDD, then it means that we allow the generation algorithm to create new transitions, transitions not belonging to the corpus. But if we use a soft MDD constraint for the $MDD_{\bar{P}}$, then this allows the generation algorithm to create sequences containing some plagiarisms. Finally, the goal of the solver is to limit this plagiarism or the number of invalid transitions. Figure 17.7 shows which MDDs are soften.

An important remark is that, if the corpus size grows, then the `maxOrder` constraint becomes satisfiable. If it grows very large, then it becomes useless to apply a `maxOrder` constraint because it becomes exponentially improbable to build a sequence containing plagiarism [1]. That's why we focus on corpus like fables and short texts.

[1] This remark came from discussions with Alexandre Papadopoulos. I found its remark very interesting.

Algo	Markov			Plagiarism		
size	18	20	22	18	20	22
inter	5,5	104,8	111,7	**4,7**	**8,1**	**9,3**
cost-MDD4R	**5,3**	**86,5**	**94,9**	23,7	44,6	67,9
ev-mdd	11,1	361,9	355,5	26,2	58,5	78,0

Table 17.1: Times needed to build the sequences with minimum of violations (Time out 1800s).

	Markov		Plagiarism	
	#nodes	#arcs	#nodes	#arcs
original	73	168	261	21.5k
arc *	73	380	261	22.1k
intersection	147	590	783	54.6k

Table 17.2: Size of the MDDs. 60 different words.

17.3.2 Experiments

The results are split depending on which MDD is softened since both gives pertinent results. For the experimentation, `"The fables of Jean de La Fontaine"` are used because they contain several sentences, not too many words and often produce funny results.

Table 17.1 gives the time results (in seconds) and Table 17.2 gives the size of the MDDs. Note that the model also contains an alldifferent constraint. `Markov` means that we apply the soft constraint on the Markovian transition, `Plagiarism` is for the plagiarism part. The creation time is similar for both MDDs, and insignificant compared to the search time.

The filtering algorithms used for the soft constraints are first the cost MDD propagators ev-mdd and cost-MDD4R up with the cost transformation from chapter 12, and then the MDD4R algorithm used with the MDD results from the intersection method from chapter 12, named inter in the tables. These tables show that both the cost and intersection methods for soft constraints are useful.

17.4 Conclusion

This chapter has presented several models for solving the MaxOrder problem, all with different complexity and efficiency. Then the problem of generating good non-solution has been considered and a model proposed. The results for

both of these problems seems to indicate that MDDs are well suited for these kinds of problems.

Chapter 18

Audio Multitrack Synchronization

Contents

18.1 Introduction

A lead sheet is a representation of a musical piece commonly used in popular music and consisting of a melody with chord labels on top, as shown on Figure 18.1 (extract of *A Day in the Life* by Lennon / McCartney).

An important aspect of this lead sheet is that melodic patterns are distributed according to a temporal structure. For example, the pattern of bars 1-2 is repeated at bars 5-6. This type of structure is commonplace in popular music. To generate lead sheets with a similar temporal structure similar, a standard CP approach is to define one variable per note. However, notes have a different duration: bar 1 contains eight short notes (including the rest) and bar 2 contains only one long note. Consequently, there is no direct correspondence between the index of a note and its temporal position. This makes it hard to post constraints stating that the first two bars should be repeated two bars later, regardless of their number of notes; or any other constraint of the same kind. In Section 18.3, we show that our approach yields a practical solution to this problem.

Automatic Accompaniment Generation The generation of musical accompaniment from audio multitrack recordings has immediate applications for computer generated musical improvisation or accompaniment generation.

The task consists in generating a new multi-track audio accompaniment by reusing an existing multitrack recording. The original tracks are segmented

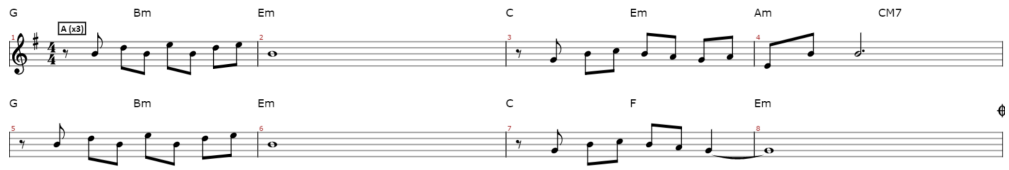


Figure 18.1: The first 8 bars of *A Day in the Life* by John Lennon and Paul McCartney

into chunks, using an onset detector [Dixon 2006]. Then new tracks are generated by recombining chunks, using concatenative synthesis [Maestre 2009]. Chunks in the generated tracks may appear in a different order than in the original track and may be used any number of times.

Such a scheme involves several types of constraints: On the one hand, we have to constrain each track to avoid awkward chunk transitions, by allowing transitions that are similar to the transitions in the training corpus. The similarity is measured using acoustic features, see Section 18.2. On the other hand, to prevent the tracks from "drifting" from one another, *e.g.,* one track becomes increasingly louder while another track fades out, we have to *synchronize* the tracks at regular points in time, for instance at the onset of every bar.

This problem raises the same issue as lead sheet generation. The chunks have different durations, therefore the index of a chunk and its temporal position are not directly depending on one another.

18.2 Description of the Benchmark

We use a three-track recording (guitar, bass, and drum) of the first 32 bars of song *Prayer in C* (Lilly Wood & The Prick). Each track is segmented using standard onset detection and quantized to $1/24^{th}$ of a beat (see Figure 18.2).

Chunks are categorized into *clusters* according to harmonic similarity (for pitched instruments) and timbre similarity for drums. The timbre is represented by Mel Frequency Cepstral Coefficients (MFCC) with 13 coefficients; the harmonic similarity is computed using the Harmonic Pitch-Class Profile (HPCP) with 36 divisions of the octave [Gómez 2006].

The guitar track contains 128 chunks with duration ranging from an eighth-note (half a beat) to a dotted quarter-note (1.5 beats), categorized into 13 harmonic clusters. The bass track contains 81 chunks (duration from half a beat to 1.5 beats) categorized into 9 bass clusters (harmonic similarity). The drum track contains 94 chunks with duration from half a beat to 20/3 beats, that is a full bar plus two thirds of a bar. There are 40 timbre-based drum

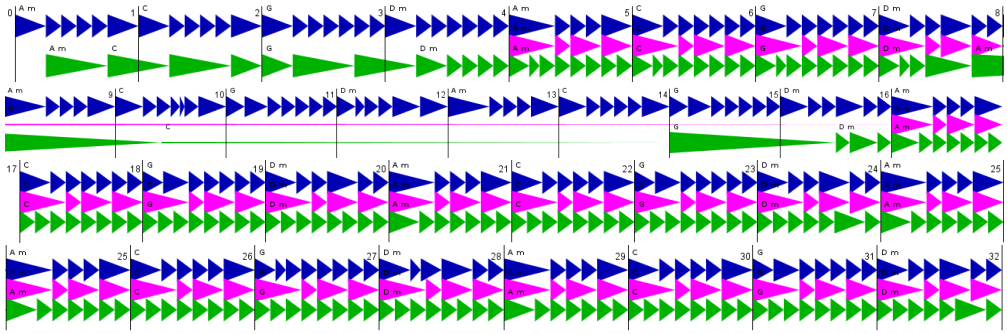


Figure 18.2: A graphical representation of the guitar (top), bass (center), and drum (bottom) tracks of *Prayer in C*. Each track contains 32 bars and each triangle represents a chunk. Vertical lines indicate bar separations.

clusters.

We state the problem of creating new multitracks as the generation of three sequences of chunks, each with an imposed total duration of n bars. Each bar has four beats, and the duration of the shortest chunks is $^1/_8$ of a bar, therefore each sequence contains at most $p = 32n$ chunks.

We define a sequence of p chunk variables: $G_1, \ldots, G_p$ (guitar), $B_1, \ldots, B_p$ (bass), and $D_1, \ldots, D_p$ (drums). The domain of each variable is the set of chunks in the corresponding recorded track, plus a dummy chunk with duration 0, called the *padding element*, which we explain below.

For each track, all chunk transitions, *e.g.*, $G_i \rightarrow G_{i+1}$, are such that the associated cluster transition exists in the original track. Additionally, we synchronize the tracks together at the beginning of every bar. More precisely, let G_i, B_j, D_k be the three chunks playing at the beginning of bar b, and $C(G_I), C(B_j), C(D_k)$ be the corresponding clusters. We enforce that the same cluster "signature" exists somewhere in the original multitrack, not necessarily at the beginning of a bar. The underlying idea is that the cluster signatures of the original track are musically acceptable. Intuitively, this constraint imposes that the generated multitrack uses acceptable chunk signatures at the beginning of every bar but can "invent" new cluster signatures (new sounds) between bar lines.

Total Duration The total duration can be handle in two different ways. For the first Allen model, it is sufficient to ensure that the last note has to terminate at $4n$. For the MDD-based Allen model, it is easy to impose the total duration of $4n$ to each track by simply removing all nodes of the graph (see Figure 18.3, left) whose label is greater than $4n$ and every node of the final layer whose label is different from $4n$. Every node with label $4n$ that is not in

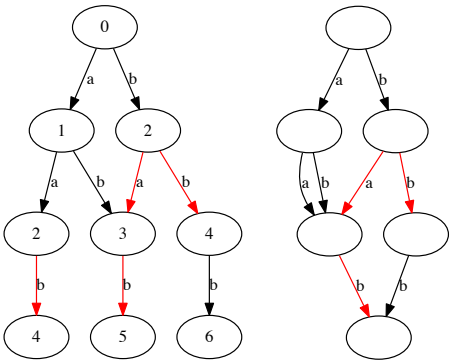


Figure 18.3: The graph (left) and MDD (right) representations of the constraint $\text{ALLEN}_{\mathbf{d}\vee\mathbf{s}\vee\mathbf{fi}\vee\mathbf{eq}[2,5]}$. Red labels correspond to values satisfying the constraint. Numbers in the graph on the left represent the temporal position.

the final layer receives a new arc, labeled with the padding element, going to the next layer to a new node with the same label $4n$, since the padding element has a 0 duration. We repeat this process to the final layer. This allows us to generate sequences with fewer than p "actual" variables, the padding value being assigned to the "extra" variables.

Markov transition The variables are subject to the binary constraints on chunk cluster transitions. These constraints are expressed as simple table constraints between consecutive chunk variables in each track. For example, $(C(G_i) \rightarrow C(G_{i+1})) \in \mathcal{C}_g$, where $\mathcal{C}_g$ is the set of all cluster transitions in the original guitar track. The same applies to the two other instruments.

Synchronization The vertical synchronization constraints are represented by an ALLEN constraint for each track and for each bar. To specify the events that are playing at the beginning of bar i, we use the Allen relation $\mathbf{o} \vee \mathbf{s}$ applied to the time interval $[4(i-1), +\infty)$. In our context, this relation, one of the 2^{13} combinations of Allen relations, specifies exactly the intervals which "contain" the temporal point $4(i-1)$, the onset of bar i.

The ALLEN constraints for the guitar track are

$$\text{ALLEN}_{\mathbf{o}\vee\mathbf{s}\,[4(i-1),\infty)}([C_1^g, \ldots, C_p^g], \mathcal{I}_i^g, \mathcal{E}_i^g)$$

where C_i^g is the variable cluster(G_i). The synchronization itself is enforced by an *ad hoc* table constraint between $\mathcal{E}_i^g$, $\mathcal{E}_i^b$, and $\mathcal{E}_i^d$, where accepted triplets are cluster signatures of the original multitrack.

This approach applies to the generation of automatic accompaniment of an imposed melody in a given style.

n	MDD size (#Vertices, #Edges)						Time (ms)
	Guitar		Bass		Drum		
6	2382	41k	848	13667	1864	73k	2301
8	4199	74k	1493	24k	3817	156k	7219
10	6530	117k	2388	39k	6513	275k	23k
12	9374	169k	3623	61k	9957	429k	57k
14	12k	231k	5085	87k	14k	617k	112k

Table 18.1: The size of the MDDs and the execution time to find 5 solutions for various multitrack lengths

.

18.3 Experiments

The experiments were run on a MacBook pro late 2013, having a I7 2.3Ghz and 8 Go of rams. The code is implemented using the OR Tools solver [Perron 2013].

18.3.1 First Allen Model

We evaluate two implementations of the scheduling model, depending on how we implement the constraint which links start times and duration (defined in Section 14.3.1). First, we enforce arc-consistency on this ternary sum constraint. The model solves the problem for two bars in 8.4 seconds. It does not solve the problem for more than two bars in less than 30 minutes, which we consider a timeout.

A lighter version has also been implemented where the ternary sums constraints only perform bound-consistency, based on the intuition that propagating information about the bounds of event duration offers a good trade off between simplicity and pruning. This model solves the problem for two bars in 5.4 seconds, but does not scale either to larger instances.

18.3.2 MDD-Based Allen Model

Note that, as said in Section 14.3.2, all the Allen constraints on a same track are represented by a single MDD.

The comparison with the performance of the simple model for Allen is clearly in favor of the MDD approach (see Table 18.1). The simple model does not solve problems longer than two bars in less than 30 minutes. In contrast, the MDD-based model solves the 14-bar problem in less than 2 minutes. The extra cost of performing the MDD construction and operations

is more than compensated for by the higher pruning offered by this model, especially regarding the treatment of the set variable $\mathcal{E}$.

Chapter 19

Geomodeling of a Petroleum reservoir

Contents

19.1 Introduction

The geomodeling of a petroleum reservoir [Pennington 2001] consists on generating images, respecting several constraint like the one from seismic images. These constraints are defined from pattern, for the geomodeling, and from seismic process. This chapter consider the latter one, the seismic constraint.

The generated images have to respect the geophysics constraint, called the convolution, consisting on applying a seismic process. An image can be seen as a vector of pixels, and constraints over images often contain other images and information. The geophysics constraint contains a siesmic image and a vector corresponding to a wavelet. This siesmic image has to correspond to the application of a process onto the generated image.

The applied processes are velocity distribution functions, that are given. Given a seismic image we want to find these velocities.

Suppose that α is a vector of 51 values corresponding to the "seismic wave", let V be the velocity image of the initial image, the image that we want to generate. We obtain this image by randomly selecting the values, considering a given distribution, under the following strong constraint.

For each pixel p_i in position i in its column, we define a pixel at the same

position in the seismic image with a value s_i:

$$s_i = \sum_{k=1}^{51} \alpha_k \log(V_{i-25+k-1}) \tag{19.1}$$

Finally we want to avoid statistical outliers.

19.2 Models

19.2.1 Problem

Velocities values are represented by a probability mass function (PMF, see chapter 9) on the model space. The PMF is given by physic laws. Velocities are discrete values of variables. For each cell c_{ij} of the reservoir, the seismic image gives a value s_{ij} and the given seismic wavelet (α_k). We define a sum constraint $\sum_{k=1}^{22} \alpha_k log(x_{i-11+k-1}j) = s_{ij}$. Locally, that is for each sum, we have to avoid outliers w.r.t. the PMF for the velocities.

We recall that the MDD of the constraint $\sum_{x_i \in X} f(x_i)$ is defined as follows. For the layer i, there are as many nodes as there are values of $\sum_{k=1}^{i} f(x_k)$. Each node is associated with such a value. A node n_p at layer i associated with value v_p is linked to a node n_q at layer $i+1$ associated with value v_q if and only if $v_q = v_p + f(a_i)$ with $a_i \in D(x_i)$. Then, only values v of the layer $|X|$ with $a \leq v \leq b$ are linked to tt. The reduction operation is applied after the definition and delete invalid nodes. The construction can be accelerated by removing states that are greater than b or that will not permit to reach a during the construction.

Outliers are avoided thanks to an `MDDProbability` constraint defined from the PMF for the velocities. P_{min} is defined by selecting only values having the 10% smaller probabilities, P_{max} is defined by selecting only values having the 10% greater probabilities.

The first experiment involves 22 variables and a constraint C_α: $\sum_{i=1}^{n} \alpha_i x_i = I$, where I is an tight interval (i.e. a value with an error variation). C_α is represented by $mdd_\alpha = \text{MDD}(\Sigma_{a_i,I}(X))$ where $a_i(x_i) = \alpha_i x_i$.

Dispersion First, we impose that the variables have to be distributed with respect to a normal distribution with μ, a fixed mean.

- $M_{\sigma<,\sigma>}$ One cost-MDD propagator on $mdd_\mu = \text{cost-MDD}(\Sigma_{n\mu}(X), \sigma)$ with σ and $\leq$ and one with σ and $\geq$. This model is similar to Pesant's model [Pesant 2015].

- M_{GCC} involves a GCC constraint [Régin 1996] where the cardinalities are extracted from the probability mass function.

- $M_{\mu\cap\sigma}$ represents the mean constraint by $mdd_\sigma = \text{MDD}(\Sigma_{n\mu}(X))$. It represents the sigma constraint by the $\text{MDD}(\Sigma_\sigma(X))$. Then the two MDDs are intersected. An MDD propagator is used on this MDD, named $mdd_{\mu\sigma}$.

- $M_{\mu\cap\sigma\cap\alpha}$ intersects mdd_α, the MDD of the constraint C_α, with $mdd_{\mu\sigma}$ the previous MDD to obtain mdd_{sol}. In this case, all constraints are combined.

PMF Then, we consider a `PMF` constraint and that μ is variable:

- M_{log}. We define a cost-MDD propagator on $mdd_{I_\mu} = \text{cost-MDD}(\Sigma_{id,[\underline{\mu},\overline{\mu}]}(X), logP)$ with $\log(P_{min})$ and $\geq$ and with $\log(P_{max})$ and $\leq$.

- $M_{log\cap\alpha}$. We define $mdd_{I_{log}} = \text{MDD}(\Sigma_{logP,I_{log}}(X))$ and we intersect it with mdd_{I_μ}. Then, we intersect it with mdd_α, the MDD of C_α, to obtain $mdd_{log\alpha}$.

19.2.2 Results

Table 19.1 shows the result of these experiments. As we can see when the problem involves many solutions, all the methods perform well (excepted M_{GCC}). We can see that an advantage of the intersection methods is that they contain all the solutions of problem. Table 19.2 shows the different sizes of the MDDs.

We consider 20 definitions of C_{ij}. We repeat the experiments 20 times and take the mean of the results.

For each constraint C_{ij}, the resulting MDD has in average 116,848 nodes and 1,239,220 edges. More than 320s are needed to compute it. Only 8 ms are required by computeMDDProbabilities algorithm in average. When a modification occurs the time to recompute the values are between a negligible value when the modifications are close to the root of the MDD and 8 ms when another part is modified.

For sampling 100,000 solutions we need 169 ms with the old C rand() function and 207 ms with the Mersenne-Twister random engine in conjunction with the uniform generator of the C++ standard library. Note that the time spent within the rand() function is 15 ms, whereas it is 82 ms with the second function. Therefore, the sampling procedures require less than 3 times the time spent in the random function.

		Fixed μ				Variable μ	
Sat?	#sol	$M_{\sigma<,\sigma>}$	M_{GCC}	$M_{\mu\cap\sigma}$	$M_{\mu\cap\sigma\cap\alpha}$	M_{log}	$M_{log\cap\alpha}$
Sat	Build	50	31	138	2,203	34	317,921
	10 sol	14	T-O	16	0	14	0
	All sol	T-O	T-O	T-O	0	T-O	0
UnSat	Build	55	28	121	151	37	133,752
	10 sol	T-O	T-O	T-O	0	T-O	0
	All sol	T-O	T-O	T-O	0	T-O	0

Figure 19.1: Comparison solving times (in ms) of models. 0 means that this is immediate. T-O indicates a time-out of 500s.

		Fixed μ					Variable μ		
Sat?	N/A	mdd_α	mdd_μ	mdd_σ	$mdd_{\mu\sigma}$	mdd_{sol}	mdd_{I_μ}	$mdd_{I_{log}}$	$mdd_{log\alpha}$
Sat	#nodes	3	3	5	67	521	2	18	24,062
Sat	#arcs	44	27	55	660	4,364	30	268	341,555
UnSat	#nodes	3	2	5	67	0	2	18	0
UnSat	#arcs	46	27	55	660	0	30	268	0

Figure 19.2: Comparison of MDD sizes (in thousands) of different models. 0 means that the MDD is empty.

19.3 Conclusion

This chapter proposes models for the geomodeling of a petroleum reservoir problem using MDDs. These different models have different efficiency, but also different costs. This chapter shows that combining MDD for designing more powerful model can be efficient but a trade-off between memory and processing has to be done.

Chapter 20

Conclusion

Contents

20.1 Conclusion

This thesis has focused on two main points. How to efficiently work with Decision Diagrams and how to efficiently integrate them in Constraints solvers.

For the first point, this thesis proposed several new methods for the most important operations which are the reduction, the combination and the creation of MDDs. Almost all these improvements come from the fact that MDDs have a fixed number of variables, and are directed acyclic graphs. Thanks to that, we have improved the existing operations by defining simple algorithm that avoid complex data structures and memory hacks. Moreover, using our new definition, we have designed an efficient parallel version of the reduction and combination algorithms. We have adapted the algorithms to deal with non deterministic and relaxed MDDs, allowing to deal with a broader range of real world problems. Finally, we have designed several algorithms allowing to sample solutions in MDDs with respect to probability distributions. All these improvements have allowed us to solve new problems of several order of magnitude in size.

For the second point, we have designed the current best three algorithms for MDDs in Constraint solvers. These algorithms allow to handle MDD constraints, cost-MDD constraint and soft MDD constraints. Furthermore, we have designed several mechanisms like the channeling constraint allowing to constrain sub-part of MDDs. Thank to these algorithms, we have shown how to reformulate several of the state of the art constraints like the dispersion constraint into MDDs and how to define new constraints like the Allen constraint. Using these new algorithms, we have solved many problems in text and music generation for entertainment, like the MaxOrder problem and the

multi-track audio synchronization problem. Furthermore, we have also solved a problem of geomodeling of a petroleum reservoir combining several knapsack problems.

20.2 Perspectives

This thesis mostly focuses on the algorithmic aspects of MDDs, but several characterization problems should be worked. First the relation between the size of an MDD and the global Hamming distance of the tuples contained in the MDD. Since a set of tuple with a global Hamming distance of 1 (Global Cut Seed) can be turned into an stick MDD, and ones with a bigger hamming distance cannot, one should describe the relation between this characteristics.

Moreover, the problem of finding the best variable ordering in an MDD is known to be NP-Complete. But some works should be done on local modifications of the variable ordering coupled with local search, for finding good local optimum size of MDDs.

In the context of constraint programming, we should try to convert several other constraints into MDDs, in the same way as done in this thesis. This allows us to reduce the number of specialized propagators in solvers. A good idea is often to extract the state machine of the constraints and to apply it to a fixed number of variables.

Part VI

Appendix

Appendix A

Implementation

In this section, I present different existing implementations for an MDD package. All these implementations are used in at least one algorithm presented in this thesis. The crucial difference between these implementations is how the arcs are represented. The first representation, using arrays, is used in most of the state of the art implementation [Cheng 2010, Cheng 2008], the second one, using lists, is used in several papers and in almost all the algorithms proposed in this thesis.

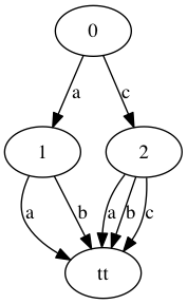


Figure A.1: An MDD representing the tuple set {(a,a),(a,b),(c,a),(c,b),(c,c)}

A.1 Array Implementation

The first MDD representation is simple and easy to implements. Each node of the MDD is represented by an array of size d (d is the domain size, see 2.1). The possible values of these arrays are the terminal node tt, the terminal node ff or another node. tt and ff are two special nodes, tt represents the true terminal node and ff the false terminal node. The index of an arc is its label, a mapping can be used if the values are not between 0 and d.

Example Consider the MDD from Figure A.1, the root node (node 0) has the following array $[1, ff, 2]$ of outgoing arcs. The mapping of the values is $[0 \to a, 1 \to b, 2 \to c]$. The outgoing arcs of all the nodes are:

Node	0/a	1/b	2/c
0	1	ff	2
1	tt	tt	ff
2	tt	tt	tt

Complexity The space complexity of such representation is $O(d)$ by node. This implies an overall space complexity of $O(nd)$. This complexity is strongly related to the domain size. The advantage of this representation is the random access for any outgoing arcs. The drawback is when the average outgoing degree is smaller than d.

Using this implementation, a random access to each outgoing edges is possible and testing the existence of a path can be implemented in $O(k)$ for an MDD with k variables. This kind of graph is often used for storing large set of words and then be used as a dictionary.

C++ possible implementation A possible implementation of the array version of MDDs is given here. The nodes ff and tt can be global (static in C++) and we will use the id 0 for the ff node and 1 for the tt node.

```
1  class NodeA{
2  private:
3      NodeA ** childs;      // Array of child
4      int size;             // size of the array childs
5      int id;               // unique id
6      int stamp;            // stamp used for the search
       algorithms
7  public:
8      NodeA(int size);      // Constructor
9      ~NodeA();             // destructor
10     int Id();             // if getter
11     int Size();           // size getter
12     int Stamp();          // stamp getter
13     void setStamp(int newStamp); // stamp setter
14     void set(int label,NodeA * end); // add or modify
       arcs
15     NodeA* get(int label);  // get the terminating
       extremities of the arc
16 };
17 class MDDA {
18 public:
```

```
19      NodeA * root;              // root of the MDD
20
21      static int opStamp;        // stamp used for the search
        algorithm
22      static int current_id;     // maximum already assigned
       id
23      static NodeA * ff;         // false terminal node
24      static NodeA * tt;         // true terminal node
25  };
```

Code A.1: Array MDDs

Construction and setter The construction of a node is easy, first we build the array of outgoing arcs and then we set all the arcs to the false terminal node ff. Adding or modifying an outgoing arc for a node is done by changing the value of the array corresponding to the label of the arc.

```
1   NodeA::NodeA(int size_):
2       childs(new NodeA*[size_]),
3       size(size_),
4       id(MDDA::current_id++),
5       stamp(-1)
6       {
7           for(int i=0; i < size; i++){
8               childs[i] = MDDA::ff;
9           }
10      }
11  NodeA::~NodeA(){
12      delete[] childs;
13  }
14  int NodeA::Id(){ return id; }
15  int NodeA::Size(){ return size; }
16  int NodeA::Stamp(){ return stamp; }
17  void NodeA::setStamp(int newStamp){ stamp = newStamp; }
18
19  void NodeA::set(int label, NodeA * end){
20      childs[label] = end;
21  }
22  NodeA* NodeA::get(int label){
23      return childs[label];
24  }
```

```
25
26 int MDDA::opStamp = 0;                    // no operation made
27 int MDDA::current_id = 0;                  // maximum already
      assigned id
28 NodeA * MDDA::ff = new NodeA(0);   // ID = 0
29 NodeA * MDDA::tt = new NodeA(0);   // ID = 1
```

Code A.2: Constructor Array MDDs

Example Building the MDD from Figure A.1 can be done using the following code:

```
1 MDDA mdda;
2 mdda.root = new NodeA(3);
3 NodeA * n1 = new NodeA(3);
4 NodeA * n2 = new NodeA(3);
5 mdda.root->set(0, n1);
6 mdda.root->set(2, n2);
7 n1->set(0, MDDA::tt);
8 n1->set(1, MDDA::tt);
9 n2->set(0, MDDA::tt);
10 n2->set(1, MDDA::tt);
11 n2->set(2, MDDA::tt);
```

Code A.3: Example of construction of an Array MDD

Depth First Search Most of the operation applied to MDDs using an array implementation use depth first search in a recursive way. The following code gives an example of how to search on an MDD using a DFS and print the *dot* version of the MDD.

```
1 void MDDA::printDot(std::ostream &ss){
2     ss << "digraph G{\n";
3     MDDA::opStamp ++;
4     printDotDFS(root,ss);
5     ss << "}\n";
6
7 }
8 void MDDA::printDotDFS(NodeA * n, std::ostream &ss){
9     if (n->Stamp() != MDDA::opStamp) {
```

```
10          n->setStamp(MDDA::opStamp);
11          for (int l = 0; l < n->Size(); l++) {
12              if (n->get(l) != MDDA::ff) {
13                  ss << n->Id() << " -> " << n->get(l)->Id
    () << " [label = " << l << " ]\n";
14                  printDotDFS(n->get(l),ss);
15              }
16          }
17      }
18  }
```

Code A.4: Example of construction of an Array MDD

Calling this method for the MDD example will give the following output:

```
1  digraph G{
2  2 -> 3 [label = 0 ]
3  3 -> 1 [label = 0 ]
4  3 -> 1 [label = 1 ]
5  2 -> 4 [label = 2 ]
6  4 -> 1 [label = 0 ]
7  4 -> 1 [label = 1 ]
8  4 -> 1 [label = 2 ]
9  }
```

Code A.5: Dot DFS output

Breath First Seach We can also apply Breath First Search (BFS) over MDDs. The following code print the *dot* version of the MDD using a BFS.

```
1  void MDDA::printDotBFS(std::ostream &ss){
2      MDDA::opStamp ++;
3      std::queue<NodeA*> q;
4      q.push(root);
5      ss << "digraph G{\n";
6      while (!q.empty()) {
7          NodeA * n = q.front();
8          q.pop();
9          for (int l = 0; l < n->Size(); l++) {
10             if (n->get(l) == MDDA::ff) {continue;}
11             ss << n->Id() << "->" << n->get(l)->Id() <<
    "[ label = " << l << "]\n";
```

```
12                if (n->get(l)->Stamp() !=MDDA::opStamp) {
13                    n->get(l)->setStamp(MDDA::opStamp);
14                    q.push(n->get(l));
15                }
16            }
17        }
18        ss << "}\n";
19    }
```

Code A.6: Example of construction of an Array MDD

Calling this method for the MDD example will give the following output:

```
1 digraph G{
2 2->3[ label = 0]
3 2->4[ label = 2]
4 3->1[ label = 0]
5 3->1[ label = 1]
6 4->1[ label = 0]
7 4->1[ label = 1]
8 4->1[ label = 2]
9 }
```

Code A.7: Dot BFS output

A.2 List Implementation

The list implementation is the one used in almost all the algorithms presented in this thesis. Using this representation, a node contains a list of outgoing arcs instead of an array and an arc has two information, its label and its terminateing extremity. While the label was implicittly given in the array implementation by the index, the list representation need to give it explicitly. We denote the list of outgoing arcs by ω^+.

Important The ω^+ list is an ordered list of arcs. The arcs are ordered by their label. All the creation, operation and reduction algorithms developed here consider as input MDDs whose node's ω^+ are sorted, and output MDDs with the same property.

Example Consider again the MDD from Figure A.1. The nodes have the following ω^+ list:

- $\omega^+(0) = ((a,1),(c,2))$
- $\omega^+(1) = ((a,tt),(b,tt))$
- $\omega^+(2) = ((a,tt),(b,tt),(c,tt))$

Complexity Using this representation, the overall space complexity is $O(n+m) = O(|N|+|E|)$.

C++ possible implementation A possible implementation of the ordered list version of MDDs is given here. Since we want to have ordered list of outgoing arcs, each node will maintains a pointer to its last arc in order to add an arc in the last position in $O(1)$.

```
1  class Node;
2  class Arc {
3      friend Node;
4      Arc * next;          // Next Edge in the list
5      Arc * prev;          // Previous Edge in the list
6      Node * start;        // starting node
7      Node * end;          // terminating node
8      int label;           // label of the arcs
9  };
10 class Node {
```

```
11      Arc * arcs;         // Double-linked List of outgoing
    edges
12      Arc * last_arc;     // pointer to the last edge
13      int id;             // unique id
14      int stamp;          // stamp used for the search
    algorithms
15  };
16  class MDD{
17      Node * root;          // root of the MDD
18      int node_id;          // id used to give a unique id
    to nodes.
19      int opStamp;          // stamp used for the search
    algorithm
20  };
```

Code A.8: List MDDs

Construction and setter The construction of a node without outgoing arc is easy. Set all the pointors to *NULL*. An Arc is define using the starting node, the terminating node and the label.

```
1   class Arc {
2   public:
3       Arc(Node * start_,int label_,Node * end_):
4       next(nullptr),
5       prev(nullptr),
6       start(start_),
7       end(end_),
8       label(label_)
9       {}
10      Node * getStart(){return start;}
11      Node * getEnd(){return end;}
12      int getLabel(){return label;}
13      Arc * Next(){return next;}
14  };
15
16  class Node {
17  public:
18      Node(int id_):
19          arcs(nullptr),
20          last_arc(nullptr),
```

```
21          id(id_),
22          stamp(-1)
23          {}
24      int Id(){return id;}
25      int Stamp(){return stamp;}
26      void setStamp(int newStamp){stamp = newStamp;}
27      Arc * getFirstArcs(){return arcs;}
28      void setLastArcs(int label, Node * dest){
29          if (arcs == nullptr) { // empty list
30              arcs = new Arc(this,label,dest);
31              last_arc = arcs;
32              return;
33          }
34          last_arc->next = new Arc(this,label,dest);
35          last_arc->next->prev = last_arc;
36          last_arc = last_arc->next;
37      }
38  };
39
40  class MDD{
41  public:
42      MDD():
43          root(nullptr),
44          node_id(0),
45          opStamp(-1)
46      {}
47      Node * Root(){return root;}
48      void setRoot(Node *n){root = n;}
49  };
```

Code A.9: List MDDs constructor

Example Using this implementation, building the MDD from Figure A.1 can be done using the following code:

```
1  MDD mdd;
2  mdd.setRoot(new Node(mdd.node_id++));
3  Node * n1 = new Node(mdd.node_id++);
4  Node * n2 = new Node(mdd.node_id++);
5  Node * tt = new Node(mdd.node_id++);
6  mdd.root->setLastArcs(0, n1);
```

```
7  mdd.root->setLastArcs(2, n2);
8  n1->setLastArcs(0, tt);
9  n1->setLastArcs(1, tt);
10 n2->setLastArcs(0, tt);
11 n2->setLastArcs(1, tt);
12 n2->setLastArcs(2, tt);
```

Code A.10: Example of construction of a List MDD

Breath First Search Most of the operations applied to MDDs using a List implementation use Breath First Search (BFS). The following code gives an example of how to search on an MDD using a BFS and print the *dot* version of the MDD.

```
1  void MDD::printDotBFS(std::ostream &ss){
2      opStamp++;
3      std::queue<Node*> q;
4      q.push(root);
5      ss << "digraph G{\n";
6      while (!q.empty()) {
7          Node * n = q.front();
8          q.pop();
9          Arc * a = n->getFirstArcs();
10         while (a != nullptr) {
11             ss << n->Id() << " -> " << a->getEnd()->Id()
       << " [label = " << a->getLabel() << " ]\n";
12             if (a->getEnd()->Stamp() != opStamp) {
13                 a->getEnd()->setStamp(opStamp);
14                 q.push(a->getEnd());
15             }
16             a = a->Next();
17         }
18     }
19     ss << "}\n";
20 }
```

Code A.11: List MDDs Breath First Search

The application of this method on the MDD from Code A.10 gives the following result:

```
digraph G{
0 -> 1 [label = 0 ]
0 -> 2 [label = 2 ]
1 -> 3 [label = 0 ]
1 -> 3 [label = 1 ]
2 -> 3 [label = 0 ]
2 -> 3 [label = 1 ]
2 -> 3 [label = 2 ]
}
```

Code A.12: Dot BFS output

Depth First Search While the BFS is usually applied when we use the List implementation, we can easily search over an MDD using Depth First Search (DFS). The following code gives an example of how to perform a DFS for printing the *dot* version of the MDD.

```
void MDD::printDot(std::ostream &ss){
    opStamp++;
    ss << "digraph G{\n";
    printDotDFS(root,ss);
    ss << "}\n";
}
void MDD::printDotDFS(Node * n, std::ostream &ss){
    if (n->Stamp() != opStamp) {
        n->setStamp(opStamp);
        Arc * a = n->getFirstArcs();
        while (a != nullptr) {
            ss << n->Id() << " -> " << a->getEnd()->Id()
    << " [label = " << a->getLabel() << " ]\n";
            printDotDFS(a->getEnd(),ss);
            a = a->Next();
        }
    }
}
```

Code A.13: List MDDs Depth First Search

The application of this method on the MDD from Code A.10 gives the following result:

```
digraph G{
0 -> 1 [label = 0 ]
1 -> 3 [label = 0 ]
1 -> 3 [label = 1 ]
0 -> 2 [label = 2 ]
2 -> 3 [label = 0 ]
2 -> 3 [label = 1 ]
2 -> 3 [label = 2 ]
}
```

Code A.14: Dot BFS output

A.2.1 Conclusion

In this appendix, I have presented the two most used MDD implementations. While both of these implementations have advantages and drawbacks, the choice depends on the problem.

Basically, for problems having a small number of different labels or who need a O(n) check for tuple validation, the the Array implementation should be chosen. For problems having a huge number of different labels or who need incremental operations and modifications, then the List implementation should be preferred.

Example Consider the problem of building a dictionary of word. In this problem, words have 26 different letters, which is not so big. And a fast look up for path existancy is required. The Array implementation should be prefered.

Consider the problem from Chapter 17. It involves more than 10 thousands of different labels, which is big. Using an Array implementation leads to memory out most of the time (see the experimental part). The List implementation scales in practice and should be preferred.

Finally, consider the problem from Chapter 18. This problem involves between 30 to 60 different labels, which start to be a lot. At first the average outgoing degree is close to 60, which implies that we can use the Array implementation, but during the search, this degree is decreasing and the whole MDD need to be modified and maintained. This is why we use the List implementation.

Appendix B

Algorithms and Data Structures

B.1 Sorting

This section presents several methods for sorting a set of values. The reading order of this section is important, since for example the radix sort use the count sort. Moreover, all the sorting method of this appendix are well know in computer science and so-called non comparison sort.

B.1.1 Indexing sort

Consider that you are correcting the exams of your students. But in order to increase the suspence (or whatever), you want to distribute the corrected exam from the lowest rank to the best one. Thus during your correction, you have to sort them.

Most of the sort, like the Fusion sort are not so natural for sorting exams. In general, we are going to make maybe 11 packs, one for each rank from 0 to 10. Then each time an exam is corrected, it is put in the good pack. Thus if you have 100 exam to correct, then you do 100 instruction *put the copy in the right pack*.

If we convert this idea into computer science programs, then we have an array of scores (integer), and we are going to put them into an array of list. Once all the exam are on the array of list, the size of this array depends on the maximal value, we just have to iterate over all the non-empty list from the lowest value to the biggest value.

Example Consider the following vector of values: $[3, 8, 2, 5, 3, 4, 8, 0, 9, 4, 7, 8, 5]$. We need to define an array of list that will be able to store all these values. Thus we define the following list array :

0	1	2	3	4	5	6	7	8	9
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

When we read the first value 3, we put it in the list of cell labeled by 3:

0	1	2	3	4	5	6	7	8	9
$\emptyset$	$\emptyset$	$\emptyset$	3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

We continue, we put the values 8,2,5 and 3 in their respective lists:

0	1	2	3	4	5	6	7	8	9
$\emptyset$	$\emptyset$	2	3, 3	$\emptyset$	5	$\emptyset$	$\emptyset$	8	$\emptyset$

Finally, by putting the remaining values we obtain:

0	1	2	3	4	5	6	7	8	9
0	$\emptyset$	2	3, 3	4, 4	5, 5	$\emptyset$	7	8, 8, 8	9

We can iterate over the list, from cell 0 to 9 in order to get back the sorted values: $[0, 2, 3, 3, 4, 4, 5, 5, 7, 8, 8, 8, 9]$.

Sometimes, we are interested in sorting element according to their values, so not necesseraly directly the values. Using this sort, you can store the element or even their indexes in order to obtain a sorted permutation.

B.1.2 Counting sort

As shown in the previous section, we can put values in an array of list to sort them. But an interesting remark is that all the values of a list are the same, moreover, the values of the list are given by the cell index.

The counting sort [Cormen 2001] use this remark, instead of placing them in an array of list, it counts the values. Then we can easily reconstruct a vector of sorted values by creating as many values as counted.

Example Consider again the following vector of values: $[3, 8, 2, 5, 3, 4, 8, 0, 9, 4, 7, 8, 5]$. We need to define an array # of integer set to 0 for counting the values :

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

We first read 3, and increments the value of cell 3:

0	1	2	3	4	5	6	7	8	9
0	0	0	1	0	0	0	0	0	0

Then we read the values 8,2,5 and 3 and increments their respective cells:

0	1	2	3	4	5	6	7	8	9
0	0	1	2	0	1	0	0	1	0

Finally when we have processed all the values, the counting array is:

0	1	2	3	4	5	6	7	8	9
1	0	1	2	2	2	0	1	3	1

From the cell 0 to cell 9 of #, we build the sorted vector of values by adding as many times the value as the value of its associated cell. We finally obtain $[0, 2, 3, 3, 4, 4, 5, 5, 7, 8, 8, 8, 9]$.

As previously said, we are often interested in sorting element according to their values, so not necesseraly directly the values. But using this method does not allow to obtain the sorted permutation, since we recreate the value according to their cell. Thus the counting sort is most of the time used as described in the next paragraph.

Sorted indexes In order to obtain the sorted elements, which is often what we want, we need to avoid creating the values based on their cell value. Instead, we process the position of the values (and so index) in the resulting vector.

First, we process the P vector of position. This vector contains for each cell, the position in the vector of the next value belonging to the cell. This position is given by $P_0 = 0$ for the first value and by $P_i = P_{i-1} + \#_{i-1}$ for all the other values.

Thus, using the previous example, the arrays C and P are:

0	1	2	3	4	5	6	7	8	9	Cells
1	0	1	2	2	2	0	1	3	1	#
0	1	1	2	4	6	8	8	9	12	P

Then, using the P vector, we first build the resulting vector R of size n (number of values to sort, 13 in our example), then we iterate over the original vector, and for each value i, we put i in the cell P_i of the result vector and then increment P_i.

Following our example, the first value of the original vector is 3, thus we put 3 in $P_3 = 2$ and obtain $R = [?, ?, 3, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]$. Then we increment P_3 which is now equal to 3.

We read the values 8,2,5 and 3 and put them in the vector $R = [?, 2, 3, 3, ?, ?, 5, ?, ?, 8, ?, ?, ?]$. The current state of the P vector is:

0	1	2	3	4	5	6	7	8	9	Cells
1	0	1	2	2	2	0	1	3	1	#
0	1	2	4	4	7	8	8	10	12	P

Note that even if $P_2 = 2$, which is an already set cell, this is not an issue since no more value 2 are in the original vector.

B.1.3 Radix sort

The counting sort [Cormen 2001] is very efficient for sorting values in a small range, but numbers in a computer are most of the time encode using 32 bits, thus an array of 2^{32} is required, which is intractable in practice.

The radix sort proposes to see a number as a list of digits, for example 28 in base 10 can be decomposed into $2*10^1+8*10^0$ and thus can be represented by $(2, 8)$. Then to apply efficient linear sorting algorithm on the digits of this decomposition, digit by digit.

Intuitively, if we have many scores between 0 and 99 to sort, we may want to first apply a counting sort using the digit of the 10^1, and then for each cell, re-aplying a sort. But this lead to applying several times (in the worst case of this example 11 times) the algorithm. This schema can be seen as sorting using first the most significant digit.

This main idea of the radix sort is to start with the least significant digit. Thus with our scores, we first sort them using the digit associating to 10^0, and then we sort all the value using the 10^1 digit, by keeping the ordering given by the first sort. The linear sort used is often the counting sort.

Example Consider that you want to sort the following array of 2 Boolean digits values:

0	1	2	3	4	5	6	7	8	9	ID
10	0**0**	**11**	**11**	0**0**	**10**	**10**	0**1**	**11**	0**1**	Value

Thus using a counting sort on the right most digit, we obtain the two counters $\#_0 = 5$ and $\#_1 = 5$. Using these counter we can permute the values :

0	1	4	5	6	2	3	7	8	9
10	**0**0	**0**0	**10**	**10**	**11**	**11**	**0**1	**11**	**0**1

Now using the second digit, we recount, we obtain the counters $\#_0 = 4$ and $\#_1 = 6$. Using these counter we can permute the values and obtain the sorted vector:

1	4	7	9	0	5	6	2	3	8
00	00	01	01	10	10	10	11	11	11

Complexity Given n values, a base b and if d digits are required for representing the biggest element, the complexity of the radix sort is given by $O(d(n+b)$. In today's computer, the radix sort is often encoded using a base $2^8 = 256$, thus since the numbers are encode into 32 or 64 bits, we obtain $d = log_{256}(2^{32}) = 4$ or$d = log_{256}(2^{64}) = 8$. Thus for sorting n number, it usualy cost $4(n + 256) = 4n + 1024$. This complexity implies that the choice of the base depends strongly on the maximal value and on the number of values. Sorting 4 integers encoded in 32 bits is not efficient using a radix sort, but for sorting hundreds or millions of them, the radix sort is recommended.

Parallel version The parallel version of this algorithm is recalled in chapter 6.

Exercises Just kidding, but the book [Cormen 2001] contains many information, exercises about these sorts and certainly more insights.

Bibliography

[Aho 1974] Alfred V Aho and John E Hopcroft. The design and analysis of computer algorithms. Pearson Education India, 1974. (Cited on page 25.)

[Akers 1978] Sheldon B. Akers. *Binary decision diagrams.* IEEE Trans. Computers, vol. 27, no. 6, pages 509–516, 1978. (Cited on pages 14 and 17.)

[Allen 1983] James F. Allen. *Maintaining Knowledge about Temporal Intervals.* Commun. ACM, vol. 26, no. 11, pages 832–843, 1983. (Cited on page 224.)

[Amilhastre 2014] Jérôme Amilhastre, Hélene Fargier, Alexandre Niveau and Cédric Pralet. *Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams.* International Journal on Artificial Intelligence Tools, vol. 23, no. 04, page 1460015, 2014. (Cited on page 108.)

[Andersen 1997] Henrik Reif Andersen. *An introduction to binary decision diagrams.* Lecture notes, available online, IT University of Copenhagen, 1997. (Cited on pages 5, 6, 18, 27, 42 and 62.)

[Andersen 1999] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams.* 1999. (Cited on page 59.)

[Andersen 2007] Henrik Reif Andersen, Tarik Hadzic, John N. Hooker and Peter Tiedemann. *A Constraint Store Based on Multivalued Decision Diagrams.* In CP, pages 118–132, 2007. (Cited on pages 6, 18, 19, 42, 113, 154, 184, 186, 209, 212 and 248.)

[bac] *Problem 30 of CSPLIB.* (`www.csplib.org`). (Cited on page 235.)

[Barbieri 2012] Gabriele Barbieri, François Pachet, Pierre Roy and Mirko Degli Esposti. *Markov Constraints for Generating Lyrics with Style.* In ECAI, volume 242, pages 115–120, 2012. (Cited on pages 14 and 128.)

[Beldiceanu 2004a] N. Beldiceanu, M. Carlsson and T. Petit. *Deriving Filtering Algorithms from Constraint Checkers.* In CP'04, pages 107–122, 2004. (Cited on pages 2, 52, 149, 156, 157 and 178.)

[Beldiceanu 2004b] Nicolas Beldiceanu and Thierry Petit. *Cost evaluation of soft global constraints.* In International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, pages 80–95. Springer, 2004. (Cited on page 200.)

[Beldiceanu 2007] N. Beldiceanu, M. Carlsson, S. Demassey and T. Petit. *Global Constraint Catalog: Past, Present and Future.* Constraints, vol. 12, no. 1, pages 21–62, 2007. (Cited on page 235.)

[Beldiceanu 2012] Nicolas Beldiceanu, Mats Carlsson and Jean-Xavier Rampon. *Global constraint catalog, (revision a)*, 2012. (Cited on page 8.)

[Bent 2004] Russell Bent and Pascal Van Hentenryck. *A two-stage hybrid local search for the vehicle routing problem with time windows.* Transportation Science, vol. 38, no. 4, pages 515–530, 2004. (Cited on page 14.)

[Bergman 2011] David Bergman, Willem Jan van Hoeve and John N. Hooker. *Manipulating MDD Relaxations for Combinatorial Optimization.* In CPAIOR, pages 20–35, 2011. (Cited on pages 3, 6, 114, 115, 184 and 205.)

[Bergman 2014a] David Bergman, Andre A Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat and Willem-Jan van Hoeve. *Parallel combinatorial optimization with decision diagrams.* In International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pages 351–367. Springer, 2014. (Cited on pages 6, 59 and 68.)

[Bergman 2014b] David Bergman, Andre A Cire and W van Hoeve. *MDD propagation for sequence constraints.* Journal of Artificial Intelligence Research, pages 697–722, 2014. (Cited on pages 6, 14, 19, 59 and 209.)

[Bergman 2016a] David Bergman and Andre A. Cire. Multiobjective optimization by decision diagrams, pages 86–95. Springer International Publishing, Cham, 2016. (Cited on pages 184 and 205.)

[Bergman 2016b] David Bergman, Andre A Cire, Willem-Jan van Hoeve and JN Hooker. *Decision Diagrams for Optimization*, 2016. (Cited on pages 3, 6 and 18.)

[Berrani 2013] Sid-Ahmed Berrani, Mohammed Haykel Boukadida and Patrick Gros. *Constraint satisfaction programming for video summarization.* In IEEE International Symposium on Multimedia, Anaheim, California, United States, December 2013. IEEE. (Cited on page 224.)

[Bessiere 1997] Christian Bessiere and Jean-Charles Régin. *Arc consistency for general constraint networks: preliminary results*. 1997. (Cited on pages 43, 148, 151 and 161.)

[Bessière 2001] C. Bessière and J-C. Régin. *Refining the Basic Constraint Propagation Algorithm*. In Proceedings of IJCAI'01, pages 309–315, Seattle, WA, USA, 2001. (Cited on page 168.)

[Bessiere 2004] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich and Toby Walsh. *Disjoint, partition and intersection constraints for set and multiset variables*. In International Conference on Principles and Practice of Constraint Programming, pages 138–152. Springer, 2004. (Cited on page 211.)

[Bessière 2005] Christian Bessière, Jean-Charles Régin, Roland HC Yap and Yuanlin Zhang. *An optimal coarse-grained arc consistency algorithm*. Artificial Intelligence, vol. 165, no. 2, pages 165–185, 2005. (Cited on page 148.)

[Bessiere 2014] Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, Émilie Picard-Cantin, Claude-Guy Quimper and Toby Walsh. *The Balance Constraint Family*. In Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, pages 174–189, 2014. (Cited on page 235.)

[Bollig 1999] Beate Bollig and Ingo Wegener. *Complexity theoretical results on partitioned (nondeterministic) binary decision diagrams*. Theory of Computing Systems, vol. 32, no. 4, pages 487–503, 1999. (Cited on pages 6 and 105.)

[Boussemart 2016] Frédéric Boussemart, Christophe Lecoutre and Cédric Piette. *XCSP3: An integrated format for benchmarking combinatorial constrained problems*. arXiv preprint arXiv:1611.03398, 2016. (Cited on pages 7 and 19.)

[Brace 1991] Karl S Brace, Richard L Rudell and Randal E Bryant. *Efficient implementation of a BDD package*. In Proceedings of the 27th ACM/IEEE design automation conference, pages 40–45. ACM, 1991. (Cited on pages 5, 6, 27, 59, 65 and 85.)

[Briggs 1993] Preston Briggs and Linda Torczon. *An Efficient Representation for Sparse Sets*. ACM Letters on Programming Languages and Systems, vol. 2, pages 59–69, 1993. (Cited on pages 165 and 166.)

[Brooks 1957] Frederick P Brooks, AL Hopkins, Peter G Neumann and WV Wright. *An Experiment in Musical Composition*. Electronic Computers, IRE Transactions on, no. 3, pages 175–182, 1957. (Cited on page 128.)

[Bryant 1986] Randal E Bryant. *Graph-based algorithms for boolean function manipulation*. Computers, IEEE Transactions on, vol. 100, no. 8, pages 677–691, 1986. (Cited on pages 3, 5, 6, 14, 18, 25, 26, 27, 29, 30, 42, 59 and 60.)

[Bryant 1992] Randal E Bryant. *Symbolic Boolean manipulation with ordered binary-decision diagrams*. ACM Computing Surveys (CSUR), vol. 24, no. 3, pages 293–318, 1992. (Cited on pages 3, 6, 41, 42, 59 and 60.)

[Carlsson] Mats Carlsson. Sicstus prolog user's manual, volume 3. (Cited on page 19.)

[Cheng 1999] BMW Cheng, Kenneth MF Choi, Jimmy Ho-Man Lee and JCK Wu. *Increasing constraint propagation by redundant modeling: an experience report*. Constraints, vol. 4, no. 2, pages 167–192, 1999. (Cited on page 210.)

[Cheng 2005] Kenil CK Cheng and Roland HC Yap. *Constrained decision diagrams*. In Proceedings of the National Conference on Artificial Intelligence, volume 20, page 366. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005. (Cited on pages 18, 42 and 209.)

[Cheng 2008] Kenil C. K. Cheng and Roland H. C. Yap. *Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints*. In CP, pages 509–523, 2008. (Cited on pages 7, 18, 19, 41, 42, 44, 149, 154, 165 and 283.)

[Cheng 2010] K. Cheng and R. Yap. *An MDD-based Generalized Arc Consistency Algorithm for Positive and Negative Table Constraints and Some Global Constraints*. Constraints, vol. 15, 2010. (Cited on pages 7, 18, 19, 27, 43, 44, 54, 149, 154, 165, 179, 263 and 283.)

[Cheng 2012] Kenil CK Cheng, Wei Xia and Roland HC Yap. *Space-Time Tradeoffs for the Regular Constraint.* In CP, pages 223–237. Springer, 2012. (Cited on pages 42 and 107.)

[Christofides 1981] Nicos Christofides, Aristide Mingozzi and Paolo Toth. *State-space relaxation procedures for the computation of bounds to routing problems.* Networks, vol. 11, no. 2, pages 145–164, 1981. (Cited on page 114.)

[Cire 2013] Andre A Cire and Willem-Jan van Hoeve. *Multivalued decision diagrams for sequencing problems.* Operations Research, vol. 61, no. 6, pages 1411–1428, 2013. (Cited on pages 6, 42, 53 and 248.)

[Ciré 2014a] André A Ciré and John N Hooker. *The Separation Problem for Binary Decision Diagrams.* In ISAIM, 2014. (Cited on pages 19 and 78.)

[Cire 2014b] Andre Augusto Cire. *Decision diagrams for optimization.* 2014. (Cited on pages 6 and 114.)

[Cook 2009] Roy T Cook. "Intensional Definition" in a dictionary of philosophical logic. Edinburgh University Press, 2009. (Cited on page 149.)

[Cormen 2001] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest and Clifford Stein. Introduction to algorithms, volume 6. MIT press Cambridge, 2001. (Cited on pages 45, 70, 298, 300 and 301.)

[Coulouris 2005] George F Coulouris, Jean Dollimore and Tim Kindberg. Distributed systems: concepts and design. pearson education, 2005. (Cited on page 90.)

[Dean 2008] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters.* Communications of the ACM, vol. 51, no. 1, pages 107–113, 2008. (Cited on page 90.)

[Dechter 1991] Rina Dechter, Itay Meiri and Judea Pearl. *Temporal Constraint Networks.* Artif. Intell., vol. 49, no. 1-3, pages 61–95, 1991. (Cited on page 224.)

[Dejemeppe 2015] Cyrille Dejemeppe, Sascha Van Cauwelaert and Pierre Schaus. *The unary resource with transition times.* In International Conference on Principles and Practice of Constraint Programming, pages 89–104. Springer International Publishing, 2015. (Cited on page 184.)

[Demassey 2006] Sophie Demassey, Gilles Pesant and Louis-Martin Rousseau. *A cost-regular based hybrid column generation approach.* Constraints, vol. 11, no. 4, pages 315–333, 2006. (Cited on pages 7, 14, 184, 186 and 194.)

[Demeulenaere 2016] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin and Pierre Schaus. *Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets.* In International Conference on Principles and Practice of Constraint Programming, pages 207–223. Springer International Publishing, 2016. (Cited on pages 11, 44, 148, 149, 153 and 180.)

[Derrien 2015] Alban Derrien, Jean-Guillaume Fages, Thierry Petit and Charles Prud'homme. *A Global Constraint for a Tractable Class of Temporal Optimization Problems.* In Principles and Practice of Constraint Programming – CP 2015, pages 105–120. Springer, 2015. (Cited on page 224.)

[Dixon 2006] Simon Dixon. *Onset detection revisited.* In Proceedings of the 9th International Conference on Digital Audio Effects, volume 120, pages 133–137. Citeseer, 2006. (Cited on page 270.)

[Finkbeiner 2001] Bernd Finkbeiner. *Language containment checking with nondeterministic BDDs.* In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 24–38. Springer, 2001. (Cited on pages 6 and 105.)

[Finkel 1974] Raphael A. Finkel and Jon Louis Bentley. *Quad trees a data structure for retrieval on composite keys.* Acta informatica, vol. 4, no. 1, pages 1–9, 1974. (Cited on page 249.)

[Flener 2001] Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel and Toby Walsh. *Matrix modelling.* In Proc. of the CP-01 Workshop on Modelling and Problem Formulation, 2001. (Cited on page 223.)

[Focacci 2001] Filippo Focacci and Michela Milano. *Global Cut Framework for Removing Symmetries.* In Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings, pages 77–92, 2001. (Cited on pages 2, 5, 42 and 47.)

[Galvane 2015a] Quentin Galvane, Marc Christie, Christophe Lino and Rémi Ronfard. *Camera-on-rails: Automated Computation of Constrained Camera Paths.* In ACM SIGGRAPH Conference on Motion in Games, Paris, France, November 2015. (Cited on page 224.)

[Galvane 2015b] Quentin Galvane, Rémi Ronfard, Christophe Lino and Marc Christie. *Continuity Editing for 3D Animation*. In AAAI Conference on Artificial Intelligence, AAAI Press, Austin, Texas, United States, January 2015. (Cited on page 224.)

[Gange 2011] G. Gange, P. Stuckey and Radoslaw Szymanek. *MDD propagators with explanation*. Constraints, vol. 16, pages 407–429, 2011. (Cited on pages 7, 19, 44, 149 and 161.)

[Gange 2013] Graeme Gange, Peter J Stuckey and Pascal Van Hentenryck. *Explaining propagators for edge-valued decision diagrams*. In Principles and Practice of Constraint Programming, pages 340–355. Springer, 2013. (Cited on pages 7, 184, 186 and 206.)

[Gavanelli 2002] Marco Gavanelli. *An algorithm for multi-criteria optimization in CSPs*. In ECAI, volume 2, pages 136–140, 2002. (Cited on page 249.)

[Gent 2007] Ian P. Gent, Christopher Jefferson, Ian Miguel and Peter Nightingale. *Data Structures for Generalised Arc Consistency for Extensional Constraints*. In Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada, pages 191–197, 2007. (Cited on pages 43, 149 and 151.)

[Gervet 1993] Carmen Gervet. *Sets and Binary Relation Variables Viewed as Constrained Objects.* In ICLP Workshop on Logic Programming with Sets, 1993. (Cited on page 211.)

[Gómez 2006] Emilia Gómez. *Tonal Description of Music Audio Signals*. PhD thesis, Universitat Pompeu Fabra, 2006. (Cited on page 270.)

[Hadzic 2004] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M Jensen, Henrik R Andersen, Jesper Møller and Henrik Hulgaard. *Fast backtrack-free product configuration using a precompiled solution space representation*. small, vol. 10, no. 1, page 3, 2004. (Cited on pages 14 and 59.)

[Hadzic 2008] Tarik Hadzic, John N. Hooker, Barry O'Sullivan and Peter Tiedemann. *Approximate Compilation of Constraints into Multivalued Decision Diagrams*. In CP, pages 448–462, 2008. (Cited on pages 6, 18, 19, 42, 113, 114, 115 and 209.)

[Hamadi 2002] Youssef Hamadi. *Optimal distributed arc-consistency*. Constraints, vol. 7, no. 3-4, pages 367–385, 2002. (Cited on page 89.)

[Hartert 2014] Renaud Hartert and Pierre Schaus. *A Support-Based Algorithm for the Bi-Objective Pareto Constraint.* In AAAI, pages 2674–2679. Citeseer, 2014. (Cited on page 249.)

[Harvey] Warwick Harvey. *CSPLib Problem 010: Social Golfers Problem.* http://www.csplib.org/Problems/prob010. (Cited on page 211.)

[Hedayati Somarin 2016] Iraj Hedayati Somarin. *DFA Minimization Algorithms in Map-Reduce.* PhD thesis, Concordia University, 2016. (Cited on page 90.)

[Hoda 2010] Samid Hoda, Willem Jan van Hoeve and John N. Hooker. *A Systematic Approach to MDD-Based Constraint Programming.* In CP, pages 266–280, 2010. (Cited on pages 6, 8, 19, 154, 184, 186, 187, 197, 209, 212 and 240.)

[Hooker 2007] John N Hooker. Integrated methods for optimization, volume 100. Springer Science & Business Media, 2007. (Cited on pages 3 and 6.)

[Hooker 2013] John N Hooker. *Decision diagrams and dynamic programming.* In International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pages 94–110. Springer, 2013. (Cited on pages 5, 18, 41, 42, 53, 184 and 247.)

[Hopcroft 2006] John E Hopcroft, Rajeev Motwani and Jeffrey D Ullman. *Automata theory, languages, and computation.* International Edition, vol. 24, 2006. (Cited on pages 19 and 51.)

[Houndji 2014] Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey and Yves Deville. *The stockingcost constraint.* In International Conference on Principles and Practice of Constraint Programming, pages 382–397. Springer International Publishing, 2014. (Cited on page 184.)

[Jiang 1993] Tao Jiang and Bala Ravikumar. *Minimal NFA problems are hard.* SIAM Journal on Computing, vol. 22, no. 6, pages 1117–1141, 1993. (Cited on pages 105 and 106.)

[Jurafsky 2014] Dan Jurafsky and James H Martin. Speech and language processing. Pearson, 2014. (Cited on pages 7, 127, 130 and 238.)

[Katsirelos 2007] G. Katsirelos and T. Walsh. *A Compression Algorithm for Large Arity Extensional Constraints.* In Proc. CP'07, pages 379–393, Providence, USA, 2007. (Cited on page 149.)

[Kimura 1990] Shinji Kimura and Edmund M Clarke. *A parallel algorithm for constructing binary decision diagrams.* In Computer Design: VLSI in Computers and Processors, 1990. ICCD'90. Proceedings, 1990 IEEE International Conference on, pages 220–223. IEEE, 1990. (Cited on pages 6 and 90.)

[Knuth 2011] Donald E Knuth. The art of computer programming, volume 4a: Combinatorial algorithms, part 1. Pearson Education India, 2011. (Cited on page 66.)

[Koriche 2015] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis and Samuel Thomas. *Compiling Constraint Networks into Multivalued Decomposable Decision Graphs.* In IJCAI, pages 332–338, 2015. (Cited on page 42.)

[Lai 1992] Y-T Lai and Sarma Sastry. *Edge-valued binary decision diagrams for multi-level hierarchical verification.* In Proceedings of the 29th ACM/IEEE Design Automation Conference, pages 608–613. IEEE Computer Society Press, 1992. (Cited on page 184.)

[Lecoutre 2009] Christophe Lecoutre. *CSP/MaxCSP/WCSP solver competitions.* In http://www.cril.univ-artois.fr/ lecoutre/benchmarks.html, 2009. (Cited on pages 37 and 179.)

[Lecoutre 2011] Christophe Lecoutre. *STR2: optimized simple tabular reduction for table constraints.* Constraints, vol. 16, no. 4, pages 341–371, 2011. (Cited on pages 43, 148, 152 and 165.)

[Lecoutre 2012a] Christophe Lecoutre, Chavalit Likitvivatanavong and Roland H. C. Yap. *A Path-Optimal GAC Algorithm for Table Constraints.* In ECAI, pages 510–515, 2012. (Cited on pages 43, 148, 152 and 165.)

[Lecoutre 2012b] Christophe Lecoutre, Nicolas Paris, Olivier Roussel and Sébastien Tabary. *Propagating soft table constraints.* In Principles and Practice of Constraint Programming, pages 390–405. Springer, 2012. (Cited on page 200.)

[Lecoutre 2015] Christophe Lecoutre, Chavalit Likitvivatanavong and Roland HC Yap. *STR3: A path-optimal filtering algorithm for table constraints.* Artificial Intelligence, vol. 220, pages 1–27, 2015. (Cited on page 148.)

[Lee 1959] Chang-Yeong Lee. *Representation of Switching Circuits by Binary-Decision Programs.* Bell Labs Technical Journal, vol. 38, no. 4, pages 985–999, 1959. (Cited on page 16.)

[Lhomme 2005] Olivier Lhomme and Jean-Charles Régin. *A Fast Arc Consistency Algorithm for n-ary Constraints.* In Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA, pages 405–410, 2005. (Cited on pages 43 and 148.)

[Lhomme 2012] Olivier Lhomme. *Practical reformulations with table constraints.* In Proceedings of the 20th European Conference on Artificial Intelligence, pages 911–912. IOS Press, 2012. (Cited on pages 9 and 147.)

[Mackworth 1977] Alan K Mackworth. *Consistency in networks of relations.* Artificial intelligence, vol. 8, no. 1, pages 99–118, 1977. (Cited on page 151.)

[Maestre 2009] Esteban Maestre, Rafael Ramírez, Stefan Kersten and Xavier Serra. *Expressive Concatenative Synthesis by Reusing Samples from Real Performance Recordings.* Comput. Music J., vol. 33, no. 4, pages 23–42, December 2009. (Cited on page 270.)

[Mairy 2012] Jean-Baptiste Mairy, Pascal Van Hentenryck and Yves Deville. *An optimal filtering algorithm for table constraints.* In Principles and Practice of Constraint Programming, pages 496–511. Springer, 2012. (Cited on page 148.)

[Mairy 2015] Jean-Baptiste Mairy, Yves Deville and Christophe Lecoutre. *The smart table constraint.* In International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pages 271–287. Springer International Publishing, 2015. (Cited on pages 2 and 149.)

[Malapert 2008] Arnaud Malapert, Christelle Guéret, Narendra Jussien, André Langevin and Louis-Martin Rousseau. *Two-dimensional pickup and delivery routing problem with loading constraints.* In First CPAIOR Workshop on Bin Packing and Placement Constraints (BPPC'08), 2008. (Cited on page 184.)

[Mateescu 2008] Robert Mateescu, Rina Dechter and Radu Marinescu. *AND/OR multi-valued decision diagrams (AOMDDs) for graphical*

models. Journal of Artificial Intelligence Research, vol. 33, pages 465–519, 2008. (Cited on page 42.)

[Miller 1998] D Michael Miller and Rolf Drechsler. *Implementing a multiple-valued decision diagram package*. In Multiple-Valued Logic, 1998. Proceedings. 1998 28th IEEE International Symposium on, pages 52–57. IEEE, 1998. (Cited on pages 59 and 64.)

[Mohr 1988] R. Mohr and G. Masini. *Good Old Discrete Relaxation*. In Proceedings of ECAI-88, pages 651–656, 1988. (Cited on pages 43, 148 and 151.)

[Mostaghim 2002] Sanaz Mostaghim, J"urgen Teich and Ambrish Tyagi. *Comparison of data structures for storing Pareto-sets in MOEAs*. In Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on, volume 1, pages 843–848. IEEE, 2002. (Cited on page 249.)

[Nair 1982] M. Nair. *On Chebyshev-type inequalities for primes*. AMM, vol. 89, pages 126–129, 1982. (Cited on page 224.)

[OscaR Team 2012] OscaR Team. *OscaR: Scala in OR*, 2012. Available from `https://bitbucket.org/oscarlib/oscar`. (Cited on pages 7, 149 and 177.)

[Pachet 1999] François Pachet and Pierre Roy. *Automatic generation of music programs*. In International Conference on Principles and Practice of Constraint Programming, pages 331–345. Springer, 1999. (Cited on pages 14 and 211.)

[Pachet 2001] François Pachet, Pierre Roy, Gabriele Barbieri and Sony CSL Paris. *Finite-length Markov processes with constraints*. transition, vol. 6, no. 1/3, 2001. (Cited on pages 14 and 237.)

[Pachet 2011] François Pachet and Pierre Roy. *Markov constraints: steerable generation of Markov sequences*. Constraints, vol. 16, no. 2, pages 148–172, 2011. (Cited on pages 14, 128 and 237.)

[Pachet 2014] François Pachet. *Avoiding plagiarism in Markov sequence generation*. 2014. (Cited on page 14.)

[Palmieri 2016] Anthony Palmieri, Jean-Charles Régin and Pierre Schaus. *Parallel Strategies Selection*. In International Conference on Principles and Practice of Constraint Programming, pages 388–404. Springer, 2016. (Cited on page 11.)

[Papadopoulos 2014] A. Papadopoulos, P. Roy and F. Pachet. *Avoiding Plagiarism in Markov Sequence Generation.* In Proceeding of the Twenty-Eight AAAI Conference on Artificial Intelligence, pages 2731–2737, 2014. (Cited on pages 5, 7, 14, 127, 128, 184, 257, 258 and 259.)

[Papadopoulos 2015] Alexandre Papadopoulos, François Pachet, Pierre Roy and Jason Sakellariou. *Exact Sampling for Regular and Markov Constraints with Belief Propagation.* In International Conference on Principles and Practice of Constraint Programming, pages 341–350. Springer, 2015. (Cited on pages 7, 127, 128, 130, 134 and 237.)

[Pennington 2001] Wayne D. Pennington. *Reservoir Geophysics.* vol. 66, no. 1, 2001. (Cited on pages 140, 243 and 275.)

[Perez 2014] Guillaume Perez and Jean-Charles Régin. *Improving GAC-4 for Table and MDD Constraints.* In Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, pages 606–621, 2014. (Cited on pages 11, 19, 44 and 140.)

[Perez 2015a] Guillaume Perez and Jean-Charles Régin. *Efficient operations on MDDs for building constraint programming models.* International Joint Conference on Artificial Intelligence, IJCAI-15, Argentina, 2015. (Cited on pages 11, 97, 184 and 263.)

[Perez 2015b] Guillaume Perez and Jean-Charles Régin. *Relations between MDDs and Tuples and Dynamic Modifications of MDDs based constraints.* arXiv preprint arXiv:1505.02552, 2015. (Cited on page 11.)

[Perez 2016] Guillaume Perez and Jean-Charles Régin. *Constructions and In-Place Operations for MDDs Based Constraints.* In Integration of AI and OR Techniques in Constraint Programming, pages 279–293. Springer International Publishing, 2016. (Cited on page 11.)

[Perez 2017a] Guillaume Perez and Jean-Charles Régin. *MDDs are Efficient Modeling Tools: An Application to Dispersion Constraints.* In Integration of AI and OR Techniques in Constraint Programming, 2017. (Cited on page 11.)

[Perez 2017b] Guillaume Perez and Jean-Charles Régin. *MDDs: Sampling and Probability Constraints.* In Principles and Practice of Constraint Programming, 2017. (Cited on page 11.)

[Perez 2017c] Guillaume Perez and Jean-Charles Régin. *Soft and Cost MDD Propagators*. In AAAI Conference on Artificial Intelligence, 2017. (Cited on pages 11, 140 and 186.)

[Perron 2013] L. Perron. *Or-tools*. In Workshop "CP Solvers: Modeling, Applications, Integration, and Standardization", 2013. (Cited on pages 7, 149, 177, 232 and 273.)

[Pesant 2004] G. Pesant. *A Regular Language Membership Constraint for Finite Sequences of Variables*. In Proc. CP'04, pages 482–495, 2004. (Cited on pages 2, 14, 41, 42, 52, 149, 156, 159, 161 and 186.)

[Pesant 2005] G. Pesant and J-C. Régin. *SPREAD: A Balancing Constraint Based on Statistics*. In CP'05, pages 460–474, 2005. (Cited on pages 9 and 235.)

[Pesant 2015] Gilles Pesant. *Achieving Domain Consistency and Counting Solutions for Dispersion Constraints*. INFORMS Journal on Computing, vol. 27, no. 4, pages 690–703, 2015. (Cited on pages 9, 235, 239, 245 and 276.)

[Petit 2001] Thierry Petit, Jean-Charles Régin and Christian Bessiere. *Specific filtering algorithms for over-constrained problems*. In Principles and Practice of Constraint Programming—CP 2001, pages 451–463. Springer, 2001. (Cited on page 200.)

[Puget 1993] Jean-François Puget. *Set constraints and cardinality operator: Application to symmetrical combinatorial problems*. In Third Workshop on Constraint Logic Programming–WCLP93, 1993. (Cited on page 211.)

[Quimper 2006] C-G. Quimper and T. Walsh. *Global Grammar Constraints*. In CP'06, pages 751–755, 2006. (Cited on pages 157 and 178.)

[Quimper 2010] Claude-Guy Quimper and Louis-Martin Rousseau. *A large neighbourhood search approach to the multi-activity shift scheduling problem*. Journal of Heuristics, vol. 16, no. 3, pages 373–392, 2010. (Cited on page 184.)

[Rabin 1959] Michael O Rabin and Dana Scott. *Finite automata and their decision problems*. IBM journal of research and development, vol. 3, no. 2, pages 114–125, 1959. (Cited on page 109.)

[Ravikumar 1996] Bala Ravikumar and Xuanxing Xiong. *A parallel algorithm for minimization of finite automata.* In Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International, pages 187–191. IEEE, 1996. (Cited on page 90.)

[Régin 1994] Jean-Charles Régin. *A filtering algorithm for constraints of difference in CSPs.* In AAAI, volume 94, pages 362–367, 1994. (Cited on page 14.)

[Régin 1996] J-C. Régin. *Generalized Arc Consistency for Global Cardinality Constraint.* pages 209–215, Portland, Oregon, 1996. (Cited on pages 243 and 277.)

[Régin 1997] Jean-Charles Régin and Jean-François Puget. *A filtering algorithm for global sequencing constraints.* Principles and Practice of Constraint Programming-CP97, pages 32–46, 1997. (Cited on page 14.)

[Régin 2002] Jean-Charles Régin. *Cost-based arc consistency for global cardinality constraints.* Constraints, vol. 7, no. 3-4, pages 387–405, 2002. (Cited on page 183.)

[Régin 2005] Jean-Charles Régin. *AC-*: a configurable, generic and adaptive arc consistency algorithm.* Principles and Practice of Constraint Programming-CP 2005, pages 505–519, 2005. (Cited on pages 2, 148, 150 and 168.)

[Régin 2011] Jean-Charles Régin. *Improving the expressiveness of table constraints.* In CP-11 ModRef Workshop, 2011. (Cited on pages 5, 42, 48, 51, 148 and 149.)

[Régin 2013] Jean-Charles Régin, Mohamed Rezgui and Arnaud Malapert. *Embarrassingly parallel search.* In International Conference on Principles and Practice of Constraint Programming, pages 596–610. Springer, Berlin, Heidelberg, 2013. (Cited on page 89.)

[Revuz 1992] Dominique Revuz. *Minimisation of acyclic deterministic automata in linear time.* Theoretical Computer Science, vol. 92, no. 1, pages 181–189, 1992. (Cited on page 30.)

[Rossi 2014] Roberto Rossi, Steven David Prestwich and S. Armagan Tarim. *Statistical Constraints.* In ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), pages 777–782, 2014. (Cited on pages 236 and 240.)

[Roy 2013] Pierre Roy and François Pachet. *Enforcing Meter in Finite-Length Markov Sequences*. In AAAI, 2013. (Cited on page 128.)

[Roy 2016] Pierre Roy, Guillaume Perez, Jean-Charles Régin, Alexandre Papadopoulos, François Pachet and Marco Marchini. *Enforcing structure on temporal sequences: The Allen constraint*. In International Conference on Principles and Practice of Constraint Programming, pages 786–801. Springer International Publishing, 2016. (Cited on pages 11 and 59.)

[Sabin 1996] Daniel Sabin and Eugene C Freuder. *Configuration as composite constraint satisfaction*. In Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop, pages 153–161. AAAI Press Palo Alto, CA, 1996. (Cited on page 14.)

[Schaus 2007a] P. Schaus, Y. Deville, P. Dupont and J-C. Régin. *The Deviation Constraint*. In CPAIOR'07, pages 260–274, 2007. (Cited on page 235.)

[Schaus 2007b] P. Schaus, Y. Deville, P. Dupont and J-C. Régin. Future and trends of constraint programming, chapter Simplification and extension of the SPREAD Constraint, pages 95–99. ISTE, 2007. (Cited on page 235.)

[Schaus 2007c] Pierre Schaus, Yves Deville and Pierre Dupont. *Bound-Consistent Deviation Constraint*. In Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings, pages 620–634, 2007. (Cited on page 235.)

[Schaus 2009a] Pierre Schaus*et al. Solving balancing and bin-packing problems with constraint programming*. These de doctorat, Université catholique de Louvain, 2009. (Cited on page 14.)

[Schaus 2009b] Pierre Schaus, Pascal Van Hentenryck and Jean-Charles Régin. *Scalable load balancing in nurse to patient assignment problems*. In International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pages 248–262. Springer Berlin Heidelberg, 2009. (Cited on page 14.)

[Schaus 2012] Pierre Schaus, Jean-Charles Régin, Rowan Van Schaeren, Wout Dullaert and Birger Raa. *Cardinality reasoning for bin-packing constraint: application to a tank allocation problem*. In Principles and

Practice of Constraint Programming, pages 815–822. Springer Berlin Heidelberg, 2012. (Cited on page 14.)

[Schaus 2013] Pierre Schaus and Renaud Hartert. *Multi-objective large neighborhood search.* In International Conference on Principles and Practice of Constraint Programming, pages 611–627. Springer, 2013. (Cited on pages 249 and 250.)

[Schaus 2014] P. Schaus and J-C. Régin. *Bound-consistent spread constraint.* vol. 2, no. 3, 2014. (Cited on pages 9 and 235.)

[Sellmann 2002] Meinolf Sellmann. *An arc-consistency algorithm for the minimum weight all different constraint.* In International Conference on Principles and Practice of Constraint Programming, pages 744–749. Springer, 2002. (Cited on page 183.)

[Sgarbas 2001] Kyriakos N Sgarbas, Nikos D Fakotakis and George K Kokkinakis. *Incremental construction of compact acyclic NFAs.* In Proceedings of the 39th Annual Meeting on Association for Computational Linguistics, pages 482–489. Association for Computational Linguistics, 2001. (Cited on page 106.)

[Shannon 1938] Claude E Shannon. *A symbolic analysis of relay and switching circuits.* Electrical Engineering, vol. 57, no. 12, pages 713–723, 1938. (Cited on page 16.)

[Shannon 1949] Claude Shannon *et al. The synthesis of two-terminal switching circuits.* Bell Labs Technical Journal, vol. 28, no. 1, pages 59–98, 1949. (Cited on page 16.)

[Srinivasan 1990] Arvind Srinivasan, Timothy Ham, Sharad Malik and Robert K Brayton. *Algorithms for discrete function manipulation.* In Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on, pages 92–95. IEEE, 1990. (Cited on pages 59, 64, 65 and 85.)

[Stornetta 1996] Tony Stornetta and Forrest Brewer. *Implementation of an efficient parallel BDD package.* In Proceedings of the 33rd annual Design Automation Conference, pages 641–644. ACM, 1996. (Cited on pages 6 and 90.)

[Tarjan 1975] Robert Endre Tarjan. *Efficiency of a Good But Not Linear Set Union Algorithm.* J. ACM, vol. 22, no. 2, pages 215–225, April 1975. (Cited on page 95.)

[Tewari 2002] Ambuj Tewari, Utkarsh Srivastava and Phalguni Gupta. *A parallel DFA minimization algorithm*. In International Conference on High-Performance Computing, pages 34–40. Springer, 2002. (Cited on page 90.)

[Trick 2003] Michael A Trick. *A dynamic programming approach for consistency and propagation for knapsack constraints*. Annals of Operations Research, vol. 118, no. 1-4, pages 73–84, 2003. (Cited on pages 41, 42, 52, 76, 183 and 238.)

[Ullmann 2007] Julian R Ullmann. *Partition search for non-binary constraint satisfaction*. Information Sciences, vol. 177, no. 18, pages 3639–3678, 2007. (Cited on page 152.)

[Van Hoeve 2004] Willem Jan Van Hoeve. *A hyper-arc consistency algorithm for the soft alldifferent constraint*. In Principles and Practice of Constraint Programming–CP 2004, pages 679–689. Springer, 2004. (Cited on page 200.)

[Van Hoeve 2006] Willem-Jan Van Hoeve, Gilles Pesant and Louis-Martin Rousseau. *On global warming: Flow-based soft global constraints*. Journal of Heuristics, vol. 12, no. 4-5, pages 347–373, 2006. (Cited on pages 200 and 203.)

[Verhaeghe 2017] Hélene Verhaeghe, Christophe Lecoutre and Pierre Schaus. *Extending Compact-Table to Negative and Short Tables*. 2017. (Cited on pages 44, 148 and 149.)

[Vilím 2004] Petr Vilím. *O (nlog n) filtering algorithms for unary resource constraint*. In International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, pages 335–347. Springer Berlin Heidelberg, 2004. (Cited on page 184.)

[Wang 2016] Ruiwei Wang, Wei Xia, Roland HC Yap and Zhanshan Li. *Optimizing Simple Tabular Reduction with a Bitwise Representation*. 2016. (Cited on pages 148, 149, 153 and 180.)

[Xia 2013] Wei Xia and Roland HC Yap. *Optimizing STR algorithms with tuple compression*. In International Conference on Principles and Practice of Constraint Programming, pages 724–732. Springer, 2013. (Cited on page 148.)

[Yip 2010] Justin Yip and Pascal Van Hentenryck. *Exponential propagation for set variables*. In International Conference on Principles and Practice of Constraint Programming, pages 499–513. Springer, 2010. (Cited on page 211.)

[Zagha 1991] Marco Zagha and Guy E Blelloch. *Radix sort for vector multiprocessors*. In Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 712–721. ACM, 1991. (Cited on page 92.)

Decision Diagrams, Algorithms and Constraints

Abstract: Multivalued Decision Diagrams (MDDs) are efficient data structures widely used in several fields like verification, optimization and dynamic programming. In this thesis, we first focus on improving the main algorithms such as the reduction, allowing MDDs to potentially exponentially compress set of tuples, or the combination of MDDs such as the intersection of the union. We go further by designing parallel algorithms, and algorithms handling non-deterministic MDDs. We then investigate relaxed MDDs, that are more and more used in optimization, and define the notions of relaxed reduction or operation and design efficient algorithms for them. The sampling of solutions stored in a MDD is solved with respect to probability mass functions or Markov chains. In order to combine MDDs with constraint Programming, we design the propagators of all the types of MMDD constraints in solvers, and introduce a new one, the channeling constraint. These new propagators outperform the existing ones and allow the reformulation of several other constraints such as the dispersion constraint, and even to define new ones easily. We finally apply our algorithm to several real world industrial problems such as text and music generation and geomodeling of a petroleum reservoir.

Keywords: Constraint Programming, Decision Diagram, MDD, Optimization.