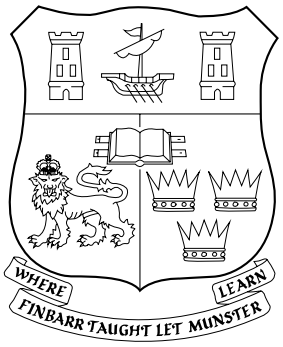
Computing Explanations for Interactive Constraint-based Systems

Alexandre Papadopoulos



A Thesis Submitted to the National University of Ireland

in Fulfillment of the Requirements for the Degree of

Doctor of Philosophy.

December, 2011

Research Supervisor: Prof. Barry O'Sullivan
Head of Department: Prof. James Bowen

Department of Computer Science,
National University of Ireland, Cork.

Contents

List of Figures

List of Tables

Abstract

Constraint programming has emerged as a successful paradigm for modelling combinatorial problems arising from practical situations. In many of those situations, we are not provided with an immutable set of constraints. Instead, a user will modify his requirements, in an interactive fashion, until he is satisfied with a solution. Examples of such applications include, amongst others, model-based diagnosis, expert systems, product configurators.

The system he interacts with must be able to assist him by showing the consequences of his requirements. Explanations are the ideal tool for providing this assistance. However, existing notions of explanations fail to provide sufficient information. We define new forms of explanations that aim to be more informative. Even if explanation generation is a very hard task, in the applications we consider, we must manage to provide a satisfactory level of interactivity and, therefore, we cannot afford long computational times.

We introduce the concept of representative sets of relaxations, a compact set of relaxations that shows the user at least one way to satisfy each of his requirements and at least one way to relax them, and present an algorithm that efficiently computes such sets. We introduce the concept of most soluble relaxations, maximising the number of products they allow. We present algorithms to compute such relaxations in times compatible with interactivity, achieving this by indifferently making use of different types of compiled representations. We propose to generalise the concept of prime implicates to constraint problems with the concept of domain consequences, and suggest to generate them as a compilation strategy. This sets a new approach in compilation, and allows to address explanation-related queries in an efficient way. We define ordered automata to compactly represent large sets of domain consequences, in an orthogonal way from existing compilation techniques that represent large sets of solutions.

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not been submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced. Parts of this work have appeared in the following publications which have been subject to peer review.

1. Barry O'Sullivan and Alexandre Papadopoulos and Boi Faltings and Pearl Pu, Representative Explanations for Over-Constrained Problems, *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, July 22-26, 2007, Vancouver, British Columbia, Canada
   (Acceptance Rate: 27%.)

2. Alexandre Papadopoulos and Barry O'Sullivan, Relaxations for Compiled Over-Constrained Problems, *Principles and Practice of Constraint Programming, 14th International Conference*, September 14-18, 2008, Sydney, Australia
   (Acceptance Rate: 32.1%.)

3. Alexandre Papadopoulos and Barry O'Sullivan, Compiling All Possible Conflicts of a CSP, *Principles and Practice of Constraint Programming, 15th International Conference*, September 20-24, 2009, Lisbon, Portugal
   (Acceptance Rate: 41.4%.)

The contents of this dissertation extensively elaborate upon previously published work and mistakes (if any) are corrected. Some sections of this work are previously unpublished, although they may appear in some journals in the future.

Alexandre Papadopoulos
March 2011.

Dedication

This dissertation is dedicated to my wife Careene and to my daughter Zoé.

Acknowledgements

When I started my PhD, I knew the road to completing it would be a long and hard one. The real surprise was that it did eventually end. Of course, this could not have happened without the help of many people that I would sincerely like to thank. Since I am not very expressive by nature, I will try my best to acknowledge them, but I already know I will not manage to express the full extent of my gratitude.

First, I wish to thank Barry O'Sullivan for welcoming me at 4C – who could dream of a better place to do a PhD in constraints – and for his support as my supervisor. Of course, he did much more than simply being my supervisor. He allowed me to work as a peer with him and the rest of the team at 4C, which constituted one of the most beneficial teaching I received during my PhD studies. I also wish to thank him on more practical aspects. For ensuring I would get financial support to allow me to complete my PhD in good comfort. For trying to teach me some proper English, and help me improve my general writing skills (this does not apply to this part, as it has not been proofread by him). And for his patience in reading this dissertation. The level of detail he went into when proofreading this dissertation was far beyond any expectation I had. Of course, all remaining errors/language mistakes/typos are all mine.

I wish to thank Science Foundation Ireland for supporting this work, under grant No. 05/IN/I886.

I wish to thank the external examiner, Professor Pascal Van Hentenryck (Brown University, USA), and the internal examiner, Dr. Ken Brown, for also having had the patience to read and examine this dissertation.

I wish to thank Dr. Christian Bessière (Université Montpellier 2, France), who was my Masters supervisor, for introducing me to Constraint Programming, and, more concretely, to Barry, thus supporting my desire to pursue a PhD.

I wish to thank a lot of people at 4C for all the good time I spent with them. In particular, I had the great chance to be able to work with both Emmanuel Hebrard and Hadrien Cambazard, and I wish to thank them for their patience; they taught me so many things.

I wish to thank my parents for having made me who I am, whatever this is.

And finally, I wish to thank Careene, my wife, who has always been there when I needed most. She wholeheartedly supported, from the very beginning, my desire to pursue a PhD, and went as far as sacrificing herself in support of this desire. My daughters gave me the inspiration to accomplish this work. Zoé, by reminding me that no work is ever too hard. Cassandre, by giving me the motivation to enjoy the birth of at least one of my children without being a PhD student anymore.

Chapter 1

Introduction

Summary. *This chapter briefly introduces the research presented in this dissertation. Constraint Programming and Explanations are introduced. The complexity of explanation generation is discussed, and an example is given of a current approach. The limitation of the current approaches is then highlighted. A statement of the thesis is presented, and the general contributions of this work are outlined. Finally, the structure of this dissertation is described.*

1.1 Context

Constraint Programming has become an active field of research in Artificial Intelligence. It emerged as a successful paradigm for modelling hard combinatorial problems, with applications such as scheduling, timetabling, planning, packing, and others arising from real-world situations [136, 118]. It also defines a framework for developing resolution and optimisation procedures for all these kind of problems.

In 1971, Cook defined the concept of *NP*-Completeness [15]. This formalises the impression that some problems are intrinsically hard to solve, problems that if one was to solve by hand, he would have to resort to intuition, or luck, or both, but could not apply a systematic strategy. Informally, *NP* is the class of decision problems, i.e. problems admitting a yes or no answer, such that, for in-

stances admitting a yes answer, the answer can be verified in polynomial time. *NP*-Complete problems are the hardest problems in *NP*: by definition, solving any of those would allow us to solve any problem in *NP*. *NP*-Completeness characterises hard problems, as it is widely believed that no efficient algorithm exists for any *NP*-Complete problem.

Constraint Programming allows us to model and solve hard problems by expressing them as a Constraint Satisfaction Problem (CSP) and by applying algorithms that can solve a CSP. The basic problem of determining if a CSP is satisfiable, i.e. if it admits a solution, is *NP*-Complete [56]. This means that, most likely, no naive, brute-force algorithm can efficiently solve a CSP. In order to solve problems of practical significance, different techniques can be used to guide search procedures towards a solution, but without any theoretical guarantee in terms of efficiency.

However, in many real-life settings, solving a problem is just the first part of the task [111, 74]. Particularly in settings involving user interaction, it can happen that the solution found is not satisfactory, or that no solution at all can be found. In these cases, further action needs to be taken. This is achieved by computing explanations.

1.2 Explanations

Explanations are a versatile concept of Artificial Intelligence, and appear in many different ways and contexts. The main purpose of an explanation is to justify in a concise and meaningful way an event that has occurred [89]. For example, suppose it is found that a particular CSP has no solution containing $X = a$, $Y = b$ and $Z = c$. An explanation can be computed that explains why this is the case. This can be needed for two reasons, corresponding to two main aspects where explanations appear in Constraint Programming.

Explanations can be used as a *technique* that help search procedures avoid parts of the search that lead to no solution, and learn from failure when they do reach a dead-end [80]. For example, an explanation could be: $X = a$, $Y = b$ and $Z = c$ is inconsistent in the CSP because $X = a$ and $Z = c$ is already inconsistent. Such an explanation is often called a *conflict*. This provides additional

understanding of the CSP to the search procedure and allows it to avoid making this failure again in the future.

In some settings, a level of user interactivity is present, establishing a two-way process between a user and the solver. The user can direct the solver towards a specific direction, and the solver can use explanations as a *tool* to present back to the user justifications about its steps or discoveries. In the previous example, suppose that it was the user who wanted $X = a$, $Y = b$ and $Z = c$, as he was looking for a preferred solution. When the solver finds this to be inconsistent in the CSP, the user wants to know why. By letting him know that $X = a$ and $Z = c$ are incompatible, the user knows he has to review his choices by relaxing one of the two requirements.

Computing explanations is a very hard task, harder than solving a CSP. The main leap in complexity follows from the complexity of recognising a solution. Intuitively, while recognising a solution to a CSP is easy – it only requires that each constraint be satisfied – recognising an explanation is already hard. In the previous example, checking that $X = a$ and $Z = c$ is a conflict is itself *CoNP*-Complete, as it requires checking it is inconsistent. There has been work that studied the complexity of explanation computation in the context of Horn clauses. Horn theories, i.e. theories that contain only Horn clauses, are an interesting restriction for which satisfiability is polynomial. In this setting, it has been shown that many problems associated with computing explanations are *NP*-Complete [123, 124, 41]. When considering the general setting where testing satisfiability is already *NP*-Complete, such as for the Constraint Satisfaction Problem, these problems move higher up in terms of complexity [40]. The Polynomial Hierarchy defines a hierarchy of complexity classes to formalise this growing level of complexity. For example, the class Δ_2^P is defined as P^{NP}, denoting the class of problems that can be solved in polynomial time using an oracle for an *NP*-Complete problem. Similarly, Σ_2^P is defined as NP^{NP}, denoting the class of problems for which the yes-instances can be verified in polynomial time using an oracle for an *NP*-Complete problem. This represents harder problems than *NP*-Complete problems. It is known that problems that are polynomial in Horn theories become complete for Δ_2^P in the general case where satisfiability is *NP*-Complete, and problems that are *NP*-Complete in Horn theories become complete

for Σ_2^P in the general case [40, 1].

Nevertheless, there are techniques to compute explanations. In Constraint Programming, the QUICKXPLAIN algorithm is considered as one of the standard techniques to compute minimal conflicts [74]. Suppose a user states n constraints $c_1, \ldots, c_n$ that are inconsistent. As shown in the previous example, a conflict is a subset of those constraints that is also inconsistent. A set-wise minimal conflict is a conflict in which the removal of any constraint would make the remaining set of constraints consistent. A minimal conflict thus shows a concise cause of inconsistency in a set of constraints. The XPLAIN algorithm [31] works as depicted in Algorithm 1. It makes use of an *NP*-Complete oracle in the form of a function $s(C)$ that checks the satisfiability of a set C of constraints.

Algorithm 1: XPLAIN

Data: $c_1, \ldots, c_n$
Result: A minimal conflict

```
1  X ← ∅
2  C ← ∅
3  while s(X) do
4      k ← 0
5      while s(C) and k < n do
6          k ← k + 1
7          C ← C ∪ {c_k}
8      if s(C) then return "No conflict"
9      X ← X ∪ {c_k}
10     C ← X
11 return X
```

This algorithm operates in a fairly intuitive way. It starts by adding constraints until inconsistency is detected (lines 5-7), after having added $c_1, \ldots, c_k$. The minimal conflict is a subset of $c_1, \ldots, c_k$, and c_k belongs to it (line 9). This is repeated again, by adding c_k before starting again from c_1 (line 10). The QUICKXPLAIN algorithm [74, 75] improves this basic scheme by making a dichotomic search of the set of constraints. This makes this algorithm scalable with regard to problem size, in particular when the size of the minimal conflict is small, a safe assumption for real-life cases. This algorithm forms the basis of the explanation facility

of ILOG Configurator[1], one of the main industrial constraint programming configuration tool.

1.3 Limitation of Current Approaches

QUICKXPLAIN can be considered as one of the standard ways of computing explanations in Constraint Programming. However, the type of explanations it computes can actually be of little utility for applications involving user interactivity. In order to highlight the difference in the use of explanations between the two aspects mentioned above, consider the same example. When the search discovers that $X = a \wedge Z = c$ is inconsistent in the CSP, it can infer, and add to the CSP, the constraint that $X \neq a \vee Z \neq c$. Even if the search could discover other failures, this will potentially be useful to avoid discovering the same failure again. However, when presenting an explanation to a user, he should obtain a complete understanding of why his requirements cannot be met. In this regard, simply telling him that $X = a$ and $Z = c$ is not possible is merely partial information. Surely enough, the user will understand he has to remove one of the two requirements, and it is up to him to decide which. However, this does not guarantee the remaining requirements can be met. In fact, there could be another conflict saying that $X = a$ and $Y = b$ is also inconsistent. Now, the user has to remove another of his requirements too. If these are the only two conflicts, that is sufficient. Indeed, the user has to "break" *all* the conflcits to have his requirements partially satisfied. However, in a real problem, there could be a large, potentially exponential, number of such conflicts. This poses the problem of how can a user be made aware of them.

There have been alternative forms of explanations that take this problem into account. Rather than explaining a cause of inconsistency, effectively answering a "why" question, they focus on providing a way to recover from it, answering instead a "how" question. For example, relaxations show a subset of requirements that can be satisfied. In the previous example, the user could be told straightaway that he can remove $Y = b$ and $Z = c$. A single explanation of this type is

[1] `www.ilog.com`

sufficient to achieve the desired effect, which is to recover consistency. However, other considerations now come into play. It is important to make suggestions to the user that maximise acceptance. For example, by suggesting to relax $X = a$, the two conflicts are also broken. But, most probably, if the user values his three requirements more or less equally, he will be happier to simply relax $X = a$ than both $Y = b$ and $Z = c$. Additionally, by knowing that simply relaxing $X = a$ is enough to recover consistency, he understands that $X = a$ appears in all conflicts, no matter how many of them. This provides a deeper understanding of the problem.

1.4 Dissertation Overview

In summary, well-known explanation techniques in Constraint Programming have a poor utility in settings involving user interaction. Alternative forms of explanations are needed that are more suited for these applications, and little work has been done in this direction in Constraint Programming. The main objective for these explanations has to be to provide to a user a good understanding of the problem in order to give him enough confidence when reviewing his requirements and making choices. For this objective to be achieved, more sophisticated explanations are required. However, the complexity of computing such explanations looks prohibitive, and it is legitimate to wonder how one can provide such a facility for problems of any real-world significance. Furthermore, interactivity implies quick response times. More specifically, according to user interface design criteria, for a user to perceive interaction as real-time, response times need to be of around 250 milliseconds in practice [109].

In this dissertation, we present a novel approach to explanation computation in Constraint Programming with user interaction. We define new types of explanations, and we claim they are more informative to the user in that they provide a more complete understanding of a problem and help him draw better conclusions.

1.4.1 Thesis

The thesis defended in this dissertation is that we *can* compute more informative explanations in constraint-based systems. Actually, we claim we can compute them with a response time compatible with interactive product configurators, the motivating domain for this research. The main strategy is to apply *compilation methods*, where extensive computation is performed before effectively computing explanations. More precisely, the thesis is the conjunction of a number of sub-theses. We present them below, along with a discussion of the way the dissertation supports each of them.

Sub-thesis 1. *We show that presenting a* set *of relaxations that is at the same time informative and compact, while giving a better picture of a problem than showing a single relaxation, can be achieved in practice.*

In Chapter 3, we define the concept of a *representative set of relaxations*. A representative set of relaxations is a compact set of relaxations that shows the user at least one way to satisfy each of his requirements and at least one way to relax them. This notion can be extended to more complex considerations, such as combination of user's requirements, or by taking into account preferences, if at least partial preferences are provided. To overcome complexity issues, we present an algorithm that heuristically converges to a representative set of relaxations.

Sub-thesis 2. *We claim that, when suggesting a relaxation, we need to take into account the solutions it allows. We propose to compute relaxations that admit the highest number of solutions, or the lowest number of solutions. A relaxation with the highest number of solutions leaves the user with the largest choice. A relaxation with the lowest number of solution corresponds the most closely to the user's original requirements.*

In Chapter 4, we define algorithms to compute such relaxations using compiled representations of a problem to cope with the complexity of computing those. We define algorithms based on automata, as a baseline, then we abstract and generalise these procedures in order to make them independent from any particular representation, by identifying the structural properties a representation must

satisfy for them to apply. An important consequence of this is that it by choosing the most compact representation possible, we obtain the most efficient procedures.

Sub-thesis 3. *We claim that complete knowledge of the conflicts of a problem helps compute more useful explanations. It is possible to circumvent the complexity associated with computing conflicts and arising from their high number by compiling a problem to a new type of representation that contains explicit knowledge of its conflicts.*

In Chapter 5, we propose a new way to compile a constraint problem to handle the requirements posed by explanation computation, with the concept of *domain consequence*. This allows us to detect conflicts inherent in a problem before the user specifies any particular requirement. For example, nogoods are a particular case of domain consequences, and thus the method we propose facilitates early detection of all the nogoods of a problem. From a logical point of view, this concept generalises the concept of prime implicates. In Chapter 6, we present a data structure and a series of algorithms to represent, in a compact way, a large set of domain consequences, and to compute the domain consequences of a problem in a more efficient way. This defines a new type of compilation that is more suited to explanation-related computation.

1.5 Outline of the Dissertation

This dissertation is structured as follows. Chapter 2 is an overview of concepts pertaining to the work in this dissertation. Particularly, we formally introduce the fundamental concepts in Constraint Programming, such as constraint satisfaction problems (CSPs), consistency, propagation, solving procedures, and we discuss the structure of CSPs. We then introduce the concept of explanation, and review the literature showing how this notion appears in many different ways and contexts, and how they relate or differ to our setting. We present knowledge compilation, as an approach, and describe the most widely known compilation techniques. We explain what configuration is, and show it is a widely studied application to which interactive constraint programming applies. Finally, we present a synthesis of this overview that defines the setting of this dissertation.

In Chapter 3, we introduce the concept of a representative set of relaxations. We discuss the complexity of enumerating explanations, and show the impact on algorithms for generating sets of explanations. We present REPRESENTATIVEXPLAIN, an algorithm we developed that heuristically computes a representative set of explanations, and evaluate its performance with an experimental study.

In Chapter 4, we study how compiled representations can be used to compute most or least soluble relaxations. We present two algorithms that operate on automata, but then show how they can be abstracted from the representation they are operating on. We show how our algorithms can be applied to other representations from the Compilation Map, a framework for defining compiled representations introduced by Darwiche and Marquis [22]. The experimental evaluation shows that by relying on a more compact representation, we can compute these relaxations in very short times.

In Chapter 5, we study a new approach for compiling a problem that is specifically designed for explanation computation. It is based on the notion of domain consequence, which we define. We show that the representations considered in the previous chapter, which correspond to the commonly used compilation techniques, do not allow us to efficiently compute some type of explanations, namely those that require knowledge of all the conflicts of a problem. Our approach aims at providing this functionality. We give ideas of applications of this method. A simple experimental study shows the behaviour of problems in terms of their number of domain consequences. This motivates for further study in the representation of the domain consequences of a problem.

In Chapter 6, we define ordered automata to encode sets of domain consequences to compactly represent a large number of domain consequences. The main contribution in this chapter is to design a range of algorithms that allow us to generate the domain consequences of a problem directly on this representation, as it has been shown in the previous chapter that no algorithm that explicitly enumerates each domain consequence can ever be tractable.

Finally, in Chapter 7, we summarise our contributions and suggest directions for future work.

Chapter 2

Background and Related Work

Summary. *We begin by defining constraint satisfaction problems, and related concepts such as consistency and propagation. We then introduce the notion of explanation, which is at the core of the work in this dissertation. We present a very general perspective, exhibit a formal point of view and introduce several areas where explanations appear, aiming at illustrating their versatility. Finally, we introduce two more domains that concern our work, namely knowledge compilation and configuration.*

2.1 Overview

In this chapter, we introduce a series of concepts, definitions and results from the literature that are used throughout this dissertation, or that are related with, and relevant to, the work presented in it. The purpose of this presentation is to make the reader familiar with the broader area to which the work presented in this dissertation pertains, in an attempt to make the presentation as self-contained as possible, and to make a critical review of the literature, in order to put our work into context. Some concepts and related work cannot be clearly presented in a general way in this chapter and for that reason are introduced in subsequent chapters, where they are more appropriate.

2.2 Constraint Satisfaction Problems

2.2.1 Formal Definition

A constraint satisfaction problem (CSP) is a simple and general way to formally describe a combinatorial problem. A CSP is defined in terms of a set of variables, a domain for each variable stating the values it can take, and a set of constraints, each holding on a subset of the variables and restricting the values those variables can simultaneously take. In this dissertation, we consider the case of variables having a finite discrete domain. The problem consists in assigning to all the variables of the problem a value from their domain, such that all constraints are satisfied.

Definition 2.2.1 (Constraint Satisfaction Problem)**.** A CSP $\mathcal{P}$ is defined as a tuple $\langle X, D, C \rangle$, where X is a finite set of n variables $\{X_1, \ldots, X_n\}$, D is a set of finite domains $\{D(X_1), \ldots, D(X_n)\}$, where $D(X_i)$ is the set of values variable X_i is allowed to take, and C is a finite set of constraints $c_1, \ldots, c_k$. In the most general form, a constraint c_j is a predicate holding on a set of variables, its scope, denoted $X(c_j) = \{X_{i_1}, \ldots, X_{i_p}\}$, which has value true for each combination of values $\langle v_{i_1,\ldots,v_{i_p}} \rangle$ of $D(X_{i_1}) \times \ldots \times D(X_{i_p})$ that satisfies this constraint. A solution to the CSP is a tuple $\langle v_1, \ldots, v_n \rangle$ of $D(X_1) \times \ldots \times D(X_n)$ that satisfies every constraint in C.

In the rest of this dissertation, we will adopt some simplifications, to avoid heavy formalism, where this does not create ambiguity. The domain of variable X_i might be simply denoted D_i. Additionally, and more often that not, a problem will be simply defined in terms of a set of constraints. Sometimes, the variables and their domains are assumed to be implicitly or trivially known by the context. In other cases, we simply assume that the set of constraints completely specifies the problems. Indeed, a domain $D(X_i)$ can be expressed in the form of a unary constraint stating that $X_i \in D(X_i)$. Additionally, as it makes little sense to introduce variables that do not appear in any constraint, we can implicitly assume that the set of variables is defined as the union of the scopes of each constraint.

2.2.2 On the Structure of CSPs

CSPs are sometimes viewed as constraint networks. We can reason about the *structure* of a constraint network (as opposed to its semantics) in terms of graph representations.

Definition 2.2.2 (Constraint Graph)**.** Given a CSP $\mathcal{P} = \langle X, D, C \rangle$, the associated constraint graph is the graph $G_{\mathcal{P}} = \langle Q, E \rangle$ defined as follows:

- $Q = \{1, \ldots, n\}$, with $n = |X|$;
- $\{i, j\} \in E$ iff there is a constraint $c \in C$ such that $X_i \in X(c)$ and $X_j \in X(c)$.

For a CSP containing only binary constraints (i.e. involving only two variables), each constraint between two variables is represented by an edge between the two corresponding vertices. For a CSP with constraints of any arity, each constraint is represented by a clique between all corresponding vertices. The constraint graph is sometimes called *primal constraint graph*. Alternatively, the structure of a CSP can be represented by its *dual graph* [34].

Definition 2.2.3 (Dual Graph)**.** The dual graph of a CSP $\mathcal{P} = \langle X, D, C \rangle$ is a labelled graph $H_{\mathcal{P}} = \langle Q, E, l \rangle$ such that:

- $Q = \{1, \ldots, k\}$, with $k = |C|$;
- for each node $i \in Q$, $l(i) = X(c_i)$;
- $E = \{\{i, j\}, i \neq j / l(i) \cap l(j) \neq \emptyset\}$;
- for each edge $e = \{i, j\}$, $l(e) = l(i) \cap l(j)$.

Example 2.2.1. Consider a CSP with variables A, B, C, D, E and three ternary constraints, holding respectively on $\{ABC\}$, $\{BCD\}$ and $\{CDE\}$. The constraint graph of this CSP and its dual graph are shown on Figure 2.1. ▲

The dual graph gives rise to a binary CSP equivalent to the original one (usually referred to as the dual encoding): the domains of the dual variables range over the allowed value combinations of the constraint, and the dual constraints

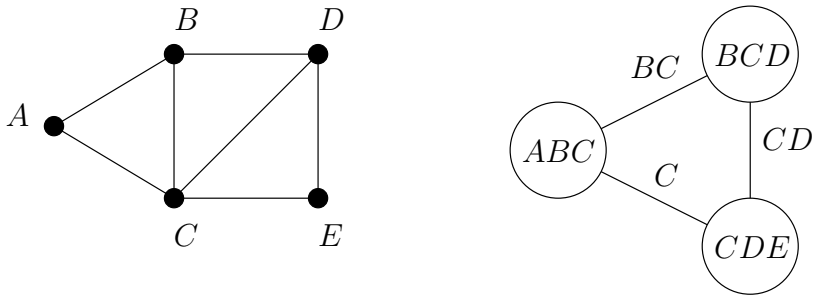


Figure 2.1: The constraint graph and the dual graph of the example CSP

impose equality of values over the shared variables. In this dual CSP, some constraints are redundant, in that their removal does not change the solutions of the CSP. Namely, an arc between two nodes can be removed if its variables appear in all the arcs of an alternative path between these two nodes. A graph obtained by removing all redundant arcs, called a *join graph*, satisfies the following property: for any two nodes sharing a variable, this variable appears in the all the arcs of at least one path linking these two nodes. A join graph that is a tree is called a *join tree*. Not all CSPs admit a join tree. Those that do are called *acyclic CSPs*, which is well-known to be a desirable property for various reasons [4]. Particularly, acyclic CSPs form a tractable category of CSPs. For example, they can be solved in polynomial time [58], while efficient algorithms that have been developed for databases can be applied to identify acyclic CSPs and compute a join tree.

Example 2.2.2. Consider the problem at Example 2.2.1. It is acyclic. Indeed, the arc between nodes ABC and CDE can be removed: C appears in all arcs of the path remaining between ABC and CDE. There is no other redundant arc to be removed, and the join graph obtained is a tree, which means the CSP is acyclic, admitting the join tree shown at Figure 2.2. ▲

A non-acyclic CSP can be turned into an acyclic CSP by merging nodes in its dual graph to bigger nodes (i.e. containing more variables) in such a way that it admits a join tree. This is usually referred to as *tree decomposition* [34]. In terms of the CSP, this amounts to merging some constraints into bigger constraints, standing for the conjunction of the merged constraints. If we reason on the primal graph, this amounts to adding edges between vertices creating bigger cliques, a process which is known a triangulation [132, 4]. The *width* of this join tree is

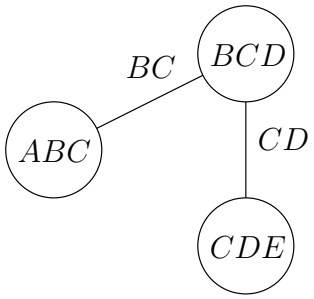


Figure 2.2: A join tree of the example

equal to size of its bigger node minus one, and the *tree-width* of a CSP is equal to the minimum width over all possible join trees. This corresponds exactly to the tree-width of the primal graph, as it has been defined in literature [114, 115, 116].

Example 2.2.3. The width of the CSP at Example 2.2.1 is 2. Consider now a CSP with five constraints holding respectively on ABC, BC, BD, CE, DE. The constraint graph and the join graph are shown on Figure 2.3 (dashed lines show redundant arcs removed from the dual graph). The join graph is not a tree, therefore this CSP is not acyclic.

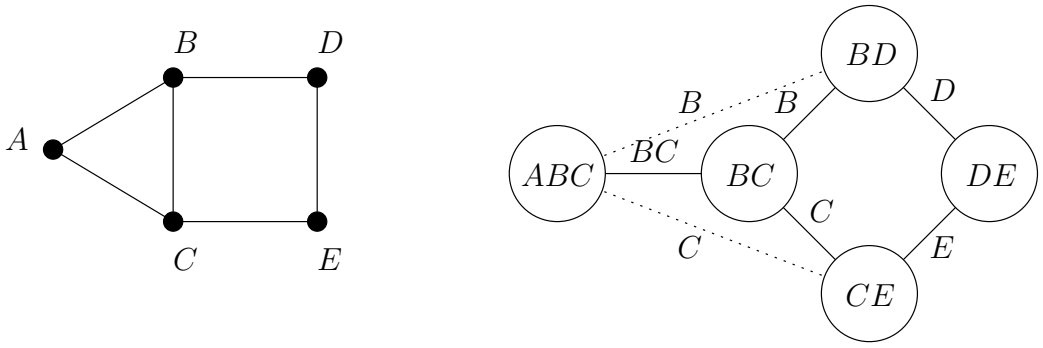


Figure 2.3: The join graph of a non-acyclic CSP

An acyclic CSP can be obtained by merging the four nodes of the join graph containing B, C, D, E into one big node. This corresponds to adding an edge BE and an edge CD in the constraint graph, creating a clique between the nodes B, C, D, E. This tree decomposition is of width 3. However, by merging the nodes BC and BD on the one hand, and the nodes CE and DE on the other hand,

which corresponds to adding an edge CD on the constraint graph, we obtain a tree decomposition shown on Figure 2.2, of width 2, which is optimal. Therefore the tree-width of the CSP is 2. ▲

The benefit of finding a join tree for a non-acyclic CSP in terms of tractability is limited by the number of constraints (corresponding to nodes of the dual graph) that have to be merged to make the CSP acyclic, more precisely by the size of the resulting merged constraints. A trivial tree-decomposition would involve merging all constraints into a single one, thus obtaining a tree-decomposition of width equal to the number of variables of the CSP minus one. For the most degenerated CSP, this would be the optimal tree-decomposition, i.e. the tree-width of the CSP would be equal to its number of variables minus one. In the other end, when a CSP is already acyclic, its tree-width is equal to the size of its largest constraint (minus one). In between the two extremes, the closer the tree-width of a CSP is to the size of its largest constraint (minus one), the more "tree-like" is the CSP, or the closer it is to being acyclic. It became very widespread in the literature to say that such a CSP *is structured*. This is just a short way of saying that such a CSP has a structure that can be taken advantage of in a variety of ways. Structured CSPs are interesting to us as we shall see later in this chapter.

2.2.3 Types of Constraints

In the definition we gave of a CSP, we did not make any special assumption of the way constraints are defined, other than that they provide a predicate defining how they are satisfied. In most cases, we assume that checking the satisfaction of a constraint is polynomial, in which case testing the satisfiability of a CSP is *NP*-Complete [56, 95].

We usually distinguish between constraints that are defined *extensionally* and *intensionally*. Extensionally defined constraints are also called table constraints. A table constraint holding on $X_{i_1}, \ldots, X_{i_p}$ is defined by a subset of $D(X_{i_1}) \times \ldots \times D(X_{i_p})$, containing the combination of values that are allowed by the constraint. For negatively defined table constraint, this is the subset of combination of values that are forbidden by the constraint. Intensionally defined constraints are usually

defined in terms of their semantics, whether a simple arithmetic expression or a complex domain-specific semantics.

We also distinguish between *global constraints* and non-global constraints. Global constraints are constraints that hold on an unbounded number of variables. For example, the *AllDifferent* constraint says that a set of variables should be assigned to mutually different values [110].

2.2.4 Problem Solving

Approaches for solving CSPs differ in the way they explore the search space. A complete solver makes use of a search procedure that considers all variable value assignments in a systematic and exhaustive way [94]. It can prove that no solution exist or that the computed solution is optimal, when some variable is to be optimised (i.e. has to be assigned the lowest or highest possible value). An incomplete solver on the other hand uses a local search procedure [68]. This procedure moves locally from a full variable value assignment to another, using heuristics to converge to a feasible and optimal solution. These solvers tend to be more efficient than complete solvers, and are used as a last resort for large-scale problems. However, they offer no guarantee of completeness (no exhaustive search, cycles are possible) or optimality (possibility of getting trapped in local optima). Meta-heuristics exist that reduce the risk of these phenomena from happening. Incomplete solvers are more often associated with optimisation problems, and are not relevant to our work.

Search Procedures

A search procedure operates by incrementally extending partial variable value assignments. When it finds that a partial assignment cannot be extended to any valid solution, it discards it by backtracking to a previous decision and considering a new one. When a full assignment has been reached, a solution has been found, or when all possible decisions have failed, no solution exists. This algorithm is referred to as *backtracking algorithm*. The space of partial assignments is called a *search tree*.

Each node of the search tree is associated with a decision, where one variable is assigned a value from it current domain. Thus, to each node corresponds an assignment of a subset of variables. Of course, the entire search tree cannot be explicitly explored. Strategies exist to refrain from considering parts of the search tree that can be proved not to contain any solution, and from repeatedly explore parts of the search tree that lead to the same failure (thrashing).

Example 2.2.4. Consider a simple CSP with $X = \{X_1, X_2, X_3, X_4\}$, $D(X_1) = \{1, 2\}$, $D(X_2) = D(X_3) = \{1, 2, 3, 4\}$, $D(X_4) = \{1, 2, 3, 4, 5\}$, and two constraints:

$$\mathit{AllDifferent}(X_1, X_2, X_3, X_4)$$

$$X_2 + X_3 < X_4$$

There is no solution containing the value $X_2 = 2$, as discovered during the search depicted at Figure 2.4.

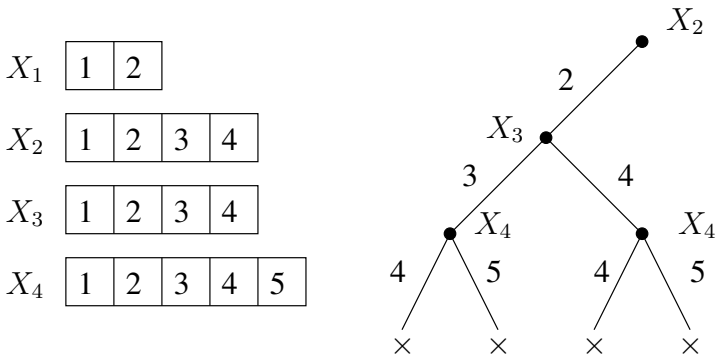


Figure 2.4: Part of the search tree finding that $X_2 \neq 2$.

After the decision $X_2 = 2$, the value 2 can safely be removed from the domains of the other variables, because of the $\mathit{AllDifferent}$ constraint. This reduces the domain of X_1, forcing it to take value 1. At this stage, only values 3 and 4 are left in the domain of X_3. After the decision $X_3 = 3$, X_4 is left with values 4 and 5. For each of those values, the constraint $X_2 + X_3 < X_4$ is violated, therefore the search backtracks by removing the decision $X_3 = 3$. It then tries the second possible value $X_3 = 4$, which similarly fails. This allows the search to backtrack even further by removing the decision $X_2 = 2$, as it did not lead to any solution.▲

Consistency and Propagation

While decisions are made, the domains of the uninstantiated variables can be updated to be consistent with those decisions. This happens by removing some values that can be inferred not to belong to any solution in the current state (i.e. current domains), repeating this process until a fixed-point is reached. This process is referred to as *propagation*. Propagation allows us to reduce the amount of the search tree that is explored. In particular, when the domain of some variable is emptied, a contradiction arises, indicating that the algorithm should backtrack to a previous decision.

A value that is detected not to belong to any solution is said to be *inconsistent*. The inconsistency of a value is defined with regard to a specific form of consistency, as different forms of consistency might not detect the same values as being inconsistent. One of the most basic forms of consistency is Generalised Arc-Consistency (GAC).[1]

Definition 2.2.4 (Generalised Arc-Consistency)**.** Let c be a constraint with $X(c) = \{X_1, \ldots, X_k\}$. A value $v_i \in D(X_i)$, with $i \leq k$, is *consistent* with regard to c in the current domains iff there exists a tuple $\langle v_1, \ldots, v_i, \ldots, v_k \rangle \in D(X_1) \times \ldots \times \{v_i\} \times \ldots \times D(X_k)$ that satisfies c. This tuple is called a *support* for v_i. c is Generalised Arc-Consistent if for every $X_i \in X(c)$ and every $v_i \in D(X_i)$, v_i is generalised arc-consistent.

Definition 2.2.5. Given a CSP $\mathcal{P} = \langle X, D, C \rangle$, a set of constraints $C' \subseteq C$ is generalised arc-consistent if every constraint $c \in C'$ is generalised arc-consistent. $\mathcal{P}$ is generalised arc-consistent if C is generalised arc-consistent.

In other words, a value that does not have a support for a given constraint will not appear in any solution, as any complete assignment containing this value will violate at least this constraint. Such a value can therefore be removed from the domain of the variable. This removal can in turn affect the consistency of another value (if its only support involved the removed value), so a series of inconsistent value removals has to be undertaken until a fixed point is reached, by a procedure called *propagator*.

[1] Originally defined on networks of binary constraints as *arc-consistency*, where an arc corresponds to a constraint in the constraint graph representation [94].

For generic constraints extensionally defined, propagators have been developed that achieve GAC, such as AC3, AC4, AC2001, etc. [5]. Constraints defined intensionally need to provide a dedicated propagator.

Example 2.2.5. Let us consider again the CSP introduced in Example 2.2.4. The initial domains are not arc-consistent. For example, value 4 in the domain of X_2 is not arc-consistent with regard to constraint $X_2 + X_3 < X_4$, as there is not value for X_3 and X_4 that would satisfy the inequality. Similarly, 4 can be removed for X_3 and 1,2 can be removed for X_4. In these reduced domains, value 3 for X_4 is no more arc-consistent with regard to $AllDifferent$, and has, in turn, to be removed (Régin [110] describes a propagator for $AllDifferent$, i.e. an algorithm that would detect this). The domains shown in Figure 2.5 are arc-consistent.

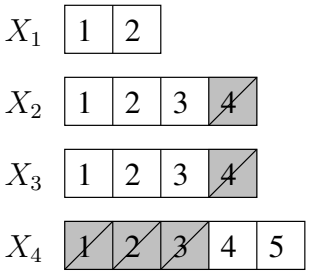


Figure 2.5: Arc-consistent domains for the example CSP.

Consider now the decision $X_2 = 2$. This decision is propagated by the search procedure by enforcing arc-consistency to the domains (see Figure 2.6). More specifically, because of the $AllDifferent$, we can remove value 2 for X_1 and X_4, which instantiates X_1 to 1, and in turn we can remove value 1 for X_3. X_3 only has value 3 in its domain. But this value is not arc-consistent with regard to $X_2 + X_3 < X_4$ (there is no value for X_4 which is strictly greater than $2 + 3$). Therefore, 3 has to be removed too, which makes the domain of X_3 empty, thus raising a contradiction.

The search can backtrack on the decision $X_2 = 2$, and try $X_2 = 3$. By enforcing arc-consistency after this decision, all domains are reduced to a single value, thus having a solution (see Figure 2.7). ▲

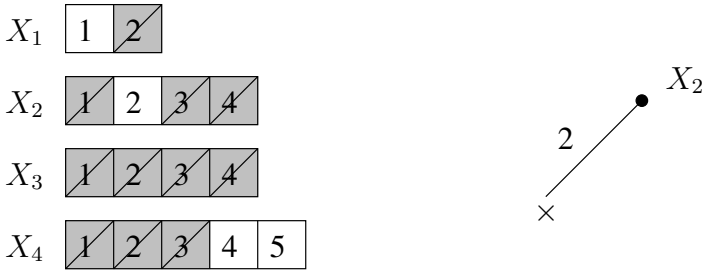


Figure 2.6: Contradiction after the decision $X_2 = 2$.

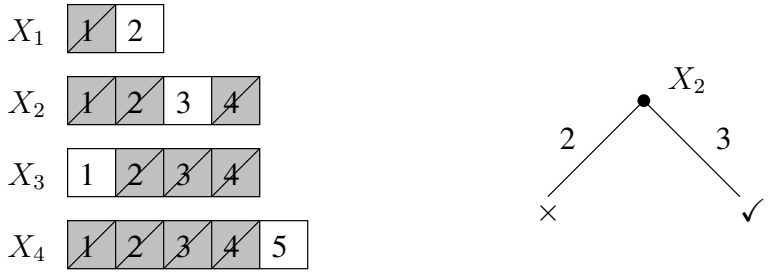


Figure 2.7: Arc-consistent domains after the decision $X_2 = 3$.

Levels of Consistency

GAC is defined only with regard to a single constraint, it is therefore a *local* type of consistency. The more values can be detected to be inconsistent, the stronger the consistency level is. The strongest type of consistency is called *global consistency*, where a value is inconsistent iff it does not appear in any solution of the complete problem. A set of constraints is globally consistent iff it is satisfiable. Of course, with a propagator that achieves global consistency, the search procedure would find a solution without backtracking.

Between the two, different *levels of consistency* exist, that can be ordered according to their strength. Generally speaking, the higher the level of consistency is enforced, the fewer nodes leading to failure will be explored, resulting in a solution being found in a smaller number of nodes, but possibly at the expense of more costly reasoning. The trade-off that exists between the amount of reasoning and the amount of search has been well established [49, 50, 51, 35]. Different types of consistency include path-consistency, k-consistency, (i, j)-consistency, pairwise consistency, singleton arc-consistency, dual consistency, and variations thereof. See Bessière [5] for a survey.

In the rest of this dissertation, we will often be interested in the consistency of a set of constraints. Depending on the applications, checking for the satisfiability of a set of constraints cannot be achieved without a search procedure, the cost of which can be prohibitive in the said application. Instead, a propagator can be used on its own, independently from any search procedure, to detect the consistency of a set of constraints, by applying the procedure and checking for empty domains. Obviously, a propagator that achieves global consistency will accurately check whether a set of constraint is satisfiable, but weaker levels of consistency will only partially detect inconsistent sets of constraints (i.e. some sets of constraints that are actually not satisfiable will be detected to be consistent), but probably at a lower cost too. Note too that when not explicitly mentioning a particular level of consistency, the terms consistent and satisfiable might be used indifferently, by an abuse of language.

2.3 Explanations

An informal, but intuitive, definition of an explanation is that it is a statement intending to make something understandable, or that provides a justification for something. Nonetheless, it is hard to formally define an explanation, and, in Artificial Intelligence, this term can be found to have many different meanings. In Artificial Intelligence, explanations are associated with the reasoning process. We can identify two ways in which this can happen [126]: explanations are part of the reasoning process itself, whether they are used by it or are the result of it, or explanations serve to make the reasoning process understandable, transparent to the user.

In either case, the task of an explanation consists in *explaining an observation*. In the context of automated reasoning, an observation can have different meanings. It can mean explaining a value removal, a variable assignment, a solution, inconsistency; all of these being obviously equivalent. The way such an explanation can be given varies too. A *simple reasoning* can be presented that allows one to deduce the explained event. This is particularly relevant to rule-based systems, where, at its simplest form, a reasoning trace of the system is presented, or to any other system that proceeds with inference (see Sqalli and Freuder [127]

for an example on constraint satisfaction). This approach is more satisfactory as to what an explanation should be, but also less pragmatic in many cases. A more restricted view of an explanation consists in providing a *subset of elementary events*, such as a constraints, value removals, variable assignments, that, combined with each other, imply another elementary event that is being explained. The term explanation most commonly refers to this type of approach in the literature, and it is this view that will retain our attention throughout this dissertation. Note that, when needed by the application, these type of explanations can still be made user-friendly [79].

Finally, explanations can be taken to have a different meaning. Within this meaning, they deserve less to be called explanation, but the term is nevertheless widely used in this context, probably because similar theory and algorithms apply. We can refer to them as "constructive" or "corrective" explanations. This is the case where instead of simply explaining an event, an explanation proposes an action to counteract the event. This will typically involve restoring consistency, restoring back a value that has been ruled out, correct a variable assignment, etc., with these concepts being equivalent. Consistency-based diagnosis, which will be introduced later in this section, falls into this category. Moreover, the rest of this dissertation will concentrate mostly on this type of explanation in configuration problems.

2.3.1 A Formal Concept: Abduction

Logic offers a way to formalise the act of explaining, with the concept of abduction. This method of reasoning underlies, more or less directly, all kind of explanations. Furthermore, many formalisms related to explanations can ben seen as particular instances of abductive reasoning, as we shall see later.

Logical reasoning has traditionally[2] been divided into three kinds: deduction, induction and abduction. Deduction allows one to infer a conclusion from a valid hypothesis: I know that a holds, I know that $a \Rightarrow b$, I can deduce that b holds. Induction refers to the fact of inferring deductive rules from a finite set of observations: I observed that every time a holds, b holds too, therefore it is likely that

[2]Following the work of the American logician and philosopher C.S. Peirce (1839-1914)

$a \Rightarrow b$. Finally, abduction refers to the fact of guessing a hypothesis given an observation: b holds, I know that $a \Rightarrow b$, so it could be that a holds.

More specifically, abductive reasoning can be regarded as an approach to finding a *likely explanation* for an observation. Let F be a set of formulas known to be true in the domain we are representing, H a set of possible hypotheses, and let O be a set of new observations. $E \subseteq H$ is an explanation of O if:

$$F \cup E \models O \text{ and}$$
$$F \cup E \text{ is consistent.}$$

Abduction is not simply about finding one such E. Abduction has also been defined as "inference of the best possible explanation" [67], i.e. finding from a set of many possible explanations one that satisfies some optimality criteria that relate to its *explanatory power*. This can refer to some succinctness property or to some probabilistic consideration. Abductive reasoning constitutes the logical background behind the many forms of explanations, and underlies many tasks in AI, as it has been widely noted [39, 106, 81], and as we will show in the rest of this overview.

2.3.2 Explanations for Problem Solving

Explanations are used as a powerful tool to counteract common shortcomings of classic backtrack search algorithms in an attempt to reduce the amount of the search tree that is explored. The general idea is that explanations can be used to record dependencies between decisions as they are made (such as assigning a value to a variable) and conclusions (such as removing a value or detecting a failure), and that this can be taken advantage of by the search procedure. In this context, an explanation corresponds to a set of decisions that are sufficient to explain a conclusion. We present here some of the most important of such techniques, with the purpose of simply giving an intuition of the role of explanation in them, rather than giving an actual technical description of the different systems introduced here, which is out of the scope of our work.

Explanations for Search Algorithms

Explanation-based search algorithms, which we will review below, can improve common weaknesses of classic backtrack algorithms, by avoiding thrashing behaviour. Thrashing occurs when the search algorithm backtracks to a decision, typically the last decision, that is not related to the failure and thus repeatedly makes a proof of the same contradiction.

Dynamic backtracking (DBT) is an alternative way to explore the search space that limits thrashing, while still remaining complete [57]. The idea is to repair the current decisions instead of simply discarding the last one. In particular, any inference that has been made by decisions that are not involved in the contradiction, in particular those decisions made after the incriminated ones, is kept intact. More specifically, when the domain of a variable becomes empty, mac-dbt [80] computes an explanation for this contradiction, consisting of the union of explanations for the removal of each value in the domain in question. An explanation for the removal of value v from the domain of variable X is a subset of the decisions that explains $X \neq v$. It then removes the most recent of those decisions. The current state is repaired by removing all inference resulting from this decision. By always removing the last decision, it can be proved that completeness is guaranteed.

Thrashing can be eliminated altogether by recording all explanations for inconsistency whenever it occurs, in an approach called *nogood recording*. This has also been called learning, as it consists in making explicit any implicit (or implied) constraints. Suppose that during search, a subset of variables $X_1, \ldots, X_k$ is instantiated to values $v_1, \ldots, v_k$, and this assignment cannot be extended to any value for the next variable. An inconsistency arises, and we discover that $X_1 = v_1 \wedge \ldots \wedge X_k = v_k$ is inconsistent with the problem: it is a *nogood*. There is no point recording this nogood as the same state will never be considered again due to backtrack. However, if a subset of this nogood is also inconsistent, recording it, in form of an explicit constraint, prevents search from ever reconsidering a state containing this nogood. This approach is not practical as it suffers from exponential growth in the number of such nogoods, and practical implementations limit the nogoods that are actually kept to those that are most likely to be useful [32, 122].

ATMS

Even if originally described in procedural terms, Assumption-based Truth Maintenance System (ATMS) were one of the first popular systems that compute abductive explanations, with the initial purpose of explaining a system's behaviour and conclusions. De Kleer's ATMS [23, 113] is a system (i.e. a set of data structures) and a set of procedures that record dependencies between nodes and a subset of those nodes, called assumptions. Nodes correspond to decisions made by a search algorithm, and the ATMS is queried during search. For example, given a justification for a new node (the set of nodes that led to this node), the ATMS can provide a set of assumptions that lead to this node. The ATMS can be used for problem solving [24], and the link between the ATMS and common CSP solving methods, in particular the different types of consistency, has been noted by de Kleer [25].

In formal terms, each node is represented by a propositional symbol. A subset $A \subseteq P$ of the propositional symbols is called the set of assumptions. The justifications form a set Σ of Horn clauses, each justification being of the form $p \leftarrow p_1, \ldots, p_n$, meaning p derives from p_1, ..., and p_n. Suppose a new justification $q \leftarrow q_1, \ldots, q_n$ is added to Σ, the ATMS computes an explicit record of q dependency on the assumptions, i.e. a set of all justifications $q \leftarrow q'_1, \ldots, q'_n$ such that each q'_i belongs to A, $q'_1, \ldots, q'_n$ is consistent with Σ and $q'_1, \ldots, q'_n$ is minimal by inclusion. The ATMS also maintains a set of nogoods, defined as subsets of A that are, in conjunction with Σ, unsatisfiable.

Example 2.3.1. Suppose Σ contains the following justifications:

$$p \leftarrow a, b$$
$$p \leftarrow a, d$$
$$q \leftarrow a, c$$

where a, b, c, d are assumptions, and the following justification is added:

$$r \leftarrow p, q$$

the ATMS computes the justifications:

$$r \leftarrow a, b, c$$
$$r \leftarrow a, c, d$$

In terms of abduction, this corresponds to computing all abductive explanations of r given the theory Σ that contain only symbols from A, which are called *assumption-based* explanations. Here, $a \wedge b \wedge c$ is an explanation of r in the theory Σ. ▲

Even if not presented as such in the original paper [23], the link between ATMS and abductive reasoning has been much noted later [113, 81, 123]. The link between the ATMS technique and the theory of diagnosis is also well established [28].

2.3.3 Explanations for the User

Explanations have been used to introduce user interaction within Constraint Programming, particularly in order to assist a user while he is solving a CSP. Explanations can be used to present the decisions of the search procedure to the user [9], and, if needed, in a user-friendly form [79]. Jussien and Ouis [79] show how explanations can be presented in a way that provides understandable and exploitable information to the user, instead of involving low-level technical information understandable only by the developer of the constraint model. This can be particularly useful for debugging purposes, so as to help understand why a model does not behave as expected. In order to make explanations even easier to understand, Sqalli and Freuder [127] suggest to use inference-based solvers, where the type of inference needed to solve a problem is context-dependent, the argument being that inference is closer to human reasoning. Explanations can be used to reveal

the reasoning performed by such a solver, by converting each inference step to a user readable explanation. As an example, consider the classic Zebra puzzle, or any similar logic puzzle. A solver can find a solution very simply with inference. However, the solution actually provided to the user usually comprises the successive steps that allow him to build the solution. An explanation revealing the steps of the solver can automatically provide this type of justification.

As we can see, *how well* an explanation does explain becomes an important consideration. Previous research has considered what characterises a "good" explanation. For example, Friedrich [55] shows how explanations can be used to *explain a solution*. This amounts to explaining a specific variable assignment that is part of the solution. This is equivalent to computing a conflict, containing the negation of the variable value assignment. In his paper, Friedrich argues that not every conflict is suitable for explaining a particular value in a solution. He claims that some conflicts can result in *spurious explanations*, in that they can imply values on the other variables that are in contradiction with the actual solution [55]. In other words, a "well-founded explanation" must explain a particular variable assignment, part of a particular solution, in such a way that the explanation not only entails the variable assignment, but also entails restrictions on the other variables that are *compatible* with the given solution. He argues that such explanations are more likely to be understood by the user, and more likely to be accepted.

Other approaches have been proposed that attempt to be more "helpful" by relying on "corrective" explanations instead. This can be done by presenting users with partial consistent solutions [108], or advising on how to relax constraints in order to achieve consistency, using corrective explanations [101]. The benefit of presenting a set of relaxations rather than a conflict, in order to give users a better understanding of the space of possible solutions, has also been suggested, in very different terms, for recommender systems [99].

Diagnosis

In the model-based diagnosis theory [112, 27], the modelled system is made of components that might work normally or not, and of observable features. We model the behaviour of the components of the system on the assumption they work

correctly. The inconsistency arises when the actual observation of the behaviour of the system is not consistent with the way the system was meant to behave. The diagnosis problem involves identifying which of the components are behaving abnormally in such a way as to explain the observed behaviour and its discrepancy with the expected behaviour: this is a diagnosis. There can be more than one diagnosis. The most basic algorithm consists in computing all diagnoses.

The fundamental framework for diagnosis as set by Reiter [112] involves working on a system made of a set of *components*, linked to each other according to a *system description* (SD), which describes the *expected* or *correct* behaviour of the system, and assume a set of *observations* (OBS) is given. SD and OBS are given in the form of a set of first-order sentences, and COMPONENTS is a finite set of constants. Given a component $c \in$ COMPONENTS, an abnormality predicate $AB(c)$ is defined that is true when c is working abnormally.

A *diagnosis* for (SD, COMPONENTS, OBS) is a minimal set $\Delta \subseteq$ COMPONENTS such that

$$\text{SD} \cup \text{OBS} \cup \{\neg AB(c)/c \in \text{COMPONENTS} \setminus \Delta\}$$

is consistent. In other words, a diagnosis is a minimal set of components that have to be assumed to be abnormally functioning to explain the observation, i.e. for the observation to be consistent with the expected behaviour as described by SD. If the system is working correctly, the only diagnosis is $\Delta = \emptyset$. If the system is working incorrectly, then SD $\cup$ OBS $\cup \{\neg AB(c)/c \in$ COMPONENTS$\}$ is inconsistent, i.e. the observation contradicts the possibility that all components work correctly.

A *conflict set* for (SD, COMPONENTS, OBS) is a set of components $\{c_1, \ldots, c_n\}$ such that

$$\text{SD} \cup \text{OBS} \cup \{\neg AB(c_1), \ldots, \neg AB(c_n)\}$$

is inconsistent.

There can be of course more than one diagnosis, and Reiter [112] gives an algorithm to compute all *potential* diagnoses. Each diagnosis is a hypothesis, or a possible explanation, of what could be wrong. However, with an observation that is only partial (as is often the case in practice, typically for cost reasons), there will probably not be sufficient information to infer the actual diagnosis. Depending on the application, one could either decide to replace all possibly faulty components,

or carry extra work by making extra measurements to refine the hypotheses. In the second case, an extra work of the diagnostic system is to identify the exact diagnoses with a minimal number of extra measurements. This is a complex task, and is out of the scope of this presentation.

This type of diagnosis is often referred to as *consistency-based diagnosis*, and was initially associated with approaches where only the correct behaviour of the system is described. Though conceptually it can be seen as a way of providing an explanation for an observation, it does not correspond to the framework of abductive reasoning. An alternative and more restrictive approach is *abductive diagnosis* [106, 107], which offers a stronger definition of diagnosis. An abductive diagnosis does not simply have to be consistent with the observation, it has to actually entail the observation. This corresponds to an instance of abductive reasoning.

2.3.4 Computing Explanations

Little has been said so far in this overview about how explanations can be computed effectively. It is a general fact that computing explanations is associated with hard classes of complexity, and often with undecidability. Therefore, generating explanations presents a considerable algorithmic challenge. Algorithms tackle this complexity in various ways, but they fall roughly in two categories: *on the fly computation*, *post-event computation*.

On the fly computation pertains to explanations that are tied with the reasoning process. As we already mentioned, in these approaches, explanations consists in *keeping a trace* of decisions and the reasoning that led to them. The reader is referred to corresponding papers for technical details [80, 78, 117], which are out of the scope of this presentation.

Post-event computation relates to computing an explanation after the event that needs to be explained has occurred. The term *non-intrusive* computation is also used, as it will not require the solver to be adapted to offer explanation functionalities, but rather will involve a separate procedure, which might, or not, make use of the solver. QuickXPlain [74, 75] is an example of such an algorithm, which takes a propagation or solving procedure as a parameter, used to detect the consistency

of a set of constraints, and simply uses it to compute a minimal set, by inclusion, of conflicting constraints. The advantage of such a method is, as we said, that it does not rely on the ability to maintain explanations by the solver, which can be potentially very costly; this method can be more efficient in some applications where methods like the ATMS lead to high overhead. Additionally, being non-intrusive, the algorithm is clearly separated from the solver, which does not have to be changed. This makes for a wide range of applications, depending on the consistency detection that is used. In particular, there are good reasons for explanations specific to a propagation engine to be interesting: they can explain the behaviour of the propagation engine, they can be computed polynomially, etc. [74].

This procedure forms the basis, at least conceptually, of other non-intrusive algorithms. Amongst the work we already mentioned, Friedrich [55] presents a non-intrusive algorithm to compute well-founded explanations based on the same ideas, which, according to his claim, incurs acceptable additional cost with regard to classic explanation computation. O'Callaghan et al. [101] presents an algorithm that is an adaptation of QuickXPlain to compute corrective explanations, similar to maximal relaxations, instead of minimal conflicts.

Another instance of post-event computation is constituted by explanation enumeration algorithms. These are discussed at a more technical level in Section 3.3 of Chapter 3.

Consequence Finding

Consequence finding is a general term that refers to the task, in Artificial Intelligence, of deriving specific knowledge that is intensionally contained in a knowledge base (see Marquis [96] for a survey). Many notions of explanations introduced earlier in this chapter fit this description and could be seen as instances of consequence finding. However, consequence finding often amounts to computing implicates and implicants of a propositional base, and several algorithms exist to achieve that. An implicate of a propositional theory Σ is a clause c that is entailed by the theory, i.e. such that $\Sigma \models c$. An implicant of Σ is a term t (a conjunction of literals) that entails the theory, i.e. such that $t \models \Sigma$. If c is an implicate, and $c \models c'$, then c' is trivially an implicate too. Therefore, we are often interested

only in the most specific implicates, i.e. those that are not entailed by any other implicate: these are the prime implicates. Similarly, the most general implicants, called prime implicants, are the most interesting. If trivial implicates (containing $x \vee \neg x$, equivalent to true) and trivial implicants (containing $x \wedge \neg x$, equivalent to false) are excluded, prime implicates and implicants are set-wise minimal ones.

There are strong connections between prime implicant/prime implicates and explanations, as shall see in Chapter 5. De Kleer at al. [28] noted the connection between prime implicate and minimal conflicts sets in diagnosis and between prime implicant and minimal diagnoses. In particular, they note that diagnoses corresponding to prime implicants can suitably characterise all diagnoses, which is not always the case with minimal diagnoses (it is not true in all settings that the superset of a minimal diagnosis is a diagnosis itself). The connection between prime implicates and the ATMS has also been noted by Reiter and de Kleer [113] and Selman and Levesque [123]. Essentially, the ATMS task can be described as computing all prime implicates involving at most one literal that is not an assumption.

2.4 Knowledge Compilation

2.4.1 The Approach

In some applications, a complex reasoning task has to be achieved under a short time limit. This could involve solving a problem, adding constraints to a problem, checking the satisfiability of a set of constraints, computing an explanation, etc., and standard techniques to tackle these artificial intelligence tasks can usually not run under these time restrictions. *Knowledge compilation* provides an methodology to tackle this issue based on two assumptions: there is an underlying instance that does not change over time, and the same type of operations needs to be performed repeatedly over time. Under this setting, we can distinguish between two phases: an *on-line phase*, where those operations must take place under time constraints, and an *off-line phase*, where as much computation as needed can be performed, beforehand, on a particular problem instance.

The computation made during the off-line is called *compilation*. The result of

the compilation procedure is a structure that offers an alternative representation of the problem, a *compiled representation*, which will explicitly contain information that will be valuable for the on-line phase. To illustrate this with an overly simplistic example, suppose we want to quickly know the number of solutions of a problem. We could spend as much time as we want in an off-line phase computing all the solutions of the problem, then in an on-line phase, we could immediately return the number of solutions found. Compilation methods work well when, for a particular instance, the compiled representation is compact. This will allow on-line queries to be efficient as their running time will typically depend on the size of the structure they are operating on. Considering our example, suppose we want to know the number of solutions containing a given variable assignment. If we went through the list of all solutions and count the ones satisfying the requirement, it would be intractably long for any problem that has any substantial number of solutions. Therefore, we need a structure that can store this information in a much more compact way, while still allowing to easily answer this query.

In summary, compilation approaches allow for fast query answering in two ways. First, a separation is drawn between an off-line phase and an on-line phase, allowing as much computation as needed for the off-line phase, as long as the resulting structure is compact enough for the operations to be efficiently performed during the on-line phase. Second, because in a compilation approach we focus on a particular instance, by examining the actual size of the compiled structure for this instance, we can have a guarantee about the efficiency of answering on-line queries. When compiling a problem, we have to reach a compromise between the two contradictory requirements of saving time during the on-line operations and saving space taken by the compiled representation. The method chosen should involve the most compact representation that allow us to efficiently perform the required operations. The relationship between the compactness of a representation and the operations it allows have been formally and systematically studied by Darwiche and Marquis [22], and will be further discussed in Section 4.3, Chapter 4.

2.4.2 Compilation Strategies

The principle of compactly representing the complete solution space of a problem, as a compilation method, has been explored from various approaches. Binary Decision Diagrams (BDDs) [10] were first proposed to encode a boolean function as a directed acyclic graph. They have further been extended to Multivalued Decision Diagrams [82] for non-boolean domains. Alternatively, the solution space of a CSP can be represented by an automaton [134], and this representation can be used as a compilation strategy for configuration [1]. These approaches form the basis of many compilation methods, such as *dDNNF* [20], *Cluster trees* [34, 103], *tree-of-BDDS* [131]. Other approaches also exist to compactly represent all solutions of a problem using *synthesis trees* [139]. Finally, there are some methods concentrate on prime implicates rather than solutions [36, 97].

2.4.3 Binary Decision Diagrams

Ordered Binary Decision Diagrams (OBDDs) were introduced by Bryant [10], as a canonical representation of a boolean function. It was first applied to formal verification [11], but BDDs have been successfully applied to many other domains, including configuration [61].

A BDD is a simple representation of a boolean function as a rooted directed acyclic graph, consisting in decisions nodes and two types of terminal nodes, representing the value true and false. A decision node is labelled by a variable, and has two outgoing edges. One represents an assignment of the variable to true, the other to false. A path from the root node to a 1-node corresponds to a variable assignment for which the represented function is true.

Two additional properties are usually enforced in order to make BDD more compact and efficient. A BDD is *Ordered* (OBDD) if for any path from the root node to a terminal node, the variables appear in the same fixed order. This allows the OBDD representation to be canonical given a variable ordering. The BDD on Figure 2.8 is ordered. Choosing a good variable ordering is critical for the efficiency of the OBDD representation [45]. An OBDD is *Reduced* if *isomorphic* subgraphs have been *merged*, and decision nodes whose outgoing edges both point to the same node, which are *redundant* nodes, have been *eliminated*. For example,

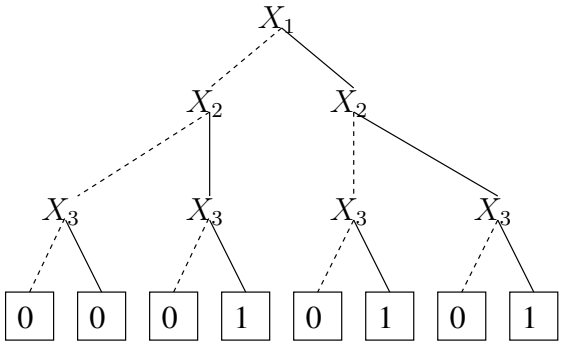
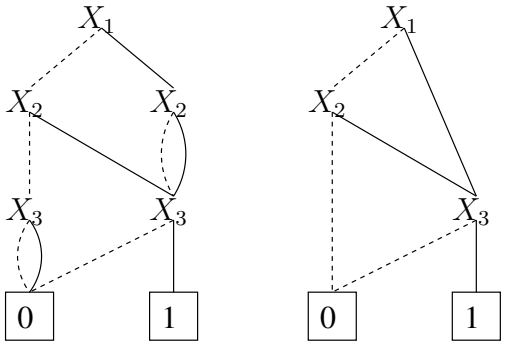


Figure 2.8: An example BDD for the function $X_1 = X_3 \vee X_2 = X_3$; dashed edges represent assignments to false, plain edges represent assignments to true

on Figure 2.8, all 0-nodes can been merged and so can all 1-nodes. Then, we can see that the last three nodes labelled by X_3 have the same successors, and can be merged too, resulting in the BDD at Figure 2.9(a). In this BDD, two nodes are redundant and can be removed, which yields the BDD at Figure 2.9(b). Most of the times only Reduced Ordered BDDs are considered, and the term BDD usually refers to Reduced Ordered BDDs.



(a) Isomorphic nodes merged

(b) Redundant nodes removed

Figure 2.9: The example BDD with reduction rules applied.

2.4.4 Automata

A generalisation to the case of multivalued CSPs has been independently provided by Vempaty [134], proposing the use of Finite State Automata, or simply

Automata. Vempaty notes that the solution set of a CSP can be expressed as a regular language, as, after having fixed an order on the variables of the CSP, every solution can be expressed by a unique string. He suggests that a Minimal Deterministic Finite state Automaton (MDFA) [69] that recognises this language can be used as a canonical representation of the CSP.

Definition 2.4.1 (Automaton)**.** A Deterministic Finite State Automaton (DFA) is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, with:

- Q a finite set of states,
- Σ, a set of symbols, the alphabet
- δ a transition function $Q \times \Sigma \to Q$
- $q_0 \in Q$ the initial state,
- $F \subseteq Q$ the final (or acceptance) states.

A DFA recognises the regular language defined in the following way.

Definition 2.4.2 (Recognised Language)**.** Given the transition function δ, we can define the function $\delta^* : Q \times \Sigma^* \to Q$ such that $\delta^*(q, \epsilon) = q$ and $\delta^*(q, a.w) = \delta^*(\delta(q, a), w)$. In other words, $\delta^*(q, w)$ is the state reached from q following the transition function by consuming w characters one by one. A word w is *recognised* (or accepted) by the DFA if $\delta^*(q_0, w) \in F$. The language recognised by the DFA is the set of words recognised by it.

A transition links state q to state q' with label a when $\delta(q, a) = q'$. An automaton can be represented by a labelled directed graph, where each node corresponds to a state, and each edge corresponds to a transition, labelled with the label of the transition. Determinism refers to the fact that a state has only one outgoing transition with a given label. Two states q_1, q_2 are said to be equivalent if they recognise the exact same words. Equivalently, they are equivalent if either both are final (i.e. they belong to F) or neither is final, and for each $a \in \Sigma$, $\delta(q_1, a)$ and $\delta(q_2, a)$ are equivalent. A DFA is *minimal* (MDFA) if no two states are equivalent. Minimisation is the process of merging equivalent states, by replacing them with a new

state having the same outgoing transitions and the union of incoming transitions. We usually only consider MDFAs, and use the simple term of automaton to refer to MDFAs.

An automaton gives a compact way of representing the set of solutions of a CSP. Informally, an automaton can be seen as a representation of the part of the search tree that leads to all the solutions of the problem, on which minimisation reduces the size. From a formal language theory point of view, each word recognised by the automaton corresponds to a solution of the problem, given that a particular ordering on the variables has been fixed in advance. Note too that this automaton only recognises words of the same length (corresponding to the number of variables in the problem). Such an automaton presents two important properties: it is acyclic, and, more specifically, it is organised in layers, or levels. Additionally, it only has only one final state.

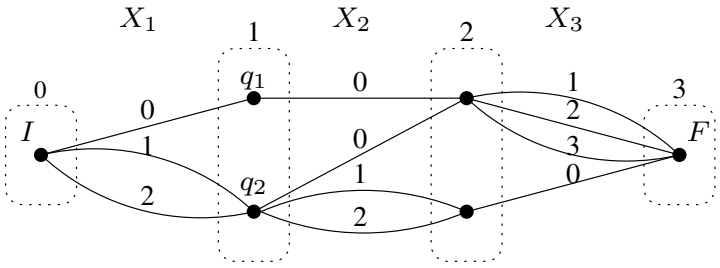


Figure 2.10: An example automaton defined on three variables.

The incoming and outgoing transitions of a state q are denoted by $in(q)$ and $out(q)$, respectively. The origin and destination state of a transition t are denoted by $in(t)$ and $out(t)$, respectively. The initial and the final states (or the source and the sink) are denoted by I and F, respectively. The level of a state q is the length of the words from I to q. The set of all states of level i is denoted $Q(i)$. The level of a transition t is the level of $out(t)$. Each level greater than 0 corresponds to a variable of the problem. Thus, each transition t provides a support for the instantiation of the variable of its level with the value labelling t.

Example 2.4.1. Figure 2.10 shows the automaton for a problem on three variables X_1, X_2 and X_3. This problem has 13 solutions, corresponding to the following words: 001, 002, 003, 101, 102, 103, 201, 202, 203, 110, 120, 210, 220. I is

the only state of level 0, q_1 and q_2 are the two states of level 1, there are two transitions between I and q_2, supporting the instantiations $X_1 = 1$ or $X_1 = 2$. Also, looking at the different transitions of level 2, we can see that from state q_1, only the instantiation $X_2 = 0$ is supported (by the unique outgoing transition of q_1), whereas from state q_2, $X_1 = 0$ or $X_1 = 1$ or $X_1 = 2$ is supported. Therefore, looking at the respective ingoing transitions of states q_1 and q_2, we can deduce that $X_1 = 0$ entails $X_2 = 0$, whereas $X_1 \in \{1, 2\}$ does not entail any restriction to the domain of X_2. Similarly, forbidding value 0 from X_3 entails $X_2 = 0$. ▲

When restricted to boolean domains, MDFAs are equivalent to BDDs with some syntactic differences. First, only the removal of isomorphic nodes reduction rule (equivalent to minimisation) is applied, but redundant nodes are kept. One has to note that keeping redundant nodes does only increase the size of the BDD by a polynomial factor, and that on non-boolean domains, redundant nodes are less likely to occur. Second, the 0-node and all edges leading to it is not explicitly represented. Again, this only reduces the size of the BDD by a polynomial factor.

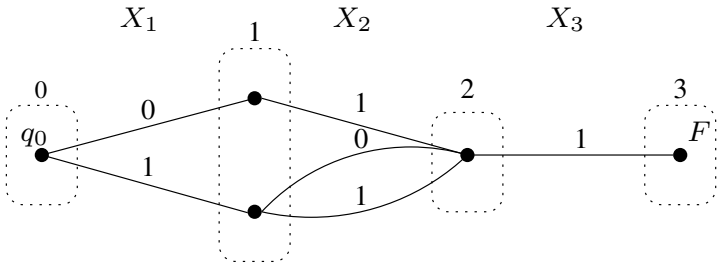


Figure 2.11: An automaton representing the BDD of Figure 2.9

The ability of MDFAs to represent CPSs and their power for supporting various types of queries relevant to configuration has been noted and exploited by Amilhastre et al. [1]. Some of the ideas introduced in this paper, in particular how MDFAs can be used to maintain global consistency, have been exploited too in a very different context to design a propagation algorithm for a regular language membership global constraint [104].

2.4.5 Other Approaches

As we mentioned in the beginning of this section, the purpose of compilation methods is to explicitly extract some knowledge contained in a theory that can be subsequently accessed by some operations in order for them to run more efficiently. All the approaches we presented so far focused on the solutions of the problem, or the solutions of parts of the problem and the connection between them. However, there has been some approaches that use prime implicates as the basic extracted knowledge: Theory Prime Implicates [97], Tractable Databases [36], Possible Conflicts [77]. These approaches are further discussed in Section 5.7, Chapter 5.

2.5 Configuration

2.5.1 What is Configuration?

Configuration is an application area that has been successful for artificial intelligence. The need for automated tools for configuration came from a widely observed trend that in many industrial sectors, there is a move from mass production to mass customisation [46, 130, 119, 76]. To allow for highly customised products while still benefitting from savings gained by increased levels of production, such products are made from a number of different components, that are themselves made of subcomponents, until some are down to only standard elementary parts. Even with a fixed number of elementary parts, such an organisation offers a very important flexibility in terms of design. When such a flexibility is allowed, it becomes very complex to configure a product, whether automatically (given the latitude in manufacturing a product, try to design one that optimise some requirements such as costs or that is tailored for a specific application) or with user interaction (allow the user to interactively configure his ideal product). With such a complexity, a software tool is needed to offer support for automated configuration. There are many commercial tools that offer configuration functionalities,

such as ILOG Configurator[3], Tacton Configurator[4], Selectica ACE Enterprise[5], Baan SalesPlus[6], Configit[7].

2.5.2 Approaches to Configuration

Product configuration involves two major tasks [119]: one consists in defining a modelling language that makes it easy to specify and maintain a formal product model, and one consists in developing a decision support tool that efficiently guides users to desirable product configurations. There are several ways to carry out the modelling part [130, 119]. These can be categorised in two main types, using rule-based reasoning or model-based reasoning.

Rule-based reasoning is used by expert systems, where the behaviour of the system is described by production rules. Production rules consists of a precondition and an action. If, in the current state of the system, the precondition is satisfied, the rule is triggered and the action is executed, modifying the current state. Rules are executed in chaining, building a solution as they are executed. Rule-based approaches suffer from a big drawback, due to the complexity such a system very quickly reaches. This implies that they are hard to debug, when one tries to understand why the knowledge base does not behave as expected or desired, and that they are hard to maintain, when one needs to update rules to reflect a change in the product specification [14] (incidentally, Clancey [14] proposes to use explanation techniques to circumvent this problem).

Model-based approaches solve this issue by making a clear separation between the knowledge of the system and how the knowledge is used. Concretely, when dealing with a configuration problem, first, we must decide how to *represent* the problem, then, we need *algorithms* that perform the desired task based on the problem representation.

The first proposal for a constraint-based generic model for configuration was presented by Mittal and Frayman [100]. They presented a generic definition of a

[3] www.ilog.com
[4] www.tacton.com
[5] www.selectica.com
[6] www.baan.com
[7] www.configit-software.com

configuration problem; it is often this description that is, implicitly, taken into account when referring to configuration. More precisely, in configuration, we focus on a system that is made of a fixed set of components. A component is described by a set of properties, and has a set of ports to connect it to other components. For each of these ports, a constraint describes which components can be connected to it and how. There can also be some additional structural constraints. This description is fixed over time. In particular, during configuration, no new component type can be created, and the way components can be connected cannot be modified. Secondly, a description is given of what a desired, or possibly optimal, configuration is. The configuration task then involves finding configurations satisfying the desires or detecting inconsistencies in the requirements. A solution defines a list of components and a description of the structure of the product. As we can see, this can be naturally described using a CSP. Generally, elementary component types are represented by variables, taking values corresponding to possible components, complex components are described as large arity table constraints holding on the variables corresponding to its subcomponents, and local constraints of small arity describe the connections between components.[8]

2.5.3 The Configuration Task

Consider a configuration problem where the artefact being configured is a product. The model fully specifies a *catalog* of all possible (or feasible) products, without explicitly representing it. In fact, with most configuration problems, this catalog would contain an extremely large number of entries. Typically, a set of options is available to the user, who can state some choices or requirements, thus restricting the number of entries that are compatible with her requirements. A *configurator* is a software tool that assists the user in making choices, by guiding her in this large set of possibilities. Specifically, the fundamental capabilities a configurator must be able to provide are [76, 119, 100, 54, 48, 88, 65]:

- show the *consequences* of the current user choices, in order to avoid future conflicts;

[8] Clib provides a configuration benchmarks library: `http://www.itu.dk/research/cla/externals/clib/`

- suggest a product or several products that satisfy *all* the user requirements;
- suggest a product or several products satisfying all user requirements that *optimise* a set of *preferences*;
- *explain* inconsistency when it failed to find any such product;
- suggest a product that satisfies a *maximum number* of the user requirements.

Example 2.5.1 (Bicycle configuration)**.** In order to clarify these ideas, consider a concrete example of a bicycle configuration. A bicycle is made of several components: the frame, the wheels, the different elements of the drivetrain, etc. A wheel is itself made of a rim, a hub, etc. The drivetrain might comprise the cogset, the crankset, the chain, and so on. Not any component can fit with any other: a single-speed bike must have a frame with horizontal dropouts, the frame has only provision for certain type of brakes, the fork have only clearance for tyres up to a certain width, etc.

A user might state that she wants a bicycle with hub gears, which limits the types of frame, with either roller brakes or disc brakes, which limits the types of the frame or the wheels, etc. She might prefer some colours too, and it is very likely that the manufacturer does not offer all of his models in any colour. These are not very restrictive requirements, and the user is very likely to feel overwhelmed by the amount of choice she is given, and at the same time frustrated she cannot find any bicycle she likes. Sometimes, despite her apparently not very restrictive requirements, no bicycle at all exists. She very quickly finds it tedious to constantly browse through the catalog (especially when it contains, say, over 10^6 entries) as a result of setting her requirements, and feels this process should be automated. This is the task of the configurator. ▲

2.6 A Synthesis

We presented the main concepts that are relevant to our work in this dissertation. Concluding this overview, we show how these concepts are related to each other, thus defining the setting to which the work in this dissertation belongs.

2.6.1 Configuration and Compilation

A configuration problem can be trivially modelled as a CSP. The only refinement to the general definition of a CSP is that we partition the set of constraints into two sets: the background constraints and the user constraints. This partition reflects the decomposition previously mentioned between modelling and representing the problem, and operating on the it. The *background constraints* model the system: they specify the components and express the relationship between them. They define what a *feasible* product is; they *cannot be violated*. For the user, the basic configuration operation involves assigning variables, corresponding to options, to values. Essentially, she is in charge of choosing values for variables, and the configurator's role is to assist her in that task rather than actually solve the problem. These user choices can be modelled with unary constraints, referred to as *user constraints*. User constraints *can be added and relaxed interactively*. Restricting user constraints to unary constraints has been the common practice in the literature [119, 1, 48, 101], and indeed in commercial configurators.

Even if constraint satisfaction techniques have been applied from a practical point of view to configuration [46, 88], compilation techniques apply particularly well to configuration. When choosing how to represent a configuration problem, a compiled representation is indeed a very good candidate. In fact, many compilation techniques have been developed specifically for configuration, such as BDDs [61], Automata [1], Cluster Trees [103].

Concretely, as the background constraints are immutable, it is worth spending offline effort to compile them to a compact representation. This defines the *offline phase*. User constraints on the other hand are dynamic and transient in essence, and, in an interactive context, during the *online phase*, the user will require quick response times. We thus have a typical cycle, during configuration, where an algorithm must operate efficiently on the compiled representation of the background constraints, while taking into account the user constraints, and provide a quick answer. For example, a configurator should almost instantly indicate whether the background restricted by the current user constraints is consistent or not.

The structure of configuration problems in a component and subcomponent hierarchy has a very interesting impact on the structure of the constraint model.

Indeed, it has been well-noted in the literature that such problems are *structured* problems, as defined in Section 2.2.2. In those problems, variables are loosely connected to each other. More precisely, in the constraint graph of the constraint model, nodes that are highly connected to each other occur only at a local level, forming a cluster. A variable will likely occur only in a few clusters, and there tends to be few constraints linking different cluster. Those clusters correspond to components of the configuration problem, and they tend to be structured close to a tree hierarchy, reflecting the component/subcomponent hierarchy. The constraint graph of a configuration problem will likely have a *low tree-width*. This property is successfully exploited by many compilation techniques (synthesis trees [139], dDNNF [20], Tree-Driven Automata [43], DDGs [42], AOMDDs [98],Cluster Trees [103], Tree-of-BDDs [131]). These compilation techniques produce compiled structures the size of which directly depends on the tree-width of the problem: the smaller its tree-width is, the more compactly it can be compiled.

These facts make configuration problems particularly good candidates for applying compilation techniques.

2.6.2 Configuration and Explanations

As we saw in Section 2.5.3, the fundamental way a configurator assists a user in making choices is by providing *explanations* of the reasoning the system can make on the current choices. Explanation is indeed a concept that is intrinsically linked to interactive configuration [48, 1, 9], and many of the explanation techniques we mentioned in this chapter have been applied to configuration [65].

In the setting we described, the elementary events are the user constraints, typically corresponding to user assignments. An explanation then consists of a set of constraints, belonging to the original problem or from the user constraints, that is sufficient to explain another constraint, such as the removal of a value or the assignment of a variable to a value, etc.

In configuration, explanations have to play a extra role compared to what they do in other applications we presented in this chapter. A user is often interested not only in understanding that some choices are incompatible, but also in having choices recommended to her. The relevance of these suggestions is very important

if they are to be accepted by the user. There has been little work in the literature on proposing *constructive* explanations for configuration. For example, Freuder and O'Sullivan [52] introduces the concept of tradeoff, which is a restricted definition of an exclusion set, which helps a user relax his requirements in order to find a feasible product. In this dissertation, we define such kinds of explanations, and provide algorithms that efficiently compute them by using compiled representations.

In this dissertation, we use the notions of *relaxation*, *exclusion set* and *conflict*, similar to the notions introduced by Junker [74, 75], O'Callaghan et al. [101]. They are formally presented in Section 3.2.3, Chapter 3.

2.6.3 Configuration vs. Diagnosis

The configuration problem and the diagnosis problem have very close connections. Just as explanations are intimately associated with configuration, diagnoses themselves are a form of explanations. More precisely, a diagnosis corresponds to a minimal exclusion set, and a conflict set corresponds to a minimal conflict. In fact, the diagnosis framework is general enough that it can be adapted and applied to configuration. Felfernig et al. [44] defines diagnosis in configuration, and shows how to use a conflict-based approach to debug a configuration knowledge base. The question that naturally arises then is what differentiates the two areas.

From an application point of view, there is a major difference in the role of the user, and specifically in the direction in which he acts. In diagnoses, the purpose is to identify faulty components. Once a correct diagnosis has been identified, the purpose is to repair faulty components, after which new observations are made, which lead to a consistent system. In configuration, this would amount to correcting the background constraints to make them consistent with the user constraints, effectively changing the product to make it match the user's desires. This is of course not what is done in configuration, where the user tries instead to modify his requirements to match the possibilities.

Furthermore, as mentioned earlier, a fundamental functionality of a diagnostic engine is to help identify, amongst possible diagnoses (hypotheses), one that is *correct*. From a theoretical point of view, this raises an important point. When

new observations are added, the current set of diagnosis changes. In particular, some diagnoses (or indeed all diagnoses) that initially explained the observation become invalid in light of the new observation. This fact that new information reduces the set of what could initially be inferred means that non-monotonic logic are needed to model diagnostic reasoning. This non-monotonicity reasoning is a core aspect of the diagnostic reasoning, but is irrelevant to configuration.

On a similar aspect, there is a parallel between diagnosis and configuration concerning the choice of the right element from a set of diagnosis or relaxations. As just mentioned, in diagnosis we need to find a correct diagnosis. To this end, different diagnoses can be ranked in terms of likelihood (this hypothesis is more likely than this other), involving some kind of probabilistic reasoning. For example, single fault diagnoses (i.e. diagnoses of single cardinality) are much more likely than multiple fault diagnoses, on the basis that it is less likely that two components will fail at the same time. In configuration, where no notion of "correct" relaxation exists, there is still the concept of acceptance by the user, where we want to maximise the probability of acceptance.

Another difference between the two frameworks concerns the modelling capacity of each. In configuration, we can model very different types of systems; indeed, any system that can modelled by CSPs, although in practice, we saw that configuration problems present some typical structural features. In the model-based diagnosis theory, the modelled system is made of components that might work normally or not, and of observable features. We model the correct behaviour of the components of the system. The inconsistency arises when the actual observation of the behaviour of the system is not consistent with the way the system was meant to behave. The diagnosis problem consists in identifying which of the components are behaving abnormally in such a way to explain the observed behaviour and its discrepancy with the expected behaviour. In model-based diagnosis, we thus model a quite specific type of problem.

On a secondary matter, there are some issues in diagnosis that are irrelevant to configuration. In diagnosis, the design of a system is a core concern. Indeed, we often want to design systems that are easily diagnosable (i.e. such that, for any abnormal behaviour, it will be possible to determine a unique diagnosis or few diagnoses), while still minimising the number of sensors in the system (typically for

cost reasons). Furthermore, when multiple diagnoses are possible, most probably only one will be correct, and further action has to be taken to identify it or at least filter out some candidate diagnoses (for example by complementing the observations with further measurements, and carry out those measurements in such a way that the number of diagnoses will indeed decrease). These are hard problems, and is an issue that is irrelevant in the context of configuration problems. It would not make sense to design a system with the concern of minimising the number of possible explanations in case of inconsistency (and typically the system preexists its modelling as a configuration problem); rather, it is the configurator's task to assist the user when inconsistency arises, in particular when many possible explanations exist.

Chapter 3

Representative Sets of Explanations: Partial Explanation Enumeration

Summary. *We propose the notion of a representative set of explanations. A representative set of explanations in this context means that every constraint that can be satisfied is shown in a relaxation and every constraint that must be excluded is shown in an exclusion set. We study the complexity of enumerating explanations in general, and of enumerating a representative set of explanations in particular. We present an algorithm for computing a minimal representative set of explanations, and demonstrate its performance on a variety of random and real-world problem instances.*

3.1 Introduction

We consider a configuration tool where a user can specify preferences for options. These preferences are expressed as constraints. When preferences conflict, we want to help the user find which preferences to relax. In an iterative process, the user might relax constraints until at least one consistent solution is found. Alternatively, the user might prefer to select a solution from a list of solutions that partially satisfy the user's constraints. It would be good to categorise solutions according to which constraints are satisfied/violated, and the benefits of that have

been shown in earlier work [108]. However, this requires that they are in some way representative.

Most current approaches to explanation generation in constraint-based settings are based on the notion of a (set-wise) minimal set of unsatisfiable constraints, also known as a minimal conflict set of constraints. However, a minimal conflict does not necessarily give an intuitive explanation, in that many users will want to be shown which subsets of their constraints they can satisfy and which they cannot satisfy. Furthermore, we argue that users need more than one explanation in order to avoid drawing false conclusions. It has also been shown, as already presented in Chapter 2, that minimal conflict-based explanations can also be spurious and misleading, when it comes to explaining a specific solution [55].

Computing a single explanation – a relaxation or a conflict – as well as not being very helpful, it is (almost) not challenging either (at least, it is no more challenging than checking for consistency or finding a solution to a problem). Computing all relaxations and/or all conflicts of a problem provides, in principle, all the knowledge that is required for generating more informative explanations. The obvious pitfalls of such a naive strategy though would be twofold. From a usability point of view, of course, an exhaustive list of all possible explanations without any kind of synthesis has little value. From a computational point of view, even accepting that a refined result must be given, enumerating all the explanations would most of the times be an expensive, if not intractable, step.

The strategy we propose in this chapter lies somewhere in the middle ground: provide a partial set of the relaxations of a problem, chosen in such a way that it is both compact and gives a good hint of the full picture, and computationally aim at minimising the number of explanations that are generated against what is actually required. Such sets will be referred to as *representative sets of explanations*. Those sets are exponentially more compact than those found using common approaches from the literature based on finding all minimal conflicts.

3.2 Definitions

3.2.1 Configuration Problem

The working assumption throughout this dissertation is that we are working on configuration problems that are solved in an interactive manner.

A configuration problem is denoted as $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{B}, \mathcal{U} \rangle$, where $\mathcal{B}$ is the set of background constraints, and $\mathcal{U}$ is the set of user constraints. In a configuration problem, a user tries to find a preferred solution to $\langle \mathcal{X}, \mathcal{D}, \mathcal{B} \rangle$ by finding a solution to $\langle \mathcal{X}, \mathcal{D}, \mathcal{B} \cup \mathcal{U} \rangle$. The configuration problem is said to be inconsistent when the CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{B} \cup \mathcal{U} \rangle$ has no solution. Often, we will simply refer to a configuration problem as $\langle \mathcal{B}, \mathcal{U} \rangle$, where, depending on the context, the knowledge of the variables and domains is either not ambiguous or is irrelevant. We assume that the set of background constraints, $\mathcal{B}$, is always consistent.

3.2.2 Consistency Oracle

Throughout this chapter, we will rely on a consistency oracle, Π, being provided, thus following, and simplifying, the formalism presented by Junker [74]. Π is simply a property that, given a problem $\langle \mathcal{B}, \mathcal{U} \rangle$, holds on a set $C \subseteq \mathcal{U}$ of constraints. It should be sound, verifying that $\mathcal{B} \cup C$ is satisfiable $\Rightarrow \Pi$ holds on C. This allows us not to specify explicitly how a set of constraints is detected to be consistent or not, but instead, we will always assume we have some way to detect that in constant time, in particular for our complexity results. Where ambiguity needs to be avoided, a set that is consistent according to Π will be said to be Π-consistent. Concretely, we might consider complete consistency, which can be checked by actually looking for a solution, or, more reasonably, by relying on a compiled representation. Alternatively, we could consider local consistency properties as a way to polynomially approximate satisfiability.

If a set of constraints does not admit a solution, one or several constraints must be excluded in order to recover consistency. This is where explanations become important.

3.2.3 Explanations

Throughout the literature, different standard types of explanations have emerged. We introduce here the most fundamental ones, which form the basis of what we are interested in. We will use the following running example in order to illustration these definitions.

Example 3.2.1. Consider a simple car configuration problem, based on an example presented by Junker, 2004 [75], with a given set of options. Note that the Boolean variable $x_i \in \{0, 1\}$ indicates whether option i is selected or not.

Option	Selector	Cost
Roof Rack	x_2	$k_2 = 500$
Convertible	x_3	$k_3 = 500$
CD Player	x_4	$k_4 = 500$
Leather Seats	x_5	$k_5 = 2600$

Assume that the technical constraints of the configuration problem forbid convertible cars having roof racks. This problem can be modelled with a straightforward set of background constraints, as follows. Let variable x_1 represent the total cost. We have one constraint maintaining this total cost, i.e. stating that $x_1 = \sum_{i=2}^{5}(x_i \cdot k_i)$ and one constraint enforcing the incompatibility between roof racks and convertible cars, i.e. stating that $x_2 + x_3 \leq 1$.

Suppose the user decides on the following user constraints.

Constraint	Semantics
c_1	$x_1 \leq 3000$
c_2	$x_2 = 1$
c_3	$x_3 = 1$
c_4	$x_4 = 1$
c_5	$x_5 = 1$

It is clear that not all user constraints can be satisfied at the same time. ▲

Definition 3.2.1 (Minimal Conflict)**.** Given a configuration problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, a subset C of $\mathcal{U}$ is a *conflict* of $\mathcal{P}$ if $\mathcal{B} \cup C$ is inconsistent. The conflict C is a *minimal conflict* if $\forall C' \subseteq \mathcal{U} / C' \subseteq C$, $\mathcal{B} \cup C'$ is inconsistent iff $C = C'$.

Example 3.2.2. Consider the problem at Example 3.2.1. All five constraints cannot be satisfied at the same time. Because of the technical background constraints, c_2 and c_3 form a conflict. Note that, given the budget constraint, if the user selects option c_5, it is not possible to have any of the options c_2, c_3, c_4.

Specifically, the minimal conflicts for this example are presented in the following table.

Table 3.1: The set of conflicts for the over-constrained problem presented in Example 3.2.1

	Constraints					
	c_1	c_2	c_3	c_4	c_5	Conflict
I	×	✓	✓	×	×	$\{c_2, c_3\}$
II	✓	✓	×	×	✓	$\{c_1, c_2, c_5\}$
III	✓	×	✓	×	✓	$\{c_1, c_3, c_5\}$
IV	✓	×	×	✓	✓	$\{c_1, c_4, c_5\}$

▲

Definition 3.2.2 (Maximal Relaxation)**.** Given a configuration problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, a subset R of $\mathcal{U}$ is a *relaxation* of $\mathcal{P}$ if $\mathcal{B} \cup R$ is consistent. The relaxation R is a *maximal relaxation* if $\forall R' \subseteq \mathcal{U}/R \subseteq R'$, $\mathcal{B} \cup R'$ is consistent iff $R = R'$.

Intuitively, a maximal relaxation defines a maximal set of constraints that the user can satisfy at the same time.

We can also define the complementary notion of minimal exclusion set. For the sake of clarity, it might be sometimes more useful to refer to this notion instead.

Definition 3.2.3 (Minimal Exclusion Set of Constraints)**.** Given a configuration problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, and a (maximal) relaxation $R \subseteq \mathcal{U}$ of $\mathcal{P}$, we define $E \stackrel{\text{def}}{=} \mathcal{U} \setminus R$ to be a (minimal) exclusion set.

Intuitively, a minimal exclusion set shows the user a minimal set of constraints he must *remove* in order to recover consistency.

Example 3.2.3. The maximal relaxations, and the associated minimal exclusion sets, of the problem at Example 3.2.1 are shown in the following table. I, II

correspond to a choice of options free from any budget limit. III, IV, V correspond to the scenario where the user decides to respect his budget limit.

Table 3.2: The set of relaxations and exclusion sets for the over-constrained problem presented in Example 3.2.1

	Constraints						
	c_1	c_2	c_3	c_4	c_5	Relaxation	Exclusion Set
I	×	×	✓	✓	✓	$\{c_3, c_4, c_5\}$	$\{c_1, c_2\}$
II	×	✓	×	✓	✓	$\{c_2, c_4, c_5\}$	$\{c_1, c_3\}$
III	✓	×	✓	✓	×	$\{c_1, c_3, c_4\}$	$\{c_2, c_5\}$
IV	✓	✓	×	✓	×	$\{c_1, c_2, c_4\}$	$\{c_3, c_5\}$
V	✓	×	×	×	✓	$\{c_1, c_5\}$	$\{c_2, c_3, c_4\}$

▲

The notion of minimal conflicts is dual to the notion of maximal relaxations. It is somehow more "explanatory" but less "constructive". More specifically, a minimal conflict shows the user a minimal set of constraints that it will never be possible to satisfy at the same time, thus showing one cause for inconsistency. Therefore, it gives a more precise explanation of the inconsistency. However, it is not necessarily enough to break *one* minimal conflict (by definition, by relaxing any one of its constraints) to recover consistency: indeed, *all* minimal conflicts have to be broken to recover consistency. So while a minimal conflict shows one cause (amongst potentially many) of inconsistency, a minimal exclusion set shows one way to recover from this inconsistency. Additionally, a conflict might involve some technical constraints of the problem, and can require more technical knowledge from the user in order to be really understood. On the other hand, a minimal exclusion set is a ready-made suggestion that the user can decide to accept or reject.

Example 3.2.4. Consider the conflicts shown at Example 3.2.2. As explanations, these conflicts are sufficient to explain, using a subset of the user's constraints, why all constraints cannot be satisfied simultaneously. However, it is not necessarily sufficient to remove one of the constraints in a minimal conflict, thus eliminating the conflict, to regain consistency.

Consider, for example, what happens if we present $\{c_1, c_2, c_5\}$ as an explanation for why the set of constraints $\{c_1, \ldots, c_5\}$ is not satisfiable. The user would be mistaken in thinking that simply breaking this conflict by removing, say, constraint c_2 is enough to recover consistency. It is not, because $\{c_1, c_3, c_5\}$ is also a conflict. Similarly, relaxing c_5 from $\{c_1, c_2, c_5\}$ would not have been enough because $\{c_2, c_3\}$ is a conflict. Minimal conflicts only explain why a set of constraints is inconsistent.

In order to recover consistency all minimal conflicts must be eliminated by relaxing a set of constraints that form a hitting set of the conflicts, i.e. an exclusion set, as shown at Example 3.2.3. For example, suppose the user is presented with the exclusion sets $\{c_1, c_2\}$ and $\{c_2, c_5\}$, as two ways to obtain a feasible car by giving up as few requirements as possible. The user can then decide which compromise seems best to him: have leather seats, or stay under the budget limit. ▲

Remark. In the above definitions, we did not make any mention to the level of consistency. If a given consistency oracle Π is specified, the explanations can be referred to as (maximal) Π-relaxations, (minimal) Π-exclusion sets and (minimal) Π-conflicts. When a weaker consistency oracle is used, it has to be noted that explanations have a lower quality. Minimal conflicts are not as small, and fewer of them might be detected, while, more importantly, maximal relaxations might be bigger too, which means they might actually not lead to a consistent problem. The setting in this dissertation being that we rely on a compiled representation to check complete consistency (see Section 4.2 of Chapter 4 for more details) ensures that this will never be an issue.

Definition 3.2.4 (Sets of all explanations)**.** Given a configuration problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ we define:

- $\mathcal{R}$ as the set of all the maximal relaxations of $\mathcal{P}$;
- $\mathcal{E}$ as the set of all the minimal exclusion sets of $\mathcal{P}$;
- $\mathcal{C}$ as the set of all the minimal conflicts of $\mathcal{P}$.

Of course, a relaxation exists (i.e $\mathcal{R} \neq \emptyset$) if and only if $\mathcal{B}$ is consistent. However, as noted earlier, we will always assume $\mathcal{B}$ is consistent. On the other hand,

if $\mathcal{P}$ is consistent, then there is only one maximal relaxation, $\mathcal{U}$, and $\mathcal{R} = \{\mathcal{U}\}$. In the non-trivial case where $\mathcal{P}$ is inconsistent, a relaxation R is a proper subset of $\mathcal{U}$. As far as conflicts are concerned, the trivial cases are inverted: $\mathcal{C} = \emptyset \Leftrightarrow \mathcal{P}$ is consistent, and $\mathcal{C} = \{\emptyset\} \Leftrightarrow \mathcal{B}$ is inconsistent.

Note too that every two distinct maximal relaxations are set-wise incomparable, i.e. for all $R, R' \in \mathcal{R}$, $R \neq R' \Rightarrow R \not\subseteq R'$ and $R' \not\subseteq R$. Similarly, all pairs of minimal exclusion sets in $\mathcal{E}$ and all pairs of minimal conflicts in $\mathcal{C}$ are set-wise incomparable. The set-wise optimality of explanations is required for various reasons. First, it helps rule out the trivial cases previously outlined. Additionally, optimal sets (maximal relaxations and minimal conflicts) have the property of covering, and thus characterising, all possible explanations. For example, a set of constraints is inconsistent iff it is a superset of at least one minimal conflict. From a user point of view, a smaller exclusion set is preferable as it implies giving up on fewer user constraints. Similarly a smaller conflict gives more precise information on what user constraints are incompatible with each other. Finally, from a computational point of view, set-wise optimal sets, as opposed to sets of maximum cardinality, as much easier to compute, as they allow for greedy optimisation, by adding user constraints as long as consistency is maintained to compute maximal relaxations, or by removing user constraints as long as inconsistency is present, to compute minimal conflicts.

The incomparability of optimal explanations (maximal relaxations or minimal conflicts) implies naturally the following result.

Proposition 1 (Worst-case Number of Explanations)**.** *Given a inconsistent problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, with $|\mathcal{U}| = n$, the worst-case number of maximal relaxations/minimal exclusion sets/minimal conflicts is $\binom{n}{\lfloor n/2 \rfloor}$.*

Proof. Immediate from Sperner's Theorem [135]. □

Throughout the examples, we mentioned the relationship that links maximal relaxations and minimal conflicts. Let us finish this section with a formal presentation of this relationship. We recall that, for a given collection of sets $\mathcal{X} = \{X_1, \ldots, X_k\}$, with $X_i \subseteq U$, a set $H \subseteq U$ is a *hitting set* of $\mathcal{X}$ if for each $X_i \in \mathcal{X}$, $X_i \cap H \neq \emptyset$. H is a *minimal hitting set* if for each $H' \subseteq H$, H' is a

hitting set of $\mathcal{X}$ only if $H' = H$. The following property holding on the set of maximal relaxations is a classic result from the literature [112, 27], [1, 60, 3, 7].

Proposition 2 (Hitting Set Relationship)**.** *Let $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ be an inconsistent problem. The following two properties hold:*

- *For each $C \in \mathcal{C}$, C is a minimal hitting set of $\mathcal{E}$.*
- *For each $E \in \mathcal{E}$, E is a minimal hitting set of $\mathcal{C}$*

Proof. This can be intuitively explained. A hitting set of all the exclusion sets of $\mathcal{P}$ is a subset of $\mathcal{U}$ that is not contained in any relaxation of $\mathcal{P}$, which means it is not consistent. A minimal hitting set of $\mathcal{E}$ is thus a minimal inconsistent subset of $\mathcal{U}$, which is the definition of a minimal conflict of $\mathcal{P}$. Conversely, let E be a hitting set of $\mathcal{C}$. By relaxing all the constraints of E, at least one constraint of each minimal conflict is relaxed. But by definition of a minimal conflict, it is enough to remove one constraint from it to render it consistent. Therefore, by relaxing all the constraints in E, all causes for inconsistency are corrected, and thus consistency is retrieved, which by definition means that E is an exclusion set. □

Furthermore, when not all maximal relaxations are taken into account, the following property also holds [60, 3].

Proposition 3 (Minimal Incomparable Relaxation)**.** *Let $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ be an inconsistent problem. Let $\mathcal{R}' \subset \mathcal{R}$ be a set of some, but not all, maximal relaxations of $\mathcal{P}$, and let $\mathcal{E}'$ be the corresponding set of minimal exclusion sets $\mathcal{E}' = \{\mathcal{U} \setminus R / R \in \mathcal{R}'\}$. Then there exists a minimal hitting set of $\mathcal{E}'$ that is consistent (and is not contained in any $R' \in \mathcal{R}'$).*

Proof. This observation is a bit more subtle. Of course, no hitting set of $\mathcal{E}'$ is included in any $R' \in \mathcal{R}'$. Now let R be a maximal relaxation of $\mathcal{P}$ not in $\mathcal{R}'$. By definition, any of its subsets is consistent. Now consider a subset H of R that is still incomparable with any element in $\mathcal{R}'$, and suppose H is minimal, i.e. any subset of H is also a subset of some maximal relaxation in $\mathcal{R}'$. Then one has simply to observe that H is actually a minimal hitting set of $\mathcal{E}'$. □

3.3 Explanation Enumeration

Enumerating explanations is an extremely challenging task. Much existing work focuses on the computation of all the diagnoses of a fault [112], which in our terms amounts to enumerating minimal exclusion sets. Many of the initial enumeration algorithms explore a search tree, searching the subsets of the user constraints. This tree might explore a number of subsets that is exponentially larger than the number of minimal exclusion sets or minimal conflicts. Pruning rules allow us to reduce this search space [70, 66], often quite significantly with the use of a variety of subtle rules [29].

However, the most efficient methods all make use of the property that links maximal relaxations and minimal conflicts [112, 92, 91, 90, 3, 59, 120, 121], and essentially, while enumerating all minimal conflicts, the algorithms also enumerate maximal relaxations. Of course, the number of maximal relaxations and minimal conflicts can still be exponentially larger than each other, but empirical evidence shows that in practice they are both smaller than the search tree that would otherwise be explored[3].[1]

For historical reasons, let us mention first that, in his original work on diagnoses, Reiter [112] presents an algorithm that outputs all (not necessarily minimal) conflict sets (with the use of a theorem prover) and all diagnoses (i.e. minimal exclusion sets) using the hitting set duality between those concepts. In the context of explanations however, two independent approaches have exploited this duality for the enumeration of explanations [3, 90]. The first, called Dualize and Advance [3], simultaneously generates the set of maximal relaxations and minimal conflicts, using consistent minimal hitting sets of minimal exclusion sets as seeds to compute the next maximal relaxation; those that are not consistent are the minimal conflicts. The algorithm terminates when no such seed exists, meaning that the two complete sets have been generated.[2]

The second method, from Liffiton and Sakallah [90, 91], on the other hand,

[1] Note too that we establish in Section 3.5 that none of the two problems of enumerating maximal relaxations or minimal conflicts is harder than the other.

[2] Reiter's algorithm works in a similar way, but in the opposite direction: conflicts are generated first (they correspond to refutations of a propositional sentence) and from those maximal relaxations (i.e. diagnoses) are computed using to hitting set computations.

computes those two sets independently. It first computes the set of maximal relaxations, and then derives the set of minimal conflicts, using minimal hitting set computations. This method claims better results on Boolean formulas, and the presented algorithm is indeed focused purely on SAT – search for maximal relaxations is done by iteratively solving a MaxSAT problem on a reformulation of the original SAT model, and the MaxSAT is solved by an ad-hoc adaptation of the SAT solver that trivially handles a single ATMOST-k constraint. More specifically, the MaxSAT problem involves maximising the number of clauses that are satisfied in an inconsistent SAT instance. This corresponds to a relaxation (of maximal cardinality).

In order to solve this MaxSAT problem, the model is reformulated by adding to every clause of the model a selector variable; when it is false, the rest of the clause, i.e. the original clause, has to be satisfied. If the number of selector variables set to true is set to k, a solution to the reformulated model corresponds to a relaxation of size at least $m - k$ (where m is the number of clauses). The search procedure can be set to look first for all solutions of the reformulated model for a given k before looking for solutions for an incremented value of k (by instantiating to 0 all remaining selector variables as soon as k selector variables have been instantiated to 1). The search is started with $k = 1$, and for every new solution, corresponding to an exclusion set, a new clause is added to the model preventing any exclusion set that is a superset of it to be considered again, thus ensuring that only minimal exclusion sets (i.e. maximal relaxations) are generated.

As an idea, this methods provides a general approach, but the actual algorithm for enumerating maximal relaxations has to be domain and solver-specific, and therefore cannot be straightforwardly adapted to our setting. In particular, it seems very hard to rely on abstracted consistency detection, which is one of the basic aspects of our formalism. For this reason [3], we consider Dualize and Advance as being the current state-of-the-art in explanation enumeration, at least for our context. This method has then been adapted to Disjunctive Temporal Problems [128, 129] by the same authors [92]. It has also been further improved by the

[3] And also because the way Dualize and Advance is presented contains an error, raising some doubts about the way it has been implemented and thus possibly about the validity of their experimental results – actually the authors themselves acknowledged it later [91].

use of a local search preprocessing that records *potential* minimal exclusion sets, which are then confirmed or rejected during the complete phase of the original approach [59].

3.4 Definition of Representative Explanations

3.4.1 The Approach

We present a sequence of examples, based on Example 3.2.1, demonstrating the approach we propose in this chapter. We presented in Table 3.2 the set of all explanations, each showing how the user can satisfy at least some of his constraints. Each explanation comprises a maximal consistent relaxation, i.e. the set of constraints that can be satisfied, and the corresponding minimal exclusion set of constraints that must be excluded. We show both the subset of the constraints in the relaxation (marked with a ✓) and those that are in the exclusion set, i.e. those that must be removed (marked with a $\times$). For example, consider Explanation I: we can simultaneously satisfy constraints c_3, c_4 and c_5, but in order to do so we must exclude c_1 and c_2.

However, in general we cannot present the user with every possible relaxation/exclusion set for his set of preferences, because the number of maximal relaxations is exponential in the number of user constraints in the worst case. The best we can hope for is to, therefore, present a subset of all possible relaxations/exclusion sets of the user's constraints. For example, we might require that every constraint that appears in a relaxation appears at least once in a relaxation in our chosen subset. This is the scenario presented in Table 3.3. However, this approach has the potential to mislead the user. In our example, the user might believe that it is never useful to exclude c_4, and might miss that it is possible to get option c_5 while still satisfying the budget constraint c_1, drawing the wrong conclusion.

A similar problem arises if we present a set of explanations that ensures that every constraint that must be relaxed once appears in at least one exclusion set as shown in Table 3.4. Here, the user might be lead to believe that constraint c_2 must always be excluded.

Table 3.3: A set of representative relaxations.

	Constraints				
Explanation	c_1	c_2	c_3	c_4	c_5
I	×	×	✓	✓	✓
II	×	✓	×	✓	✓
III	✓	×	✓	✓	×

Table 3.4: A set of representative exclusion sets.

	Constraints				
Explanation	c_1	c_2	c_3	c_4	c_5
I	×	×	✓	✓	✓
III	✓	×	✓	✓	×
V	✓	×	×	×	✓

Instead, we propose that the subset of explanations that are presented to the user should be both representative of the relaxations and exclusion sets of the problem. Specifically, a set of explanations should be presented that contains at least one maximal relaxation containing each constraint that can be satisfied at least once, and at least one minimal exclusion set containing each constraint that must be excluded at least once. The set of explanations must satisfy the property that a constraint should only appear in all relaxations if it is always satisfied, or that one should only appear in all exclusion sets if it never participates in a maximal relaxation. A set of explanations that satisfies these properties is presented in Table 3.5.

Table 3.5: A set of representative explanations.

	Constraints				
Explanation	c_1	c_2	c_3	c_4	c_5
I	×	×	✓	✓	✓
IV	✓	✓	×	✓	×
V	✓	×	×	×	✓

In the following subsections we formalise our approach, and prove the intractability of one of the fundamental decision problems that underpins the approach. We discuss the implications this has for what we can hope to achieve

from an algorithmic perspective. The succeeding section presents an algorithm for finding representative sets of explanations for over-constrained problems.

3.4.2 Representative Explanations

A maximal relaxation defines a maximal set of constraints that the user can satisfy, while the corresponding minimal exclusion set tells the user which constraints he must remove. However, the user might not be satisfied with an arbitrary explanation. Since the size of $\mathcal{R}$ can be exponential in the number of constraints in $\mathcal{U}$, presenting all explanations is not practical in general. We are, therefore, interested in selecting a subset of the explanations that are "representative" of all possibilities.

Definition 3.4.1. [Representative Set of Explanations] Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ that is inconsistent, $\mathcal{R}' \subseteq \mathcal{R}$ a set of maximal relaxations of $\mathcal{P}$, $\mathcal{E}' = \{\mathcal{U} \setminus R / R \in \mathcal{R}'\}$ the corresponding set of minimal exclusion sets, and $\mathcal{X} \stackrel{\text{def}}{=} (\mathcal{R}', \mathcal{E}')$, we say that $\mathcal{X}$ is a representative set of explanations iff

1. $\forall R \in \mathcal{R}, \forall c \in R$, there exists a relaxation $R' \in \mathcal{R}'$ such that $c \in R'$, and

2. $\forall E \in \mathcal{E}, \forall c \in E$ there exists an exclusion set $E' \in \mathcal{E}'$ such that $c \in E'$.

Clearly, the example set presented in Table 3.5 is a representative set of explanations since it contains a relaxation (respectively, an exclusion set) containing each constraint that appears in a relaxation (respectively, an exclusion set). Of course, we wish to restrict the size of our representative sets to an optimal size. It is more computationally efficient to focus on set-wise minimal sets of explanations. We therefore, define the notion of minimal representative set.

Definition 3.4.2 (Minimal Representative Set of Explanations)**.** Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ that is inconsistent, we say that a set of representative set of explanations is minimal if any strict subset of it is not representative.

For sake of simplicity, we will implicitly assume that a representative set is minimal, unless ambiguity is to be avoided.

Theorem 1 (Number of Explanations)**.** *The size of any minimal representative set of explanations is at most $|\mathcal{U}|$, and this bound is exact.*

Proof. Let $(\mathcal{R}, \mathcal{E})$ be a minimal representative set of explanations, and let $\mathcal{R} = R_1, ..., R_m$ and $\mathcal{E} = E_1, ..., E_m$. R_2 is incomparable to R_1 so it must contain a constraint that is not in R_1, and E_2 is incomparable to E_1 so it must contain a constraint not in E_1. Let $|R_1| = k$ and thus $|E_1| = |\mathcal{U}| - k$. Since $(\mathcal{R}, \mathcal{E})$ is minimal, for every explanation (R_i, E_i) either R_i has to cover a constraint not covered by any other R_j or E_i has to cover a constraint not covered by any other E_j. There are only $|\mathcal{U}| - k - 1$ constraints not covered by R_1 or R_2, and $k - 1$ constraints not covered by E_1 or E_2, so there are no more than $|\mathcal{U}| - 2$ such constraints to be covered by the remaining $m - 2$ explanations. Thus, $m \leq |\mathcal{U}|$. To show that the bound is tight, consider an example where the only minimal conflict is the set $\mathcal{U}$. Then there are $|\mathcal{U}|$ minimal exclusion sets, each containing one of the constraints. □

Of course, two minimal representative sets of explanations can have a very different size. The following example illustrates a case where one of them has a linear size while the other has a constant size.

Example 3.4.1. Let $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ be a problem on the boolean variables $x_1, \ldots, x_n$, with n even, with a single background constraint stating $\sum_{i=1}^{n} x_i \leq n/2$ and $\mathcal{U} = \{c_1, \ldots, c_n\}$ with c_i stating $x_i = 1$. Then we have $\mathcal{E} = \{S \subseteq \mathcal{U} / |S| = n/2\}$. We have $|\mathcal{E}| = \binom{n}{n/2} = \frac{n!}{(n/2)!^2} \geq 2^{\frac{n}{2}}, \forall n \geq 0$. Let $\mathcal{E}_1 = \{\{c_1, ...c_{n/2-1}, c_i\} / \frac{n}{2} \leq i \leq n\}$: $\mathcal{E}_1$ is a representative subset of $\mathcal{E}$ of size $n/2 + 1$. Now let $\mathcal{E}_2 = \{\{c_1, ..., c_{n/2}\}, \{c_{n/2+1}, ..., c_n\}\}$: $\mathcal{E}_2$ is a representative subset of $\mathcal{E}$, of size 2. ▲

3.4.3 Complexity

Let us consider the fundamental decision problem associated with finding representative sets of explanations. To ensure that we have found a representative set of explanations, we must at least find a minimal exclusion set for each constraint, provided that each constraint appears in at least one minimal exclusion set. Unfortunately, checking that this condition holds is *NP*-Complete.

Theorem 2 (Complexity)**.** *Given an inconsistent constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, a procedure Π for testing consistency of a set of constraints, and a constraint $c \in \mathcal{U}$, deciding whether there is a minimal exclusion set E such that $c \in E$, is* NP*-Complete.*

Proof. Given a relaxation R, we can test in polynomial time whether it is maximal by running the consistency checker Π on $R \cup \{c'\}$ for every $c' \notin R$. We can easily test that $c \notin R$. Thus, the problem is in *NP*. To show completeness, we will use a reduction from 3SAT. Let $x_1, \ldots, x_n$ be the variables and f the CNF formula of an instance of 3SAT. We build an instance of the problem of deciding whether there exists a maximal relaxation of an inconsistent problem that does not contain a given constraint. For each variable x_i, let c_i (resp. c'_i) be the constraint that enforces $x_i = 1$ (resp. $x_i = 0$) and $\bot$ the constraint that holds iff the variables form an instantiation that violates one of the clauses of f. We define $\mathcal{P}$ with $\mathcal{U} = (\cup_{i=1}^{n} \{c_i, c'_i\}) \cup \{\bot\}$ and no background constraint. Clearly, $\mathcal{P}$ is inconsistent (any $\{c_i, c'_i\}$ set is inconsistent). There are 2^n maximal relaxations, each corresponding to a different value assignment to each of the n variables. Let R be a relaxation corresponding to an assignment of the n variables. This assignment satisfies f iff $\bot$ cannot be added to R, i.e. R is a maximal relaxation. Therefore, f has a solution iff there is a maximal relaxation not containing $\bot$. □

Corollary. *Given an inconsistent problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, a set of explanations $(\mathcal{R}', \mathcal{E}')$, with $\mathcal{R}' \subseteq \mathcal{R}$, testing if $(\mathcal{R}', \mathcal{E}')$ is representative is* CoNP*-Complete.*

Proof. To test if $(\mathcal{R}', \mathcal{E}')$ is representative, one must particularly check point 2 of Definition 3.4.1. In particular, if some constraint $c \in \mathcal{U}$ is not contained in any $E \in \mathcal{E}'$, one must make sure c is not contained in any exclusion set of $\mathcal{P}$ at all. If there exists an exclusion set containing c, then $(\mathcal{R}', \mathcal{E}')$ is not representative. □

3.5 Complexity of Explanation Enumeration

As we saw in Section 3.3, the most efficient enumeration algorithms effectively enumerate both sets of maximal relaxations and minimal conflicts. Therefore, the complexity of such algorithms depends on both the number of maximal relaxations and minimal conflicts. However, as mentioned earlier, these two numbers

are independent and each can be exponentially larger than the other. As a consequence, if one is interested only in the maximal relaxations of an over-constrained problem, he will incur a potential computational overhead resulting from the number of minimal conflicts. Strangely enough, no formal complexity discussion, to our knowledge, has ever justified the need for this overhead. We can actually prove that enumerating the maximal relaxations of an over-constrained problem, with a consistency oracle, is "intractable", even in terms of the output, as it will be discussed in the following sections.

3.5.1 Complexity of Enumerating Maximal Relaxations

Let us consider, given a problem, its enumeration version, i.e. the problem of finding all the solutions instead of one solution. When the number of solutions to a problem is not polynomial in the size of the input, clearly the enumeration problem is not in *P*. But as this happens for any but the most trivial enumeration problems, we need a more subtle way to analyse the complexity of enumeration. An obvious way to circumvent this issue is to bring into consideration the size of the output too. An algorithm is *output-polynomial* [102] if it runs in time polynomial in the input *and* the output. For enumeration procedures, this means that the algorithm can enumerate all solutions in polynomial time in the number of solutions. When considering enumeration algorithms, the notion of *incremental-polynomial* [73, 102] is more suitable a characterisation of what one might consider as an efficient enumeration procedure. An enumeration algorithm runs in *incremental-polynomial* time if it can generate k solutions in polynomial time in the size of the input and k, for any arbitrary k. Clearly, an incremental-polynomial enumeration algorithm is also output-polynomial. However, this property has an added practical interest in that it implies an *anytime behaviour*: the algorithm can be stopped at any particular moment if the current output is satisfactory; in particular, if we only want to enumerate a polynomial number of solutions, an incremental-polynomial algorithm will be polynomial, whereas we have no complexity guarantee that an output-polynomial algorithm will not be exponential.

An equivalent definition of an incremental-polynomial algorithm is that an algorithm is incremental-polynomial if it can compute a new solution in polynomial

time in the number of solutions computed so far. Indeed, given $k-1$ solutions, we can generate a new one by generating k solutions: at least one solution will be new, the equivalency in the other direction being trivial.

Let us consider our problem of enumerating the maximal relaxations of a problem. Even though generating one maximal relaxation is easy; with what we mean by easy having already been discussed, enumerating them is actually, and somehow counter-intuitively, hard. We can prove that enumerating the maximal relaxations of an over-constrained problem is not incremental-polynomial, unless *P=NP*.

Theorem 1. *Given a set of maximal relaxations and an consistency oracle, computing a maximal relaxation different from any in the given set is* NP*-Hard.*

In order to prove that theorem, the following proposition holding on the decision version of the problem must be proved:

Proposition 2. *Given a set $R_1, \ldots, R_n$ of maximal relaxations, deciding if there exists another maximal relaxation that is different from any of those is* NP*-Complete.*

Proof. Consider the problem slightly reformulated: given a set $R_1, \ldots, R_n$ of maximal relaxations, does a consistent set exist that is minimally incomparable with any of those relaxations? Let COMPMR denote that problem. Obviously, these two formulations are equivalent: if this set exists, it can be extended polynomially to a new maximal relaxation.

COMPMR is clearly in *NP*: given a set $T \subseteq \mathcal{U}$, it is polynomial to check that it is incomparable with any R_i, that it is set-wise minimal with regards to this property, and that it is consistent (using the consistency oracle).

Now, let MINHST denote the problem of deciding whether, given a collection of subsets $S_1, \ldots, S_n$ of a universe U and an integer k, there exists a hitting set H of $S_1, \ldots, S_n$ of cardinality $|H| \leq k$. In order to show completeness, we will reduce MINHST to COMPMR.

A problem $\mathcal{P}$ is built as follows:

- a boolean variable X_i corresponds to each element of $U = \{1, \ldots, M\}$;
- let $C_1 = \vee_{i \leq n}(\wedge_{j \in S_i} \neg X_j)$;

- let $C_2 = \wedge_{i \leq n}((\vee_{j \in S_i} X_j) \wedge (\vee_{j \notin S_i} \neg X_j))$;
- let c_i be the constraint stating $X_i = 1$;
- let $\mathcal{B} = \{C_1 \vee (C_2 \wedge \text{ATMOST}k(X_1, \ldots, X_M))\}$;
- let $\mathcal{U} = \{c_i | i \leq M\}$.

Checking the consistency of $\mathcal{B} \cup U$ for some set $U \subseteq \mathcal{U}$ is polynomial. U is consistent with C_1 if, for some $i \leq n$, it is included in $\{c_j / j \notin S_i\}$. It is consistent with C_2 if, for each $i \leq n$, $\exists j \in S_i$ with $c_j \in U$ and $\exists j \notin S_i$ with $c_j \notin U$; in other words, for each $i \leq n$, U is incomparable with $\{c_j / j \notin S_i\}$. It follows that U is consistent with $\mathcal{B}$ if either it is included in one of the sets $\{c_j / j \notin S_i\}$ or if it is incomparable with any of those sets and it has a cardinality of at most k.

Obviously, $\mathcal{B} \wedge \mathcal{U}$ is inconsistent. The sets $R_i = \{c_j | j \notin S_i\}$ are all maximal relaxations, i.e. the sets S_i correspond to minimal exclusion sets. Amongst all the sets minimally incomparable with any R_i, all and only those of cardinality less than or equal to k are consistent, and they correspond to a hitting set of the S_i sets of cardinality $\leq k$. □

Example 3.5.1. As an illustration of the proof, consider the following instance of MINHST: let $U = \{1, 2, 3, 4, 5\}$, $S_1 = \{1, 2\}$, $S_2 = \{1, 3\}$, $S_3 = \{1, 5\}$, $S_4 = \{2, 5\}$ and $k = 2$. The minimal hitting sets are $\{1, 2\}$, $\{1, 5\}$ and $\{2, 3, 5\}$, two of which are of size ≤ 2.

$R_1 = \{c_3, c_4, c_5\}$, $R_2 = \{c_2, c_4, c_5\}$, $R_3 = \{c_2, c_3, c_4\}$ and $R_4 = \{c_1, c_3, c_4\}$ are maximal relaxations. Let $H_1 = \{c_1, c_2\}$, $H_2 = \{c_1, c_5\}$, $H_3 = \{c_2, c_3, c_5\}$. None of these sets is consistent with C_1. However they are all consistent with C_2, but only H_1 and H_2 are also consistent with $\text{ATMOST}k(X_1, \ldots, X_M)$, and, therefore, with $\mathcal{B}$. H_1 and H_2 are therefore minimal hitting sets of cardinality 2. ▲

The complexity of enumerating all the maximal relaxations, in other words the existence of an output-polynomial algorithm for the enumeration of maximal relaxations, has not been considered here, and is to our knowledge an open question. However similar complexity results, both positive and negative, have been found in the parallel field of abduction with Horn theories [41, 121, 84].

3.5.2 Complexity of Enumerating Minimal Conflicts

The previous result implies that enumerating the minimal conflicts is also intractable, with intractable having the same meaning as per the previous discussion. Given any problem, we can create a new problem whose exclusion sets correspond to the conflicts in the initial problem, as follows.

Let $\mathcal{P} = \langle \mathcal{B}, \mathcal{U} = \{c_1, \ldots, c_n\} \rangle$ be a problem. We build a problem $\mathcal{P}' = \langle \mathcal{B}', \mathcal{U}' \rangle$ as follows. We have n boolean variables $X_1, \ldots, X_n$, one background constraint stating that an assignment of $X_1, \ldots, X_n$ is valid iff $\mathcal{B} \cup \{c_i \in \mathcal{U} / X_i = 0\}$ is inconsistent, and n user constraints c'_i each stating that $X_i = 1$. If $\mathcal{B}$ is consistent, $\mathcal{B}' \cup \mathcal{U}'$ is inconsistent; if $\mathcal{B} \cup \mathcal{U}$ is inconsistent, $\mathcal{B}'$ is consistent (setting all variables to 0 is a solution); for a given set $U \subset \mathcal{U}'$, $\mathcal{B}' \cup U$ is consistent iff $\mathcal{B} \cup \{c_i \in \mathcal{U} / c'_i \notin U\}$ is inconsistent. We thus have a constant time consistency checker for $\mathcal{P}'$ iff we have a consistency oracle for $\mathcal{P}$, and the minimal exclusion sets of $\mathcal{P}'$ are in one to one correspondence with the minimal conflicts of $\mathcal{P}$.

3.6 Computing Representative Sets of Explanations

The previous discussions in Section 3.4.3 and Section 3.5 show two crucial inherent intractabilities in computing representative sets of explanations, and therefore prove that we cannot expect to have a polynomial algorithm for computing representative sets of explanations unless *P=NP*. An exact polynomial algorithm would consist of a partial enumeration algorithm the behaviour of which can be roughly depicted as follows:

1. is the current set of explanations representative? (halting condition)
2. if not, generate a new maximal explanation that covers a new constraint and go back to step 1.

This algorithm would only generate a polynomial number of explanations, according to Theorem 1. However, we saw that such an algorithm cannot exist, unless *P=NP*: both of the steps just depicted are intractable. In particular, step 2 is intractable because of the two intractabilities that have been raised: knowing if a

relaxation exists that does *not* contain a constraint is *NP*-Complete, and computing just a new relaxation (without any additional property) is *NP*-Hard too. As a result, we have to compromise between run-time and the guarantee of the result. In our approach, we decided to guarantee the maximality of relaxations, which means the complexity of generating a new maximal relaxation will have to be accepted, as with any explanation enumeration algorithm, but we relax the need to guarantee representativeness. Ideally, we would like to relax the second requirement in a soft way, by having an anytime behaviour. Such an algorithm, in the best case, would quickly find, in practice, a representative set of explanations, while it might need to spend most of its time actually proving representativeness or, in the worst case, computing only a few explanations that are missing to form a representative set. We cannot have a guarantee that, although only looking for a partial and polynomially sized subset of explanations, this algorithm would not possible incur the cost of a complete enumeration. With this consideration in mind, we decided to adapt an existing enumeration procedure to fit these requirements.

3.6.1 The Algorithm

As we saw in Section 3.3, amongst the best algorithms for computing all minimal conflicts [3, 59], one is an algorithm tailored for SAT problems [59], while the other, called Dualize and Advance, has the advantage in that it is for general constraints [3], and we consider it as being state-of-the-art. The algorithm we propose, REPRESENTATIVEXPLAIN, is based on a modification of this existing algorithm.

This algorithm operates in three steps: first REPRESENTATIVEDA computes a set of explanations such that every constraint that belongs to at least one exclusion set of the problem belongs to an exclusion set of the returned set of explanations. Next, this set is rendered representative by adding into it, for every constraint that belongs to at least one relaxation of the problem (i.e. such that $\mathcal{B} \cup \{c\}$ is consistent) and that belongs to no relaxation in the returned set (i.e. $\forall E \in \mathcal{E}, c \in E$), a relaxation containing this constraint. As a post-processing step, we can reduce the set of explanations to obtain a *minimal* representative set of explanations, provided (or assuming) the original set was representative too. This post-processing,

which we call *minimisation*, involves greedily removing explanations not required to ensure representativeness. This is also the opportunity to apply heuristics on the order of removal to try and minimise the cardinality of the final set, although we did not give any such consideration. The guarantee, or not, of representativeness of the final set follows from how REPRESENTATIVEDA terminated. We can decide either to stop it when a time limit or any other heuristical criterion has been met (number of constraints covered, number of explanations generated), or let it reach its halting condition, when representativeness has been proved.

Algorithm 2: REPRESENTATIVEXPLAIN

Data: $\mathcal{P} \stackrel{\text{def}}{=} (\mathcal{B},\mathcal{U})$, an inconsistent problem, $\mathcal{B}$ consistent.
Result: $\mathcal{X}$ a minimal representative set of explanations.

```
1 E ← RepresentativeDA(P)
2 R ← {U \ E/E ∈ E}
3 foreach c ∈ U do
4     if consistent(B ∪ {c}) ∧(∀E ∈ E, c ∈ E) then
5         R ← grow({c},U)
6         R ← R ∪ {R}
7         E ← E ∪ {U \ R}
8 X ← (R, E)
9 minimise(X)
```

The `grow()` function takes a consistent subset as a seed and grows it to a maximal relaxation by greedily adding from the remaining constraints in M those that do not create inconsistency, as detected by our propagator, called by the `consistent()` function.

The key property that is exploited in the algorithm is the duality between minimal exclusion sets and minimal conflicts that was previously outlined. At each point of the iteration of the **repeat** loop in function REPRESENTATIVEDA (Line 6), $\mathcal{H}$ contains all the minimal hitting sets of the current $\mathcal{E}$. The important property used in this function is that when $\mathcal{E}$ contains some but not all of the minimal exclusions, there must be at least one consistent minimal hitting set in $\mathcal{H}$ (see Proposition 3). Therefore, the condition $R = \emptyset$ is satisfied iff all elements in $\mathcal{H}$ are inconsistent iff $\mathcal{E}$ contains all the minimal exclusions. The hitting sets are incrementally computed in Line 12 by computing the cross product of two sets,

Algorithm 3: REPRESENTATIVEDA($\mathcal{P}$)

Data: $\mathcal{P} \stackrel{\text{def}}{=} (\mathcal{B}, \mathcal{U})$, an inconsistent problem, $\mathcal{B}$ consistent.

$\mathcal{C} \leftarrow \emptyset$
$\mathcal{E} \leftarrow \emptyset$
$\mathcal{H} \leftarrow \{\emptyset\}$
$R \leftarrow \emptyset$
$U \leftarrow \mathcal{U}$
repeat
  $R \leftarrow \texttt{grow}(R, \mathcal{U} \setminus U)$
  $R \leftarrow \texttt{grow}(R, U)$
  if $(\mathcal{U} \setminus R) \cap U \neq \emptyset$ **then**
    $U \leftarrow U \setminus (\mathcal{U} \setminus R)$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{U} \setminus R\}$
  $\mathcal{H} \leftarrow \texttt{Min}(\mathcal{H} \otimes \{\{c\}, c \in \mathcal{U} \setminus R\})$
  $R \leftarrow \emptyset$
  for $H \in \mathcal{H} \setminus \mathcal{C}$ **do**
    if $\texttt{consistent}(\mathcal{B} \cup H)$ **then**
      $R \leftarrow H$
      break
    else $\mathcal{C} \leftarrow \mathcal{C} \cup \{H\}$
until *limit reached* $\vee (U = \emptyset \vee R = \emptyset)$
return $\mathcal{E}$

Algorithm 4: Auxiliary functions

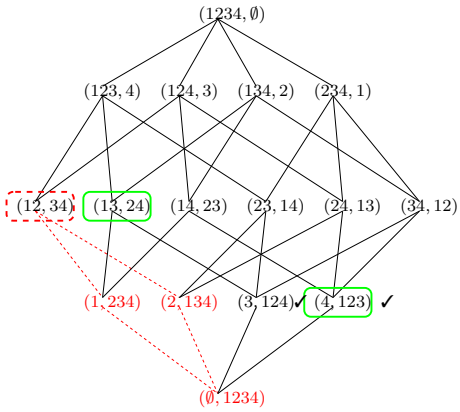
function $\texttt{minimise}((\mathcal{R}, \mathcal{E}))$
  foreach $R \in \mathcal{R}$ **do**
    $E \leftarrow \mathcal{U} \setminus R$
    if $(\forall c \in R, \exists R' \neq R, c \in R') \wedge (\forall c \in E, \exists E' \neq E, c \in E')$ **then**
      $\mathcal{R} \leftarrow \mathcal{R} \setminus \{R\}$
      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{E\}$

function $\texttt{grow}(S, M)$
  foreach $c \in M \setminus S$ **do**
    if $\texttt{consistent}(\mathcal{B} \cup S \cup \{c\})$ **then** $S \leftarrow S \cup \{c\}$
  return S

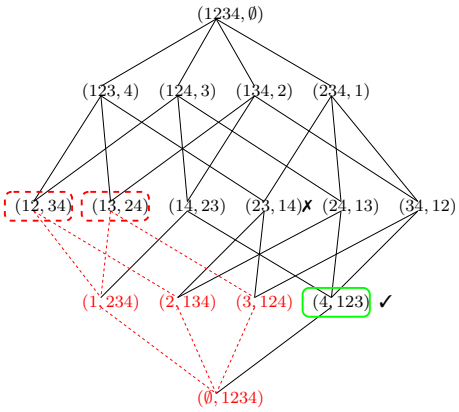
function $\texttt{Min}(\mathcal{H})$
  if $\mathcal{H} = \emptyset$ **then return** $\emptyset$
  Let $H \in \mathcal{H}$ such that $\forall H' \in \mathcal{H}, |H| \leq |H'|$
  return $\{H\} \cup \texttt{Min}(\{H' \in \mathcal{H} / H \not\subseteq H'\})$

where $\mathcal{H}_1 \otimes \mathcal{H}_2 = \{H_1 \cup H_2 / H_1 \in \mathcal{H}_1 \wedge H_2 \in \mathcal{H}_2\}$, and then reducing the result, where `Min()` keeps all set-wise minimal elements of the given set. Example 3.6.1 shows an illustration of this behaviour.

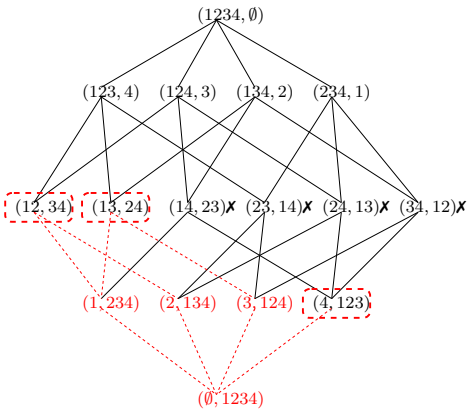
Example 3.6.1. Suppose we have a problem such that the maximal relaxations are $\{1,2\}$, $\{1,3\}$, $\{4\}$; the minimal exclusion sets are $\{3,4\}$, $\{2,4\}$, $\{1,2,3\}$; the minimal conflicts are $\{1,4\}$, $\{2,3\}$, $\{2,4\}$, $\{3,4\}$, as depicted on Figure 3.1.



(a) Enumeration of 12



(b) Enumeration of 13



(c) Enumeration of 4

Figure 3.1: Enumeration of three maximal relaxations

Consider Figure 3.1(a). Suppose the maximal relaxation 12 has been produced. The dashed part of the lattice correspond to consistent sets, i.e. to relaxations, but that are all included in 12. In order to guarantee a new maximal

relaxation is found, we need to start from a consistent set that is minimally incomparable with 12, i.e. from a consistent minimal hitting set of the corresponding minimal exclusion set 34. There are two such sets: 3 and 4, both of which are consistent.

Suppose we pick 3. We can still add 1 to form a new maximal relaxation. Now, as shown on Figure 3.1(b), there are two minimal hitting sets of the two minimal exclusions sets 34 and 24. 23 is inconsistent, and is therefore a minimal conflict, while 4 is consistent, and is in fact a maximal relaxation.

Now, all minimal hitting sets, shown in Figure 3.1(c), are inconsistent: they all are minimal conflicts, and all maximal relaxations have been found. ▲

The algorithm used in REPRESENTATIVEDA is an adaptation of Bailey and Stuckey's [3] Dualize and Advance algorithm. The essentials of the modification are that we change the termination condition at Line 19 to either (a) having reached some limit, or (b) having proved representativeness. This happens after either (a) having covered all the user constraints, or (b) having found all minimal exclusions, whichever occurs first.

Of course, when not all the user constraints belong to at least one minimal exclusion set, only condition (b) is met, and we cannot expect to do much better than generating all minimal exclusion sets. However, even in this case, our algorithm can still find a representative set of explanations quickly, while the rest of the effort is spent *proving* representativeness, as shown in the next section.

Additionally, we prefer that user constraints be covered as quickly as possible, in order to converge to a representative set of explanations as quickly as possible. An obvious heuristic (which we refer to as Heuristic 1) to speed up the convergence is simply to consider the order in which the constraints are added to the current relaxation by the `grow()` function. We try to first add constraints not in U so that as many constraints as possible in U will remain in the exclusion set (Line 7).

We considered two additional heuristics which are not depicted in the algorithm's pseudo-code. A second heuristic (which we refer to as Heuristic 2) would be to consider which consistent hitting set we choose in $\mathcal{H}$ as a seed for the next maximal relaxation. If we have a consistent $H \in \mathcal{H}$ and $\exists c \in U$ such that the addition of c in H causes a conflict, then by choosing H as a seed, we will find

a maximal relaxation that will relax c. A third heuristic (which we refer to as Heuristic 3) would be to choose the $H \in \mathcal{H}$ that causes a conflict with the largest number of constraints in U (instead of just one at least). Of course it could be the case that none of the constraints in U can create on its own an inconsistency with any of the consistent hitting sets.

3.6.2 Experiments

We studied the performance of our algorithm on both random and real-world problems[4]. The objective of the experiment was to study the reduction in the number of explanations one achieves by considering a minimal set of representative explanations, as computed using REPRESENTATIVEXPLAIN, rather than all minimal exclusions, as computed using the algorithm of Bailey & Stuckey [3], which we will refer to as the *baseline algorithm*.

Random Problems

The random problems consisted of fifteen boolean variables with one 15-ary background constraint defined on those variables. We varied the proportion of all possible assignments that were consistent with this background constraint, i.e. its satisfiability, taking 20 settings in all. Since we have assumed that the background constraints are always consistent, the least satisfiable background constraint accepted only one assignment.

For a given level of satisfiability, we randomly generated 10 background constraints. For every background constraint, we generated 10 inconsistent queries for which each algorithm generated a set of explanations. Each query was defined in terms of a set of user constraints that assigned a random value to each variable such that the whole set of constraints was inconsistent. We plot the average results over 100 queries at each satisfiability setting in the next figures.

Consider the size of the representative set of explanations in Figure 3.2. The data in this plot represents the number of representative explanations found, before the minimisation process. For most settings of satisfiability we observe a

[4]Experiments were implemented in Java and run on an iMac (2.33GHz Intel Core 2 Duo), 3GB RAM, running Mac OSX 10.4.8.

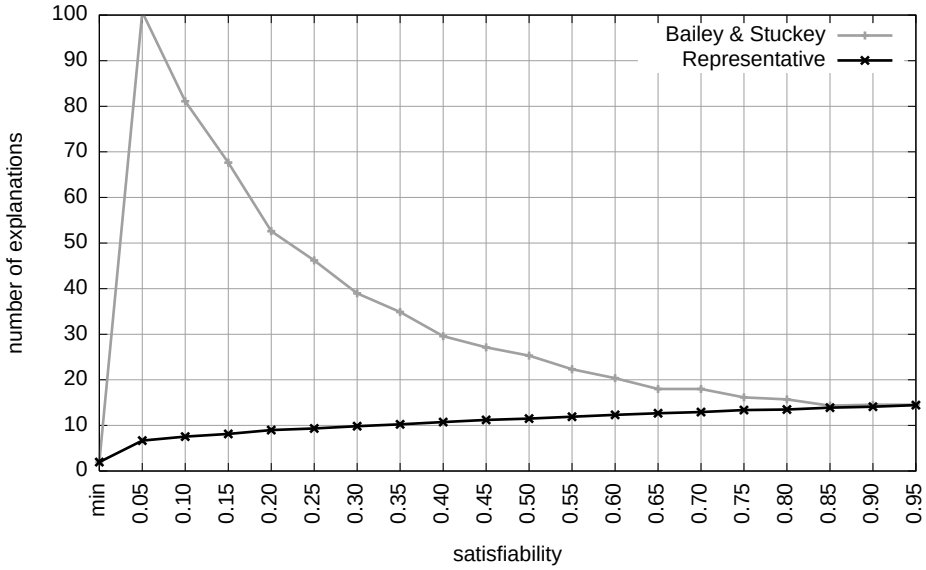


Figure 3.2: Cardinality of the sets of explanations

significant gap between the total number of exclusions, as found by the baseline algorithm, and the number of representative explanations found by our algorithm. We noted that in the vast majority of cases, the set of explanations was already (set-wise) minimal, and was already representative. However, with Heuristic 2 and Heuristic 3, the total computation time was almost always higher than the time needed for a complete enumeration with the baseline algorithm, while the number of the exclusions found was almost the same as with the basic algorithm with Heuristic 1. This suggests that these heuristics are too complex to give an advantage on convergence speed, at least with these random problems. Therefore, we will now on only consider the algorithm with Heuristic 1.

From Figure 3.3 we can see that the difference between algorithms in terms of running time mimics the difference in the size of the sets of explanations they generate. As we saw, REPRESENTATIVEXPLAIN can avoid enumerating all the relaxations of one instance if all the user constraints of this problem are involved in at least one exclusion set. We refer to instances in which this occurs as "true" instances. As we highlighted before, we can hope for a potentially large decrease in the total execution time on these "true" instances. Figure 3.4 confirms this, as we see that the difference in running times tends to decrease as the proportion of

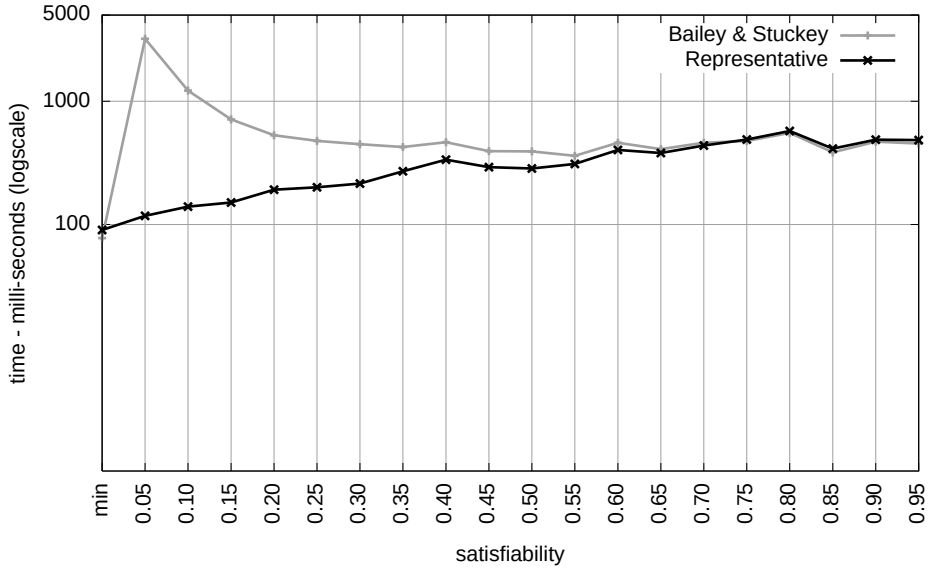


Figure 3.3: Times required to generate sets of explanations.

"true" instances decreases.

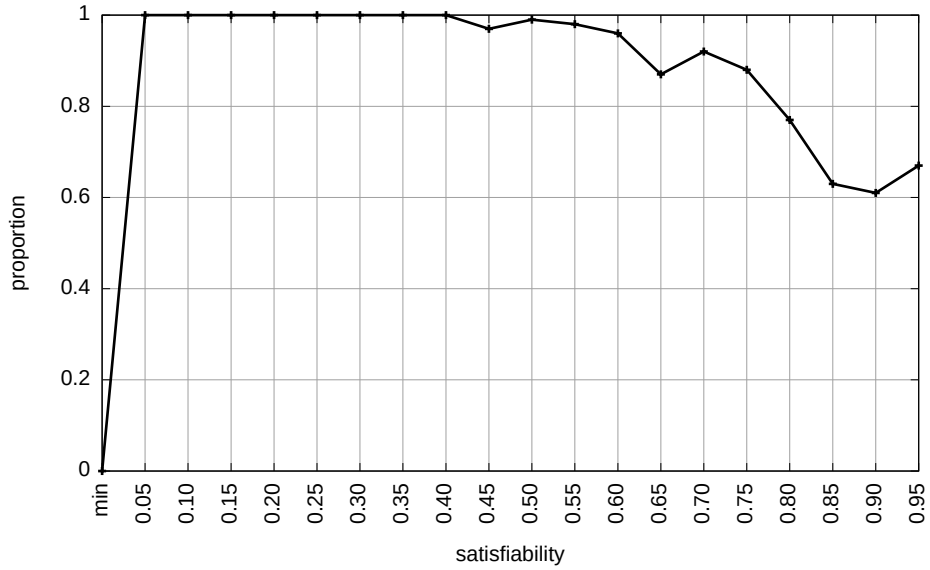


Figure 3.4: Proportion of queries per instances in which all constraints were involved in at least one exclusion set.

To analyse this behaviour more deeply on pure "false" instances, we ran a second kind of experiment. We used exactly the same process for generating

random instances as before, and just added a trivially satisfied constraint to each instance, so that it belongs to all maximal relaxations. The results are presented in Figure 3.5. On these instances, if we let REPRESENTATIVEXPLAIN reach termination, we cannot hope to be much faster than a full enumeration of all maximal relaxations. We measured two different times: the time when the last relaxation has been found by REPRESENTATIVEXPLAIN and the time when it terminates, i.e. the time to find a representative set of explanations and the time to also prove representativeness, respectively. Here again, the results are positive. We observe that we actually find a representative set much faster than it takes to find all relaxations. This is very interesting in an interactive context, where we can be confident a user will stop the computation quickly once he is indeed satisfied with the few relaxations he has been shown (because they form a representative set of explanations), regardless of whether they are *guaranteed* to be representative or not. However, unexpectedly, REPRESENTATIVEXPLAIN can terminate a little quicker than the baseline algorithm.

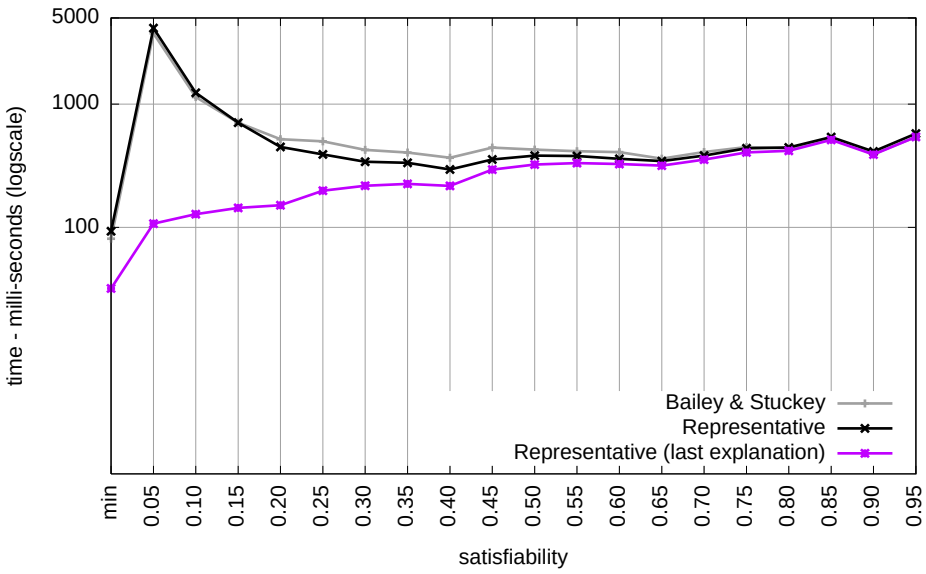


Figure 3.5: Average times for finding all relaxations, a representative set of explanations and the last explanation.

A Real-world Problem

We also ran experiments on a real-world problem, the Renault Megane car configuration problem [1]. This problem is defined by 99 variables and has 2.8×10^{12} solutions. We extracted four problem instances of this problem by restricting it in the following way. We ordered the variables by increasing domain size. Then, by a dichotomic search, we instantiated the variables with the largest domain sizes in order to reduce the number of solutions to a more reasonable level for an interactive application, while still honouring the real-world structure of the problem. The restricted instances of the problem provided four possible sets of background constraints, reducing the number of solutions by a factor of $10^6, 10^7, 10^8$ and 10^9 in each case. We compiled each instance into a tree-driven automaton [43], similar to that presented in [1], and further detailed in the next chapter. In short, this allows for very efficient consistency checks (in time polynomial in the size of the automaton). The user's set of constraints was generated by randomly assigning 30 of the remaining uninstantiated problem variables. The results of this experiment are presented in Table 3.6; the instances are labelled by the reduction factor in the number of solutions as compared with the original Renault problem. For each instance of the background constraint we computed explanations for 15 inconsistent queries, presenting the medians (due to the size of the instances) in this table.

These results confirm those on the random problems, except on one point. All of the instances are "false" instances, which was expected, and the total running time of REPRESENTATIVEXPLAIN is a little higher than the time for the baseline algorithm. This can be explained by the fact that REPRESENTATIVEXPLAIN performs more set operations (intersection and filtering on U) which have not been optimised for this bigger instance. However, the important results that the set of representative explanations is much more compact than the set of all explanations, and very compact in absolute terms indeed, and the last explanation can be found faster, are very encouraging.

Instance	Baseline time	Baseline #exps	REPRESENTATIVEXPLAIN time last	REPRESENTATIVEXPLAIN time all	REPRESENTATIVEXPLAIN #exp
renault 10^6	474.76	17	318.87	618.76	3
renault 10^7	263.95	11	125.51	324.71	3
renault 10^8	205.82	8	97.98	232.32	3
renault 10^9	293.00	12	139.67	350.51	3

Table 3.6: The results for the Renault problem.

3.7 Possible Extensions

We can generalise the definitions to richer forms of representativeness. For example, we might want to insist that all pairs, triples, etc. of constraints appearing in a relaxation or exclusion set also appear as such in our representative explanations.

Given a set of exclusion sets $\mathcal{E}$ and a set of maximal relaxations $\mathcal{R}$, let $C^k_{\mathcal{E}} = \bigcup_{E\in\mathcal{E}} \mathcal{P}_k(E)$, and $C^k_{\mathcal{R}} = \bigcup_{R\in\mathcal{R}} \mathcal{P}_k(R)$, where $\mathcal{P}_k(S)$ is the set of all subsets of S of size k, be the set of the combinations of k user constraints that can be excluded in at least one minimal exclusion set of $\mathcal{E}$ or satisfied in at least one maximal relaxation of $\mathcal{R}$, respectively.

Definition 3.7.1 (Strictly k-Representative Set of Explanations)**.** Let $\mathcal{R}' \subseteq \mathcal{R}$ and $\mathcal{E}'$ be the corresponding set of exclusion sets. $(\mathcal{R}', \mathcal{E}')$ is a *strictly k-representative* set of explanations if it is a minimal set of explanations containing all the possible ways of satisfying and excluding k constraints in $\mathcal{U}$, i.e. such that $C^k_{\mathcal{R}'} = C^k_{\mathcal{R}}$ and $C^k_{\mathcal{E}'} = C^k_{\mathcal{E}}$, and any strict subset of $\mathcal{R}'$ does not satisfy this property.

Definition 3.7.2 (k-Representative Set of Explanations)**.** $(\mathcal{R}', \mathcal{E}')$ is a *k-representative* set of explanations if $\forall l \leq k$, it is strictly l-representative and any strict subset of Φ does not satisfy with this property.

Obviously, the representative sets of explanations and the 1-representative sets are the same. Obviously enough, ensuring a stronger level of representativeness can only be done at the expense of compactness.

Property 3.7.1. *If $(\mathcal{R}', \mathcal{E}')$ is minimal k-representative then, $\forall k' \leq k$, $(\mathcal{R}', \mathcal{E}')$ is (not necessarily minimal) k'-representative.*

Example 3.7.1. Slightly abusing notation, suppose we have a problem such that $\mathcal{R} = \mathcal{E} = \{c_1, c_2c_3, c_2c_4, c_3c_4\}$. Then $C^1 = \{c_1, c_2, c_3, c_4\}$, $C^2 = \{c_2c_3, c_2c_4, c_3c_4\}$, and $\mathcal{R}_1 = \{c_1, c_2c_3, c_2c_4\}$ is 1-representative, $\mathcal{R}_2 = \{c_2c_3, c_2c_4, c_3c_4\}$ is strictly 2-representative, and $\mathcal{R}_1 \cup \{c_3c_4\}$ is 2-representative. ▲

These extensions are straightforward (albeit involving heavy formalism) to the basic definitions used throughout this chapter. The algorithms we presented can be simply adapted to take into account these extensions, and therefore we will not present further details on the matter. The practical usefulness of these extensions are very context-dependent and model-dependent. For example, depending on the type of user constraints, one user "requirement" could be made up of one or (at least) two user constraints, and the user could then wish to have either both constraints satisfied at the same time or none, whereas it would not make sense to him to have only one constraints satisfied at a same time, and therefore would not be interested in being shown any such explanation. Such considerations could be taken into account either in the way the problem is modelled or by the property of representativeness that is ensured by the algorithm.

Chapter 4

Compiled Representations for Explanation Generation

Summary. *We consider the problem of generating maximal relaxations by reasoning about their solubility, in the context of product configuration, where the constraint model of the problem has been compiled into an automaton. Two novel algorithms are presented. The first finds from amongst the longest relaxations to a set of inconsistent user constraints, the one that is consistent with the most/fewest solutions; while the second considers the problem for maximal relaxations. Based on a large real-world configuration problem we demonstrate the value of our approach. Finally, we generalise our results by identifying the properties that the target compilation language must have for our approach to apply.*

4.1 Introduction

In the previous chapter, we discussed how considering some well chosen sets of explanations instead of single explanations can be more informative. We proposed a definition of representative sets of explanations, that are representative of which user constraints may be kept and which may be relaxed. On the other hand, when we restrict the explanations we consider, we also restrict the solutions that the

user can eventually reach. If we ignore this aspect altogether, we are missing in some sense a part of the challenge, as the final purpose of the user is indeed to find a solution, not just a relaxation. We will now look how we can define sets of explanations keeping in mind the solutions they allow.

In many real-world settings, e.g. product configuration, constraint satisfaction problems are compiled into automata or binary decision diagrams, which can be seen as instances of Darwiche's negation normal form. In this chapter, we consider a setting in which the background constraints are compactly represented in a compiled form. We further assume, as already explained earlier in this dissertation, that user constraints are unary constraints. This chapter is organised as follows. We first describe the general framework used in this chapter. In particular, we provide insights into the compilation methods we use for our work. Then, we present two novel algorithms for finding relaxations based on automata. The first algorithm finds from amongst the *longest relaxations* to a set of inconsistent user constraints, the one that is consistent with either the fewest/largest number of solutions; this algorithm is *linear in the size of the automaton*. The second algorithm considers the same objectives in the context of set-wise *maximal relaxations*. Furthermore, we generalise our approach to other compiled representations by studying the properties that these must have in order for our results to carry over. We show that basically our algorithms do not need all the power of automata and that they work with more general, *i.e.* more compact, representations. Finally, we show empirically that, while it is not polynomial in the size of the automaton, the second algorithm, by being specific to our setting, can be on average more than 500 times faster than a much more general-purpose state-of-the-art algorithm.

4.2 General Framework

We recall here that we work in a setting where we consider configuration problems. As explained in Section 2.6.1, Chapter 2, these problems are ideal candidates for compilation methods. First, the partition between immutable background constraints and transient user constraints makes compilation well worth the investment for the background, to allow for fast computation times when dealing with

the user constraints. Additionally, configuration problems are real-life problems that often benefit from good compression levels by compilation methods, especially those that take advantage of their hierarchical structure. Moreover, we also recall that we restrict user constraints to unary constraints. In many cases, algorithms can take advantage of this specific case and operate more efficiently on compiled representations.

To summarise the precise setting used in this chapter and in the rest of this dissertation, we operate as follows: a given configuration problem is converted to some form of compiled structure; a user query, i.e. a set of user constraints, is stated during an interactive process; an algorithm provides a result concerning this user query, such as whether it is consistent or not, within an acceptable response time, by querying the compiled structure.

Concerning relaxations, we can impose on maximal relaxations and minimal exclusion sets the stronger property that they be of maximum/minimum cardinality.

Definition 4.2.1 (Longest Relaxation)**.** Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ that is inconsistent, a relaxation R of $\mathcal{P}$, R is a *longest relaxation* if for any other each maximal relaxation R', $|R'| \leq |R|$.

Accordingly, shortest exclusion sets are defined as follows.

Definition 4.2.2 (Shortest Exclusion Set)**.** Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ that is inconsistent, an exclusion set E of $\mathcal{P}$ is *shortest exclusion* if $\mathcal{U} \setminus E$ is a longest relaxation.

While intuitively we might believe that longer relaxations have fewer solutions, the story is not that simple. Theoretically, there is no reason why the number of solutions of two maximal relaxations should be similar. We can illustrate this both theoretically and with a concrete example.

Example 4.2.1. Consider variables $x_0, \ldots, x_n$ with respective domains $D(x_0) = \{0, 1\}$ and $D(x_i) = \{0, \ldots, d\}, i > 0$, and the constraints $x_0 < x_i, \forall i > 0$, $x_0 > x_i, \forall i > 0$ and $x_0 = 0$. This problem is inconsistent, and $R_1 = \{x_0 < x_i, \forall i > 0\} \cup \{x_0 = 0\}$ and $R_2 = \{x_0 > x_i, \forall i > 0\}$ are two maximal relaxations of the constraints. The number of solutions to R_1 is d^n, while there is only one

solution to R_2. The problem of selecting the maximal relaxation consistent with the largest set of solutions is intractable, in general. ▲

Example 4.2.2. More concretely, in Figure 4.1 we show the results of a simple experiment on the Renault Mégane configuration problem [1], which has been compiled in an automaton. This problem has 99 variables and about 2.8×10^{12} solutions. We built inconsistent user queries that instantiated 40 randomly chosen variables with a random value. We ran 20 such queries. For each query, we generated the complete set of maximal relaxations using the Dualize & Advance algorithm [3]. Using the automaton we could efficiently count the number of solutions consistent with each relaxation. In Figure 4.1, we plot, for each maximal relaxation, its length and the number of solutions of the problem consistent with it. It is clear from this figure that the number of solutions of a maximal relaxation is not necessarily correlated with its length. ▲

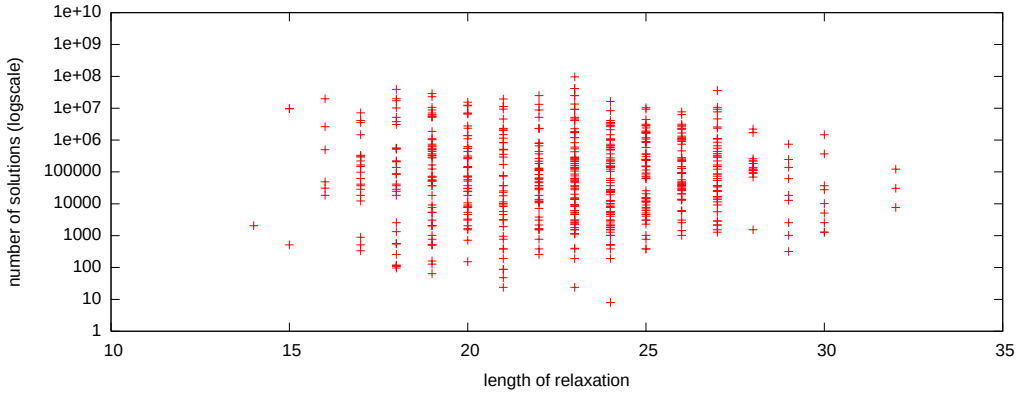


Figure 4.1: Results from a simple experiment showing that number of solutions of a maximal relaxation is not necessarily correlated with its length.

It is also important to understand how maximal relaxations partition the solutions to the background. In particular, it is important to note we cannot cover the same set of solutions with fewer maximal relaxations.

Definition 4.2.3 (Solution Set)**.** Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$, a subset $C \subseteq \mathcal{U}$, the solution set of C, denoted $sol(C)$, is the set of solutions to $\mathcal{B} \cup C$.

Proposition 1 (Disjoint Solutions between Maximal Relaxations)**.** *Given a constraint problem* $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ *that is inconsistent, let* R_1 *and* R_2 *be two maximal relaxations of* $\mathcal{P}$*. We have* $sol(R_1) \cap sol(R_2) = \emptyset$*.*

Proof. Let $s \in sol(R_1)$ be a solution consistent with R_1. Suppose $s \in sol(R_2)$ too. In particular, $s \in sol(R_2 \setminus R_1)$, i.e. s must be consistent with any constraint c in $R_2 \setminus R_1$. As R_1 and R_2 are incomparable, $R_2 \setminus R_1$ is not empty, and thus there must exist at least one such constraint c. Then $R_1 \cup \{c\}$ has at least one solution s, which is in contradiction with R_1 being maximal. □

This tells us that, from the moment we decide not to consider some maximal relaxations, we are necessarily excluding some potential solutions. In other words, if we decide not to show the user all maximal relaxations, we are preventing him access to some solutions. As it is indeed unreasonable to show all possible maximal relaxations, as it has been discussed in the previous chapter, in order to maximise acceptance of a maximal relaxation, care has to be taken not only concerning the constraints it involves, but more generally the solutions it allows. However, it is not possible or desirable to reason at the solution level. On the one hand, we have to note that some solutions to the background – i.e. some possible products – will be left out by any maximal relaxation, and, therefore, if the user wants to reach one of these solutions, more user constraints, if not all, have to be relaxed. This is not an option however, as we do not want to give up on the notion of maximal relaxation, i.e. on the idea that we want to satisfy as many user constraints as possible. On the other hand, even if the desirable solutions are actually covered by some maximal relaxations, we have to keep in mind that the number of solutions covered by any maximal relaxation can sometimes be very high. This implies that it is not possible to show all the solutions of a maximal relaxation for the purpose of assisting the user in his decision of accepting the maximal relaxation or not.

This leads to the idea of aggregating the information on solutions by considering instead the *number of solutions*. More specifically, we propose to rank maximal relaxations depending on the number of solutions they allow, or their solubility for short. The solubility can serve as a preference metric: the higher the number of solutions for some maximal relaxation, the more preferred it is,

in the sense that it leaves the user with the largest choice. On the other hand, a low solubility has also an interest. In their work on partial constraint satisfaction, Freuder and Wallace [53] defined the distance between two problems by the number solutions not shared between the two problems. The problem containing the relaxation with lowest solubility is, according to this metric, the closest to the original inconsistent problem. In this regards, it is the closest to the user's initial intentions.

4.2.1 The Basics of Automaton-based Configuration

As we explained in Section 2.4.4, Chapter 2, in a configuration context, a typical approach is to compile the problem into a minimal deterministic finite state automaton (MDFA), or simply an automaton, in order to facilitate interactive solving [134, 1].

Different operators are efficiently implemented on problems represented as MDFAs [134, 1]. To give a few examples, given an automaton, one can: compute its negation (i.e. the automaton representing the negation of the problem represented by the input automaton); count the number of solutions of the problem it represents; answer entailment and consistency queries; and given two automata, one can compute the conjunction or the disjunction of the two automata. By efficiently, we mean that the running times of those operators depend only in the size of the automata themselves, and not the size of the problem they represent. On instances where the reduction factor between the size of the automaton and the number of solutions of the problem it represents is exponential, this means the operators will run exponentially faster than a naive algorithm.

Using these operators it is very easy to compile a problem consisting only of table constraints, i.e. constraints given in terms of their list of allowed or forbidden tuples. For a given constraint, we can compile it by converting each of its tuples into an automaton (in a straightforward way) and then applying the disjunction between each of them, and negating the result in the case of constraints defined with forbidden tuples. Then, all the constraints can be combined by applying the conjunction operator. Note that the size of the output automaton of the conjunction and disjunction automaton can theoretically be exponentially related to the size of

the input automata. Compilation to an MDFA works well for those cases where, in practice, the automaton obtained by the successive incorporation of constraints through conjunction increases in a tractable way.

4.2.2 Compiling a Single Constraint Into an MDFA

Relying on a constraint being extensionally defined is often too restrictive. In fact, a constraint might have a very high number of solutions, while still having a good automaton representation. A good compromise can be to rely simply on an arc-consistent propagator. In that case, an automaton can be greedily constructed, by adding transitions level by level, each of them being guaranteed to reach a final state (the automaton is indeed representing the backtrack-free search-tree). Additionally, it is possible to maintain a minimal automaton at each stage, thus avoiding an intermediate result having a size equal to the number of solutions of the constraint. That procedure is still not ideal, as we would like to operate at a declarative level, and not at the resolution level, to which propagation pertains. In this situation, the user can specify a constraint at the modelling level, which, for example, could be modelled with a set of intensional constraints, and the compiler will build an automaton that represents it. As a side effect, this has the added benefit of providing a generic arc-consistent propagation procedure for constraints that do not already have one, that will be efficient when the resulting automaton has a reasonable size (particularly against generic GAC schemes operating on table constraints [6]). We show an algorithm that performs this task, using the ideas introduced by Daciuk et al. [17].

Algorithm 5 compiles a single constraint into an automaton. As we want to keep the space usage as low as possible, not only in the end but also during the computation, the algorithm always maintains an (almost) minimal automaton.

In order to obtain the solutions of $\mathcal{C}$, we must solve a problem $\mathcal{P}$ consisting (semantically at least) of the single constraint $\mathcal{C}$, and search for all its solutions. For that, we are only assuming that we have one way of solving this constraint, and that solutions can be output in increasing lexicographic order. For example, this could mean that we have an equivalent decomposition, or that we have a propagator that achieves some weaker level of consistency. In practice however,

Algorithm 5: CONSTRAINTCOMPILATION

Data: A constraint $\mathcal{C}$
Result: A minimal deterministic automaton representing all the solutions of $\mathcal{C}$

$\mathcal{P} \leftarrow \langle scope(\mathcal{C}), \{\mathcal{C}\}\rangle$
$\mathcal{A} \leftarrow$ empty automaton with only an initial state q_0
while *exists next solution of* $\mathcal{P}$ **do**
  $t \leftarrow$ next solution of $\mathcal{P}$ in increasing lex. order
  $t_p \leftarrow$ common prefix in $\mathcal{A}$
  $q \leftarrow \delta(q_0, t_p)$
  $t_s \leftarrow t \setminus t_p$
  if $hasChildren(q)$ **then** `MinimiseLastWord(`q`)`
  $addSuffix(q, t_s)$
`MinimiseLastWord(`q_0`)`

function `MinimiseLastWord(`q`)`
  $qLast \leftarrow lastChild(q)$ `// the successor with the biggest value`
  $i \leftarrow level(qLast)$
  if $hasChildren(qLast)$ **then** `MinimiseLastWord(`$qLast$`)`
  if $\exists q'' \in Q(i)$ *such that* $equiv(q'', qLast)$ **then**
    $lastChild(q) \leftarrow q''$
    $delete(qLast)$

in a typical CP solver, most constraints will have an arc-consistent propagator, so that resolution will consist indeed in a single backtrack-free traversal of the solution tree. In particular, for a table constraint, this algorithm will effectively be equivalent to adding each tuple one by one to the current automaton.

The invariant that is kept during the procedure is that, before the $addSuffix$ call, the current automaton is minimal. After the suffix is added (as a new disjoint path in the automaton), at the beginning of next iteration, we know that nothing will change that will affect state equivalence, for the part of this last suffix starting from q to the end of the automaton. Therefore, at that point, we can merge the states on this path with equivalent existing ones, which is what the `MinimiseLastWord` function does.

4.2.3 Tree-Driven Automata

We briefly described how CSPs can be compiled into MDFAs. Unfortunately, even if MDFAs reduce the space required to represent the solution set of real-world configuration problems, to levels that make it an acceptable compilation strategy, they still suffer from extremely high memory usage and long compilation times. Fargier and Vilarem [43] introduced the concept of tree-driven automata to improve the achieved compactness. Roughly, the idea is, instead of representing the whole solution set of the problem, to exploit independence between parts of the problems in order to represent smaller parts of the problems and only encode how these parts combine to each other[1]. This is particularly relevant to configuration problems, which are indeed structured in terms of loosely interconnected components, as we noted in Chapter 2. For all practical applications throughout this dissertation, we used our own-developed CSP compiler, which compiles any problem modelled in Choco [87] to a tree-driven automaton. For this reason, we will briefly and informally present this concept here, illustrating the concepts with the help of Figure 4.2.

Example 4.2.3 (Tree-Driven Automaton)**.** Consider a problem defined on three variables X_1, X_2, X_3, with two constraints $X_1 = X_2$ and $X_1 \Rightarrow X_3$. There are three solutions to this problem: 000, 001, 111. Figure 4.2 shows a support for this problem, and the tree-driven automaton representation of this problem for this support. The dashed part of the tree-driven automaton corresponds to the solution 001.

Definition 4.2.4 (Support)**.** A *support* for a problem is a directed tree, the edges of which are labelled by the variables of the problem (where each variable is associated with exactly one edge of the tree). This tree gives rise to a partial order $\leq$ between the variables of the problem, where $X_i \leq X_j$ iff there is a path from the edge labelled with X_i to the edge labelled with X_j. We impose an additional restriction that for each constraint, the variables in its scope must be

[1]The idea is conceptually similar to the idea, in databases, of breaking large database (i.e. holding on a large set of variables) to smaller ones, by identifying *functional dependencies*, thus avoiding redundant information being copied to many tuples in the original database – the original database can still be generated by the join of the smaller tables without information loss [71].

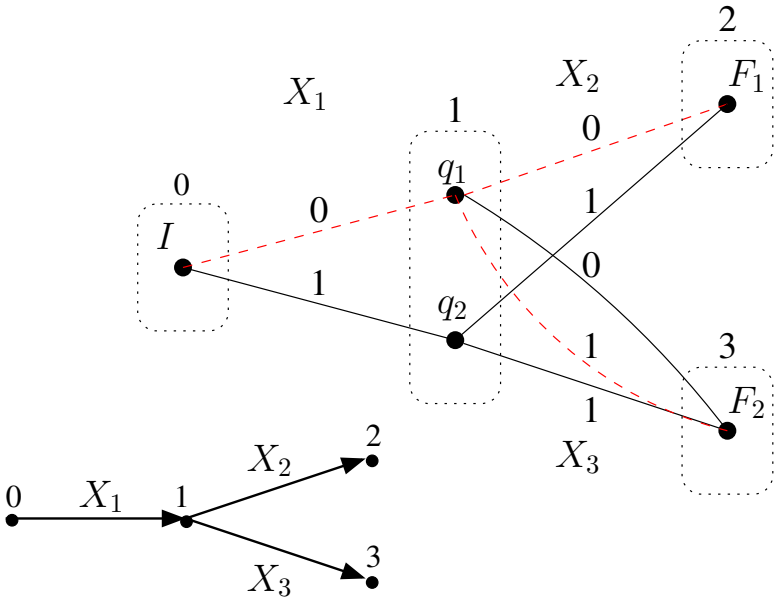


Figure 4.2: An example tree-driven automaton and its associated support.

totally ordered by the order $\leq$, or, in other words, that all the edges labelled with the variables of the constraint must be on one path of the tree.

In the example at Figure 4.2, the support is a tree with four nodes, rooted at node 0, such that $X_1 \leq X_2$ and $X_1 \leq X_3$. X_1, X_2 are on the same path indeed, and X_1, X_3, the scope of the second constraint, are on a same path too, but different from the first. X_2 and X_3 are incomparable, and they indeed do not appear in any common constraint.

Note that this generalises the concept of automaton previously introduced, in that it does not require a total order to be defined between all the variables of the problem, but instead only a partial order satisfying some additional conditions.

Definition 4.2.5 (Tree-Driven Automaton)**.** A *tree-driven automaton* for a problem and an associated support is a labelled directed graph such that:

- Every node (or state, to follow automaton terminology) corresponds to a node of the support,
- there is an edge (or transition) between two nodes only if there is an edge between the corresponding nodes in the support, and

- every solution to the problem represented by the tree-driven automaton corresponds to a subgraph of the tree-driven automaton that is isomorphic with the support.

This generalises the concept of automaton previously introduced, which can be referred to as a linear automaton (the support of which contains just one single path). The level of a state is the node of the support it corresponds to. The level of a transition is the level of its destination state. Every node in the support other than the root corresponds to a unique variable.

In the example at Figure 4.2, state I is of level 0, corresponding to the root 0 of the support, and is therefore the initial state. State F_1 and F_2 are of level respectively 2 and 3, corresponding respectively to the leaf nodes 2 and 3 of the support, and are therefore final states. q_1 and q_2 are both of level 1. The transition between I and q_1, labelled by 0, is of level 1, and therefore supports $X_1 = 0$. The transition between q_1 and F_2 labelled by 0 supports $X_3 = 0$. The dashed subgraph of Figure 4.2 corresponds to the solution 001.

It is not easy to see on simple examples how tree-driven automata allow for a more compact representation than classic linear automata. We show in Figure 4.3 the constraint graph of a simple problem that acts as a basic pattern. Bigger problems can be recursively defined by appending by its root a copy of the basic pattern to any leaf node. These problems admit a straightforward support that allow for a tree-driven automaton representation that grows polynomially in size as the problem size grows, whereas any linear automaton representation (i.e. for any total order of the variables of the problem) grows exponentially in size as the problem size grows.

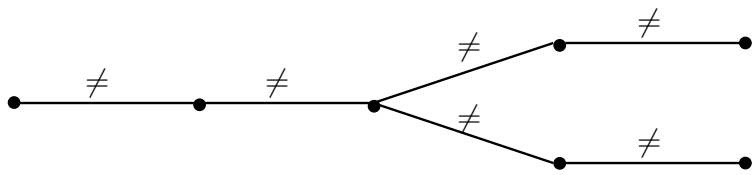


Figure 4.3: A problem that illustrates the compactness of tree-driven automata against linear automata

All the usual properties on automata, such as determinism and minimality, can be generalised to tree-driven automata in a straightforward way. Additionally, all

usual operators on automata, such as union and intersection, can be generalised to tree-driven automata. Therefore, the compilation procedure can operate as described for linear automata, once a support has been extracted.

In practice, it is easy to extract a support from a CSP that is *acyclic*, by converting its join-tree to a support (see Fargier and Vilarem [43] for the procedure). The compilation procedure will be straightforward and will have a worst case running time and result size that is exponential only in the tree-width of the CSP, but polynomial in the size of the CSP (which is an effect of a well-known result about the tractability of acyclic CSPs [50]). For non-acyclic CSPs, one has first to find a cluster-tree decomposition of the constraint graph – as first explained by Dechter and Pearl [34]. The compilation can be performed on the resulting acyclic CSP as just outlined, and the performance will depend on the width of the decomposition. It is therefore important to find a good decomposition, i.e. one with the lowest possible width, or equivalently that is the closest to the tree-width of the CSP. As we mentioned in Section 2.6.1 of Chapter 2, configuration problems are structured, in the sense that they have a low tree-width. Therefore, they can be efficiently compiled into tree-driven automata. It has to be noted though that it is generally intractable (it is *NP*-Hard [2]) to determine the cluster-tree decomposition with the lowest width, but good heuristics exist to find one that is close enough [132, 8, 86]. On the real-world configuration problems we compiled, we observed a reduction in size and compilation time by a factor ranging between 10 and 100.

4.3 A Unifying Framework: the Compilation Map

Although automata, and their boolean counterparts BDDs (Reduced Ordered BDDs to be exact), are becoming widely used in real-world applications, other ways of compiling and representing a problem exist. An extensive survey of such representations for propositional formulas has been carried out by Darwiche and Marquis [22], which classifies the different representations from the perspective of their compactness, while showing which operations each representation efficiently supports. The fact that this work concerns propositional formulas means it does not apply directly to our framework. However, it does capture the complexity, com-

pactness and representation power of each language, and gives a methodology to compare and abstract different CSP compilation methods by relating them to languages in the Compilation Map. To quote Darwiche and Marquis, "*[this map] also provides an example paradigm for studying and evaluating further target compilation languages*" [22], which is precisely our motivation for introducing their work here.

After presenting the Compilation Map, we will study, in this rest of this chapter and dissertation, how non-boolean compilation methods relate to the languages introduced in the Compilation Map, and conversely how some languages introduced in this map can be generalised to our setting.

4.3.1 Compiling Propositional Formulas

The Compilation Map [22] presents a set of compiled representations of a problem, referred to as *target compilation languages*. Only boolean problems are considered. The most general language is **NNF** (for Negation Normal Form). A *sentence* Σ in **NNF**, holding on the set of variables $vars(\Sigma)$, can be represented as a directed acyclic graph (DAG), where the leaf nodes are labelled with $\top$ (true) or $\bot$ (false), X, $\neg X$ (with $X \in vars(\Sigma)$), and the internal nodes are labelled with $\wedge$ or $\vee$, and can have arbitrarily many children.

Example 4.3.1. Figure 4.4 shows an example (taken from Darwiche [19]) of a DAG representing an NNF for the odd-cardinality function [2]. This function holds on four boolean variables X_1, X_2, X_3 and X_4 and is satisfied when there is an odd number of variables set to true. The formula reads as a big disjunction of the form: $((X_1 \wedge \neg X_2) \vee (\neg X_1 \wedge X_2)) \wedge ((X_3 \wedge X_4) \vee (\neg X_3 \wedge X_4)) \bigvee \cdots$. ▲

Additionally, some key properties are defined that a sentence can satisfy.

decomposability An NNF satisfies *decomposability* if the children of every and-node hold on disjoint sets of variables, *i.e.* if for every and-node $\wedge_i \alpha_i$, $vars(\alpha_i) \cap vars(\alpha_j) = \emptyset$, for $i \neq j$. Every and-node of Figure 4.4 satisfies this property and therefore this sentence satisfies decomposability.

[2] We use a bold-face font to refer to a language, e.g. **NNF**, while a regular font is used to denote a formula in a specific language, e.g. NNF.

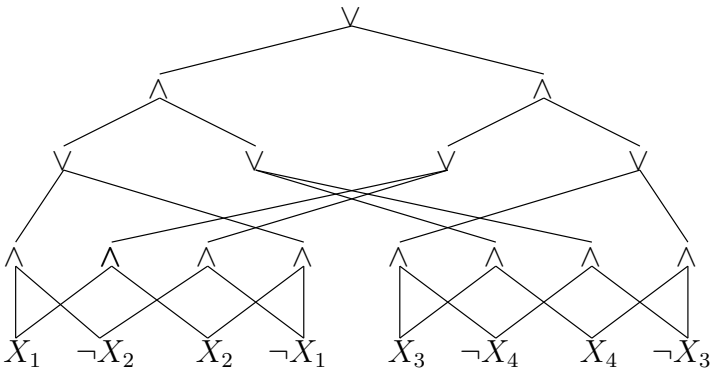


Figure 4.4: An example NNF for the odd-cardinality function.

determinism An NNF satisfies *determinism* if the children of every or-node are mutually inconsistent, *i.e.* if for every or-node $\vee_i \alpha_i$, $\alpha_i \wedge \alpha_j \models \bot$, for $i \neq j$. Every or-node in Figure 4.4 satisfies this property and therefore this sentence also satisfies determinism.

smoothness An NNF satisfies *smoothness* if the children of every or-node hold on the same set of variables, *i.e.* if for every or-node $\vee_i \alpha_i$, $vars(\alpha_i) = vars(\alpha_j)$, for $i \neq j$. The sentence of Figure 4.4 also satisfies smoothness. It is important to mention that smoothness can be enforced on any NNF. In fact, any child α_i of an or-node $C = \vee_i \alpha_i$ such that $vars(\alpha_i) \subset vars(C)$ can be replaced by $\alpha_i \wedge (\wedge_{X \in vars(C) \setminus vars(\alpha_i)} (X \vee \neg X))$. This preserves equivalence, decomposability, determinism, and results in a new sentence whose size is $\mathcal{O}(|\Sigma|.|vars(\Sigma)|)$. From a practical perspective, this means that any language is equivalent to its smooth version as far as its succinctness and the satisfiability of queries are concerned.

Classic Reduced Ordered BDDs [10], which are effectively equivalent to automata restricted to boolean domains, can be represented under this framework by imposing some additional properties, which essentially describe the form of or-nodes. Figure 4.5 shows a BDD in both its usual representation (where a plain line stands for a transition of value 1 and a dashed line for a transition of value 0) and its NNF representation, where the decision nodes of the BDD are encoded by an or-node of the form $(X \wedge \alpha) \vee (\neg X \wedge \beta)$. On any path from the root to a terminal

node of a BDD, a variable should appear only once. This can be obtained by enforcing decomposability on the associated NNF. Finally the usual properties can be described in terms of the NNF framework to define Reduced Ordered BDDs.

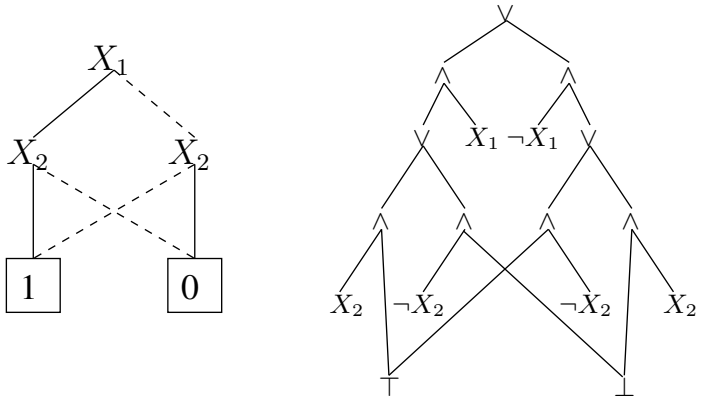


Figure 4.5: A simple BDD and its NNF representation

We introduce here some of the languages defined by Darwiche and Marquis [22]. The languages **DNNF**, **d-NNF** and **s-NNF** are the subset of **NNF** that satisfy decomposability, determinism, smoothness, respectively. The language **d-DNNF** is the subset of **DNNF** that also satisfies determinism. The languages **s-DNNF**, **sd-DNNF**, **sd-NNF** are corresponding subsets that also satisfy smoothness. The language $\mathbf{OBDD}_<$ is the language corresponding to Reduced Ordered BDDs, and is a subset of **d-DNNF**. Some other languages are worth mentioning here. **DNF** is the subset of **DNNF** containing DNF sentences, **CNF** is the set of CNF sentences, and **PI** (for prime implicates) is the subset of **CNF** of the sentences represented as the conjunction of their prime implicates. Finally, **MODS** is the language where sentences are represented by their set of models. It is the subset of **DNF** satisfying determinism and smoothness.

Those languages can be ranked according to their succinctness. Informally, a language $\mathbf{L}_1$ is at least as succinct as $\mathbf{L}_2$ if any sentence in $\mathbf{L}_2$ is equivalent to a sentence in $\mathbf{L}_1$ that is at least as small. $\mathbf{L}_1$ is strictly more succinct, or, simply, more succinct, than $\mathbf{L}_2$ if $\mathbf{L}_2$ is not at least as succinct as $\mathbf{L}_1$. This relation between languages is fundamental, as one typically wants to find the most succinct language that satisfies some desired properties. The relation between all the aforementioned languages in terms of succinctness is given in Figure 4.6, where

NNF is the most succinct language. A plain arrow links a language $\mathbf{L_1}$ to another language $\mathbf{L_2}$ when $\mathbf{L_1}$ is more succinct than $\mathbf{L_2}$. When no arrow links $\mathbf{L_1}$ to $\mathbf{L_2}$ (and no relationship can be inferred from transitivity), $\mathbf{L_1}$ is not more succinct than $\mathbf{L_2}$. When a dotted arrow links $\mathbf{L_1}$ to $\mathbf{L_2}$, the relationship is unknown. For example, **MODS** is not more succinct than **PI**, but it is unknown whether **PI** is more succinct than **MODS**, though we can easily be tempted to conjecture it is not, and see **PI** and **MODS** as the two least succinct languages.

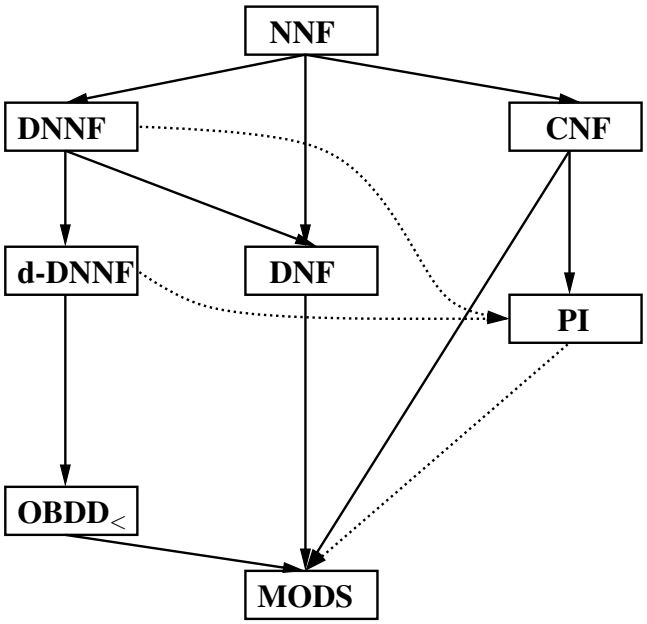


Figure 4.6: The succinctness of the different languages; more succinct languages are higher up.

While a language might be more succinct than another, it might provide for efficient answers to fewer questions. Darwiche and Marquis [22] introduce the term of query to refer to those "questions". This can be misleading, as we already introduced this term for a set of user constraints, of course, an entirely different concept. Nevertheless, we decide to preserve the terminology and we will use the same term with its two meanings, making the effort to avoid any lack of clarity and making sure the proper concept is understood.

A *query* allows us to efficiently retrieve information from a given sentence. A language satisfies the query **CO** (Consistency) if there exists a polynomial algorithm that decides the consistency of any sentence in it; it satisfies **CT** (Model

counting) if there exists a polynomial algorithm that counts the number of solutions of any sentence in it; it satisfies **ME** (Model enumeration) if there exists an algorithm that enumerates the models of any sentence in time polynomial in its size and the number of models.

A *transformation* transforms a sentence into another one. We will only recall one such transformation. A language satisfies the transformation **CD** (Conditioning) if there exists a polynomial algorithm that maps a sentence Σ of that language and a term γ to a sentence, denoted $\Sigma|\gamma$, in the same language, equivalent to $\Sigma \wedge \gamma$. Queries, and transformations, are said not to be supported when no polynomial algorithm exists unless *P=NP*.

4.3.2 Generalisation to Non-Boolean Settings

Although the Compilation Map only involves representations for propositional formulas, in our setting we are interested in CSP compilation methods. This methodology allows us to formally compare different compilation methods, by attaching them to their NNF counterpart, consisting of their restriction to boolean domains. Conversely, it can be interesting to define new compilation methods by generalising an NNF language to non-boolean domains.

Let us briefly mention the link between the previously mentioned compilation methods and the Compilation Map. Automata restricted to boolean domains correspond to Reduced Ordered BDDs, and therefore correspond to the $\mathbf{OBDD}_<$ language. Conversely, by design BDDs encode boolean functions. Non-boolean problems are dealt with either by encoding them to an equivalent boolean model, using standard boolean encodings of multi-valued variables [137], and representing it by a BDD, where the log encoding appears to give the most compact result [62], or by generalising BDDs to multivalued decision diagrams (MDDs) [82].

There is a strong connection between tree-driven automata and dDNNF. This connection is better seen by considering AND/OR Multi-valued Decision Diagrams, or AOMDDs [98]. Clearly, AOMDDs and TDAs are equivalent, and can be considered as two independently discovered and differently presented versions of the same concept. Clearly too, AOMDDs are a subset of dDNNF. However, they are a *strict* subset. An additional property has to be enforced on dDNNF,

which has been formalised as structured decomposability [105]. Very informally, this imposes and-nodes to be in correspondence with a node in a tree defining a partial order between the variables of the formula. This generalises the ordered property on BDDs, exactly in the same way the support of a TDA generalises the order of an automaton. It is interesting to note that apparently the c2d compiler [21] that compiles a CNF to a DNNF implicitly enforces determinism and structured decomposability.

Finally, the generalisation of the **PI** language to non-boolean problems will be dealt with in the next chapter.

4.4 Enriching the Compilation Map with Explanation Related Queries

In this section, we introduce new queries relevant to explanation generation. We first show in an algorithmic way how these queries can be computed on automata. We then formalise these queries in the terms of the Compilation Map formalism, and we study how to generalise our algorithms beyond automata so that more succinct representations can be used. We present some sufficient conditions that the compiled representations must satisfy in order to guarantee that efficient algorithms exist for finding the kinds of relaxations we study in this chapter.

4.4.1 Algorithms for Automata

We present two novel algorithms for finding relaxations that are consistent with either the largest or fewest number of solutions, based on an automaton representation of the configuration problem. For a particular user query, comprising a set of unary constraints that restrict the domain of each variable, a valuation $\phi(t)$ is associated with each transition t of the automaton: $\phi(t) = 0$ meaning that this transition supports a valid instantiation (i.e. is labelled by an allowed value) and $\phi(t) > 0$ meaning it does not. Thus, to each complete path from I to F there corresponds a relaxation of the user's constraints, composed of the user's constraints supported by the transitions of the path with a valuation of 0.

If we restrict the valuation of the transitions only to 1 in case of a violation, the cost of a path from the source to the sink, which is the sum of the valuations of the transitions it is composed of, corresponds to the number of user constraints violated. If no such path of cost 0 exists, then the set of user constraints is inconsistent. A procedure is described in [1] that associates with each transition t of an automaton a cost $cost(t)$ of the best path (i.e. of minimal cost) of the automaton that uses t. This allows us to explore only the shortest paths in the automaton and, thus, only the longest relaxations of the user's constraints. Therefore, this can give our first exact algorithm (Algorithm 6) that finds, amongst all the longest relaxations, the one that is consistent with the largest or the smallest number of solutions, in time linear in the size of the automaton.

Algorithm 6: Finding a most or least soluble longest relaxation.

Data: An automaton updated for a user query.
Result: An optimally soluble longest relaxation.

```
1  relax(I) ← ∅
2  nsols(I) ← 1
3  for i ← 1 to n do
4      forall q' ∈ Q(i) do
5          candidates ← ∅
6          forall t ∈ in(q') such that t is optimal do
7              q ← in(t)
8              if φ(t) = 0 then R ← relax(q) ∪ {c_i}
9              else R ← relax(q)
10             if R ∉ candidates then
11                 candidates ← candidates ∪ {R}
12                 nsols(R) ← 0
13             nsols(R) ← nsols(R) + nsols(q)
14         nsols(q') ← opt_{R∈candidates} nsols(R) // opt ∈ {max, min}
15         relax(q') ← R ∈ candidates such that nsols(R) = nsols(q')
16 return relax(F)
```

Obtaining the most, or least soluble, longest relaxation is simply a matter of selecting the appropriate relaxation at line 14. Let us consider the $\max$ case; a similar presentation can be used for the $\min$ case. Algorithm 6 associates with

each state q' of the automaton, the most soluble longest relaxation restricted from I to q' (the automaton "to the left of" q'), stored in $relax(q')$, with $nsols(q')$ storing the corresponding number of solutions. This is valid because of the following property.

Proposition 1. *After the end of the for loop starting at line 6, the value of $nsols(R)$ is the number of solutions of the part of the automaton from I to q' that supports R.*

Proof. In the implementation of the algorithm, at some state q', equal relaxations must be uniquely identified, and thus, for the same relaxation R, $nsols(R)$ takes into account all ways to reach R, *i.e.* it is the sum of the numbers of solutions of all the known occurrences of $R \cap \{c_1, \ldots, c_{i-1}\}$, which, by induction, are assumed to admit a maximal number of solutions. □

The set $candidates$, after the iteration (starting at line 6), will contain all the longest relaxations ending at q'. At this point, we can choose the most soluble relaxation, as a less soluble one could not result in a longer relaxation globally more soluble. As the size of $candidates$ is bounded by the number of states of the previous level, this algorithm runs in time linear in the size of the automaton.

However, restricting ourselves to the longest relaxation can prove to be too strong. For example, the plot in Figure 4.1 suggests that there is quite a concentration of long maximal relaxations, but very few of maximum length. Focusing on candidates amongst the maximal (by inclusion) relaxations seems to be a good trade-off between solubility and maximality. Therefore, we can adapt the previous algorithm to explore the whole automaton so as to consider all relaxations. The difference is that we cannot now greedily keep partial optimal solutions, because what is locally a maximal relaxation may not eventually be maximal. For example, in the automaton of Figure 2.10, suppose we post three unary constraints c_1, c_2, c_3 forcing every variable to be 0. Two maximal relaxations start in the second state of level 1: c_2 and c_3. The first has three solutions while the second has only two. But the first will be, at the next step, included in the relaxation c_1c_2, which has three solutions, while c_3 will still be a maximal relaxation, but with four solutions. Therefore, we need to maintain for each state the list of all the maximal relaxations. This procedure has, therefore, a complexity linear in the size

Algorithm 7: Finding all maximal relaxations.

Data: An automaton updated for a user query.
Result: All maximal relaxations and their number of solutions

$relax(I) \leftarrow \{\emptyset\}$
$nsols(I, \emptyset) \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  forall $q' \in Q(i)$ **do**
    forall $t \in in(q')$ **do**
      $q \leftarrow in(t)$
      forall $R \in relax(q)$ **do**
        if $\phi(t) = 0$ **then** $R' \leftarrow R \cup \{c_i\}$
        else $R' \leftarrow R$
        $relax(q') \leftarrow relax(q') \cup \{R'\}$
        $nsols(q', R') \leftarrow nsols(q', R') + nsols(q, R)$
    Sort $relax(q')$ by decreasing cardinality
    forall $R \in relax(q')$ **do** Remove in $relax(q')$ subsets of R
return $relax(F)$

of the automaton times the number of maximal relaxations. The corresponding modification is presented in Algorithm 7.

This is an ad-hoc procedure that lists all maximal relaxations of a query. However, being specifically designed for our context, it can be more efficient than generic algorithms, such as Dualize & Advance [3] (which can be theoretically exponential in the number of maximal relaxations), and gives, at the same time, the number of solutions of each relaxation. In this algorithm $relax(q)$ is a set of relaxations, and for each of them, say R, $nsols(q, R)$ stores its number of solutions. At lines 12 and 13, any relaxation that is a subset of another is removed, so as to keep only the maximal elements. As the size of the list $relax(q)$ is bounded by the total number of relaxations, the complexity is linear in the size of the automaton times the number of relaxations.

4.4.2 Generalisation to Other Compiled Representations

We define two new queries: **SES** (Shortest exclusion set, in Section 4.4.2) and **MR** (Maximal Relaxation, in Section 4.4.2). Table 4.1 summarises the results known

for the queries previously introduced and our contribution of two new queries. The support of those two queries by **PI** is left unanswered; it will be dealt with in the next chapter.

Table 4.1: The queries supported by the introduced languages

L	CO	CT	ME	CD	SES	MR
NNF	×	×	×	✓	×	×
DNNF	✓	×	✓	✓	✓	✓
d-DNNF	✓	✓	✓	✓	✓	✓
FBDD	✓	✓	✓	✓	✓	✓
OBDD	✓	✓	✓	✓	✓	✓
OBDD$_<$	✓	✓	✓	✓	✓	✓
DNF	✓	×	✓	✓	✓	✓
CNF	×	×	×	✓	×	×
PI	✓	×	✓	✓	?	?
MODS	✓	✓	✓	✓	✓	✓

Counting solutions

Counting the number of solutions can be efficiently performed on **d-DNNF** [19]. We recall here the function that counts the number of solutions of an sd-DNNF, adapted to take into account $\bot$ and $\top$ nodes. Note that because of smoothness, a $\top$ node can appear only as a child of an and-node, making it therefore neutral with respect to the conjunction, hence its value. Decomposability is required for case (4.5): the set of models of every child of the and-node hold on disjoint sets of variables, so they combine by simple product. Determinism and smoothness are required for case (4.6): the set of models of every child of the or-node hold on the same set of variables and are mutually disjoint, so the overall number of

solutions is simply the sum.

$$count(\top) = 1 \tag{4.1}$$
$$count(\bot) = 0 \tag{4.2}$$
$$count(l) = 1, \text{ if } \neg l \notin S \tag{4.3}$$
$$count(l) = 0, \text{ if } \neg l \in S \tag{4.4}$$
$$count(\wedge_i \alpha_i) = \prod_i count(\alpha_i) \tag{4.5}$$
$$count(\vee_i \alpha_i) = \sum_i count(\alpha_i) \tag{4.6}$$

This methodology is the skeleton to define other functions on DNNF.

Longest relaxations

Let Σ be a sentence. A set of user choices on it can be defined as a term S, *i.e.* an assignment of a subset of $vars(\Sigma)$. If $\Sigma \wedge S$ is inconsistent, then we can find a longest relaxation of S, *i.e.* a way to satisfy the largest number of assignments, or equivalently a shortest relaxation set of S, *i.e.* a way to give up on the fewest possible assignments. Formally, for a sentence Σ and a query S, $ses(\Sigma, S)$ is the size of a shortest exclusion set. In particular, $\Sigma \wedge S$ is consistent iff $ses(\Sigma, S) = 0$. Also, if Σ is inconsistent, $ses(\Sigma, S)$ is undefined. We must consequently assume that only consistent nodes are considered. Practically, because of decomposability, inconsistent nodes are only $\bot$-nodes, and-nodes that have at least one inconsistent child, and or-nodes that have only inconsistent children.

We define a new query **SES**: a language **L** satisfies **SES** iff there exists a polynomial algorithm that computes $ses(\Sigma, S)$ for every formula Σ from **L** and

every term S.

$$ses(\top, S) = 0 \tag{4.1}$$

$$ses(l, S) = 0, \text{ if } \neg l \notin S \tag{4.2}$$

$$ses(l, S) = 1, \text{ if } \neg l \in S \tag{4.3}$$

$$ses(\wedge_i \alpha_i, S) = \sum_i ses(\alpha_i, S) \tag{4.4}$$

$$ses(\vee_i \alpha_i, S) = \min_i ses(\alpha_i, S) \tag{4.5}$$

Case (4.4) only works with decomposable languages. In fact, if the exclusion sets of two children of the and-node hold on variables that do not overlap, they can be combined in a new one whose size is the sum. However, if the variables overlap, no greedy choice can be made, and different possibilities must be tested. Case (4.5) needs only smoothness: the exclusion sets of the children of the or-node hold on the same set of variables, it is just a matter of choosing the optimal one. Any language satisfying decomposability supports **SES**, as for example **DNNF**, **d-DNNF**, **OBDD**$_<$. On the other hand, this definition does not hold on non-decomposable languages.

Obviously, any language that does not support **CO** cannot support **SES**. Indeed, every language supports **CD**, and $ses(\Delta, \gamma) = 0$ iff $\Delta|\gamma$ is consistent. Notably, **CNF** does not support CO. However, **PI** is the only non decomposable language that supports consistency. It remains, therefore, a question to know whether it supports **SES** or not.

Once $ses(\Sigma, \gamma)$ is known, one or all the shortest exclusion sets can be found. The only point of choice is at an or-node: children that have a minimal value correspond to the shortest exclusion sets, and thus, we only need to explore those. If we rely on an oracle for counting the number of solutions of a given exclusion set (like a complete enumeration or an incomplete estimation), enumerating the shortest exclusion sets to pick a most soluble longest relaxation is enough. On the other hand, for languages satisfying determinism, we can also combine the computation of one shortest exclusion set with the solution counter to find a most soluble longest relaxation.

Let $mse(\Sigma, S)$ be the function that returns the number of solutions of the most

soluble shortest exclusion set. For the sake of clarity, we do not keep track of the corresponding exclusion set at each level, but of course that can be easily done. $mse(\Sigma, S)$ is defined as follows:

$$mse(\top, S) = 1 \tag{4.1}$$

$$mse(l, S) = 1, \text{ if } \neg l \notin S \tag{4.2}$$

$$mse(l, S) = 0, \text{ if } \neg l \in S \tag{4.3}$$

$$mse(\wedge_i \alpha_i, S) = \prod_i mse(\alpha_i, S) \tag{4.4}$$

$$mse(\vee_i \alpha_i, S) = \max_{i \text{ such that } ses(\alpha_i,S)=ses(\vee_i \alpha_i,S)} mse(\alpha_i, S) \tag{4.5}$$

Algorithm 6 (Section 4.4.1) is effectively a special application of this procedure for automata (compare lines 6 with the condition in Case (4.5) and 14 with the selection of the maximal child in Case (4.5)). Of course, we achieve, in the same way, a minimally soluble longest relaxation by changing the $\max$ of Case (4.5) to a $\min$.

Maximal Relaxations

We now consider the enumeration of all maximal relaxations. A language **L** satisfies **MR** if, for each Σ of **L** and each query S, there exists an algorithm that enumerates all the maximal relaxations of $\Sigma \wedge S$ in time polynomial in the size of Σ and the number of maximal relaxations. Similar to the previous case, the set of all maximal relaxations can be found in time linear in their number in all languages satisfying decomposability. Moreover, we can observe again that any language that does not satisfy **CO** does not satisfy **MR**, and thus there only remains the question about **PI**, which is not decomposable but supports **CO**.

For any decomposable sentence Σ, the function $mr(\Sigma, S)$ defined as follows.

returns the set of all maximal relaxations of $\Sigma \wedge S$:

$$mr(\top) = 1 \tag{4.1}$$
$$mr(l) = \{\{l\}\}, \text{ if } \neg l \notin S \tag{4.2}$$
$$mr(l) = \{\emptyset\}, \text{ if } \neg l \in S \tag{4.3}$$
$$mr(\wedge_i \alpha_i) = \otimes_i mr(\alpha_i) \tag{4.4}$$
$$mr(\vee_i \alpha_i) = \text{simplify}(\cup_i mr(\alpha_i)) \tag{4.5}$$

where $\mathcal{R} \otimes \mathcal{R}' = \{R \cup R' | R \in \mathcal{R} \wedge R' \in \mathcal{R}'\}$ and simplify$(\mathcal{R}) = \{R \in \mathcal{R} | \forall R' \in \mathcal{R}\ R \not\subset R'\}$, *i.e.*, we retain only set-wise maximal elements. Assuming we have determinism, we can associate with each maximal relaxation its number of solutions, exactly in the same way we did for mse (which does not depend at all on the nature of the relaxations). This way, we obtain a generalisation of Algorithm 7.

It is very interesting to note that basically this shows that, under some assumptions, we have a procedure that lists all maximal relaxations of an over-constrained problem that is theoretically more efficient than the best known one [3] (which is not linear in the number of maximal relaxations). These assumptions are that we post unary constraints, which is relevant in a configuration context, and that we have a problem compiled at least in a DNNF. Again, compilation is common in configuration, and DNNF is one of the most general, and succinct, forms to which a problem can be compiled.

4.5 Empirical Evaluation

The objective of our evaluation was to demonstrate the effectiveness of Algorithm 7 against a state-of-the-art algorithm for enumerating all maximal relaxations. We also considered two generic heuristic methods. We did not evaluate Algorithm 6 since it is an exact algorithm, linear in the size of the automaton. We based our experiments on the Renault Mégane Problem, also introduced in [1], which was compiled to a tree-driven automaton. This problem has 99 variables and over 2.8×10^{12} solutions. We built inconsistent sets of user choices that in-

stantiated 40 randomly chosen variables with a random value. We ran 20 such queries. For each of them, we generated the complete set of relaxations using the state-of-the-art Dualize & Advance algorithm [3], for finding all maximal relaxations in a constraint satisfaction context, and compared its performance against that of Algorithm 7 from this chapter. For both algorithms, we recorded the time for each to find the most satisfiable maximal relaxation.

In addition to comparing Algorithm 7 against Dualize & Advance, we compared two heuristic techniques in terms of the number of solutions of the best relaxation they found. The goal is to find efficient heuristics for deciding which user constraints to add first when building a maximal relaxation. Each heuristic chooses as its next assignment the one that would reduce the number of solutions of the remaining problem by the least (respectively, largest) amount (`minimise/maximise solution loss`). The key measurements taken in each case were the number of solutions of the relaxation found as well as the time taken to find that relaxation.

In Figure 4.7 we compare each method in terms of the solubility of the best relaxation each found based on each query; note that we sorted the queries by the solubility of the most satisfiable relaxation for the purposes of clarity. We observe that the heuristics find very good relaxations, and very often an optimal one.

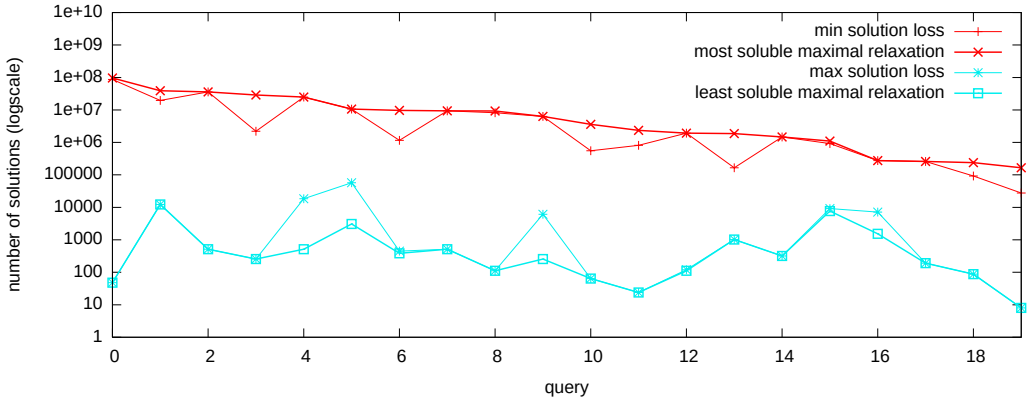


Figure 4.7: Solubility of the best relaxation found by each method.

Concerning running time, the heuristics `minimise/maximise solution loss` were prohibitively slow (i.e. not much better than a complete method). We,

Table 4.2: Running times in seconds for both algorithms

Algorithm	times (seconds) minimum	maximum	average
Dualize and Advance [3]	255	726	416
Most soluble maximal relaxation (Algorithm 2)	0.4	1.3	0.8

therefore, do not discuss time results for these. The most interesting comparison regarding time is between the Dualize & Advance algorithm and our exact algorithm (Algorithm 7) for finding the most soluble maximal relaxation. Our results, summarised in Table 4.2, show the obvious advantage of our algorithm. Not only does our algorithm guarantee that it will find the maximal relaxation consistent with the most solutions of the problem, it is over 500 times faster than a current-state-of-the-art algorithm. Also, the times required by Algorithm 7, is of the order of one second, ideal for interactive applications.

4.6 Related Work

As we saw in Chapter 2, much work has focused on finding all conflicts and all relaxations. Our approach is complementary to these by providing a basis for selecting from amongst the set of alternative explanations. Furthermore, it can be seen as a specialisation of these algorithms to the case where consistency is determined using an automaton. Furthermore, the concept of minimum cardinality (*i.e.* the minimum number of literals that are set to false in the models of a sentence), although not conceptually identical to the one of relaxations, involves similar procedures on DNNF [18]. Our work establishes a link between this work and the work on automata [1], and extends it.

Chapter 5

Towards Explanation-Oriented Compilation

Summary. *We present a new approach for compiling a problem into a form that is particularly well-suited for the context of explanations. This approach basically generalises the concept of prime implicate to constraint problems. We show how domain consequences can be used for computing different types of explanations, and explain the basics of their computation. As a proof of concept, we set up a framework under which further work will be carried out.*

5.1 Introduction

In the previous chapter, we saw how the use of common compilation techniques can, under a certain framework, drastically improve the performance of some types of explanation-related queries. However, not all explanation queries can be dealt with using classic compilation techniques. Indeed, most of compilation techniques were not conceived with explanation generation in mind. Compiled representations tend to encode, more or less explicitly, which values or sets of values are *consistent* with other values or set of values; we can characterise them as being "solution driven". However, these representations do not tend to encode which values or sets of values are *in conflict* with other values or sets of values;

we can characterise these as being "conflict driven"). However, precise and extensive information about conflicts is key for determining explanations; obviously explanations are only needed when inconsistency arises.

From this point of view, all the representations previously introduced are ruled out. Consider a very simple example: given an inconsistent query, we might simply want to find a set of conflicting constraints of minimum cardinality. This is a hard task indeed, and, as we will see, none of the previously seen representations can provide any help in this regard. Another motivating example is provided by representative sets of explanations. Deciding if a set of explanations is representative requires global knowledge of the conflicts of a problem and, as we saw in Chapter 3, this is indeed the bottleneck of the algorithm we presented. Nevertheless, the effective power of compilation in reducing computation times (after an initial time investment), and the high complexity typically associated with explanation computation, give both a strong motivation for the development for a new type of compiled representation, fulfilling the requirements set out above.

Our approach to deal with this objective will consist in computing in advance all possible conflicts, as a compilation step. Of course, this must happen in an offline phase, before a user made any choices. Therefore, the concept of "possible" conflict does not refer to the concept of conflict as used throughout this dissertation, but rather to inherent incompatibilities between values of the variables of the problem, independently from any user query. This idea, or more precisely the complementary idea, is formalised by the concept of *domain consequence*, which we define in Section 5.4.

5.2 Preliminaries

Following the practice observed throughout this dissertation, we recall first some relevant notations. We focus on problems that are solved in an interactive manner, in which we distinguish between a background set of constraints, $\mathcal{B}$, that cannot be relaxed, and a set of user constraints, $\mathcal{U}$, that are added by the user as he finds a preferred solution to $\mathcal{B}$ by finding a solution to $\mathcal{B} \cup \mathcal{U}$, the constraint problem we denote as $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$. We assume that the set of background constraints, $\mathcal{B}$, is consistent. If $\mathcal{B} \cup \mathcal{U}$ is not consistent, we can provide an explanation of the

inconsistency: a *relaxation* is a subset of $\mathcal{U}$ that is consistent with $\mathcal{B}$, an *exclusion set* is the complement in $\mathcal{U}$ of a relaxation, a *conflict* is a subset of $\mathcal{U}$ that is inconsistent with $\mathcal{B}$.

We recall the relationship between maximal relaxations, or rather minimal exclusion sets, and minimal conflicts (Chapter 3, Proposition 2). A minimal conflict is a minimal hitting set of all minimal exclusion sets, and a minimal exclusion set is a minimal hitting set of all the minimal conflicts.

5.3 Prime Implicates and Explanations

In Chapter 4, amongst all languages introduced in the Compilation Map, one in particular did not attract our attention, and this is **PI**, standing for Prime Implicates. We will concentrate on this language in this chapter, and see how it can be of interest for constraint problems, beyond proposition logic. But before, we need to recall first some basic notions from proposition logic, along with some main results from the literature.

Let Φ be a finite set of clauses in Conjunctive Normal Form (CNF) (or any NNF for the purposes of the following definitions), and let C be a disjunction of literals (a *clause*).

Definition 5.3.1 (Implicate)**.** Let C and C' be two clauses.

- C is an *implicate* of Φ if $\Phi \models C$.
- C *subsumes* C' iff $C \models C'$.

A clause that is a tautology (i.e. a clause that contains a literal in both negated and non-negated form) is a trivial implicate of any formula, and can be discarded. In the following, we will consider only non-trivial implicates, i.e. we will assume implicates to be non-trivial ones. For two non-tautology clauses, $C \models C'$ is equivalent to $C \subseteq C'$, where the inclusion refers, with a slight abuse of notation, to the inclusion of the set of literals of the clause.

A prime implicate is defined as follows.

Definition 5.3.2 (Prime Implicate)**.** C is a *prime implicate* of Φ iff:

1. C is a (non-trivial) implicate of Φ and
2. $\forall C'$ which is an implicate of Φ, $C' \models C \Rightarrow C \models C'$.

Taking into account that entailment and inclusion are equivalent for non-tautological clauses (as noted above), an implicate, when considered as a set of literals, is a prime implicate iff it is a set-wise minimal implicate, so that we can reformulate Condition 2 of Definition 5.3.2 as:

$$\forall C' \subset C, \Phi \not\models C'.$$

Finally, we introduce a notation to refer to the set of all prime implicates of a formula.

Definition 5.3.3 (Set of Prime Implicates)**.** Given a formula Φ, the set $PI(\Phi)$ is the set of all the prime implicates, i.e.

$$PI(\Phi) = \{C / C \text{ is a prime implicate of } \Phi\}.$$

Prime implicates, variants of them, and their generation, are used, amongst other things, for consequence finding. Consequence finding is a general term that refers to the task, in Artificial Intelligence, of deriving specific knowledge that is intensionally contained in a knowledge base (see [96] for a survey). But of particular concern to us is that they are of very valuable use in the context of explanations (as first noted for diagnosis [28]).

In order to illustrate that, let us first slightly adapt the previously introduced notations for the boolean case as follows. Suppose the background set of constraints is given by a single formula Φ, and a user query is a conjunction of literals γ. The literals involved in γ are denoted by $lit(\gamma)$. Suppose the query is inconsistent, i.e. $\Phi \wedge \gamma \models \bot$. From the definition of a prime implicate, $\exists C \in PI(\Phi)$ such that $lit(\neg C) \subseteq lit(\gamma)$. Indeed, $\Phi \wedge \gamma \models \bot \Leftrightarrow \Phi \models \neg\gamma$, which means that $\neg\gamma$ is an implicate of Φ. Because C is a prime implicate, $\neg C$ is also a set-wise minimal subset of the literals of γ that is inconsistent with Φ, and so corresponds to a minimal conflict of the query γ. And indeed, the subset of $PI(\Phi)$ of prime implicates the literals of which are included in those of γ is the set of all the minimal conflicts of the user query γ.

Example 5.3.1. Let Φ be a CNF of the form $(\neg a \vee b) \wedge (a \vee d) \wedge (\ldots)$. $b \vee d$ is a prime implicate. That tells us that if we want $\neg b \wedge c \wedge \neg d \wedge (\ldots)$, it is inconsistent, and a minimal conflict for that query is $\neg b \wedge \neg d$. ▲

Concerning the computation of prime implicates, most existing algorithms rely on repeatedly resolving pairs of clauses. Consider two clauses of the form $x \vee C$ and $\neg x \vee C'$. Then the resolvent $C \vee C'$ is another implicate, which has been obtained by resolving the two clauses on x. If there exists another variable y such that $y \in lit(C)$ and $\neg y \in lit(C')$, then $C \vee C'$ is a tautology, and should not be kept. This is the basis of the Tison method [133]. Note that the method described in [125] is the first one to claim a scalable method for representing and computing the prime implicates of a problem.

Prime implicates are also interesting from a compilation viewpoint. The prime implicates of a given CNF, provide a classic technique for representing this formula in a canonical way. As such, it can be seen also as a compilation technique. We presented the Compilation Map [22] in the previous chapter. Amongst the languages introduced, the one holding our attention in this context is the **PI** language, characterised as follows:

Definition 5.3.4 (**PI** language)**.** An NNF Φ belongs to **PI** iff Φ is in CNF and $Clauses(\Phi) = PI(\Phi)$, where naturally $Clauses(\Phi)$ is the set of clauses of Φ.

In the previous chapter, we explained what it means for a language to support a query, and how different languages of the Compilation Map can support different queries. It introduced some queries relevant to explanations. We recall that: **SES** is the query that asks for the size of the shortest exclusion set of a given user query; **MR** is the query that asks for the set of all maximal relaxations of a given user query. We can now add the query **SC** that asks for the size of the shortest conflict of a given query. We already explained that **PI** supports **SC**. We can now complete Table 4.1 of Chapter 4 with the following results, showing that other languages *do not support* **SC**, and showing the queries supported by **PI** PI.

Proposition 1. ***PI*** *supports* ***MR****.*

Proof. Given a query γ inconsistent with Φ, we first compute the set of minimal conflicts $\mathcal{C}(\Phi, \gamma) = \{C \in \Phi / lit(\neg C) \subseteq lit(\gamma)\}$ then compute all the minimal hitting sets of the minimal conflicts $\mathcal{R}(\Phi, \gamma)$ = $\{\gamma \setminus$

e/e is a minimal hitting set of $\mathcal{C}(\Phi, \gamma)\}$. Obviously the size of each of these two sets (as well as the time to compute them) is linear in the size of Φ times the number of maximal relaxations. □

On the other hand, **PI** does not support **SES**. Before proving this, let us first introduce some auxiliary notation.

Definition 5.3.5. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$, with $S_i \subseteq T$, be a collection of subsets of some set T. By definition, the CNF associated with $\mathcal{S}$, denoted $\Phi(\mathcal{S})$, is the CNF built such that:

- a literal x_v is introduced for every element $v \in T$;
- a clause C_i is built for every set $S_i \in S$, with $x_v \in C_i$ iff $v \in S_i$;
- $\Phi(\mathcal{S}) = \bigwedge_{i \leq n} C_i$.

Consider now the following intermediate result.

Proposition 2. *Let $\mathcal{S} = \{S_1, \ldots, S_n\}$, with $S_i \subseteq T$, be a collection of subsets of some set T, and let $\Phi(\mathcal{S})$ be the CNF associated with it. We have $\Phi(\mathcal{S}) \in$ **PI**.*

Proof. A solution of $\Phi(\mathcal{S})$ corresponds to a hitting set of $\mathcal{S}$. By definition, a (resp. minimal) clause satisfied by every solution of $\Phi(\mathcal{S})$ is a (resp. prime) implicate of $\Phi(S)$. A minimal clause satisfied by every solution of $\Phi(\mathcal{S})$ corresponds to a minimal hitting set of all the hitting sets of $\mathcal{S}$. Thus a minimal hitting set of all the hitting sets of $\mathcal{S}$ corresponds to a prime implicate of $\Phi(\mathcal{S})$. But the minimal hittings of the hittings sets of $\mathcal{S}$ are precisely the sets in $\mathcal{S}$ themselves. Thus the clauses of $\Phi(\mathcal{S})$ are all and the only prime implicates of $\Phi(\mathcal{S})$. □

Theorem 3. ***PI*** *does not support* ***SES*** *unless $P = NP$.*

Proof. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$, with $S_i \subseteq T$, be an collection of subsets of T. Consider the sentence $\Phi(\mathcal{S}) \in$ **PI**, and the user query $\gamma = \bigwedge_{v \in T} \neg x_v$, where x_v is the literal corresponding to the element $v \in T$. Clearly $\Phi(\mathcal{S}) \wedge \gamma$ is inconsistent. A shortest exclusion set gives a smallest subset of literals we must set to true in order to satisfy $\Phi(\mathcal{S})$, which corresponds to a smallest subset of elements T that is a hitting set of $\mathcal{S}$. In other words, the size of the shortest exclusion set of $\Phi \wedge \gamma$ is the size of the smallest hitting set of $\mathcal{S}$, which is NP-Hard to compute. □

Finally, we can prove that most other languages do not support **SC**.

Theorem 4. ***MODS*** *does not support* ***SC****, unless* $P = NP$.

Proof. The proof is similar to the one of Theorem 3. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$, with $S_i \subseteq T$, be a collection of subsets of some set T. Now we define Φ as the DNF $\bigvee_{i \leq n} t_i$, with $t_i = (\wedge_{v \in S_i} x_v) \wedge (\wedge_{v \notin Si} \neg x_v)$. Clearly $\Phi \in$ **MODS**. The user query $\Phi \wedge \gamma$ with $\gamma = \bigwedge_{v \in T} \neg x_v$ is inconsistent too, as it is incompatible with any of the models of the DNF Φ. A conflict of the user query corresponds to a hitting set of $\mathcal{S}$: a subset of γ is inconsistent with Φ if, for each model in the DNF, the query contains a negative literal that is positive in the model, and the positive literals of each model correspond to the elements in the corresponding set in $\mathcal{S}$. Therefore, a minimal subset of γ inconsistent with Φ corresponds to a minimal hitting set of $\mathcal{S}$. □

Theorem 5. ***OBBD***$_<$ *does not support* ***SC*** *unless* $P = NP$.

Proof. This follows from Theorem 4. A sentence in **MODS** is equivalent to a sentence in **OBDD**$_<$ of smaller or same size. Furthermore, it is polynomial to compute this sentence (with classic BDD operations). Supposing **OBBD**$_<$ supported **SC**, it would thus be polynomial to compute the size of the smallest conflict for any user query on any sentence in **MODS**, which contradicts the previous theorem. □

We can summarise these results in the extended Table 5.1.

Having given some insights into the application of prime implicates to explanations, we will now generalise these concepts to our framework, that is, define the **PI** language for constraint problems.

5.4 Domain Consequences

Suppose we have a problem defined on a set of variables $X_1, \ldots, X_n$, taking their values from the domains $D(X_1), \ldots, D(X_n)$. We are interested in simple preferences consisting of domain restrictions, expressed as unary constraints of the form $X_i \in D_i$, with $D_i \subseteq D(X_i)$, holding on a subset of the variables of the

L	CO	CT	ME	CD	SES	MR	SC
NNF	×	×	×	✓	×	×	×
DNNF	✓	×	✓	✓	✓	✓	×
d-DNNF	✓	✓	✓	✓	✓	✓	×
FBDD	✓	✓	✓	✓	✓	✓	×
OBDD	✓	✓	✓	✓	✓	✓	×
OBDD$_<$	✓	✓	✓	✓	✓	✓	×
DNF	✓	×	✓	✓	✓	✓	×
CNF	×	×	×	✓	×	×	?
PI	✓	×	✓	✓	×	✓	✓
MODS	✓	✓	✓	✓	✓	✓	×

Table 5.1: The queries supported by the different languages – extended

problem. Given an inconsistent user query, giving one or all or some, according to some criteria, minimal conflicts can be a highly intractable problem. Furthermore, compilation with any of the classic techniques, such as automata, BDDs, dDNNF, does not help either, as we saw in the previous section. It is, therefore, interesting to have a way to compute in advance all the potential minimal conflicts of a problem, or, to reuse the terminology of the literature, to infer all the consequences of the given problem.

In the absence of any particular user query, as we are working in an offline phase, we must define a new, mathematically richer, notion of minimal conflict, which we will call *domain conflicts*.

Definition 5.4.1 (Domain Conflict)**.** A *domain conflict* for given problem is given by the sequence of domains $C = \langle D_1, \ldots, D_n \rangle$, such that imposing $X_i \in D_i$ for each X_i is inconsistent. Such a conflict can be seen as a conjunction of unary constraints, which is inconsistent.

Given two domain conflicts $C_1 = \langle D_1, \ldots, D_n \rangle$ and $C_2 = \langle D'_1, \ldots, D'_n \rangle$, we say that $C_1 \subseteq C_2$ if $\forall X_i$, $D_i \subseteq D'_i$.

Definition 5.4.2 (Maximal Domain Conflict)**.** A *maximal* domain conflict is a domain conflict C such that no domain conflict $C' \neq C$ exists with $C \subseteq C'$. In other words, every component D_i of C is maximal.

Another way of seeing a domain conflict is by considering it defines the Cartesian product $\times_{i \leq n} D_i$. This product contains all the tuples that this domain conflict recognises as being inconsistent. However, defining this list of inconsistent tuples as the elements of a Cartesian product is far more informative than giving an explicit list. More specifically, a sequence of sets states that *all* the tuples that can be obtained from their product are conflicts of the problem. One implication of that observation is that a domain conflict allows us to take into account conflicts resulting from unary constraints, rather than simply variable assignments.

In this regard, we can observe that the notion of maximal domain conflict can be seen as a generalisation of the classic one of minimal conflict, in the sense that for a given i having $D_i = D(X_i)$ is equivalent to having no constraint at all on X_i. Thus, the more values that are in the D_i sets, the fewer constraints there are on the corresponding X_i variables.

As to why this definition can be regarded as being "richer" than the classic one, consider the following example. Suppose that $\langle \{a, b, c\}, \{a, b\}, \{b, c\} \rangle$ is a minimal domain conflict (with the domain of each variable being $\{a, b, c\}$). If the user asks $X_1 \in \{a, b\}$, $X_2 \in \{a\}$, $X_3 \in \{b\}$, it is inconsistent, and a minimal conflict is $c_2 c_3$ (c_i being the constraint holding on X_i). However, seen as a domain conflict, this conflict is not minimal. For example, it does not tell the user that $X_2 \in \{a, b\}$, $X_3 \in \{b, c\}$ is also inconsistent. Also, we can see that a single maximal domain conflict can cover many potential minimal conflicts resulting from a later user query. Conversely, a minimal conflict can be recognised by more than a single maximal domain conflict, thus the potential minimal conflicts covered by different maximal domain conflicts may overlap. Finally, let us point out that this notion is in its definition very close to the one of generalized nogood [83], global cut seed [47], or clausal constraints [12], but used in a very different context.

With that definition in mind, we can define the consequence of a given problem, thus generalising the concept of prime implicate.

Definition 5.4.3 (Domain Consequence)**.** A *domain consequence* of a problem is given by $P = \langle D_1, \ldots, D_n \rangle$ such that $\langle \overline{D_1}, \ldots, \overline{D_n} \rangle$, with $\overline{D_i} = D(X_i) \setminus D_i$, is a domain conflict.

Given two domain consequences P and P', whose corresponding domain con-

flicts are C and C', we have $P \subseteq P'$ if $C' \subseteq C$. To reuse classic terminology, we may say that P *subsumes* P'.

Definition 5.4.4 (Maximal Domain Consequence)**.** A domain consequence P is *minimal* if its corresponding domain conflict is maximal. In other words, no domain consequence $P' \neq P$ exists such that $P' \subseteq P$.

A given consequence of the problem is read as a disjunction. In other words, for any solution of the problem, it must be true that $X_1 \in D_1$ or $X_2 \in D_2$ or... and so on.

Definition 5.4.5 (Set of domain consequences)**.** Let Π be a problem. $Cons(\Pi)$ is the set of all the minimal domain consequences of Π, that is if P is a domain consequence of Π, then $\exists P' \in Cons(\Pi)$ such that $P' \subseteq P$, and $\forall P, P' \in Cons(\Pi)$, $P \subseteq P' \Rightarrow P = P'$.

A set of domain consequences must be seen as a conjunction, and so $Cons(\Pi) \equiv \Pi$. The strategy will thus be, given a problem, to compute all of its consequences, as efficiently as possible, and represent them as compactly as possible. In other words, we intend to compile a problem to a new representation, in a manner orthogonal to the existing approaches like automata, that supports queries related to conflicts (e.g. finding shortest conflicts, minimal conflicts, specific subsets of all the conflicts with a completeness guarantee).

By representing a problem by minimal consequences that characterise all potential consequences (or, equivalently, all potential conflicts), we soundly and completely characterise this problem. The main focus in this chapter is to introduce the concepts and the approaches we propose. How the consequences of a problem can be compactly represented, and how we can operate on this representation to compute the consequences or to perform the subsequent online queries is addressed in the next chapter.

We show of how domain consequences can help answer conflict-related queries. Suppose we are given the user query $\mathcal{U}$, where each constraint $c \in \mathcal{U}$ is of the form $X \in D$, with $D \subseteq D(X)$. We additionally assume that every unary constraint holds on a distinct variable. From that query, we can build the domain sequence $S = \langle D_1, \ldots, D_n \rangle$, where $D_i = D$ if there is a constraint $X_i \in D$

in $\mathcal{U}$, or $D_i = D(X_i)$ otherwise. To see whether $\mathcal{U}$ is inconsistent, we simply have to check if S is a domain conflict, by checking if its associated domain consequence is subsumed by a minimal domain consequence of the problem. Let $C = \langle D'_1, \ldots, D'_n \rangle$ be a maximal domain conflict such that S is included in it. The set $\{c \in \mathcal{U}/c \text{ holds on } X_i \wedge D'_i \neq D(X_i)\}$ is a minimal conflict of the query. If we want a smallest conflict of the query, we only have to iterate over all maximal domain conflicts and pick the one the corresponding minimal conflict of which is of minimal cardinality. Additionally, when looking for specific subsets, i.e. satisfying a given property, of all minimal conflicts of a query, reasoning on the set of minimal domain consequences allows us to efficiently make consistency checks as well as guarantee the desired property, without generating online all the minimal conflicts of a specific user query.

5.5 Computation of all Domain Consequences

5.5.1 Generation

Similar to the way implicates can be deduced from other prime implicates, new domain consequences can be deduced from existing consequences.

Definition 5.5.1. Let $P_1 = \langle D_1, \ldots, D_n \rangle$ and $P_2 = \langle D'_1, \ldots, D'_n \rangle$ be two domain consequences of a given problem. Then, for any X_i, we denote $res_{X_i}(P_1, P_2) = \langle D_1 \cup D'_1, \ldots, D_i \cap D'_i, \ldots, D_n \cup D'_n \rangle$

Proposition 1. *Let $P_1 = \langle D_1, \ldots, D_n \rangle$ and $P_2 = \langle D'_1, \ldots, D'_n \rangle$ be two domain consequences of a given problem. Then, for any X_i, $res_{X_i}(P_1, P_2)$ is also a domain consequence of the problem.*

Proof. Let us fix, without loss of generality, $i = 1$. P_1 tells us that $X_1 \notin D_1 \Rightarrow X_2 \in D_2 \vee \ldots \vee X_n \in D_n$. Similarly, $X_1 \notin D'_1 \Rightarrow X_2 \in D'_2 \vee \ldots \vee X_n \in D'_n$. In either case, we have $(X_1 \notin D_1 \vee X_1 \notin D'_1) \Rightarrow (X_2 \in D_2 \cup D'_2 \vee \ldots \vee X_n \in D'_n \cup D_n)$, i.e. $(X_1 \in D_1 \cap D_2) \vee (X_2 \in D_2 \cup D'_2) \vee \ldots \vee (X_n \in D'_n \cup D_n)$. □

We can say, reusing the existing terminology, that P has been inferred on X_i from P_1 and P_2. Note that if $D_i \cap D'_i$ is empty, the consequence becomes

independent from the variable X_i, thus retrieving the classical notion of resolvent in the boolean setting.

Some consequences are trivial and therefore useless:

- D_i and D'_i have to be incomparable; if one is included in the other, say $D_i \subseteq D'_i$, the consequence that is obtained will necessarily be non-minimal, as the one containing D_i will be included in it. For example, suppose we have $\langle ab, ab, c\rangle$ and $\langle a, a, cd\rangle$, we can infer, on X_1, the consequence $\langle a, ab, cd\rangle$, which will not be minimal as $\langle a, a, cd\rangle$ is included in it. This condition is equivalent to the fact that the resolvent of two clauses exists only when one contains some literal X and the other $\neg X$.

- if some $D_j \cup D'_j$, for any $j \leq n$, contains all values allowed for X_j, the inferred consequence is always true. For example, suppose we have $\langle ab, ab\rangle$ and $\langle bc, bc\rangle$, with $D_1 = D_2 = abc$, we can infer on X_1 the consequence $\langle b, abc\rangle$, which is trivially true. This condition is equivalent to the fact that two clauses that are resolved on some literal and that also contain another literal in both the negated and non-negated form will result in a trivial clause.

The following example will help illustrate the concepts of conflicts, consequences, minimal consequences, resolution, as well as the representation capabilities of these concepts.

Example 5.5.1. Suppose we have a constraint on three variables X_1, X_2, X_3 with domain $\{a, b, c\}$, given by the nogoods in Table 5.2. We can notice that a nogood corresponds to a trivial domain conflict. For example, knowing that aaa is forbidden is equivalent to saying that $X_1 \neq a \vee X_2 \neq a \vee X_3 \neq a$. The domain consequence thus corresponding to each nogood is given in the second column of the table.

From this set, different new consequences can be inferred, thus resulting in the set of minimal domain consequences for that constraint show in Table 5.3, along with their corresponding domain conflict.

The consequences that we end up with somehow "factorise" common parts amongst existing consequences. For example, the consequence $\langle bc, ac, a\rangle$ has been obtained from the two initial consequences $\langle bc, ac, ac\rangle$ and $\langle bc, ac, ab\rangle$ (on

nogood	domain consequence
aaa	$\langle bc, bc, bc\rangle$
aac	$\langle bc, bc, ab\rangle$
abb	$\langle bc, ac, ac\rangle$
abc	$\langle bc, ac, ab\rangle$
bab	$\langle ac, bc, ac\rangle$
bbb	$\langle ac, ac, ac\rangle$

Table 5.2: An example constraint

minimal consequence	maximal domain conflict
$\langle c, ac, ac\rangle$	$\langle ab, b, b\rangle$
$\langle bc, c, ab\rangle$	$\langle a, ab, c\rangle$
$\langle ac, c, ac\rangle$	$\langle b, ab, b\rangle$
$\langle bc, bc, b\rangle$	$\langle a, a, ac\rangle$
$\langle bc, ac, a\rangle$	$\langle a, b, bc\rangle$

Table 5.3: The minimal domain consequences of the problem introduce in Table 5.2

the variable X_3). In terms of conflicts, we obtain the conflict $\langle a, b, bc\rangle$, which factorises the common prefix of the two conflicts abb and abc. That is, of course, very similar to what automata perform, with some differences. First, automata merge only common prefixes, while a conflict resulting from two other conflicts might "factorise" the common part of an arbitrary subset of the variables, and is, therefore, independent of any order on the variables.

But most importantly, automata are merely just a compact representation of a list of tuples. As previously stressed, this does not allow us to easily answer queries consisting of domain restrictions. Consider a problem where the forbidden tuples are aaa, aab, baa. By inferring domain conflicts, we eventually obtain the two maximal domain conflicts $\langle a, a, ab\rangle$ and $\langle ab, a, a\rangle$. Suppose we required the three variables to be assigned the value a. While an automaton representing these nogoods would easily determine that the query is inconsistent no matter what value we assign to X_3, it would not easily show that the query is also inconsistent regardless of the value we assign to X_1. In either case, it is also not obvious that these two subconflicts are minimal. ▲

Having considered the production rules, we can now present how consequences can be initialised. The basic idea is to start by representing each constraint as the set of its minimal domain consequences. We already saw that trivial domain consequences can be initialised from the nogoods of a constraint. However, most problems that are to be compiled, especially in the context we will eventually address, define constraints as their lists of valid tuples, and it would be intractable to first generate all the corresponding nogoods. Our working assumption will, therefore, be that we are given the valid tuples of each constraint.

Definition 5.5.2. Let $\mathcal{P}_1, \mathcal{P}_2$ be two sets of minimal domain consequences, $\mathcal{P}_1 \cup \mathcal{P}_2 = \mu\{P_1 \cup P_2 / P_1 \in \mathcal{P}_1, P_2 \in \mathcal{P}_2\}$, where $\mu\mathcal{P}$ is the subset of $\mathcal{P}$ where subsumed elements have been removed.

It is quite easy to see that $\mathcal{P}_1 \cup \mathcal{P}_2$ contains the minimal domain consequences equivalent to the logical disjunction between $\mathcal{P}_1$ and $\mathcal{P}_2$. The following proposition shows how minimal consequences are built by iteratively taking into account the tuples of a constraint. Slightly abusing notation, for a list of tuples T, we use $Cons(T)$ to denote $Cons(C)$ (which in turn is an abuse of notation), where C is the constraint defined by the valid tuples T.

Definition 5.5.3. Let $t = a_1 \dots a_n$ be a tuple defined on the variables $X_1, \dots, X_n$. For each a_i, we define the consequence $P(a_i) = \langle D_1, \dots, D_n \rangle$ as follows:

$$D_j = \begin{cases} \{a_i\}, & \text{if } i = j \\ \emptyset, & \text{otherwise.} \end{cases}$$

We thus associate with tuple t the set of domain consequences $\mathcal{P}(t) = \{P(a_i), i \leq n\}$.

Proposition 2. *For a list T of tuples, we have:*

$$Cons(T) = \begin{cases} \emptyset, & \textit{if } T = \emptyset \\ \mu(Cons(T \setminus \{t\}) \cup \mathcal{P}(t)), & \textit{otherwise.} \end{cases}$$

Example 5.5.2. Consider a constraint with two valid tuples 012 and 021. We have:

$$\begin{aligned} Cons(012) = \mathcal{P}(012) = &\langle\{0\},\emptyset,\emptyset\rangle,\\ &\langle\emptyset,\{1\},\emptyset\rangle,\\ &\langle\emptyset,\emptyset,\{2\}\rangle \end{aligned}$$

$$\begin{aligned} Cons(012) \cup \mathcal{P}(021) = &\langle\{0\},\emptyset,\emptyset\rangle, \langle\{0\},\{2\},\emptyset\rangle, \langle\{0\},\emptyset,\{1\}\rangle,\\ &\langle\{0\},\{1\},\emptyset\rangle, \langle\emptyset,\{1,2\},\emptyset\rangle, \langle\emptyset,\{1\},\{1\}\rangle,\\ &\langle\{0\},\emptyset,\{2\}\rangle, \langle\emptyset,\{2\},\{2\}\rangle, \langle\emptyset,\emptyset,\{1,2\}\rangle \end{aligned}$$

and finally:

$$\begin{aligned} Cons(012,021) = &\langle\{0\},\emptyset,\emptyset\rangle,\\ &\langle\emptyset,\{1,2\},\emptyset\rangle,\\ &\langle\emptyset,\{1\},\{1\}\rangle,\\ &\langle\emptyset,\{2\},\{2\}\rangle,\\ &\langle\emptyset,\emptyset,\{1,2\}\rangle \end{aligned}$$

Although the size of the final set is greater than the number of tuples, it is smaller than the number of nogoods (i.e. 25), and so is the size of the intermediate set, that is before removal of non-minimal domain consequences. ▲

5.5.2 Algorithm

We present an algorithm for compiling a CSP as its set of all minimal domain consequences as Algorithm 8. In order to establish the correctness of the algorithm, let us first introduce some notation and results.

Definition 5.5.4. Let $\mathcal{P}$ be a set of domain consequences, and X_i be a variable. P is a domain consequence of $\mathcal{P}$ on X_i if $\exists P_1, P_2$ such that $P = res_{X_i}(P_1, P_2)$, where P_1 and P_2 either belong to $\mathcal{P}$ or are themselves consequences of $\mathcal{P}$ on X_i.

Definition 5.5.5. Let $\mathcal{P}$ be a set of domain consequences, and X_i be a variable. $Res_{X_i}(\mathcal{P})$ is the set containing all the minimal domain consequences that can be inferred on X_i from $\mathcal{P}$. Formally, $\forall P$ that is a domain consequence of $\mathcal{P}$ on X_i, $\exists P' \in Res_{X_i}(\mathcal{P})$ such that $P' \subseteq P$, and $\forall P, P' \in Res_{X_i}(\mathcal{P})$, $P \subseteq P' \Rightarrow P = P'$.

Example 5.5.3. Let $\mathcal{P} = \{\langle ab, bc\rangle, \langle ac, bc\rangle, \langle bc, bc\rangle\}$. $\langle ab,\ bc\rangle$, $\langle ac,\ bc\rangle$, $\langle bc,\ bc\rangle$, $\langle a,\ bc\rangle$, $\langle b,\ bc\rangle$, $\langle c,\ bc\rangle$ and $\langle \emptyset, bc\rangle$ are all domain consequences of $\mathcal{P}$ on X_1, and $Res_{X_1}(\mathcal{P}) = \{\langle \emptyset, bc\rangle\}$. This tells us that given $\mathcal{P}$, we can infer that a is forbidden for X_2. ▲

Definition 5.5.6. Let $\mathcal{P}$ be a set of domain consequences, and $[X_1, \ldots, X_k]$ be a sequence of variables. $Res_{[X_1,\ldots,X_k]}(\mathcal{P}) = Res_{X_k} \circ \cdots \circ Res_{X_2} \circ Res_{X_1}(\mathcal{P})$ (where $\circ$ represents function composition).

A very well-known result used in most algorithms for PI generation (as first established in [85]) can be formulated in our context as follows:

Proposition 3. *Let $\mathcal{P}$ be a set of domain consequences, X_i and X_j two variables. Then $Res_{[X_i,X_j,X_i]}(\mathcal{P}) = Res_{[X_i,X_j]}(\mathcal{P})$.*

In other words, it is useless for new consequences inferred on some variable to be tested again with another consequence on a variable that has already been considered. This is a direct consequence of the associativity and the commutativity of the res operation. Indeed, $res_{X_j}(P_3, res_{X_i}(P_1, P_2)) = res_{X_i}(P_2, res_{X_j}(P_1, P_3))$. This result is of huge computational importance, as, by fixing in advance an order on the variables, a given consequence will be discovered in a unique way.

Example 5.5.4. Let $\mathcal{C} = \{\langle a, \emptyset, a\rangle, \langle b, a, \emptyset\rangle, \langle \emptyset, b, b\rangle\}$, $Res_{X_1}(\mathcal{C}) = \mathcal{C} \cup \{\langle \emptyset, a, a\rangle\}$, and $Res_{X_2}(\mathcal{C}) = Res_{X_1}(\mathcal{C}) \cup \{\langle b, \emptyset, b\rangle, \langle \emptyset, \emptyset, ab\rangle\}$. Take for example $\langle b, \emptyset, b\rangle$, it does not need to be resolved again on X_1 with any other consequence, as it will inevitably give an element already in $Res_{X_2}(\mathcal{C})$. ▲

Corollary. *We have $Cons(\Pi) = Res_{[X_1,\ldots,X_n]}(\mathcal{P}_0)$ where $\mathcal{P}_0$ is the initial set of domain consequences corresponding to each constraint.*

Algorithm 8: CONSEQUENCECOMPILATION

Data: A problem Π
Result: $N = Cons(\Pi)$

$N \leftarrow \emptyset$
foreach $C_i \in \Pi$ **do**
  $N \leftarrow N \cup Cons(C_i)$
foreach $X_i \in \Pi$ **do**
  $S \leftarrow N$
  $N \leftarrow \emptyset$
  while $S \neq \emptyset$ **do**
    $D \leftarrow first(S)$
    foreach $D' \in N$ **do**
      if `non-trivial` (X_i, D, D') **then**
        $D^* \leftarrow resolve(X_i, D, D')$
        if *No element of* $N \cup S$ *subsumes* D^* **then**
          Remove from $N \cup S$ any element subsumed by D^*
          Add D^* to the end of S
    if D *is still in* S **then** Remove D from S and add to N
return N
function `non-trivial` $(X_i, \langle D_1, \ldots, D_n \rangle, \langle D'_1, \ldots, D'_n \rangle)$
  if $D_i \subseteq D'_i \vee D'_i \subseteq D_i$ **then**
    return *false*
  if $\exists X_j \neq X_i$ *such that* $D_j \cup D'_j = D(X_j)$ **then**
    return *false*
  return *true*

Proposition 4. *Algorithm 8 computes* $Cons(\Pi)$.

Proof. Lines 6-16 basically perform the Res_{X_i} operation, where S contains the initial domain consequences. During each iteration, the invariant that is maintained is that $N \cup S$ contains only minimal consequences. Every pair of elements is tested exactly once, and for each newly inferred consequence, it is added to $N \cup S$ only if it is currently minimal. It is added specifically to S so that it can be, in turn, tested against other consequences. When a domain consequence D in S is processed, it is resolved against all known minimal consequences contained in N. If D is not subsumed by any new consequence thus inferred, it is added to N. Therefore, when, in the end, S is empty, N contains exactly $Res_{X_i}(S)$. □

Concerning implementation details, each domain consequence has been implemented using a single bitset, which allows for very efficient operations, like the `non-trivial` test, the subsumption test, and the $resolve$ operation, which are by far the most frequent operations in this algorithm. Also, some simple optimisations can be performed when computing the disjunction of two sets of domain consequences while generating the domain consequences of each constraint (line 3). For example, when the union P of two domain consequences is equal to one of them, then this original domain consequence can be discarded, as any subsequently generated consequence will be subsumed by P, as it can be observed in Example 5.5.2.

5.5.3 Complexity

The complexity of Algorithm 8 is determined by the size of $Cons(\Pi)$. We saw that introducing a new valid tuple to the domain consequences of a constraint of arity k can add up to k times more new domain consequences. Therefore, we have an upper bound on $|Cons(\Pi)|$ of $n^{|S|}$, where $|S|$ is the number of solutions of the problem. $|S|$ itself is in $O(d^n)$, which gives an idea of the potential number of domain consequences. Additionally, the number of intermediate domain consequences that have been generated during the procedure can be far larger than the final number, as most become eventually subsumed by fewer domain consequences, as we shall see in the next section. These simple facts show that there

is very little hope that an efficient way to generate all the consequences explicitly will exist, no matter what algorithm we design.

5.6 Experimental Study

We show how the concepts we introduced are put into practice on some very simple instances. The purpose is not to test the validity of the algorithm, whose implementation has been kept intentionally naive, for the reasons stated in the last section. Indeed, we do not expect that, in the present stage, this algorithm can scale up to any other than trivial problems.

We tested the algorithm on random uniform constraint networks of binary constraints[1], using the seed 100. These instances contained 10 variables, with 10 values, 10 constraints. The tightness of each constraint varied from 1 to 19 allowed tuples, then from 95 to 99. We expected our approach to work better on problems with many conflicts. In Figure 5.1, we show the average number of consequences produced after the resolution of each variable, as well as the final total. This average number was in direct correspondence with the overall number of iterations. This shows that the tighter the constraints are, the fewer iterations it takes for the algorithm to finish. Not surprisingly, no instance with a tightness outside the extreme values could be resolved in any reasonable amount of time. Note that for the harder instances (1 to 19 allowed tuples per constraint), the final number of consequences is only 1, as they were unsatisfiable. On satisfiable instances, this number ranged from 250000 to 120. It is particularly striking, especially on the unsatisfiable instances, that most of the consequences discovered are not minimal in the end, as they end up being subsumed by the empty consequence.

To further illustrate this point, we show some more detailed figures on two instances, an unsatisfiable instance with 17 allowed tuples per constraint, and a satisfiable one with 95. Table 5.4 shows for each instance and at the end of each basic iteration (i.e. the resolution on each variable) the number of generated consequences (i.e. that passed the `non-trivial` test), the number of non-subsumed consequences (i.e. the part of those that have been successfully added to S), and

[1] we used the generator at: `http://www.lirmm.fr/~bessiere/generator.html`.

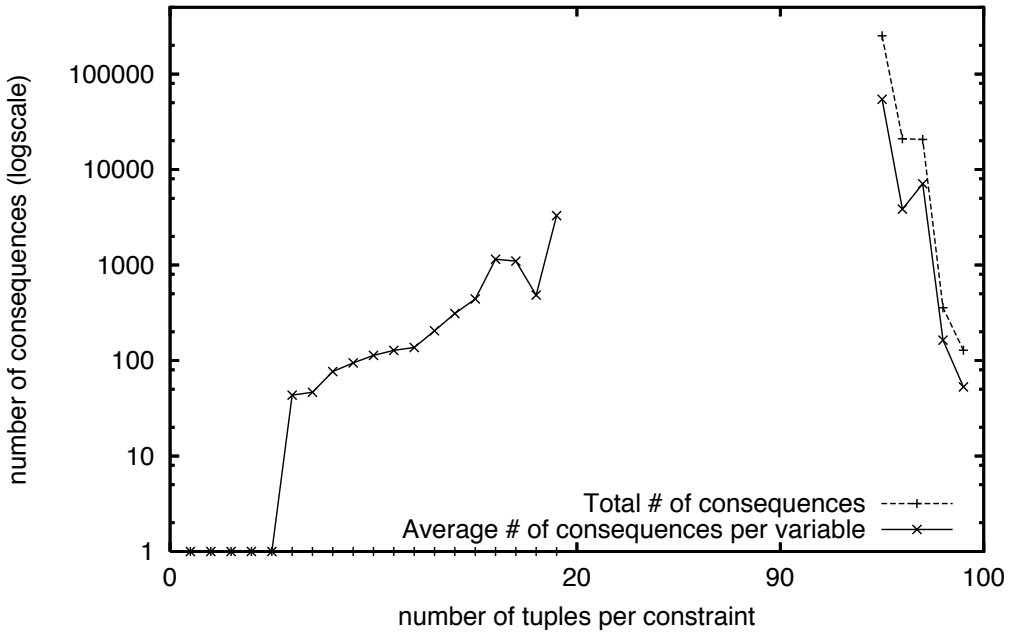


Figure 5.1: The total number of consequences and the average for each variable per instance

eventually the final variation of the current number of minimal consequences (i.e. the difference in size between N at the end and at the beginning of the iteration).

Table 5.4: The intermediate number of consequences generated

Variable	17 tuples			95 tuples		
	generated	non-subsumed	variation	generated	non-subsumed	variation
0	239179	6227	730	423	85	85
1	26843	1027	105	0	0	0
2	88841	4894	85	266531	2894	2894
3	16102	609	145	377234	2882	2056
4	397594	21603	-1545	6027194	13708	13708
5	259702	7844	481	13674500	29420	8121
6	58563	2987	-980	51688417	40006	40006
7	1371	225	-263	70889441	50362	18689
8	18	5	-22	63317577	0	0
9	0	0	0	1009075634	263964	165954

These results are encouraging. They show that the size of the final set we aim to produce is tractable on those instances. With an adequately compact representation of the set of minimal consequences, we can manage very satisfactory

sizes. As a comparison, it is a notable fact that automata can represent very efficiently a number of solutions several orders of magnitude higher than the figures shown here. The practical ability of such a representation to represent effectively a large number of potential conflicts will depend a lot on different parameters of a problem. These could be parameters like the number of actual conflicts for a user query, the size of such conflicts, value interchangeability, etc. That is of course a fundamental point that will need to be studied in further work. We conjecture that the properties yielding compact consequence compilation will be different from those needed for an automaton representation, where structural considerations (the topological properties of the constraint graph) and variable orderings are crucial.

5.7 Related Work

Work on consequence-finding has been an important task in Artificial Intelligence for the past thirty years, with a particular interest for diagnosis, which closely relates to our setting. A very complete survey of the field and its applications has been presented in Marquis [96], in particular of the many existing algorithms for prime implicate generation. de Kleer et al. [28] presents the application of prime implicants and implicates to diagnosis, noting in particular this link between conflicts and implicates. Prime implicates generation is also valuable for computing abductions. An abduction has to be consistent with the knowledge base, but checking for consistency can be an expensive operation. A subset of the hypotheses that is inconsistent with the knowledge base forms a conflict, and cannot be part of an explanation. These conflicts corresponds to the prime implicates of the theory that contain only hypothesis literals, and detecting that a knowledge base is conflict-free can allow us to speed up abduction computation [64].

One of the foundation works about prime implicate generation dates from 1967 [133]. Kernel resolution [37, 38] generalises this idea. At each step of the algorithm, a clause is partitioned in the skip and the kernel. The skip contains the variables smaller that the variable upon which the clause has been resolved and the kernel those that are greater (according to a given order on the variables). Kernel resolution generalises the idea of [133] (referred to as ordered resolution),

by resolving a given clause only upon kernel variables, i.e. variables that are greater than the variable by which this clause has been resolved. Consequence finding aims at computing a set of clauses implied by a sentence belonging to some given clause language, like clauses of a bounded size, clauses containing only some literals. Kernel resolution facilitates focused resolution, by only accepting resolvents whose skip belongs to the target language. In 1992, de Kleer [26] proposed a practical implementation, where clause bases are stored using TRIE structures, which allows for a more compact representation and more efficient subsumption operations, such as testing if there exists a smaller implicate, adding a prime implicate, removing all the bigger implicates, which are the most frequent and costly. Another approach is to work on the representation of all the prime implicates. The TRIE representation [26] and clause resolution can be extended to a ZBDD representation and multi-resolution [13, 125], which we discuss further in the next chapter. Alternatively, Coudert and Madre [16] proposes an implicit representation based on meta-products. In particular, the main contribution of Simon and del Val [125], is an extensive experimental study; it seems to be the first method for full prime implicates generation that scales to instances of real practical interest. Finally, Junquera and González [77] presents a similar approach to ours (though the concepts are different) in using a pre-analysis of the problem as a compilation technique, in order to facilitate conflict and diagnosis computation. Approximation anytime algorithms have also been proposed to deal with the computational complexity of this task. Arguments, or explanations, can be scored by relevance, according to probabilistic considerations, and an anytime approximation algorithm has been proposed that outputs all arguments in decreasing order of relevance [63]. This algorithm is the state of the art approximation method for computing abductive explanations.

There has been previous work that use prime implicate generation as a compilation technique, with the common approach of limiting the amount of prime implicates that are generated. In 1995, Marquis [97] introduced the concept of theory prime implicates, which allows us to reduce the number of generated implicates by defining a stronger notion of entailment. In 1994, del Val [36] introduced the concept of tractable databases, allowing to reduce the number of generated implicates by only selecting a subset that is refutation-complete (a set of clauses

is said refutation-complete if any clause can be tested to be an implicate only by unit resolution deduction). Both papers show encouraging experimental results. We present here the technical aspects of these papers.

Theory prime implicates. Marquis defines a more general entailment: $\sigma \models_\Phi \tau$ (σ entails τ given Φ) if $\Phi \cup \{\sigma\} \models \tau$, where Φ is a *theory*. Given that operator, *theory prime implicates* are defined in the same way as prime implicates are defined with regards to the classic $\models$ operator. Let $TPI(\Sigma, \Phi)$ be the set of all the theory prime implicates of Σ w.r.t. Φ, for some Φ such that $\Sigma \models \Phi$. The formula Σ is compiled into $COMP_\Phi(\Sigma) =_{def} \langle TPI(\Sigma, \Phi), \Phi \rangle$. In order to test if a clause π is entailed by Σ, one must check if there exists some π' in $TPI(\Sigma, \Phi)$ such that $\pi' \models_\Phi \pi$. That can be done in time $O(|\pi'| \times |\Phi \cup \{\pi\}|^\alpha)$, if Φ is tractable, that is if clausal entailment on Φ (checking if $\Phi \models \pi$ for some clause π) is polynomial. Thus, the whole query answering process takes $O(|\,TPI(\Sigma, \Phi)| \times |\Phi \cup \{\pi\}|^\alpha)$ time, thus depending on the number of theory prime implicates.

This method allows us to take advantage of computationally tractable entailment for formulas more complicated than just clauses. Whereas $\pi' \models \pi$ can be trivially decided (by checking inclusion), being able to make that test in polynomial time for more general formulas allows us to reduce the number of theory prime implicates we produce, without giving up on tractability. It is indeed shown that the stronger Φ is, the smaller is the number of theory prime implicates (and as a corollary the number of theory prime implicates is always less than or equal to the number of prime implicates).

The key point of the compilation process is the choice of the theory Φ. In the paper, it is chosen according to some heuristics, but a better study of that choice is left for future work. Some preliminary experimental results show indeed good savings in the number of theory prime implicates.

Tractable Databases. The approach here is to compile a formula Σ to a logically equivalent Σ^* which is *unit refutation complete*. A set of clauses Σ is unit refutation complete if for some clause C, $\Sigma \models C$ iff $\Sigma \wedge \neg C$ can be detected inconsistent by unit resolution deduction. Σ^* is effectively some subset of the prime implicates of Σ. Several compilations schemes are introduced, each of which

computes the prime implicates that can be obtained using a specific resolution deduction; each have or may have advantages in the number of produced clauses. Some experimental results show savings in the size compared to full prime implicate generation, with a factor raging from 5 to 40.

5.8 Applications and Extensions

Having a complete picture of the potential conflicts of a problem allows us to generate explanations with much richer properties. As a straightforward example, let us consider back representative sets of explanations that were defined in Chapter 3. We saw that the main bottleneck of any algorithm that generates a representative set of explanations for a particular query lied in the fact that is hard to verify the representativeness of a set of explanations. More particularly, it is hard to check if a constraints appears in all exclusion sets. However, when a constraint appears in all minimal exclusion sets, it appears also in all minimal conflicts. This means that if a problem is represented in terms of all its domain consequences, the representativeness check is easy to perform, i.e. it is polynomial in the size of the problem representation. Indeed the set of conflicts of a query is given by the set of domain conflicts that are supersets of the query, and we only need to check if some given user constraint of the query belongs to all those domain conflicts.

Based on this remark, we can further extend the notion of representativeness. A constraint that appears in all conflicts can clearly never be satisfied, and the user must be told that. However, a constraint that appears in most conflicts *can* be satisfied, but, coming close to inconsistency, it will be at the expense of other constraints, typically leaving very small flexibility on the amount of other constraints that the user will be able to keep. It would be good if the user could be told this information too. We can therefore define the concept of "constrainedness" of a constraint within a specific user query as the ratio of conflicts of the query where this constraint appears, a ratio of 1 meaning the constraint appears in all conflicts and cannot be satisfied. The constrainedness of each constraint can be given as such to the user, so as to give a better understanding of the inconsistency. Alternatively, it can be used to refine the notion of representative set of explanations. For example, we can chose that a constraint must be relaxed by a relaxation in the

representative set with a probability equal to its constrainedness. Such sets would obviously be also representative according to the definition of Chapter 3.

Example 5.8.1. Consider again Example 3.2.1 at Page 52. The conflicts in this example are $\{c_2, c_3\}$, $\{c_1, c_2, c_5\}$, $\{c_1, c_3, c_5\}$, $\{c_1, c_4, c_5\}$. We can see that both constraints c_1 and c_5 appear in all but one conflict. From this we can deduce that they are somehow the "hardest" to satisfy. Indeed, they correspond to keeping below a budget limit and choosing the most expensive option. The user can then be told that it will be hard to either stay within his budget limit, or have leather seats, or both (and c_1c_5 is indeed a maximal relaxation).

Chapter 6

Representing and Reasoning on Large Sets of Domain Consequences

Summary. *We present how ordered automata can be used to represent sets of domain consequences. Operators are defined that, combined with each other, define a compilation procedure of a problem to an ordered automaton encoding its set of domain consequences. The operators constitute the basic pieces to query a database of domain consequences encoded as an ordered automaton.*

6.1 Introduction

As we saw in Section 5.4, although having interesting properties, representing a problem in terms of the set of all its domain consequences is, as such, a very poor compilation technique. Indeed, we unequivocally revealed the number of consequences a problem entails can be intractably high. Therefore, the actual set of domain consequences has to be, in turn, compactly represented. It is worth noting that all compilation techniques essentially involve compactly representing the set of all the solutions of the problem being compiled: the least compact compiled representation involves simply listing all the solutions, and more and more compact representations encode more and more implicitly those solutions. Representing a problem in terms of its domain consequences would be indeed as brutal as

representing it in terms of its solutions; in this section we introduce a more compact method of representing the domain consequences of a problem. In terms of the compilation map, note that **PI** and **MODS** are possibly the two least succinct languages (see Figure 4.6). "Solution-driven" compilation methods aim at representing **MODS** more succinctly. In this chapter, we propose a "conflict-driven" compilation method that represents **PI** more compactly.

Zero-suppressed Binary Decision Diagrams (ZBDDs) have already been showed to be an efficient data structure for representing a collection of subsets of a universe, by representing the characteristic function of each subset contained in the collection [72]. This property has been successfully exploited by Chatalic and Simon [13] to represent sets of clauses (which can be seen as sets of literals), with a very efficient compression power. Exploiting this particular semantics, i.e. that it is a set of literals and not any set that is represented (see Figure 6.1), Chatalic and Simon propose some additional specific operators that allow in the end to generate and represent very efficiently all the prime implicates of a CNF. This approach is the first (and, to our knowledge, single) approach of a consequence finding algorithm that scales up to real problems. It can, remarkably, implement and use the original Davis and Putnam resolution procedure. We propose to adapt this approach for our context.

Example 6.1.1. Consider the ZBDD in Figure 6.1. Each node is labelled by a literal, and has two outgoings arcs: a 1-labelled arc, represented by a plain line, meaning the literal is present, a 0-labelled arc, represented by a dashed line, meaning the literal is omitted. Paths from the root node to a terminal node, those labelled by **0** or **1**, correspond to sets of literals, only those of which end at node **1** being kept.

A domain consequence can be seen as merely a subset of a universe, which would contain the union of the domains of all the variables. Consequently, at first glance, one could think that the ZBBD approach could be straightforwardly applied to our case. However, this approach would need to be extended to take into account the particular semantics of domain consequences. This is actually a fundamental difference and makes this extension not so straightforward. In order to do this, we propose a slightly different approach, based on automata, on which

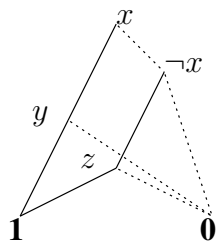


Figure 6.1: An example ZBDD encoding a collection of two sets of literals $\{xy, \neg xz\}$.

it will be more convenient to perform this generalisation.

6.2 Ordered Automata for Collection of Subsets

6.2.1 General Definition

We propose to use regular automata to represent a collection of subsets, with an added property that values must be ordered. Concretely, we impose a total order on the alphabet, and this order affects how values can be introduced: a recognised word can be composed only of symbols in strictly increasing order.

Definition 6.2.1 (Ordered Automaton)**.** An ordered automaton M is defined as a 6-tuple $\langle Q, \Sigma, \leq, \delta, q_0, F \rangle$, with:

- Q a finite set of states,
- Σ a set of symbols (the alphabet),
- $\leq$ a total order on Σ,
- δ a function $Q \times \Sigma \to Q$ (the ordered transition function), such that for any string s recognised by M, $\forall i < j \leq |s|, s[i] < s[j]$,
- $q_0 \in Q$ the initial state,
- $F \subseteq Q$ the final states.

We additionally impose the usual properties on the automaton, precisely that it should be deterministic, trimmed (remove transitions that cannot reach final states) and minimal. These properties will be implicitly assumed.

We introduce some additional notation:

- $\sigma(q) = \{a \in \Sigma / \delta(q, a) \text{ is defined}\}$, the set of labels of the outgoing transitions of a state;
- $\delta(q) = \{\delta(q, a) / a \in \sigma(q)\}$, the successors of a state;
- $\delta^*(q) = \{q' \in Q / (q' = q) \vee (q' \in \delta(q'') \wedge q'' \in \delta^*(q))\}$, the set of states accessible from a state;
- the digraph $D = (Q, E)$, with $E = \{(q, q') / q, q' \in Q \wedge \exists a \in \Sigma$ such that $\delta(q, a) = q'\}$, is called the underlying digraph of the automaton.

Let us remark that, as an ordered automaton can only recognise strings of finite length, its underlying digraph must be acyclic.

The semantics that we attach to an ordered automaton M is defined as follows. Given a collection of subsets of a universe U, we build a corresponding ordered automaton M such that:

- Σ is in one-to-one correspondence with U;
- $\leq$ can be the natural total order holding on U, or any arbitrarily chosen total order;
- a subset $S \subseteq U$ is uniquely represented by the string composed of the symbols corresponding to each element in S, in increasing order;
- a subset $S \subseteq U$ is in the collection *iff* the corresponding string is recognised by M.

Let us observe that, with this semantics, a minimal DFA representing a collection of subsets has a single final state if all the subsets of the collections are incomparable. Indeed, if a minimal DFA has more than one final state, then at

least one final state must have a successor (two final states with no successor being equivalent). This state recognises a given set which is a subset of any set recognised by any successor of this state.

Some particular cases of this semantics include the empty collection, and the collection containing the empty set only. The empty string represents the empty set. An automaton with only one state that is both initial and final recognises the empty string only, and thus interprets as $\{\emptyset\}$. An automaton that has no final state does not recognise any string, and thus interprets as $\emptyset$ (if it is minimal, the initial state is the unique state).

6.2.2 Domain Sequences

We can use this data structure to represent a collection of domain sequences. Indeed, if we assume, without loss of generality, that the domains of each variable do not share common values, we can notice that a domain sequence is merely a subset of the universe defined by the union of all domains. Therefore, we can apply the approach just described.

From a logical point of view, if the considered domain sequences represent domain consequences, a collection of domain consequences interprets as a conjunction (of disjunctions). In particular, $\emptyset$ interprets as $true$ and $\{\emptyset\}$ interprets as *false*.

In order to encode a collection of domain sequences with an ordered automaton, we first have to match the values of the initial domains to unique values for each variable, then to impose a total order on the resulting universe (which can result from an existing total order on the original domains). More formally, given a sequence of variables $X_1, \ldots, X_n$, with domains $D(X_i)$, we define an automaton M as follows:

- $\forall a \in D(X_i)$, we create a unique symbol, denoted a_{X_i}, thus $\Sigma = \{a_{X_i} / a \in D(X_i)\}$;
- we set an order $\leq$ such that $a_{X_i} \leq b_{X_j}$ iff $i \leq j$ or $i = j \wedge a \leq b$.

Then, we can encode a specific domain consequence as a string over Σ, and

a collection of domain consequences as the automaton recognising the set of the corresponding strings.

Example 6.2.1. Let $\mathcal{P} = \{\langle ab, ab, a\rangle, \langle b, bc, a\rangle, \langle b, \emptyset, b\rangle\}$, on the variables X, Y, Z (we assume $a < b < c$). We define $\Sigma = \{a_X, b_X, c_X, a_Y, b_Y, c_Y, a_Z, b_Z, c_Z\}$, declared in increasing order, and the MinDFA encoding $\mathcal{P}$ is given in Figure 6.2. Conceptually, we are basically labelling the transitions of the automaton with variable assignments.

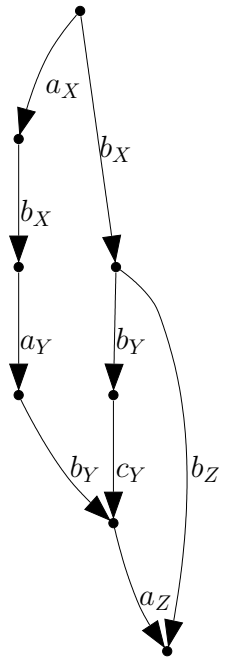


Figure 6.2: An automaton representing three domain consequences

6.3 Querying Ordered Automata

With the given semantics, we can realise operations on sets of domain consequences as operations on automata. There are two types of operations: operations that, given one or two automata return a new automaton (a transformation) and operations that given one automaton answers a question (a query). The different

transformation operations are used to actually compile a problem into the automaton representing the set of its domain consequences, while queries are used to answer questions that make use of this representation.

We will make use of the following notation:

- Given a state q_0 belonging to a certain automaton, as a shortcut notation, we denote by $\mathcal{P}(q_0)$ the collection of domain consequences *encoded by the automaton* whose initial state is q_0.
- Let $q_\emptyset$ be a constant dummy state.
- Let $M_\emptyset = \langle\{q_\emptyset\}, \emptyset, \emptyset, \emptyset, q_\emptyset, \emptyset\rangle$. We have here $\mathcal{P}(q_\emptyset) = \emptyset$; $q_\emptyset$ is not an accepting state.
- Let $M_{\{\emptyset\}} = \langle\{q_\emptyset\}, \emptyset, \emptyset, \emptyset, q_\emptyset, \{q_\emptyset\}\rangle$. We have here $\mathcal{P}(q_\emptyset) = \{\emptyset\}$; $q_\emptyset$ is an accepting state.

In order to simplify presentation, we will assume that, for a given state q and symbol a, $\delta(q, a)$ being undefined is equivalent to having $\delta(q, a) = q_\emptyset$. In particular, we can express $\sigma(q)$ as $\{a \in \Sigma / \delta(q, a) \neq q_\emptyset\}$. This allows us, in the algorithms, not to consider explicitly the case where a transition for a given symbol is undefined.

Finally, we assume that the considered automata are always defined with the same alphabet and the same order.

6.3.1 Auxiliary Functions

Register State. All operators use a function `register-state`, which is given a newly built state that will not be further modified (i.e. no transition will be added or removed and no successor will be modified). If the newly built state has no outgoing transition, or equivalently, that all transitions lead to $q_\emptyset$, $q_\emptyset$ is returned; this ensures that the automaton is trimmed. If a state already exists that is equivalent to the newly built state, i.e. the same symbols lead to the same states, that state is returned instead. Otherwise, the newly built state itself is returned. This way, the invariant that all states in the register are pairwise inequivalent is maintained.

If this function is called in a bottom-up order, i.e. when called on a state, it has already been called on all of its successors, and as the considered automata are acyclic, this is sufficient to ensure minimality.

Function `register-state(`q`)`

if $\sigma(q) = \emptyset \wedge \neg isFinal(q)$ **then return** $q_\emptyset$
if $\exists q'$ *in the register such that* q *is equivalent to* q' **then return** q'
Add q to the register
return q

Merge. This function is given two states and merges them to one state such that it remains deterministic. Essentially, transitions on common values between the given states are merged into a single transition, the destination states of which are recursively merged.

Function `merge(`q_1, q_2`)`

$q \leftarrow$ Create new State
if $isFinal(q_1) \vee isFinal(q_2)$ **then** $isFinal(q) \leftarrow true$
forall $a \in \sigma(q_1)$ **do** $\delta(q, a) \leftarrow \delta(q_1, a)$
forall $a \in \sigma(q_2)$ **do** $\delta(q, a) \leftarrow \delta(q_2, a)$
forall $a \in \sigma(q_1) \cap \sigma(q_2)$ **do**
  $\delta(q, a) \leftarrow$ `merge(`$\delta(q_1, a), \delta(q_2, a)$`)`
return q

6.3.2 Subsumed Removal

This is a fundamental operator. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two collections of domains consequences. We define $\mathcal{P}_1 \setminus_{\models} \mathcal{P}_2$ as the set of all the domain consequences of $\mathcal{P}_1$ that are not subsumed by any domain consequence of $\mathcal{P}_2$.

Let M_1 and M_2 be two ordered automata, with respective initial state q_0^1 and q_0^2. We define the operator $q_0^1 \setminus_{\models} q_0^2$ in the following function:

Proposition 1. $\mathcal{P}(q_0^1 \setminus_{\models} q_0^2) = \mathcal{P}(q_0^1) \setminus_{\models} \mathcal{P}(q_0^2)$.

Function $q_0^1 \setminus_{\models} q_0^2$

$S \leftarrow \{a \in \sigma(q_0^1) / a \leq \max \sigma(q_0^2)\}$
if $S = \emptyset$ **then**
  if q_0^2 *is final* **then**
    return $q_\emptyset$
  else
    return q_0^1
else
  $q_0 \leftarrow$ Duplicate q_0^1
  forall $a \in S$ **do**
    $q_0^2 \leftarrow \texttt{merge}(\delta(q_0^2, a), q_0^2)$
    $\delta(q_0, a) \leftarrow \delta(q_0^1, a) \setminus_{\models} q_0^2$
  return `register-state`(q_0)

Proof. Let $\mathcal{P}_1 = \mathcal{P}(q_0^1)$, $\mathcal{P}_2 = \mathcal{P}(q_0^2)$ and $\mathcal{P} = \mathcal{P}(q_0^1 \setminus_{\models} q_0^2)$. Consider first the trivial cases, i.e. the cases where $\mathcal{P}_1$ or $\mathcal{P}_2$ is equal to $\emptyset$ or $\{\emptyset\}$. If $\mathcal{P}_2 = \{\emptyset\}$, i.e. q_0^2 is final, $\mathcal{P} = \emptyset$. If $\mathcal{P}_2 = \emptyset$, q_0^2 is not final, so $\mathcal{P} = \mathcal{P}_1$. If $\mathcal{P}_1 = \emptyset$, i.e. $q_0^1 = q_\emptyset$, then $q_\emptyset$ is returned, regardless of q_0^2, and so $\mathcal{P} = \emptyset$. If $\mathcal{P}_1 = \{\emptyset\}$ (and $\mathcal{P}_2 \neq \{\emptyset\}$), $\mathcal{P} = \mathcal{P}_1$.

If $S = \emptyset$ and none of the above cases hold, then all the domain consequences of $\mathcal{P}_2$ contain a symbol which is not contained in any domain consequence of $\mathcal{P}_1$, because of the ordered property, and so no domain consequence of $\mathcal{P}_2$ subsumes any one of $\mathcal{P}_1$. In this case, as q_0^2 is not final, q_0^1 is returned, and $\mathcal{P} = \mathcal{P}_1$.

Consider now the non-terminal case, i.e. $S \neq \emptyset$. Let $a \in S$. Suppose a domain consequence P_1 from $\mathcal{P}_1$ containing a is subsumed by a domain consequence P_2 of $\mathcal{P}_2$. Either P_2 contains a too, and then the domain consequence $P_2 \setminus \{a\}$ of $\mathcal{P}(\delta(q_0^2, a))$ subsumes the domain consequence $P_1 \setminus \{a\}$ of $\mathcal{P}(\delta(q_0^1, a))$, or P_2 does not contain a, in which case P_2 itself also subsumes $P_1 \setminus \{a\}$ of $\mathcal{P}(\delta(q_0^1, a))$. Let q be the state returned the end of the recursive call. After the call to `merge()`, $\mathcal{P}(q_0^2) = \mathcal{P}(\delta(q_0^2, a)) \cup \mathcal{P}_2$, and by the induction hypothesis, $\mathcal{P}(q)$ is equal to $\mathcal{P}(\delta(q_0^1, a))$ where all the domain consequences such as $P_1 \setminus \{a\}$ have been removed. If $a \notin S$, then all domain consequences of $\mathcal{P}_2$ contain a symbol which is not in any domain consequence of $\mathcal{P}_1$, so nothing needs to be changed.

In order for the resulting automaton to be minimal it is enough to ensure that

all states of M_1 have been added to the register prior to calling this operator. □

Example 6.3.1. As an example, consider the automata in Figure 6.3. The transitions of M_1 labelled by a symbol greater than $\max(\sigma(q_0^2))$ are ignored, by definition of S. Here, the transition labelled by d will not be traversed, and its subautomaton will remain unmodified. Similarly, transitions of M_2 labelled by a symbol less than $\min(\sigma(q_0^1))$ are also ignored, as they are never taken into account by the function. Here, the transition labelled by a is never considered and indeed its presence or absence does not affect the result of the function.

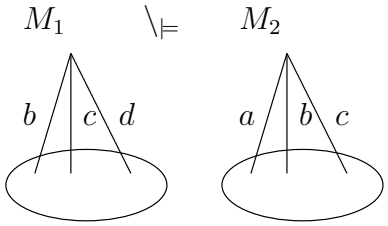


Figure 6.3: Subsumed removal on two automata

6.3.3 Minimisation

Let $\mathcal{P}$ be a collection of domain consequences. We denote $\mu\mathcal{P}$ the set of domain consequences of $\mathcal{P}$ that are not subsumed by any other domain consequence of $\mathcal{P}$.

Let M be an ordered automata, with initial state q_0. We define the operator $\mu(q_0)$ in the following function:

Function $\mu\,(q_0)$

if q_0 *is final* **then**
  return q_0
else
  $q_0' \leftarrow$ Create new state
  forall $a \in \sigma(q_0)$ *in decreasing order* **do**
    $\delta(q_0', a) \leftarrow \mu(\delta(q_0, a)) \backslash\models q_0'$
  return `register-state`(q_0')

Proposition 2. $\mathcal{P}(\mu(q_0)) = \mu\mathcal{P}(q_0)$.

Proof. At a given iteration of the forall statement, let $\mathcal{P}_1 = \mathcal{P}(\mu(\delta(q_0, a)))$ and $\mathcal{P}_2 = \mathcal{P}(q'_0)$. By induction hypothesis, both are free from subsumed elements. Any element from $\mathcal{P}_2$ subsumed by an element P of $\mathcal{P}_1$ will not be subsumed by $P \cup \{a\}$. However, if an element from $\mathcal{P}_1$ is subsumed by an element of $\mathcal{P}_2$, it will still be after adding a to it. By removing those subsumed elements, we obtain a subsumption-free automaton.

□

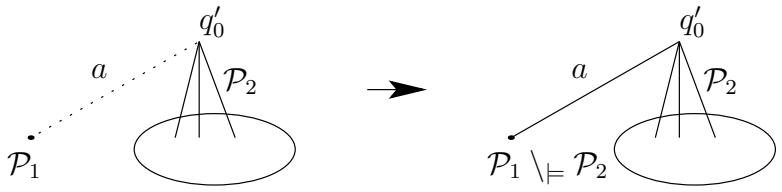


Figure 6.4: Adding $\mathcal{P}_1$ without introducing subsumed elements

6.3.4 Union

Let $\mathcal{P}_1, \mathcal{P}_2$ be two collections of domains consequences free from subsumed elements. We define $\mathcal{P}_1 \cup_\mu \mathcal{P}_2$ as the set $\mu(\mathcal{P}_1 \cup \mathcal{P}_2)$, i.e. the union of $\mathcal{P}_1$ and $\mathcal{P}_2$ where only non-subsumed elements are kept. From a logical point of view, $\mathcal{P}_1 \cup_\mu \mathcal{P}_2$ is the conjunction between $\mathcal{P}_1$ and $\mathcal{P}_2$.

Let M_1 and M_2 be two ordered automata free from subsumed elements, with respective initial states q_0^1 and q_0^2. We define the operator $q_0^1 \cup_\mu q_0^2$ in the following function:

Function $q_0^1 \cup_\mu q_0^2$

if q_0^1 *is final* **then return** q_0^1
if q_0^2 *is final* **then return** q_0^2
else
  $q_0 \leftarrow$ Create new state
  forall $a \in \sigma(q_0^1) \cup \sigma(q_0^2)$ *in decreasing order* **do**
    $\delta(q_0, a) \leftarrow (\delta(q_0^1, a) \cup_\mu \delta(q_0^2, a)) \setminus_{\models} q_0$
  return `register-state`(q_0)

Proposition 3. $\mathcal{P}(q_0^1 \cup_\mu q_0^2) = \mathcal{P}(q_0^1) \cup_\mu \mathcal{P}(q_0^2)$.

Proof. The proof is similar to that of Proposition 1. For a given symbol a, if both states have an outgoing transition labelled with a, then the union of respective consequences starting with a must be performed. If only one state has an outgoing transition with a, then the recursive call will be performed with $q_\emptyset$ as one of the operands, and so the other operand will be returned unmodified. Finally, as we assume that the result of the recursive call will be free from subsumed consequences, and with the same reasoning as for the proof of Proposition 1, the resulting automaton will also be free from subsumed consequences. □

6.3.5 Product

This is also one of the key operators. Let $\mathcal{P}_1, \mathcal{P}_2$ be two collections of domain consequences free from subsumed elements. We define $\mathcal{P}_1 \otimes \mathcal{P}_2 = \mu\{P_1 \cup P_2 / P_1 \in \mathcal{P}_1, P_2 \in \mathcal{P}_2 \wedge P_1 \cup P_2 \text{ is not trivial}\}$. Intuitively, $\mathcal{P}_1 \otimes \mathcal{P}_2$ contains all the domain consequences formed by the union of a domain consequence respectively from $\mathcal{P}_1$ and from $\mathcal{P}_2$ that are not trivial and that are not subsumed by any other such domain consequence. From a logical point of view, $\mathcal{P}_1 \otimes \mathcal{P}_2$ is the disjunction between $\mathcal{P}_1$ and $\mathcal{P}_2$.

In order to deal with trivial consequences, we need to introduce the following notation. For each $X_i, 1 \leq i \leq n$, we define the domain sequence $P_{X_i} = \langle D_1^i, \ldots, D_n^i \rangle$, with $D_i^i = D(X_i) \setminus \{\min D(X_i)\}$, and $D_j^i = \emptyset$ otherwise if $j \neq i$. For each $X_i, 1 \leq i \leq n$, we define the state $q_{X_i}^\wedge$ as the initial state of the ordered automaton recognising $\{P_{X_i}\}$. For example, suppose $n = 3$, $i = 2$ and $D(X_2) = \{abc\}$. We have $\mathcal{P}(q_{X_2}^\wedge) = \{\langle \emptyset, \{bc\}, \emptyset \rangle\}$.

Let M_1 and M_2 be two ordered automata free from subsumed elements, with respective initial states q_0^1 and q_0^2. We define the operator $q_0^1 \otimes q_0^2$ in the following function:

Proposition 4. $\mathcal{P}(q_0^1 \otimes q_0^2) = \mathcal{P}(q_0^1) \otimes \mathcal{P}(q_0^2)$.

Proof. Let $\mathcal{P} = \mathcal{P}(q_0^1 \otimes q_0^2)$, $\mathcal{P}_1 = \mathcal{P}(q_0^1)$ and $\mathcal{P}_2 = \mathcal{P}(q_0^2)$. Consider first the trivial cases. When $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is equal to $\{\emptyset\}$, i.e. q_0^1 (resp. q_0^2) is final, $\mathcal{P} = \mathcal{P}_2$ (resp. $\mathcal{P} = \mathcal{P}_1$), so q_0^2 (resp. q_0^1) is returned. If $\mathcal{P}_1 = \emptyset$ or $\mathcal{P}_2 = \emptyset$, then one of q_0^1 or

Function $q_0^1 \otimes q_0^2$

if q_0^1 *is final* **then return** q_0^2
if q_0^2 *is final* **then return** q_0^1
if $\sigma(q_0^1) = \emptyset \vee \sigma(q_0^2) = \emptyset$ **then return** $q_\emptyset$
else
  $q_0 \leftarrow$ Create new state
  forall $a \in \sigma(q_0^1) \cup \sigma(q_0^2)$ *in decreasing order* **do**
    $q_0^{1\prime} \leftarrow$ Create new state
    forall $b \in \sigma(q_0^1)/b > a$ **do**
      $\delta(q_0^{1\prime}, b) \leftarrow \delta(q_0^1, b)$
    $q_0^{1\prime} \leftarrow$ `register-state`$(q_0^{1\prime})$
    $q_0^{2\prime} \leftarrow$ Create new state
    forall $b \in \sigma(q_0^2)/b > a$ **do**
      $\delta(q_0^{2\prime}, b) \leftarrow \delta(q_0^2, b)$
    $q_0^{2\prime} \leftarrow$ `register-state`$(q_0^{2\prime})$
    $$\delta(q_0, a) \leftarrow \begin{pmatrix} \delta(q_0^1, a) \otimes \delta(q_0^2, a) \\ \cup_\mu \; \delta(q_0^1, a) \otimes q_0^{2\prime} \\ \cup_\mu \qquad q_0^{1\prime} \otimes \delta(q_0^2, a) \end{pmatrix} \setminus_\models q_0$$
    Let X_i be the variable to which a belongs
    if $a = \min D(X_i)$ **then**
      $\delta(q_0, a) \leftarrow \delta(q_0, a) \setminus_\models q_{X_i}^\wedge$
  return `register-state`(q_0)

q_0^2 is not final and has no successor, in which case $q_\emptyset$ is returned, and so $\mathcal{P} = \emptyset$. Consider now the non-terminal case. For some given a, $q_0^{1\prime}$ (resp. $q_0^{2\prime}$) contain the domain consequences of $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) involving symbols strictly greater than a. The domain consequences of $\mathcal{P}$ starting by a are of the form $\{a\} \cup P \cup P'$, such that P and P' involve only symbols strictly greater that a, where either $\{a\} \cup P \in \mathcal{P}_1$ and $\{a\} \cup P' \in \mathcal{P}_2$, or $\{a\} \cup P \in \mathcal{P}_1$ and $P' \in \mathcal{P}_2$, or $P \in \mathcal{P}_1$ and $\{a\} \cup P \in \mathcal{P}_2$. By induction hypothesis, we assume $\mathcal{P}(\delta(q_0, a))$ will contain all such $P \cup P'$. Finally, with the same reasoning as for above, the result automaton will be free from subsumed consequences. Filtering out trivial consequences on X_i is only a matter of performing subsumed removal with $q_{X_i}^\wedge$. Note that this needs only be performed when the return state contains an outgoing symbol that is the smallest value of the domain of X_i, as this is a necessary condition for a trivial consequence

on X_i to be present.

For the resulting automaton to be minimal, we have to ensure that all states of M_1 and M_2 are in the register prior to the call to this operator, which implies that they are all pairwise inequivalent (i.e. states from M_1 and M_2 that would be equivalent have been merged). □

6.4 Compiling a Problem to an Ordered Automaton

We can make use of the different operators we defined in the previous section on ordered automata, in order to compile a problem in the representation we suggest. Essentially, we want to generate all the domain consequences of a problem and represent them as an ordered automaton, with both steps being carried at the same time.

At the core of this compilation procedure is the closure by resolution. Put simply, given an ordered automaton M encoding a collection $\mathcal{P}$ of domain consequences, we need to apply an operator that builds an ordered automaton $Cons(M)$ that encodes $Cons(\mathcal{P})$. We define this operator in this section.

6.4.1 The Operator for the Boolean Case[1]

To simplify presentation, we will first consider the boolean case. Let us first define the "distribution" operator as follows.

Definition 6.4.1 (Distribution)**.** Let Φ be a collection of sets of literals. The distribution of Φ, denoted $\boxtimes\Phi$, is the collection of sets of literals obtained by keeping exactly one literal from each set in Φ, and such that non-minimal and trivial sets (i.e. sets containing a literal in non-negated and negated forms) are omitted.

If Φ is a CNF, the operator consists in distributing the conjunction over the disjunction, and thus converting from a CNF to an equivalent DNF. Conversely if Φ is a DNF, the operator distributes the disjunction over the conjunction, thus converting it to an equivalent CNF. In case a DNF is obtained, trivial sets correspond

[1] I would like to thank Hélène Fargier for her very helpful explanations and for pointing me to some very relevant papers, which helped me develop this section.

to contradictions (containing $l \wedge \neg l$), and in the case a CNF is obtained, trivial sets correspond to tautologies (containing $l \vee \neg l$).

Example 6.4.1. Let $\Phi = \{xy, \neg xz\}$, $\boxtimes\Phi = \{xz, \neg xy, yz\}$. If $\Phi = (x \wedge y) \vee (\neg x \wedge z)$, $\boxtimes\Phi = (x \vee z) \wedge (\neg x \vee y) \wedge (y \vee z)$.

Quite trivially, if we invert the literals in $\boxtimes\Phi$, for a given CNF (resp. DNF) Φ, we obtain a CNF (resp. DNF) that is the logical negation of Φ.

Proposition 1. *Let* $\Phi' = \{C'/l \in C' \Leftrightarrow \neg l \in C, C \in \boxtimes\Phi\}$. *We have* $\Phi' \equiv \neg\Phi$.

Corollary. $\boxtimes \boxtimes \Phi \equiv \Phi$.

Let us now make some observations. Let Φ be a CNF, Φ' be the DNF defined as $\Phi' = \bigvee_{C \text{ such that } C \models \Phi} C$, and Φ'' the CNF defined as $\Phi'' = \bigwedge_{C' \text{ such that } \Phi' \models C'} C'$, where both Φ' and Φ'' are free from tautologies, contradictions and subsumed elements. We have:

- $\Phi \equiv \Phi' \equiv \Phi''$;
- $\forall C \in \Phi'$, C is a prime implicant of Φ;
- $\forall C' \in \Phi''$, C' is a prime implicate of Φ.

Theorem 2. $\Phi'' = \boxtimes \boxtimes \Phi$.

Proof. Let us simply observe the following:

- A (minimal) hitting set of the sets in Φ that is not a contradiction is a (prime) implicant of Φ: indeed, if every clause in Φ is satisfied, then Φ is satisfied too.
- Conversely a (minimal) hitting set of the sets in Φ' that is not a tautology is a (prime) implicate of Φ', and thus of Φ too. □

In summary, this discussion tells us that converting a CNF to a DNF and then back to a CNF produces all the prime implicates of the initial CNF. In other words, applying twice the $\boxtimes$ operator is an alternative way to compute all prime implicates of a formula.

Example 6.4.2. Consider again Example 6.4.1, with $\Phi = \{xy, \neg xz\}$. We have $\boxtimes\Phi = \{xz, \neg xy, yz\}$, and $\boxtimes \boxtimes \Phi = \{xy, \neg xz, yz\}$. The new clause $y \vee z$ is a prime implicate of $(x \wedge y) \vee (\neg x \wedge z)$.

The last step in our approach is to implement this operator on ZBDDs encoding collections of sets of literals. Let Φ be a collection of sets of literals, and suppose we interpret it as a DNF. Let l be a literal involved in Φ. We can rewrite this DNF as $\Phi = (l \wedge f_l) \vee \bar{f}_l$, where f_l is a DNF containing all the sets in Φ involving l, from which l has been removed, and $\bar{f}_l$ is a DNF containing all the sets in Φ that do not involve l. Then $\boxtimes\Phi = (l \vee \boxtimes\bar{f}_l) \wedge (\boxtimes\bar{f}_l \vee \boxtimes f_l)$. This allows us to represent $\boxtimes\Phi$ as a ZBDD as shown in Figure 6.5.

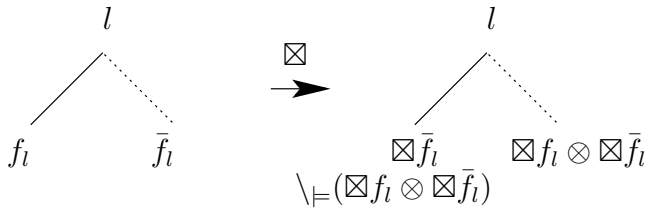


Figure 6.5: Implementing the $\boxtimes$ operator on ZBDDs

6.4.2 Generalisation to Domain Consequences

The operator for the boolean case can be generalised in a straightforward way to the non-boolean case. Most of the previous discussion holds irrespective of the size of the domains. The only point of attention arises from the definition of trivial sets.

Let $\mathcal{P}$ be a set of domain sequences. So far, we interpreted $\mathcal{P}$ as a set of domain consequences. In that case, we can say that $\mathcal{P}$ is interpreted in conjunctive form, i.e. that $\forall P = \langle D_1, \ldots, D_n \rangle \in \mathcal{P}$, P is interpreted as the disjunction $X_1 \in D_1 \vee \ldots \vee X_n \in D_n$, and $\mathcal{P}$ is interpreted as the conjunction $\bigwedge_{P \in \mathcal{P}} P$ of its domains consequences. Conversely, we say that $\mathcal{P}$ is interpreted in disjunctive form when $\forall P = \langle D_1, \ldots, D_n \rangle \in \mathcal{P}$, P is interpreted as the conjunction $\forall i \leq n \forall a \in D_i, X_i = a$, and $\mathcal{P}$ is interpreted as the disjunction $\bigvee_{P \in \mathcal{P}} P$. In particular, if $\mathcal{P} = \emptyset$, it interprets as $true$ in conjunctive form and $false$ in disjunctive form,

and if $\mathcal{P} = \{\emptyset\}$, it interprets as $false$ in conjunctive form and $true$ in disjunctive form.

For a given $P \in \mathcal{P}$, where $\mathcal{P}$ is interpreted in conjunctive form, $P = \langle D_1, \ldots, D_n \rangle$ is a tautology if $\exists i$ such that $D_i = D(X_i)$, i.e. all possible values are allowed for X_i. For a given $P \in \mathcal{P}$, where $\mathcal{P}$ is interpreted in disjunctive form, $P = \langle D_1, \ldots, D_n \rangle$ is a contradiction if $\exists i$ such that $|D_i| > 1$, i.e. X_i must have two distinct values at the same time. Note that the definitions of tautologies and contradictions are equivalent in the case of domains of size 2. This implies that the $\boxtimes$ operator could be defined simply on collections of sets of literal, irrespective of whether they interpret as CNF or DNF. When generalising to domains of arbitrary size, two distinct operators must be defined, taking into account the interpretation of $\mathcal{P}$: one for the conversion from conjunctive to disjunctive form, denoted $\boxtimes_\vee$, and another for the conversion back to conjunctive form, denoted $\boxtimes_\wedge$. At the end of this double conversion, we obtain that $\boxtimes_\wedge \boxtimes_\vee \mathcal{P} = Cons(\mathcal{P})$.

6.4.3 The Algorithm

Let M be an ordered automata free from subsumed elements, with initial state q_0. The two operators $\boxtimes_\vee$ and $\boxtimes_\wedge$ can be described in a general fashion as follows, where either operator is simply denoted by $\boxtimes q_0$.

Trivial elements may be filtered similarly to what was done with the product operator. We need to introduce some additional notation to deal with the case of the disjunctive form, i.e. when trivial elements are in the form of contradictions. For each $X_i, 1 \leq i \leq n$, we define, for each $a \in D(X_i)$, the domain sequence $P^a_{X_i} = \langle D^i_1, \ldots, D^i_n \rangle$, with $D^i_i = \{a\}$, and $D^i_j = \emptyset$ if $i \neq j$. For each $X_i, 1 \leq i \leq n$, we define the state $q^\vee_{X_i}$ as the initial state of the ordered automaton recognising $\{P^a_{X_i} / a > \min D(X_i)\}$. For example, suppose $n = 3$, $i = 2$ and $D(X_2) = \{abc\}$. We have $\mathcal{P}(q^\vee_{X_2}) = \{\langle \emptyset, \{b\}, \emptyset \rangle, \langle \emptyset, \{c\}, \emptyset \rangle\}$.

Proposition 3. *$\mathcal{P}(\boxtimes_\vee q_0) = \boxtimes_\vee \mathcal{P}(q_0)$ and $\mathcal{P}(\boxtimes_\wedge q_0) = \boxtimes_\wedge \mathcal{P}(q_0)$.*

Proof. Consider first the terminal cases.

If $\mathcal{P}(q_0) = \{\emptyset\}$, q_0 is accepting, and $q_\emptyset$ is returned, and thus $\mathcal{P}(\boxtimes q_0) = \emptyset$.

If $\mathcal{P}(q_0) = \emptyset$, $q_0 = q_\emptyset$. In that case, the **forall** loop is not executed, and q'_0 is returned as such, in which case $\mathcal{P}(\boxtimes q_0) = \{\emptyset\}$.

Function $\boxtimes q_0$

if q_0 *is final* **then return** $q_\emptyset$
else
  $q'_0 \leftarrow$ Create new state
  $isFinal(q'_0) \leftarrow true$
  $q'_0 \leftarrow$ `register-state`(q'_0)
  forall $a \in \sigma(q_0)$ *in decreasing order* **do**
    $q''_0 \leftarrow$Duplicate $q'_0 \otimes \boxtimes\delta(q_0, a)$
    Let X_i be the variable to which a belongs
    if *Operator implemented is* $\boxtimes_\vee$ **then**
      $q'_0 \leftarrow q'_0 \setminus\!\models q_{X_i^\vee}$
    if *Operator implemented is* $\boxtimes_\wedge$ **then**
      if $a = \min D(X_i)$ **then**
        $q'_0 \leftarrow q'_0 \setminus\!\models q_{X_i^\wedge}$
    $\delta(q''_0, a) \leftarrow (q'_0 \setminus\!\models q''_0)$
    $q'_0 \leftarrow$ `register-state`(q''_0)
  return q'_0

Consider now the general case, without taking into account trivial element filtering. Figure 6.6 shows how the operator can be recursively represented by an ordered-automaton, very similarly to the ZBDD representation previously discussed. The invariant that is maintained in the **forall** loop is that $q'_0 = \boxtimes\mathcal{P}_2$ (with $\mathcal{P}_2$ referring to the notation used in Figure 6.6). It is indeed the case at the first iteration, as $\mathcal{P}_2 = \emptyset$. During an iteration, q''_0 is built according to the method depicted in Figure 6.6, and at the end of the iteration, the result is assigned to q'_0.

Consider the filtering of trivial elements, which is what differentiates the two actual operators. q'_0 is the state that will be reached by value a. Initially, it is set to $\boxtimes\mathcal{P}_2$. In the case of the distribution to disjunctive form, all elements in $\boxtimes\mathcal{P}_2$ that contain at least one value (but never more than one, by induction hypothesis) belonging to the domain of the same variable X_i must be removed from $\boxtimes\mathcal{P}_2$. This is achieved by applying $q'_0 \setminus\!\models q^\vee_{X_i}$, as any such element will be subsumed by one of the elements in $\mathcal{P}(q^\vee_{X_i})$. On the other hand, in the case of the distribution to conjunctive form, all elements in $\boxtimes\mathcal{P}_2$ that contain all the values but a from of the domain of the same variable X_i must be removed from $\boxtimes\mathcal{P}_2$. This is achieved by applying $q'_0 \setminus\!\models q^\wedge_{X_i}$, as any such element will be subsumed by the unique element

in $\mathcal{P}(q^{\wedge}_{X_i})$. Note that we have to assume also that the automaton returned by $\boxtimes\mathcal{P}_1 \otimes \boxtimes\mathcal{P}_2$, to which q''_0 points in the algorithm, is free from trivial elements. This is the case when the $\boxtimes_{\wedge}$ operator is considered, as the product operator, as it has been presented in 6.3.5, filters out tautologies, but in the case of the $\boxtimes_{\vee}$ operator, the product operator has to be adapted to filter out contradictions in the same way as it has been presented here. □

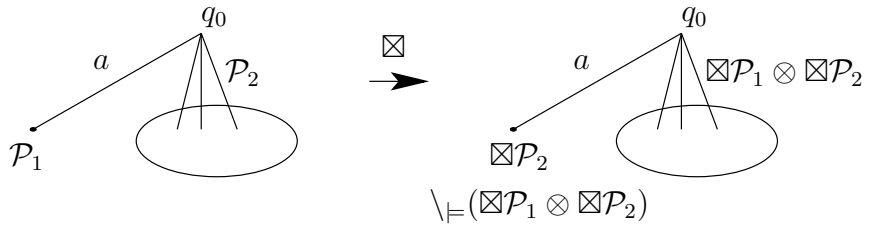


Figure 6.6: The general form of the distribution operator on ordered automata

It follows that these operators allow us to compute the closure by resolution of a set of domain consequences encoded by an ordered automaton.

Corollary. $\mathcal{P}(\boxtimes_{\wedge} \boxtimes_{\vee} q_0) = Cons(\mathcal{P}(q_0))$.

Concerning the complexity, note that computing the closure by resolution does not have a complexity that is polynomial in the size of the initial input and final output. Indeed, the size of the intermediate automaton in disjunctive form is unrelated to that of the initial and final input. But this is not surprising and cannot be overcome. Indeed, if one could generate all the domain consequences of a problem in a time polynomial in the number of the domain consequences, then this would provide a method to generate all conflicts of a problem in a time polynomial in the number of conflicts. But this contradicts the complexity result in Section 3.5, stating that generating all the minimal conflicts or all the maximal relaxations of a problem can only be achieved by incurring the cost related to the number of both (unless *P = NP*).

6.4.4 Initialising the Compilation

The last step for compiling a problem into an ordered automaton, representing its set of domain consequences, consists in describing how we build the initial

set of domain consequences that is equivalent to the problem, and on which the closure by resolution can be computed to infer *all* domain consequences. Just as Algorithm 8 of Chapter 5 initialises the set of domain consequences with the domain consequences of each constraint, we first build an ordered automaton for each constraint, then combine each of them to a single ordered automaton, and then apply the closure by resolution operator to infer all the domain consequences of the problem. All these steps are straightforward with the use of our operators.

The first step consists in inferring all the domain consequences of a single constraint. Assuming we have an ordered automaton representing the constraint in disjunctive form, it is simply a matter of applying the $\boxtimes_\wedge$ operator to it. But then, an ordered automaton in disjunctive form is simply a classic automaton. We can therefore use the following procedure. Using the procedure described in Section 4.2.2, we build an automaton encoding the list of tuples of the constraint. We then map the values in this automaton to the values of the alphabet of the target ordered automaton. For example, the tuple 000 on variables X_1, X_2, X_3, with domain $D(X_i) = \{012\}$ will be mapped to 036. Finally, we apply the $\boxtimes_\wedge$ to the resulting automaton to obtain an ordered automaton representing all the domain consequences of the constraint.

The second step consists in combing the automata corresponding to each constraint to a single one. This can be simply done using the $\cup_\mu$ operator. We then obtain an ordered automaton representing a set of domain consequences that is equivalent to the problem. We can now apply the closure by resolution operator to it.

Chapter 7

Conclusion

Summary. *In this chapter, we conclude this dissertation with a summary of our contributions and the thesis we have defended. We then briefly examine several directions for future work.*

7.1 Thesis Defence

The thesis defended throughout this dissertation was the following. Existing methods for computing explanations in constraint programming do not provide satisfactory explanations to users of interactive configuration systems. We study new approaches to enable improved explanations. We claim that we can compute such type of explanations by defining and operating within a framework where user interaction consists of unary constraints, and where different compilation strategies are applied to allow for fast computation times during interactive phases. This thesis can be regarded as the conjunction of three sub-theses, which we present here along with a summary of their defence by this dissertation.

Sub-thesis 1. *Showing a set of relaxations that is at the same time informative and compact, while giving a better picture of a problem than showing a single relaxation, can be achieved in practice.*

In Chapter 3, we defined the concept of a representative set of relaxations, which shows the user at least one way to satisfy each of his requirements and at

least one way to relax them. We studied the complexity of computing such a set in Sections 3.5 and 3.4.3, which we overcome with an algorithm that heuristically converges to a representative set of relaxations (Section 3.6). We showed how this notion can be extended to take into account combinations of a user's requirements, or a user's preferences between his requirements in Section 3.7.

Sub-thesis 2. *The choice of a relaxation needs to take into account the solutions it allows. This can be done in an aggregate form by computing most soluble or least soluble relaxations. A relaxation with the highest number of solutions leaves the user with the largest choice. A relaxation with the lowest number of solution corresponds the most closely to the user's original requirements.*

In Chapter 4, we defined algorithms to compute such relaxations using compiled representations of a problem to cope with the complexity of computing those. We defined algorithms based on automata in Section 4.4.1 and generalised these procedures to make them independent from any particular representation in Section 4.4.2, by identifying the structural properties a representation must satisfy for them to apply. We showed through an experimental study in Section 4.5 how the choice of a compact representation resulted in very efficient procedures.

Sub-thesis 3. *We claim that a complete knowledge of the conflicts of a problem helps compute more useful explanations. It is possible to circumvent the complexity associated with computing conflicts and arising from their high number by compiling a problem to a new type of representation that contains explicit knowledge of its conflicts.*

In Chapter 5, we defined the concept of domain consequence, which generalises the concept of prime implicate to constraint problems, and proposed to represent a problem in terms of its domain consequences. This defines a new compilation strategy that represents the conflicts inherent to a problem, before the user specifies any particular requirement. In Sections 5.3 and 5.8, we demonstrated how doing this computation in advance can benefit to the computation of explanations. In Chapter 6, we presented a data structure and a series of algorithms to represent in a compact way a large set of domain consequences, and that enable to compute the domain consequences of a problem in a more efficient way.

7.2 Directions for Future Work

There are several possible directions for the work carried out in this dissertation to progress, some dealing with theoretical issues, but other concerning practical aspects. We outline these below.

Implementation of Ordered Automata. We need to efficiently implement the operators outlined throughout Chapter 6. The algorithms we presented intended to be as close as possible to a function definition, rather than be of practical interest concerning their implementation. The actual implementation of these procedures require additional technical work. In particular, the operators need to be expressed in a more procedural way, and attention has to be given to careful memory management. As this can lead to minute technical considerations, it is worth instead implementing our data structures and operators on top of an existing powerful BDD package, such as BuDDy [93], thus benefitting from experience in the matter. Their efficiency, as well as the efficiency of the overall compilation procedure has to be experimentally evaluated. What is the size of the resulting structure in relation to the number of domain consequences it encodes? What is the size of the intermediate structure? As such a procedure could be used to enumerate explanations, is the overall procedure competitive against explanation enumeration procedures?

Performance of the Compilation Method. We need to understand the characteristics of problems for which this new compilation method achieves the best results. What influences the number of domain consequences of a problem? Does this method work better on problems presenting a particular structure, similar to the way traditional compilation methods exploit tree-structured problems? In an orthogonal way, another, maybe more realistic, direction will be to define interesting restrictions on the domain conflicts we compute.

Querying Ordered Automata. In order to make use of the result of this compilation method, we need to be able to efficiently query this structure. We need

to define what type of queries are needed. Examples of tasks we need to achieve include the following:

- Add to the current database a new minimal domain consequence if it is not subsumed by any element currently in the database.
- Find a maximal domain conflict in the database for a user query.
- Compute a smallest conflict of a user query.
- Count the number of conflicts for a user query.

Some of them only require a straightforward application of one or several of the operators we defined, but some require further algorithmic study. We will have to carefully measure the efficiency of explanation computation from this representation, as well as, of course, compare it with other existing representations and algorithms.

Explanation Enumeration Algorithms. Explanation enumeration algorithms is still a challenging research direction, and there is potential for new approaches. In particular for the case of partial enumeration, there are two important requirements for an enumeration procedure: it should find each new relaxation quickly, and it should generate diverse relaxations, where the notion of diversity is underpinned by the notion of representativeness. It seems promising to apply an approach similar to the approach of Liffiton and Sakallah [90] to constraint programming. In particular, propagating an ATMOST constraint in a constraint model can be highly beneficial, in particular in our setting where user constraints consist in unary equality or membership constraints. The advantage of an enumeration procedure based on a CP model is that it has the flexibility to easily support the additional requirements we mentioned at the beginning of this paragraph.

Representative Explanations. Future work should focus on developing suitable user interfaces for presenting representative explanations to users, informed by in-depth user studies. Also, we should study how to inform our choice of representative explanations by considering the user's preferences over constraints. The acceptance by users of the explanations provided has to be evaluated.

Most Soluble Explanations. One direction for future work concerns the design of non-exact methods to compute most soluble explanations. We saw in Section 4.5, Chapter 4, that using compiled presentations resulted in very efficient exact procedures. Heuristic procedures are, therefore, needed in situations where compilation is not practical. The heuristics mentioned in Section 4.5, `minimise/maximise solution loss`, are interesting as an indication for future work; we want to look for heuristics that achieve the same purpose but in a less brutal way.

Toolkit. Finally we should develop a toolkit based on the various explanation algorithms and compilation procedures developed in this dissertation.

Bibliography

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artif. Intell.*, 135(1-2):199–234, 2002. 4, 34, 38, 43, 44, 57, 78, 84, 86, 99, 106, 108

[2] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM. J. on Algebraic and Discrete Methods*, 8(2):277–284, April 1987. 92

[3] James Bailey and Peter J. Stuckey. Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *PADL*, volume 3350 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2005. ISBN 3-540-24362-3. 57, 58, 69, 73, 74, 84, 101, 106, 107, 108

[4] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983. 14

[5] Christian Bessière. *Constraint Propagation*, chapter 3, pages 29–84. In Rossi et al. [118], 2006. ISBN 0444527265. 20, 21

[6] Christian Bessière and Jean-Charles Régin. Arc Consistency for General Constraint Networks: Preliminary Results. In *IJCAI (1)*, pages 398–404, 1997. 87

[7] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsis-

tent systems: structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1): 25–46, 2003. 57

[8] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010. 92

[9] James Bowen. Using dependency records to generate design coordination advice in a constraint-based approach to concurrent engineering. *Computers in Industry*, 33(2-3):191–199, 1997. ISSN 0166-3615. doi: http://dx.doi.org/10.1016/S0166-3615(97)00024-9. 27, 44

[10] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. 34, 94

[11] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. 34

[12] Thierry Castell and Hélène Fargier. Propositional Satisfaction Problems and Clausal CSPs. In *ECAI*, pages 214–218, 1998. 117

[13] Philippe Chatalic and Laurent Simon. Multi-resolution on compressed sets of clauses. In *ICTAI*, pages 2–10. IEEE Computer Society, 2000. ISBN 0-7695-0909-6. 130, 136

[14] William J. Clancey. The Epistemology of a Rule-Based Expert System - A Framework for Explanation. *Artif. Intell.*, 20(3):215–251, 1983. 40

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971. 1

[16] O. Coudert and JC Madre. A New Method to Compute Prime and Essential Prime Implicants of Boolean Functions. In *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*. Mit Pr, 1992. 130

[17] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard Watson. Incremental Construction of Minimal Acyclic Finite State Automata. *Computational Linguistics*, 26(1):3–16, 2000. 87

[18] Adnan Darwiche. Decomposable negation normal form. *J. ACM*, 48(4): 608–647, 2001. 108

[19] Adnan Darwiche. On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001. 93, 102

[20] Adnan Darwiche. A Compiler for Deterministic, Decomposable Negation Normal Form. In *AAAI/IAAI*, pages 627–634, 2002. 34, 44

[21] Adnan Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In de Mántaras and Saitta [30], pages 328–332. ISBN 1-58603-452-9. 98

[22] Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002. 9, 33, 92, 93, 95, 96, 113

[23] Johan de Kleer. An Assumption-Based TMS. *Artif. Intell.*, 28(2):127–162, 1986. 26, 27

[24] Johan de Kleer. Problem Solving with the ATMS. *Artif. Intell.*, 28(2): 197–224, 1986. 26

[25] Johan de Kleer. A Comparison of ATMS and CSP Techniques. In *IJCAI*, pages 290–296, 1989. 26

[26] Johan de Kleer. An Improved Incremental Algorithm for Generating Prime Implicates. In *AAAI*, pages 780–785, 1992. 130

[27] Johan de Kleer and Brian C. Williams. Diagnosing Multiple Faults. *Artif. Intell.*, 32(1):97–130, 1987. 28, 57

[28] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing Diagnoses and Systems. *Artif. Intell.*, 56(2-3):197–222, 1992. 27, 32, 112, 129

[29] Maria J. García de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *PPDP*, pages 32–43. ACM, 2003. ISBN 1-58113-705-2. 58

[30] Ramon López de Mántaras and Lorenza Saitta, editors. *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 2004. IOS Press. ISBN 1-58603-452-9. 163, 166

[31] J. L. de Siqueira N. and Jean-Francois Puget. Explanation-based generalisation of failures. In *ECAI*, pages 339–344, 1988. 4

[32] Rina Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artif. Intell.*, 41(3):273–312, 1990. 25

[33] Rina Dechter, editor. *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, 2000. Springer. ISBN 3-540-41053-8. 169, 172, 175

[34] Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artif. Intell.*, 38(3):353–366, 1989. 13, 14, 34, 92

[35] Rina Dechter and Peter van Beek. Local and Global Relational Consistency. *Theor. Comput. Sci.*, 173(1):283–308, 1997. 21

[36] Alvaro del Val. Tractable Databases: How to Make Propositional Unit Resolution Complete Through Compilation. In *KR*, pages 551–561, 1994. 34, 39, 130

[37] Alvaro del Val. A New Method for Consequence Finding and Compilation in Restricted Languages. In *AAAI/IAAI*, pages 259–264, 1999. 129

[38] Alvaro del Val. The Complexity of Restricted Consequence Finding and Abduction. In *AAAI/IAAI*, pages 337–342. AAAI Press / The MIT Press, 2000. ISBN 0-262-51112-6. 129

[39] Marc Denecker and Antonis C. Kakas. Abduction in Logic Programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic*

Programming and Beyond, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer, 2002. ISBN 3-540-43959-5. 24

[40] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995. 3, 4

[41] Thomas Eiter and Kazuhisa Makino. On Computing all Abductive Explanations. In *AAAI/IAAI*, pages 62–67, 2002. 3, 67

[42] Hélène Fargier and Pierre Marquis. On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae. In *AAAI*. AAAI Press, 2006. 44

[43] Hélène Fargier and Marie-Catherine Vilarem. Compiling CSPs into tree-driven automata for interactive solving. *Constraints*, 9(4):263–287, 2004. 44, 78, 89, 92

[44] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artif. Intell.*, 152(2):213–234, 2004. 45

[45] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic variable reordering for BDD minimization. In *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93. European*, pages 130–135, Sep 1993. doi: 10.1109/EURDAC.1993.410627. 34

[46] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998. 39, 43

[47] Filippo Focacci and Michela Milano. Global Cut Framework for Removing Symmetries. In Walsh [138], pages 77–92. ISBN 3-540-42863-1. 117

[48] Eugene Freuder, Chavalit Likitvivatanavong, Manuela Moretti, Francesca Rossi, and Richard Wallace. Computing Explanations and Implica-

tions in Preference-Based Configurators. *Recent Advances in Constraints*, pages 315–336, 2003. URL http://dx.doi.org/10.1007/3-540-36607-5_6. 41, 43, 44

[49] Eugene C. Freuder. Synthesizing Constraint Expressions. *Commun. ACM*, 21(11):958–966, 1978. 21

[50] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *J. ACM*, 29(1):24–32, 1982. 21, 92

[51] Eugene C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *J. ACM*, 32(4):755–761, 1985. 21

[52] Eugene C. Freuder and Barry O'Sullivan. Generating Tradeoffs for Interactive Constraint-Based Configuration. In Walsh [138], pages 590–594. ISBN 3-540-42863-1. 45

[53] Eugene C. Freuder and Richard J. Wallace. Partial Constraint Satisfaction. *Artif. Intell.*, 58(1-3):21–70, 1992. 86

[54] Eugene C. Freuder, Chavalit Likitvivatanavong, Manuela Moretti, Francesca Rossi, and Richard J. Wallace. Computing Explanations and Implications in Preference-Based Configurators. In Barry O'Sullivan, editor, *International Workshop on Constraint Solving and Constraint Logic Programming*, volume 2627 of *Lecture Notes in Computer Science*, pages 76–92. Springer, 2002. ISBN 3-540-00986-8. 41

[55] Gerhard Friedrich. Elimination of Spurious Explanations. In de Mántaras and Saitta [30], pages 813–817. ISBN 1-58603-452-9. 28, 31, 50

[56] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7. 2, 16

[57] Matthew L. Ginsberg. Dynamic Backtracking. *J. Artif. Intell. Res. (JAIR)*, 1:25–46, 1993. 25

[58] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001. 14

[59] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a Complete Technique to Find MSS and MUS Thanks to a Local Search Oracle. In Manuela M. Veloso, editor, *IJCAI*, pages 2300–2305, 2007. 58, 60, 69

[60] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharm. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003. 57

[61] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast Backtrack-Free Product Configuration using a Precompiled Solution Space Representation. In *In Proceedings of PETO Conference*, pages 131–138. DTU-tryk, June 2004. 34, 43

[62] T. Hadzic, E.R. Hansen, and B. O'Sullivan. On automata, mdds and bdds in constraint satisfaction. In *Proceedings of the ECAI 2008 Workshop on Inference Methods based on Graphical Structures of Knowledge*. Citeseer, 2008. 97

[63] Rolf Haenni. A Query-Driven Anytime Algorithm for Argumentative and Abductive Reasoning. In David W. Bustard, Weiru Liu, and Roy Sterritt, editors, *Soft-Ware*, volume 2311 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2002. ISBN 3-540-43481-X. 130

[64] Rolf Haenni. Detecting Conflict-Free Assumption-Based Knowledge Bases. *Intelligent systems for information processing: from representation to applications*, page 203, 2003. 129

[65] Albert Hagg, Ulrich Junker, and Barry O'Sullivan. A Survey of Explanation Techniques for Configurators. In *Proceedings of ECAI-2006 Workshop on Configuration*, August 2006. 41, 44

[66] Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999. 58

[67] G.H. Harman. The inference to the best explanation. *The Philosophical Review*, 74(1):88–95, 1965. ISSN 0031-8108. 24

[68] H.H. Hoos and T. Stützle. *Stochastic local search: Foundations and applications*. Morgan Kaufmann, 2005. ISBN 1558608729. 17

[69] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 3. Addison-wesley Reading, MA, 1979. 36

[70] Aimin Hou. A Theory of Measurement in Diagnosis from First Principles. *Artif. Intell.*, 65(2):281–328, 1994. 58

[71] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.*, 42(2):100–111, 1999. 89

[72] Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993. 136

[73] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On Generating All Maximal Independent Sets. *Inf. Process. Lett.*, 27(3):119–123, 1988. 65

[74] Ulrich Junker. QuickXplain: conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle, WA, USA, August 2001. URL `http://www.lirmm.fr/~bessiere/ws_ijcai01/junker.ps.gz`. 2, 4, 30, 31, 45, 51

[75] Ulrich Junker. QuickXplain: preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004. ISBN 0-262-51183-5. 4, 30, 45, 52

[76] Ulrich Junker. *Configuration*, chapter 24, pages 837–874. In Rossi et al. [118], 2006. ISBN 0444527265. 39, 41

[77] Belarmino Pulido Junquera and Carlos Alonso González. Possible conflicts: a compilation technique for consistency-based diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(5):2192–2206, 2004. 39, 130

[78] N. Jussien. e-Constraints: Explanation-based constraint programming. In *Proc. CP01 Workshop on User-Interaction in Constraint Satisfaction, Paphos, Cyprus*, 2001. 30

[79] Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *WLPE*, 2001. 23, 27

[80] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining Arc-Consistency within Dynamic Backtracking. In Dechter [33], pages 249–261. ISBN 3-540-41053-8. 2, 25, 30

[81] AC Kakas, RA Kowalski, and F. Toni. The Role of Abduction in Logic Programming. *Handbook of Logic in Artificial Intelligence and Logic Programming: Logic programming*, page 235, 1998. 24, 27

[82] T. Kam, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4:9–62, 1998. 34, 97

[83] George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 390–396. AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X. 117

[84] Dimitris J. Kavvadias, Martha Sideri, and Elias C. Stavropoulos. Generating all maximal models of a Boolean expression. *Inf. Process. Lett.*, 74 (3-4):157–162, 2000. 67

[85] Alex Kean and George K. Tsiknis. An Incremental Method for Generating Prime Implicants/Impicates. *J. Symb. Comput.*, 9(2):185–206, 1990. 124

[86] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. Treewidth: Computational Experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57, 2001. 92

[87] F. Laburthe. CHOCO: implementing a CP kernel. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, pages 71–85, 2000. 89

[88] François Laburthe and Yves Caseau. Using Constraints for Exploring Catalogs. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 883–888. Springer, 2003. ISBN 3-540-20202-1. 41, 43

[89] D.B. Leake. *Evaluating Explanations*. Lawrence Erlbaum Associates, 1992. ISBN 0-8058-1064-1. 2

[90] Mark H. Liffiton and Karem A. Sakallah. On Finding All Minimally Unsatisfiable Subformulas. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 173–186. Springer, 2005. ISBN 3-540-26276-8. 58, 158

[91] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008. 58, 59

[92] Mark H. Liffiton, Michael D. Moffitt, Martha E. Pollack, and Karem A. Sakallah. Identifying Conflicts in Overconstrained Temporal Problems. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 205–211. Professional Book Center, 2005. ISBN 0938075934. 58, 59

[93] J. Lind-Nielsen. BuDDy, A Binary Decision Diagram Package. http://buddy.sourceforge.net/. 157

[94] Alan K. Mackworth. Consistency in Networks of Relations. *Artif. Intell.*, 8 (1):99–118, 1977. 17, 19

[95] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artif. Intell.*, 25(1):65–74, 1985. 16

[96] P. Marquis. Consequence finding algorithms. *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, 5:41–145, 2000. 31, 112, 129

[97] Pierre Marquis. Knowledge Compilation Using Theory Prime Implicates. In *IJCAI (1)*, pages 837–845, 1995. 34, 39, 130

[98] Robert Mateescu and Rina Dechter. Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs). In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2006. ISBN 3-540-46267-8. 44, 97

[99] K. McCarthy, J. Reilly, L. McGinty, and B. Smyth. Thinking positively-explanatory feedback for conversational recommender systems. In *Proceedings of the European Conference on Case-Based Reasoning (ECCBR-04) Explanation Workshop*, pages 115–124. Citeseer, 2004. 28

[100] Sanjay Mittal and Felix Frayman. Towards a generic model of configuraton tasks. In *IJCAI'89: Proceedings of the 11th international joint conference on Artificial intelligence*, pages 1395–1401, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. 40, 41

[101] Barry O'Callaghan, Barry O'Sullivan, and Eugene C. Freuder. Generating Corrective Explanations for Interactive Constraint Satisfaction. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 445–459. Springer, 2005. ISBN 3-540-29238-1. 28, 31, 43, 45

[102] Christos H. Papadimitriou. NP-Completeness: A Retrospective. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 2–6. Springer, 1997. ISBN 3-540-63165-8. 65

[103] B. Pargamin. Extending cluster tree compilation with non-boolean variables in product configuration: A tractable approach to preference-based configuration. In *Proceedings of the IJCAI'03 Workshop on Configuration*, volume 3, 2003. 34, 43, 44

[104] Gilles Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004. ISBN 3-540-23241-9. 38

[105] Knot Pipatsrisawat and Adnan Darwiche. New Compilation Languages Based on Structured Decomposability. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 517–522. AAAI Press, 2008. ISBN 978-1-57735-368-3. 98

[106] D. Poole, R. Goebel, and R. Aleliunas. Theorist: a logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer Verlag, 1987. 24, 30

[107] David Poole. Normality and Faults in Logic-Based Diagnosis. In *IJCAI*, pages 1304–1310, 1989. 30

[108] Pearl Pu, Boi Faltings, and Marc Torrens. Effective Interaction Principles for Online Product Search Environments. In *Web Intelligence*, pages 724–727. IEEE Computer Society, 2004. ISBN 0-7695-2100-2. 28, 50

[109] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison Wesley, 2000. 6

[110] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994. 17, 20

[111] Jean-Charles Régin, Thierry Petit, Christian Bessière, and Jean-Francois Puget. An original constraint based approach for solving over constrained problems. In Dechter [33], pages 543–548. ISBN 3-540-41053-8. 2

[112] Raymond Reiter. A Theory of Diagnosis from First Principles. *Artif. Intell.*, 32(1):57–95, 1987. 28, 29, 57, 58

[113] Raymond Reiter and Johan de Kleer. Foundations of Assumption-based Truth Maintenance Systems: Preliminary Report. In *AAAI*, pages 183–189, 1987. 26, 27, 32

[114] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. 15

[115] Neil Robertson and Paul D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms*, 7(3):309–322, 1986. 15

[116] Neil Robertson and Paul D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *J. Comb. Theory, Ser. B*, 52(2):153–190, 1991. 15

[117] G. Rochart, N. Jussien, and F. Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*. Citeseer, 2003. 30

[118] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. ISBN 0444527265. 1, 161, 168

[119] Daniel Sabin and Rainer Weigel. Product Configuration Frameworks-A Survey. *IEEE Intelligent Systems*, 13(4):42–49, 1998. ISSN 1541-1672. doi: http://dx.doi.org/10.1109/5254.708432. 39, 40, 41, 43

[120] Ken Satoh and Takeaki Uno. Enumerating Minimally Revised Specifications Using Dualization. In Takashi Washio, Akito Sakurai, Katsuto Nakajima, Hideaki Takeda, Satoshi Tojo, and Makoto Yokoo, editors, *JSAI Workshops*, volume 4012 of *Lecture Notes in Computer Science*, pages 182–189. Springer, 2005. ISBN 3-540-35470-0. 58

[121] Ken Satoh and Takeaki Uno. Enumerating Minimal Explanations by Minimal Hitting Set Computation. In Jérôme Lang, Fangzhen Lin, and Ju Wang, editors, *KSEM*, volume 4092 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 2006. ISBN 3-540-37033-1. 58, 67

[122] Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *ICTAI*, pages 48–55, 1993. 25

[123] Bart Selman and Hector J. Levesque. Abductive and Default Reasoning: A Computational Core. In *AAAI*, pages 343–348, 1990. 3, 27, 32

[124] Bart Selman and Hector J. Levesque. Support set selection for abductive and default reasoning. *Artif. Intell.*, 82(1-2):259–272, 1996. 3

[125] Laurent Simon and Alvaro del Val. Efficient Consequence Finding. In Bernhard Nebel, editor, *IJCAI*, pages 359–370. Morgan Kaufmann, 2001. ISBN 1-55860-777-3. 113, 130

[126] Frode Sørmo, Jörg Cassens, and Agnar Aamodt. Explanation in Case-Based Reasoning-Perspectives and Goals. *Artif. Intell. Rev.*, 24(2):109–143, 2005. 22

[127] Mohammed H. Sqalli and Eugene C. Freuder. Inference-Based Constraint Satisfaction Supports Explanation. In *AAAI/IAAI, Vol. 1*, pages 318–325, 1996. 22, 27

[128] Kostas Stergiou and Manolis Koubarakis. Backtracking Algorithms for Disjunctions of Temporal Constraints. In *AAAI/IAAI*, pages 248–253, 1998. 59

[129] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artif. Intell.*, 120(1):81–117, 2000. 59

[130] Markus Stumptner. An Overview of Knowledge-Based Configuration. *AI Commun.*, 10(2):111–125, 1997. 39, 40

[131] Sathiamoorthy Subbarayan. Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems. In Roman Barták and Michela Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005. ISBN 3-540-26152-4. 34, 44

[132] Robert Endre Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984. 14, 92

[133] P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *IEEE Transactions on Electronic Computers*, 16(4):446–456, 1967. 113, 129

[134] Nageshwara Rao Vempaty. Solving Constraint Satisfaction Problems Using Finite State Automata. In *AAAI*, pages 453–458, 1992. 34, 35, 36, 86

[135] I.M. Vinogradov and M. Hazewinkel. *Encyclopaedia of mathematics*. Kluwer, 1988. 56

[136] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996. 1

[137] Toby Walsh. SAT v CSP. In Dechter [33], pages 441–456. ISBN 3-540-41053-8. 97

[138] Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, 2001. Springer. ISBN 3-540-42863-1. 165, 166

[139] Rainer Weigel and Boi Faltings. Compiling constraint satisfaction problems. *Artif. Intell.*, 115(2):257–287, 1999. 34, 44