MONOGRAFIES DE L'IIIA

PROOF PROCEDURES FOR MULTIPLE-VALUED PROPOSITIONAL LOGICS

Felip Manyà Serres

MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 9

Institut d'Investigació
en Intel·ligència Artificial

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Num. 1 J. Puyol, *MILORD II: A Language for Knowledge-Based Systems*

Num. 2 J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions*

Num. 3 Ll. Vila, *On Temporal Representation and Reasoning in Knowledge-Based Systems*

Num. 4 M. Domingo, *An Expert System Architecture for Identification in Biology*

Num. 5 E. Armengol, *A Framework for Integrating Learning and Problem Solving*

Num. 6 J.Ll. Arcos, *The Noos Representation Language*

Num. 7 J. Larrosa, *Algorithms and Heuristics for Total and Partial Constraint Satisfaction*

Num. 8 P. Noriega, *Agent Mediated Auctions: The Fishmarket Metaphor*

Num. 9 F. Manyà, *Proof Procedures for Multiple-Valued Propositional Logics*

Proof Procedures for Multiple-Valued Propositional Logics

Felip Manyà

Foreword by Gonzalo Escalada-Imaz
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Gonzalo Escalada-Imaz
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Felip Manyà
Universitat de Lleida

Institut d'Investigació
en Intel·ligència Artificial

ISBN: 84-00-07793-8
Dip. Legal: B-17127-99
© 1999 by Felip Manyà

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.
Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

Printed by CPDA-ETSEIB.
Avinguda Diagonal, 647.
08028 Barcelona, Spain.

Als meus pares

Contents

Foreword iii

Abstract v

Acknowledgements vii

1 Introduction 1
- 1.1 Context and objectives . . . 1
- 1.2 Contributions . . . 6
- 1.3 Overview of the thesis . . . 7

2 Multiple-Valued Logics 11
- 2.1 Introduction . . . 11
- 2.2 Syntax and semantics of multiple-valued logics . . . 12
  - 2.2.1 Syntax . . . 12
  - 2.2.2 Semantics . . . 13
  - 2.2.3 Examples of multiple-valued logics . . . 14
- 2.3 Multiple-valued signed formulas . . . 15
- 2.4 Regular CNF formulas . . . 17
- 2.5 Monosigned CNF formulas . . . 19
- 2.6 Fuzzy logics as regular logics . . . 19

3 The Satisfiability Problem in Signed CNF Formulas 23
- 3.1 Introduction . . . 23
- 3.2 Resolution-based proof methods . . . 24
  - 3.2.1 Signed resolution . . . 25
  - 3.2.2 Regular resolution . . . 26
  - 3.2.3 Monosigned resolution . . . 28
- 3.3 The Davis-Putnam procedure . . . 28
- 3.4 Maximal truth value set . . . 32
- 3.5 A satisfiability checking procedure for signed CNF formulas . . . 35
  - 3.5.1 Branching heuristics . . . 38
- 3.6 A satisfiability checking procedure for regular CNF formulas . . . 40
  - 3.6.1 Alternative branching rules . . . 44

3.6.2 Branching heuristics 46
3.6.3 Data Structures 46
3.6.4 Deletion strategies 49

4 Polynomially Solvable Satisfiability Problems 53
4.1 Introduction 53
4.2 The Horn SAT problem 54
4.2.1 The Horn SAT problem in mv-formulas 55
4.2.2 The Horn SAT problem in regular CNF formulas 67
4.3 The 2-SAT problem 72
4.3.1 The 2-SAT problem in signed CNF formulas 72
4.3.2 The 2-SAT problem in regular CNF formulas 74
4.3.3 The 2-SAT problem in monosigned CNF formulas 81

5 An Interpreter for Multiple-valued Propositional Logic Programs 89
5.1 Introduction 89
5.2 The logic of programs 91
5.2.1 Syntax 91
5.2.2 Semantics 91
5.2.3 Logical inference 93
5.3 The interpreter 96
5.3.1 A simple version 96
5.3.2 An improved version 101
5.4 Negation 103
5.5 Cut 104
5.6 Implementation 105

6 Conclusions 109

References 111

Index 121

Foreword

Automated theorem proving in multiple-valued logics has received increasing interest and activity in the last years. This fact is motivated by attractive applications relying on multiple-valued logics that have been suggested in fields such as Expert Systems, Logic Programming, Error Correcting Codes, Deductive Databases, Non-Monotonic Reasoning and Multi-Agent Systems. Needless to say that all these applications demand efficient proof procedures for solving real-world problems.

This book contains the description of original satisfiability checking procedures for signed formulas and an efficient interpreter for multiple-valued logic programs; and the ultimate goal of this work is to achieve computationally competitive algorithms in the multiple-valued setting. Because of that, special attention is devoted to the definition of appropriate calculi for mechanical proofs, complexity issues, clever heuristics for scanning the proof search space and suitable data structures for representing instances of problems.

Several authors have already pointed out that the logic of signed formulas offers a suitable formal framework for representing and solving problems for practical applications. The results presented here reinforce such idea by providing new and fast algorithms that can improve the performance of existing applications.

In Chapter 1, the context and objectives of the book are presented and the main contributions are listed. This chapter also provides a summary of the contents of the remaining chapters.

In Chapter 2, the logics used throughout this book are defined and their relevance in the field of automated theorem proving in multiple-valued logics is discussed.

In Chapter 3, the propositional satisfiability problem in both signed and regular CNF formulas is considered. In both cases, Davis-Putnam-style satisfiability checking procedures are described in detail. In particular, this chapter contains original branching rules, branching heuristics, data structures and deletion strategies.

In Chapter 4, it is shown that the propositional satisfiability problem in regular Horn formulas can be solved in linear time when the truth value set is finite, whereas its time complexity is almost linear for infinite truth value sets. On the other hand, it is proved that the propositional satisfiability problem in signed CNF formulas whose clauses are binary is NP-complete. In the case that the clauses are regular or monosigned it is shown that this problem is polynomially solvable by designing quadratic-time algorithms.

In Chapter 5, a linear-time interpreter for infinitely-valued propositional logic programs, enhanced with a cut operator and a negation as failure rule, is described in detail. The low complexity achieved is due to the definition of suitable data structures for representing logic programs and strategies for pruning the proof search space.

In Chapter 6, the conclusions, as well as some open problems and future research perspectives, are presented.

The approach presented here to design and analyze algorithms should be particularly useful to those interested in developing efficient implementations. To identify new polynomially solvable problems, define more powerful calculi for signed formulas and develop encodings of problems for testing and comparing the proof procedures designed are some of the open problems that may attract the attention of other researchers working on multiple-valued logics.

Gonzalo Escalada-Imaz
Bellaterra, 1998

Abstract

In this thesis we design proof procedures for solving both satisfiability and deduction problems in multiple-valued propositional logics. As our ultimate goal is to obtain competitive algorithms from a computational point of view, our approach is not confined to present complete calculi and sketch pseudo-code of algorithms that automate their application. We also pay special attention to the definition of suitable data structures for representing formulas, heuristics for exploring the proof search space and techniques for avoiding redundant and useless computations.

Regarding satisfiability problems we focus on signed CNF formulas, since the satisfiability problem in any finitely-valued propositional logic is polynomially reducible to the satisfiability problem in signed CNF formulas. We design Davis-Putnam-style decision procedures for both signed and regular CNF formulas. To this end, we extend the concept of one-literal rule to the signed setting, introduce the new concept of maximal truth value set and define original branching rules and branching heuristics. For the subclass of regular CNF formulas, we also define suitable data structures for representing formulas and some deletion strategies for eliminating redundant and irrelevant clauses.

Starting out from the fact that there exist polynomial-time algorithms for testing the satisfiability of classical Horn and 2-CNF formulas, we then investigate what happens with these particular subclasses of formulas in the logic of signed CNF formulas. First, we develop a complete unit resolution-style calculus for regular Horn formulas. We then design efficient Horn satisfiability checking procedures with linear-time complexity when the truth value set is finite (if infinite, the complexity is almost linear). Second, we demonstrate that the satisfiability problem in arbitrary signed 2-CNF formulas is NP-complete. We then show that this problem is polynomially solvable when the formulas are regular or monosigned. We describe efficient satisfiability checking procedures for these subclasses of signed 2-CNF formulas. These procedures reach a quadratic-time complexity in the worst case.

Regarding deduction problems we focus on a wide and important family of infinitely-valued logics, including Łukasiewicz logic. First, we define a propositional logic programming language for this family of logics and prove that a modus ponens-style inference rule is complete for determining the maximum degree of logical consequence of a goal in a given infinitely-valued logic program.

Second, we define a multiple-valued negation as failure rule and a cut operator. Finally, we define suitable data structures for representing logic programs and describe an interpreter with a linear-time complexity in the worst case.

Acknowledgements

This book contains my PhD thesis, submitted to the *Departament d'Informàtica, Universitat Autònoma de Barcelona*, in November 1996.

This thesis would never exist without the help and encouragement of many people. My sincerest thanks go to:

– Gonzalo Escalada-Imaz, my advisor, for his support and advice during my work on this thesis;

– Jaume Agustí, Francesc Esteva, Lluís Godo, Reiner Hähnle and Ventura Verdú for accepting to join my examining committee and for valuable comments and suggestions;

– the *Institut d'Investigació en Intel·ligència Artificial* (IIIA-CSIC) whose members contributed with ideas, discussions and in many other ways to this thesis.

– the *Escola Universitària Politècnica* and the *Departament d'Informàtica* at the *Universitat de Lleida* for providing a very special work environment;

– the CICYT, *La Paeria* and the *Universitat de Lleida* for partially supporting this research;

– my colleagues and friends Teresa Alsinet, Cèsar Fernàndez, Jordi Levy and Lluís Vila for the good times we have shared;

– and especially my family and Teresa for their love and support over the years.

Chapter 1

Introduction

1.1 Context and objectives

The first time a computer was used to prove theorems was in 1954: Martin Davis did a straightforward implementation of the Presburger's decision procedure for additive number theory, but led to no publication. Three years later, Newell, Shaw and Simon published the first paper that reported results of a computer program that could prove some theorems in classical propositional logic, the Logic Theory Machine (Newell et al., 1957). Since then, a lot of work has been done in the field of Automated Theorem Proving, and its results have not only been applied to prove mathematical theorems, but also to different areas of Computer Science such as Program Synthesis and Verification, Data Bases, Logic Programming and Artificial Intelligence.[1]

The idea of automating the reasoning task leads to a new vision of Logic in which the notions of validity and logical consequence gain a computational dimension. Martin-Löf (1987) says that Computational Logic must take into consideration four aspects:

- The language.
- The semantics.
- The proof theory.
- An automated deduction system.

The design of most proof procedures, which are the core of any automated deduction system, offers serious obstacles because of the computational complexity of the problems they have to solve: on the one hand, it is not always possible

[1]See (Davis, 1983a) for a description of the implementation of the Presburger's decision procedure. See (Davis, 1983b; Loveland, 1984) for an overview of the history of automated theorem proving. In (Siekmann and Wrightson, 1983a; Siekmann and Wrightson, 1983b) are collected the main contributions to this field until the early 1970's.

to devise a procedure for determining the validity of a formula in a finite number of steps. In classical first-order logic, this problem was first posed by Hilbert and Ackermann (1928), who called it the *Entscheidungsproblem*. Church (1936) and Turing (1936), independently, gave a negative answer to the *Entscheidungsproblem*. On the other hand, even though such a decision procedure exists in classical propositional logic, a wide variety of problems are computationally *intractable*, in the sense that the known algorithms for solving them require an exponential number of steps in the length of the input for some problem instances. Cook (1971) proved that the propositional satisfiability (SAT) problem in classical logic is NP-complete.

Several approaches to solving computationally intractable problems have been suggested in Computational Logic. For the purposes of this thesis, we emphasize the following ones:

- To design relatively fast proof procedures based on logic calculi such as resolution and semantic tableaux. These calculi are generally more appropriate for automated theorem proving than Hilbert-style systems or natural deduction.

- To identify subclasses of computationally intractable problems which are polynomially solvable. This leads to develop logic calculi, or refinements of existing ones, which are complete for such subclasses. This is the case, for instance, of the classical SAT problem in Horn formulas (Horn SAT) and 2-CNF formulas (2-SAT).[2]

Nevertheless, there is a gap between a high-level description of a proof procedure and a competitive implementation from a computational point of view. Bridging this gap involves to make non-trivial decisions concerning, among other things, the definition of clever heuristics for solving the non-determinism inherent in logic calculi (i.e. what inference rule is applied and over what formulas) and suitable data structures for representing instances of the problem. A clever choice of heuristics and a suitable design of data structures are decisive factors to get implementations that run as fast as possible on a wide range of computationally difficult instances.

The primary concern of this thesis is the design of *fast* decision procedures for solving problems in multiple-valued propositional logics. In particular, our research will focus on the design of satisfiability checking procedures for multiple-valued formulas and interpreters for a particular family of infinitely-valued logic programs. Before going into more technical details about the solutions we propose, we will motivate and set our objectives.

For tackling the SAT problem in multiple-valued logics we will develop satisfiability checking procedures based on resolution-style logic calculi and, therefore, we will need to have some kind of normal form. Baaz and Fermüller

[2]A Horn formula is a set of clauses such that each clause contains at most one positive literal. A 2-CNF formula is a set of clauses such that each clause contains exactly two literals.

(1995) suggested a two-level approach to resolution-based theorem proving in finitely-valued logics which is suitable for putting our work into context:[3]

- The first level consists in translating arbitrary formulas of a particular finitely-valued propositional logic into clause form. Given the definition of a finitely-valued logic by means of the truth tables of its connectives, it can be devised a *logic-dependant* translation calculus that derives a clause form from a formula of the source logic. This translation can be done in such a way that the language of the clause forms is independent from the particular language of the source logic except for the number of truth values; i.e. clause forms are *logic-independent.*
- The second level consists in the application of a logic-independent resolution calculus to the clause forms obtained in the first level.

These logic-independent clause forms are known as *signed formulas in conjunctive normal form* (signed CNF formulas). They are the basis for understanding most of the results of this thesis, because the satisfiability checking procedures we develop in subsequent chapters take as input a signed CNF formula. Next, we succinctly introduce the logic of signed CNF formulas.[4]

A signed CNF formula is a set of signed clauses and a signed clause is a set of signed literals. A signed literal is an expression of the form $S:p$, where S, called the sign of the literal, is a subset of the truth value set and p is a propositional atom. An interpretation is mapping that assigns to every propositional atom an element of the truth value set. An interpretation satisfies a signed literal $S:p$ iff it assigns to p a truth value that appears in S. An interpretation satisfies a signed clause iff it satisfies at least one of its literals. A signed CNF formula is satisfiable iff there exists an interpretation that satisfies all its clauses; otherwise, it is unsatisfiable.

One significant subclass of signed CNF formulas that we consider in this thesis is the class of *regular CNF formulas.* Roughly speaking, regular CNF formulas are those signed CNF formulas such that the signs of the literals are either of the form $\{j \in N \mid j \geq i\}$ or of the form $\{j \in N \mid j \leq i\}$, where N is the truth value set, $\leq$ is a total order on N and $i, j \in N$. Another subclass that we consider is the class of *monosigned CNF formulas.* They are those signed CNF formulas whose signs are singletons instead of arbitrary subsets of the truth value set.

It is interesting to note that the main semantic difference between classical and signed CNF formulas arise in the definition of interpretation, where the number of truth values is greater than two. Then, the concept of satisfiability for signed CNF formulas is the same as the classical one, but with respect to

[3]The signs considered by Baaz and Fermüller (1995) are singletons, whereas we will deal with signs that are arbitrary subsets of the truth value set (sets as signs). Nevertheless, our interest in the present section is their description of the two-level approach to resolution-based theorem proving in finitely-valued logics. This approach remains valid for signs that are subsets of the truth value set.

[4]See Chapter 2 for further details.

multiple-valued interpretations. It is in this sense that we will say that the semantics of signed CNF formulas is classical above the literal level. Signed literals, for instance, do not have the notions of positive and negative polarity, and more than two literals are required for detecting unsatisfiability.[5] Therefore, classical proof methods may be generalized naturally to work with signed CNF formulas provided that special care be taken at the literal level.

Murray and Rosenthal (1994) view the logic of signed CNF formulas as a meta-logic for representing and solving certain problems in multiple-valued logics. These problems are often represented more easily or cannot be always expressed in the source logic. One of such problems is the transformation of a formula into a normal form that contains only connectives of the source language.

Two remarkable contributions to the first level of Baaz and Fermüller's approach to resolution-based theorem proving are the following ones:

- The system MUltlog (Baaz et al., 1993; Baaz et al., 1996) which computes an optimized logic-dependant translation calculus for deriving signed CNF formulas from the definition of the operators and quantifiers of any finitely-valued first-order logic. It produces a sequent calculus and a natural deduction system as well.[6]
- Hähnle's method for deriving a satisfiability equivalent[7] signed CNF formula from an arbitrary formula of any finitely-valued logic (Hähnle, 1994b). He has described a structure-preserving method for generating short conjunctive normal forms (i.e. short signed CNF formulas) whose length is linear in the length of the input formula and polynomial in the cardinality of the truth value set.[8]

Concerning the second level, there exist several resolution calculi for signed CNF formulas and their subclasses. We review them in Section 3.2.

Accordingly, one way to solve the SAT problem in finitely-valued logics is as follows: first, we derive a signed CNF formula which is satisfiability equivalent to the input finitely-valued formula by using, for instance, the methods described in (Baaz and Fermüller, 1995; Hähnle, 1994b); and second, we check the satisfiability of the signed CNF formula so obtained with a satisfiability checking procedure for signed CNF formulas.

However, to the best of our knowledge, little attention has been paid so far to the design of resolution-based satisfiability checking procedures for signed CNF formulas equipped with clever heuristics for exploring the proof search space and

[5]Remember that, in classical propositional logic, many satisfiability testing algorithms detect unsatisfiability when a propositional atom p is derived with positive (p) and negative ($\neg p$) polarity.

[6]A funny feature of the system MUltlog is that the calculi derived are presented as a paper written in $\LaTeX$.

[7]A formula A is satisfiability equivalent to a formula B if it holds that A is satisfiable iff B is satisfiable. This property is weaker than logical equivalence. For proofs by refutation, satisfiability equivalence suffices.

[8]In this thesis, we always refer to worst-case time complexity.

suitable data structures for representing formulas. In view of this, our intention is to look into this question and find some new results. To this end, our first objective is to extend the Davis-Putnam (DP) procedure (Davis and Putnam, 1960) to the setting of signed CNF formulas. Our motivation to choose this procedure is twofold: on the one hand, it is one of the fastest and most widely used methods for solving the SAT problem in classical logic; on the other hand, the semantics of signed CNF formulas is classical above the literal level.

Our second objective is to refine the results obtained and to design a DP-style procedure for regular CNF formulas. We think that it is justified to consider separately the SAT problem in regular CNF formulas since simpler logic calculi exist for this subclass of signed CNF formulas. Moreover, classical proof procedures can be easily extended to the regular setting.

Taking into account that there exist polynomial-time algorithms for testing the satisfiability of classical Horn and 2-CNF formulas, another objective of this thesis is to investigate what happens with these particular subclasses in the framework of signed CNF formulas. The Horn SAT problem will be only considered in regular CNF formulas because signed and monosigned literals do not have the concept of polarity (cf. Section 2.4). The 2-SAT problem will be investigated in signed, regular and monosigned CNF formulas.

Unfortunately, there is no general method as the one explained above for solving the SAT problem in arbitrary infinitely-valued logics; i.e. it has not been developed so far a method for obtaining a logic-independent clause form from any infinitely-valued formula. Hähnle (1994a) defined a reduction of the SAT problem in certain infinitely-valued logics to the mixed integer programming problem. Among such infinitely-valued logics are Łukasiewicz's and Post's logics. Automated theorem proving in Łukasiewicz logic was also studied by Mundici (1991,1994) and in Post logic was considered by Zabel (1993).

In this thesis we will address the SAT problem in infinitely-valued logics focusing on regular CNF formulas. This is possible because regular CNF formulas admit a finite representation of infinite signs, and the search space that has to be explored for solving the SAT problem is finite.

In the infinitely-valued case, we will also focus on a family of infinitely-valued propositional logics, including Łukasiewicz logic, which are chiefly characterized by the following points: the interpretation function of the conjunction connective has to fulfill the properties of triangular norm, the implication connective is defined by residuation with respect to the conjunction connective, and the interpretation function of the negation connective is the unit complement function (cf. Section 2.6).

This family of infinitely-valued logics was well studied, among others, by Pavelka (1979), Valverde and Trillas (1985) and Godo (1990). They showed that, in this case, the triangular norm that gives meaning to the conjunction connective is a modus ponens generating function for the implication. Starting out from the fact that the modus ponens inference rule that they defined is sound, one of our objectives is to define a complete calculus for deducing infinitely-valued facts from a set of infinitely-valued facts and rules. Notice that if we use this infinitely-valued modus ponens inference rule, we do not need any transformation into normal forms.

Another objective regarding the infinitely-valued facts and rules mentioned above is to design an efficient proof procedure that could act as a propositional interpreter for infinitely-valued logic programs, as well as defining a negation as failure rule and a cut operator.

The design of competitive algorithms is common to all the previous objectives. Consequently, throughout this thesis we will pay special attention to the definition of appropriate calculi for mechanical proofs, clever heuristics for exploring the proof search space and suitable data structures for representing instances of problems.

It is worth mentioning that the interest in multiple-valued theorem proving goes beyond the theoretical level. In recent years, interesting applications have been suggested in several fields such as functional analysis (Mundici, 1986), logic programming (Fitting, 1988), adaptive error-correcting codes (Mundici, 1990), formal hardware verification (Hähnle and Kernig, 1993c) and deductive databases (Subrahmanian, 1994).[9] This fact justifies to some extent the practical interest of our work because all these applications demand efficient proof procedures.

Finally, we would like to point out that there exist some automated theorem provers for multiple-valued logics. The most representative among them is $_{3}T^{A}P$ (Beckert et al., 1996), developed at the University of Karlsruhe. It is a generic tableau-based theorem prover for finitely-valued first-order logics with sorts and equality; it was implemented in Prolog.

1.2 Contributions

Once we have stated and motivated the main objectives of this thesis, we summarize its contributions in the subfield of automated theorem proving in multiple-valued logics.

The SAT problem in signed CNF formulas:

> We have designed a DP-style decision procedure for signed CNF formulas. To this end, we have extended the one-literal rule to the signed setting and defined an original branching rule based on a new concept that we have coined as maximal truth value set. We have also designed a satisfiability checking procedure for regular CNF formulas that can be seen as a refinement of the former. We have equipped it with suitable data structures for representing instances of the regular SAT problem, and proposed several alternative branching rules and deletion strategies.[10] In both cases we have defined heuristics for selecting the next literal to which the branching rule is applied and proved the completeness of the proof procedures described.

[9]See (Hähnle, 1993a; Chapter 7) for an exposition of other applications.

[10]By deletion strategies we mean ways of eliminating irrelevant and redundant clauses.

The Horn SAT problem in regular CNF formulas:

First, we have developed a refutation complete unit resolution calculus for a subclass of infinitely-valued regular Horn formulas that we have called Horn mv-formulas. Second, we have designed an almost linear-time satisfiability checking procedure for Horn mv-formulas. Third, we have extended our results to arbitrary regular Horn formulas. The satisfiability checking procedures we have obtained reach a linear-time complexity when the truth value set is finite (if infinite, the complexity is almost linear). This low complexity is due to the definition of suitable data structures.

The 2-SAT problem in signed CNF formulas:

We have first demonstrated that the 2-SAT problem in signed CNF formulas is NP-complete, contrary to what happens in classical propositional logic.[11] Then, we have shown that this problem is polynomially solvable when the formulas are regular or monosigned. We have described quadratic-time satisfiability checking procedures for these subclasses of signed 2-CNF formulas, proved their completeness, and defined suitable data structures and different branching rules.

Interpretation of infinitely-valued logic programs:

Regarding deduction problems we have focused on a wide and important family of infinitely-valued logics, including Łukasiewicz logic. First, we have defined a propositional logic programming language for this family of logics and proved that a modus ponens-style inference rule is complete for determining the maximum degree of logical consequence of a goal in a given infinitely-valued logic program. Second, we have defined a multiple-valued negation as failure rule and a cut operator. Third, after analyzing carefully several factors that contribute to improve the efficiency of algorithms, we have designed a linear-time propositional interpreter.

1.3 Overview of the thesis

The thesis is organized in six chapters, whose contents are summarized below.

Chapter 2: Multiple-valued logics

After defining the syntax and semantics of multiple-valued propositional logics, we introduce the logic of signed formulas and motivate their use in the field of automated theorem proving. For our purposes, the clause forms of signed formulas, so-called signed CNF formulas, become a suitable formalism for representing and solving multiple-valued SAT problems. Then, we define regular CNF formulas and monosigned CNF formulas, which are two significant subclasses of signed CNF formulas. Finally, we define a

[11]There exist linear-time algorithms for solving the classical 2-SAT problem (cf. Section 4.3).

family of fuzzy logics (which in turn are infinitely-valued logics) as a subclass of signed formulas. Such infinitely-valued logics give meaning to the logic programs of the propositional interpreter we develop later on.

Chapter 3: The SAT problem in signed formulas

In this chapter we investigate the SAT problem in both signed and regular CNF formulas. After reviewing the existing resolution calculi for signed, regular and monosigned CNF formulas, we introduce the DP procedure, which is our starting point for facing the SAT problem in multiple-valued logics. Then, we present the concept of maximal truth value set, which we use to define an optimized multiple-valued branching rule. Next, we extend the DP procedure to signed CNF formulas, prove its completeness and propose heuristics for choosing the propositional atom selected for doing branching. Finally, we describe a satisfiability checking procedure for regular CNF formulas that can be seen as a refinement of the former. We prove its completeness and analyze carefully those aspects that are able to improve the computational performance of satisfiability testing algorithms. In particular, we offer different options for doing branching, suitable data structures for representing formulas, heuristics for choosing the next literal to which the branching rule is applied and several deletion strategies.

Chapter 4: Polynomially solvable SAT problems

The SAT problem in classical logic is polynomially solvable for Horn and 2-CNF formulas. In this chapter we address the Horn SAT and 2-SAT problems in signed CNF formulas. Our objective is to investigate under which circumstances such signed SAT problems are polynomially solvable. Concerning the Horn SAT problem, we first describe an almost linear-time decision procedure for a subclass of infinitely-valued regular Horn formulas. Then, we extend this result to arbitrary regular Horn formulas and give a satisfiability checking procedure that reaches a linear-time complexity when the truth value set is finite. The complexity is almost linear in the infinitely-valued case. Concerning the 2-SAT problem, we start by proving that it is NP-complete in signed CNF formulas, contrary to what happens in classical logic. The 2-SAT problem, however, turns out to be polynomially solvable in regular and monosigned CNF formulas. We describe quadratic-time decision procedures for solving the 2-SAT problem in such subclasses of signed CNF formulas. To this end, we define appropriate calculi, design decision procedures equipped with suitable data structures, prove their completeness and analyze their time complexity.

Partial and preliminary results about the Horn SAT problem in multiple-valued formulas appeared in (Escalada-Imaz and Manyà, 1993b; Escalada-Imaz and Manyà, 1994a; Escalada-Imaz and Manyà, 1994c).

A communication about the 2-SAT problem in signed CNF formulas was presented in a workshop (Escalada-Imaz and Manyà, 1996b).

Chapter 5 An interpreter for multiple-valued logic programs

In recent years, new logic programming languages have been developed for reasoning and representing knowledge in situations where there is vague, incomplete or imprecise information. Some of such languages rely on multiple-valued logics and demand fast deduction procedures. In this chapter, we describe an efficient interpreter of logic programs that can deal with a wide family of infinitely-valued propositional logics. To this end, we first define a logic programming language, a complete calculus that contains a modus ponens-style inference rule, suitable data structures for representing logic programs and strategies for pruning the proof search space. Then, we define a negation as failure rule and a cut operator adapted to our multiple-valued setting. Finally, we describe an interpreter whose worst-case time complexity is linear in the total number of occurrences of propositional atoms in the input logic program.

Part of the material included in this chapter appeared in (Escalada-Imaz and Manyà, 1993a; Escalada-Imaz and Manyà, 1994b; Escalada-Imaz and Manyà, 1995; Escalada-Imaz and Manyà, 1996a).

Chapter 6: Conclusions

We conclude the thesis with a brief summary of its main contributions, and we point out some open problems and future research perspectives.

Finally, the reader can find the references appearing in the text and an index that is intended to be a pointer to the main concepts used throughout this thesis.

Chapter 2

Multiple-Valued Logics

Abstract:

After defining the syntax and semantics of multiple-valued propositional logics, we introduce the logic of signed formulas and motivate their use in the field of automated theorem proving. For our purposes, the clause forms of signed formulas, so-called signed CNF formulas, become a suitable formalism for representing and solving multiple-valued SAT problems. Then, we define regular CNF formulas and monosigned CNF formulas, which are two significant subclasses of signed CNF formulas. Finally, we define a family of fuzzy logics (which in turn are infinitely-valued logics) as a subclass of signed formulas. Such infinitely-valued logics give meaning to the logic programs of the propositional interpreter we develop later on.

2.1 Introduction

This chapter contains some basic definitions and terminology that will be used in this thesis. We start by introducing a formal framework for defining the syntax and semantics of multiple-valued propositional logics[1] and showing how some well-known logics can be represented in it. We then define multiple-valued signed formulas and motivate their use in the subfield of automated theorem proving in multiple-valued logics. We also present the syntax and semantics of signed CNF formulas, which are the clause forms that the satisfiability checking procedures we describe in subsequent chapters take as input. Next, we introduce two significant subclass of signed CNF formulas, so-called regular CNF formulas and monosigned CNF formulas, and discuss the computational advantages derived from developing specific proof procedures for these kinds of formulas.

It turns out that the infinitely-valued facts and rules of the logic programming language of the interpreter we will design in Chapter 5 can be defined as a subclass of signed formulas. The family of infinitely-valued logics that give meaning to such facts and rules are chiefly characterized by the interpretation

[1]Multiple-valued logics are also called many-valued and multi-valued logics in the literature.

functions of the conjunction and implication connectives, which have to fulfill the properties of triangular norm and residuated implication, respectively. All these topics are discussed at the end of the chapter.

The main difference between classical and multiple-valued logic is that formulas are not determinedly either true or false. They take a truth value from a truth value set with cardinality greater than two. The motivation to deal with a wider range of truth values, as well as the election of the subset of designated truth values and the interpretation functions of the logical connectives, has arisen from the semantic analysis of logical statements based on real-world problems or philosophical considerations. For instance, Kleene (1938) defined a 3-valued logic whose truth value set, $\{f, i, t\}$, can be interpreted as follows: given an evaluation of a predicate p by a computer, f means that p was evaluated to false, i means that the computer was unable to evaluate p and t means that p was evaluated to true. The reader interested in a deeper discussion about these questions can consult some standard references; e.g. (Rosser and Turquette, 1952; Zinovev, 1963; Ackermann, 1967; Rescher, 1969; Urquhart, 1986; Gottwald, 1989; Bolc and Borowik, 1992).

This chapter is organized as follows. In Section 2.2 we introduce a formal framework for defining the syntax and semantics of arbitrary multiple-valued propositional logics; in Section 2.3 we present the syntax and semantics of signed formulas and signed CNF formulas; in Section 2.4 we define regular CNF formulas; in Section 2.5 we define monosigned CNF formulas; and in Section 2.6 we show that the infinitely-valued facts and rules of the interpreter we describe in Chapter 5 can be understood as a subclass of signed formulas. We motivate the use of all these kinds of formulas as well. Part of the material included in this chapter has been borrowed from (Ryan and Sadler, 1992; Hähnle, 1993a).

2.2 Syntax and semantics of multiple-valued logics

A logic has a syntax (form) and a semantics (meaning). The formulas of a logic are defined by means of a formal language, and meaning is attached to them stating precisely the meaning of the logical connectives by means of a matrix and the meaning of the atomic formulas by means of an interpretation. The concept of logical consequence is introduced to capture the notion of valid argument.

2.2.1 Syntax

Definition 2.1 propositional language *A propositional language L is an ordered pair (P, O) consisting of a denumerable set $P = \{p_0, p_1, \ldots\}$ of propositional variables, called* **atomic formulas or propositional atoms,** *together with a finite or denumerable set $O = \{o_0, o_1, \ldots\}$ of operators, called* **logical connectives.** *Each operator o_i comes with an arity.*

Definition 2.2 formulas of a propositional language *The formulas of a propositional language L are inductively defined as follows:*

- *every propositional atom is a formula;*
- *if $\phi_1, \ldots, \phi_m$ are formulas and o_i is a logical connective with arity m, then $o_i(\phi_1, \ldots, \phi_m)$ is is a formula.*

Throughout this thesis we make the usual conventions on elimination of parentheses, as well as on precedence and infix notation for well-known logical connectives.

2.2.2 Semantics

Definition 2.3 matrix *A matrix $\mathcal{M}$ for a propositional language $L = (P, O)$ is a triple $\mathcal{M} = (N, D, F)$ where:*

- *N is a set with at least two elements, the* **truth value set***;*
- *D is a non-empty proper subset of N, the set of* **designated truth values***;*
- *$F = \{f_{o_1}, f_{o_2}, \ldots, f_{o_n}\}$ is a set of functions, one corresponding to each operator in $O = \{o_1, o_2, \ldots, o_n\}$ such that $f_{o_i} : N^{n_{o_i}} \to N$, where n_{o_i} is the arity of the operator o_i. We say that f_{o_i} interprets o_i and F is the* **set of interpretation functions.**

We will usually take as truth value set either a finite set of equidistant rational numbers of the form $\{0, \frac{1}{n-1}, \ldots, \frac{n-2}{n-1}, 1\}$ or the unit interval $[0, 1]$ on the rational numbers. Nevertheless, our results concerning arbitrary signed CNF formulas and monosigned CNF formulas are valid for any other finite truth value set, and our results concerning regular CNF formulas are valid for any totally ordered truth value set.

Definition 2.4 n-valued propositional logic *Let L be a propositional language, let $\mathcal{M} = (N, D, F)$ be a matrix for L and let n be the cardinality of N. The pair $\mathcal{L} = (L, \mathcal{M})$ is called n-valued propositional logic with designated truth values D.*

Definition 2.5 interpretation *Let $\mathcal{L} = (L, \mathcal{M})$ be an n-valued propositional logic and let N be the truth value set of the matrix $\mathcal{M}$. An interpretation I for the propositional language $L = (P, O)$ is a mapping that assigns to every propositional atom of P an element of the truth value set N. An interpretation I is extended to any arbitrary formula ϕ of L as follows:*

- *if $\phi = p$ is a propositional atom, then $I(\phi) = I(p)$;*
- *if $\phi = o_i(\phi_1, \ldots, \phi_m)$, then $I(o_i(\phi_1, \ldots, \phi_m)) = f_{o_i}(I(\phi_1), \ldots, I(\phi_m))$.*

Definition 2.6 satisfiability *Let $\mathcal{L} = (L, \mathcal{M})$ be an n-valued propositional logic with designated truth values D. A formula ϕ of L is satisfiable iff there exists an interpretation I for L such that $I(\phi) \in D$. We say then that I satisfies ϕ or that I is a model of ϕ, and we write $I \models \phi$. A set of propositional formulas Γ is satisfiable iff there exists an interpretation that satisfies all the formulas of Γ. A formula (a set of formulas) that is not satisfiable is unsatisfiable.*

Definition 2.7 tautology *A formula ϕ is a tautology, denoted by $\models \phi$, iff every interpretation satisfies ϕ.*

Definition 2.8 logical consequence *A formula ϕ is a logical consequence of a set of formulas Γ, denoted by $\Gamma \models \phi$, iff every interpretation that satisfies Γ also satisfies ϕ.*

2.2.3 Examples of multiple-valued logics

Here we define the multiple-valued propositional logics of Łukasiewicz and Post using the formal framework explained in the preceding subsections.

a) 3-valued propositional Łukasiewicz logic (Lukasiewicz, 1920)

1. Operators: $\wedge, \vee, \rightarrow, \neg$.
   The arity of $\neg$ is 1 and for the remaining operators is 2.
2. Truth value set: $N = \{t(rue), p(ossible), f(alse)\}$
3. Designated truth values: $D = \{t\}$
4. Interpretation functions (truth tables):

$\phi_1 \wedge \phi_2$	t	p	f
t	t	p	f
p	p	p	f
f	f	f	f

$\phi_1 \vee \phi_2$	t	p	f
t	t	t	t
p	t	p	p
f	t	p	f

$\phi_1 \rightarrow \phi_2$	t	p	f
t	t	p	f
p	t	t	p
f	t	t	t

$\neg\phi$	
t	f
p	p
f	t

b) n-valued propositional Łukasiewicz logics (Lukasiewicz and Tarski, 1930)

1. Operators: $\wedge, \vee, \rightarrow, \neg$.
   The arity of $\neg$ is 1 and for the remaining operators is 2.
2. Truth value set: $N = \{0, \frac{1}{n-1}, \ldots, \frac{n-2}{n-1}, 1\}$ for finitely-valued logics and the real unit interval $[0, 1]$ for infinitely-valued logics.
3. Designated truth values: $D = \{1\}$

4. Interpretation functions:

$$\begin{aligned} f_{\rightarrow}(x,y) &= min(1, 1-x+y) \\ f_{\neg}(x) &= 1-x \\ f_{\vee}(x,y) &= \max(x,y) \\ f_{\wedge}(x,y) &= \min(x,y) \end{aligned}$$

c) **n-valued propositional Post Logics** (Post, 1921)

1. Operators: $\wedge, \vee, \sigma$.
   The arity of σ is 1 and for the remaining operators is 2.
2. Truth value set: $N = \{0, \frac{1}{n-1}, \ldots, \frac{n-2}{n-1}, 1\}$
3. Designated truth values: $D = \{1\}$
4. Interpretation functions:

$$\begin{aligned} f_{\sigma}(x) &= \min(1, x + \tfrac{1}{n-1}) \\ f_{\vee}(x,y) &= \max(x,y) \\ f_{\wedge}(x,y) &= \min(x,y) \end{aligned}$$

2.3 Multiple-valued signed formulas

In classical logic, the idea of associating a truth value (sign) with a formula can be found, for example, in the setting of semantic tableaux (Smullyan, 1995). In multiple-valued tableau systems, this idea was first introduced by Suchón (1974) in a tableau system for Lukasiewicz logics and since then appeared in other works (Surma, 1984; Carnielli, 1987; Zabel, 1993; Baaz and Fermüller, 1995). But in all of them signs are only single truth values. Hähnle (1990) used subsets of the truth value set as signs (sets as signs) and the formulas extended with this kind of signs were called *signed formulas.* He showed that using truth value sets as signs enables one to optimize tableau systems: on the one hand, they avoid to build more than one tableau in order to prove that a multiple-valued formula is valid, as it happened for instance in (Surma, 1984; Carnielli, 1987); on the other hand, expansion rules become more concise. A similar concept of signed formulas was independently introduced by Murray and Rosenthal (1991a).

There are several reasons for introducing signed formulas in this thesis. We summarize them below:

- Signed formulas are the expressions used to translate any finitely-valued formula into a satisfiability equivalent signed CNF formula (cf. Section 1.1).
- Signed CNF formulas are a subclass of signed formulas.
- Signed CNF formulas are the logic-independent clause forms of the second level of Baaz and Fermüller's approach to resolution-based theorem proving in finitely-valued logics (cf. Section 1.1).

- Signed CNF formulas are the input to the satisfiability checking procedures we describe in Chapter 3 and Chapter 4.
- The infinitely-valued facts and rules of the logic programming language of the interpreter we design in Chapter 5 are defined as a subclass of signed formulas.

It is worth mentioning that it was shown that annotated logics (Lu et al., 1993) and fuzzy operator logics (Lu et al., 1994) are special cases of the logic of signed formulas.

Definition 2.9 signed formula *Let S be a subset of the truth value set N and let ϕ be a propositional formula. An expression of the form $S : \phi$ is a signed formula and S is its sign.*

The semantic notions of matrix and interpretation remain the same and the notion of satisfiability is defined as follows:

Definition 2.10 satisfiability *Let $\mathcal{L} = (L, \mathcal{M})$ be an n-valued propositional logic. A signed formula $S : \phi$ is satisfiable iff there exists an interpretation I for L such that $I(\phi) \in S$. Otherwise, $S : \phi$ is unsatisfiable.*

The concept of satisfiability for signed formulas is more general than that of Definition 2.6. Here, instead of considering a formula to be satisfiable when an interpretation assigns an element of the set of designated truth values to the formula, it is allowed to consider a *local* set of designated truth values for each propositional formula. Observe that the multiple-valued formulas defined in Section 2.2 are a particular case of signed formulas (we only have to put to each formula the set of designated truth values as sign).

As said before, one of our concerns in this thesis is the design of satisfiability checking procedures whose inputs are signed CNF formulas. Next, we introduce the basic concepts of these signed clause forms.

Definition 2.11 signed literal *Let S be a subset of the truth value set N and let p be a propositional atom. An expression of the form $S : p$ is a signed literal and S is its sign.*

Definition 2.12 complement of a signed literal *Let $L = S : p$ be a signed literal and let N be the truth value set. $\overline{L} = (N \setminus S) : p$ denotes the complement of L.*

Definition 2.13 subsumption of signed literals *A signed literal $S : p$ subsumes a signed literal $S' : p'$, denoted by $S : p \subseteq S' : p'$, iff $p = p'$ and $S \subseteq S'$.*

Definition 2.14 signed clause *A signed clause is a finite set of signed literals. A signed clause that contains exactly one literal is a signed unit clause. A signed clause that contains exactly two literals is a signed binary clause. The signed empty clause is denoted by $\Box$.*

Definition 2.15 signed CNF formula *A signed CNF formula is a finite set of signed clauses. A signed CNF formula whose clauses are binary is a signed 2-CNF formula.*

Definition 2.16 length *The length of a signed clause C, denoted by $|C|$, is the total number of occurrences of signed literals in C. The length of a signed CNF formula Γ, denoted by $|\Gamma|$, is the sum of the lengths of its signed clauses. The number of distinct signed literals occurring in Γ is denoted by $lit(\Gamma)$.*

Definition 2.17 satisfiability *An interpretation I satisfies a signed literal $S{:}p$ iff $I(p) \in S$. An interpretation satisfies a signed clause iff it satisfies at least one of its signed literals. A signed CNF formula Γ is satisfiable iff there exists at least one interpretation that satisfies all the signed clauses in Γ. A signed CNF formula that is not satisfiable is unsatisfiable. The signed empty clause is always unsatisfiable and the signed empty CNF formula is always satisfiable.*

The clauses of a signed CNF formula are implicitly conjunctively connected. The literals in a signed clause are implicitly disjunctively connected. Sometimes we will use $S_1{:}p_1 \vee \cdots \vee S_k{:}p_k$ to represent a signed clause $\{S_1{:}p_1, \ldots, S_k{:}p_k\}$.

Hähnle (1994b) described a structure-preserving method for translating formulas from any finitely-valued logic into a satisfiability equivalent signed CNF formula. This method has the advantage that it produces signed CNF formulas whose length is linear in the length of the input formula and polynomial in the cardinality of the truth value set.[2] Consequently, the SAT problem in finitely-valued logics is polynomially reducible to the SAT problem in signed CNF formulas. Murray and Rosenthal (1994) also addressed the derivation of signed normal forms but their method is exponential in the length of the input formula.

On the one hand, signed CNF formulas are sufficient to solve any SAT problem in finitely-valued logics (Hähnle, 1994b). On the other hand, there exist sound and complete calculi for signed clauses (cf. Section 3.2). Therefore, satisfiability checking procedures for signed CNF formulas are sufficient to solve the SAT problem in any finitely-valued logic.

2.4 Regular CNF formulas

In the present section we concentrate on a particular subclass of signed CNF formulas known as regular CNF formulas. Roughly speaking, we will now assume that there is a total order over the truth value set and signs have a specific form. As we will see later on, these restrictions turn out to be beneficial for developing simple logic calculi and designing proof procedures which bear a close resemblance to the classical ones.

Definition 2.18 regular sign *Let $\boxed{\geq i}$ denote the set $\{j \in N \mid j \geq i\}$ and let $\boxed{\leq i}$ denote the set $\{j \in N \mid j \leq i\}$, where N is the truth value set, $\leq$ is a total*

[2]In the rest of this chapter, the complexities given are in the worst case.

order on N and $i \in N$. If a sign S is equal to either $\boxed{\geq i}$ or $\boxed{\leq i}$, for some i, then it is a regular sign and i is its value.

Definition 2.19 regular literal *Let S be a regular sign and let p be a propositional atom. An expression of the form $S:p$ is a regular literal. If S is of the form $\boxed{\geq i}$ ($\boxed{\leq i}$), then we say that $S:p$ has positive (negative) polarity.*

The notions of positive and negative polarity are useful for extending classical concepts such as Horn clauses (i.e. clauses having at most one positive literal) or positive unit resolution to our multiple-valued setting (cf. Section 4.2.2). In the following, when we say regular positive (negative) literal we mean a regular literal with positive (negative) polarity.

Definition 2.20 regular clause *A regular clause is a finite set of regular literals. A regular clause that contains exactly one literal is a regular unit clause. A regular clause that contains exactly two literals is a regular binary clause. A regular clause that contains at most one regular positive literal is a regular Horn clause. A regular clause that contains only regular literals with positive (negative) polarity is a regular positive (negative) clause.*

Definition 2.21 regular CNF formula *A regular CNF formula is a finite set of regular clauses. A regular CNF formula whose clauses are binary is a regular 2-CNF formula. A regular CNF formula whose clauses are Horn is a regular Horn formula.*

Since regular CNF formulas are a subclass of signed CNF formulas, the definitions of the semantics of signed CNF formulas remain valid for regular CNF formulas.

When considering arbitrary signed CNF formulas we assume that the truth value set is finite. It is interesting to note that regular CNF formulas admit a finite representation of infinite signs. So, they are attractive for dealing with an infinite truth value set. Moreover, the search space that has to be explored for solving the SAT problem in regular CNF formulas is finite even in the infinitely-valued case. In this case, we should first scan all regular signs occurring in the formula and restrict N to the finite truth value set formed by all the different values of signs occurring in the formula. We will assume this approach of dealing with infinitely-valued regular CNF formulas in the rest of the thesis.

Murray and Rosenthal (1994) and Hähnle (1996) proved that for every signed CNF formula there exists a regular CNF formula such that both are logically equivalent, but their transformation can generate regular CNF formulas whose length is exponential in the length of the signed CNF formula.

Regular formulas originated in the setting of multiple-valued semantic tableaux. Hähnle (1991) showed that there exists a wide and important class of logics, so-called *regular logics*, whose minimal expansion tableau rules can be expressed by regular signs and have a uniform notation as the one proposed by Smullyan (1995) for classical semantic tableaux. The fact of using regular signs enables one to easily extend classical proof methods to the multiple-valued setting. Moreover, there exist algorithms for obtaining short normal forms for regular logics that are linear in both the length of the input formula and the cardinality of the truth value set (Hähnle, 1994b).

The concept of regular formulas also appears in (Lu et al., 1993), where their relationship with annotated logics is established. Nevertheless, these regular formulas are a little bit different; they assume that the truth value set is a complete lattice whereas we assume that it is a totally ordered set.

2.5 Monosigned CNF formulas

Definition 2.22 monosigned CNF formula *A monosigned literal is a signed literal whose sign is a singleton. A monosigned clause is a finite set of monosigned literals. A monosigned clause that contains exactly one literal is a monosigned unit clause. A monosigned clause that contains exactly two literals is a monosigned binary clause. A monosigned CNF formula is a finite set of monosigned clauses. A monosigned CNF formula whose clauses are binary is a monosigned 2-CNF formula.*

Monosigned CNF formulas were studied, among others, by Baaz and Fermüller (1995). They developed a structure-preserving translation method into monosigned CNF formulas – which is valid for any finitely-valued logic – and a resolution calculus, as well as some refinements.

2.6 Fuzzy logics as regular logics

Fuzzy logic is a term generally used to refer to a wide range of logics which are suitable for representing and manipulating vague, incomplete or imprecise information. In particular, several infinitely-valued logics are considered as fuzzy logics. In this thesis we will deal with a family of fuzzy logics, which in turn are infinitely-valued logics, that can be defined as a subclass of signed formulas. Our interest in such logics is motivated by the following points:

- These logics have been carefully investigated by Pavelka (1979), Valverde and Trillas (1985) and Godo (1990), among others. In particular, a sound modus ponens inference rule has been defined for this family of logics.
- Knowledge-based systems have used these logics as a powerful knowledge representation language in expert systems for diagnosis applications (Sierra, 1989; Puyol-Gruart, 1994).
- The design of efficient deduction algorithms for these logics has not been investigated so far despite their significance in real-world applications.

In Chapter 5 we will design a linear-time proof procedure for deducing, using a modus ponens-style inference rule, infinitely-valued facts from a set of infinitely-valued facts and rules. These results could be incorporated, for example, into the inference engine of knowledge-based systems such as Milord II (Puyol-Gruart, 1994). In addition, we will define a negation as failure rule and a cut operator. Such a proof procedure can be seen as an interpreter of multiple-valued logic programs.

In this section we introduce the family of infinitely-valued logics mentioned above and integrate them into the framework of signed formulas. In Chapter 5 we present the sub-logic of facts and rules. We define them as the subclass of signed formulas that fulfill the following conditions:

- the set of operators is $O = \{\neg, \wedge, \rightarrow\}$; $\wedge$ and $\rightarrow$ are binary and $\neg$ is unary;
- each formula is a signed formula whose sign is regular and has positive polarity;
- the truth value set of the matrix is the unit interval on the real numbers, denoted by $[0, 1]$;
- the choice of the set of interpretations functions of the matrix (i.e. $\{f_\neg, f_\wedge, f_\rightarrow\}$) is not arbitrary, they must satisfy some properties we explain below.

  In order to be standard with the literature, we will denote $f_\neg$, $f_\wedge$ and $f_\rightarrow$ by N, T and I_T, respectively. In fact, we offer different options for defining T and I_T. Therefore, we have a different logic for every election of T and I_T. It is in this sense that we say that we will deal with a family of infinitely-valued logics.

  N, T and I_T must satisfy the following properties:

  negation: the negation operator N has to be a unary operation that fulfills the following properties:

  N1: if $a < b$ then $N(a) > N(b)$, $\forall a, b \in [0, 1]$

  N2: $N^2 = Id$

  The only unary function that fulfills the previous properties in the finitely-valued case is $N(a) = 1 - a$. In our infinitely-valued case we take $N(a) = 1 - a$ as the negation operator.

  conjunction: the conjunction operator T has to be a triangular norm (T-norm), i.e. it has to be a binary continuous operation such that for all $a, b, c \in [0, 1]$ satisfies:

  T1: $T(a, b) = T(b, a)$

  T2: $T(a, T(b, c)) = T(T(a, b), c)$

  T3: $T(0, a) = 0$

  T4: $T(1, a) = a$

  T5: if $a \leq b$ then $T(a, c) \leq T(b, c)$ for all c

implication: the implication operator I_T is defined by residuation with respect to T, i.e.

$$I_T(a,b) = Sup\{c \in [0,1] \,|\, T(a,c) \leq b\},$$

and satisfies the following properties:

I1: $I_T(a,b) = 1$ if, and only if, $a \leq b$

I2: $I_T(1,a) = a$

I3: $I_T(a, I_T(b,c)) = I_T(b, I_T(a,c))$

I4: If $a \leq b$, then $I_T(a,c) \geq I_T(b,c)$ and $I_T(c,a) \leq I_T(c,b)$

I5: $I_T(T(a,b),c) = I_T(a, I_T(b,c))$

This kind of implications are known as residuated implications (R-implications) in the fuzzy literature. Observe that once we have defined a T-norm we can obtain its corresponding R-implication.

Example 2.1 *The following signed formulas are examples of our infinitely-valued formulas:*

$$\boxed{\geq 0.3} : p_1 \wedge \neg p_3 \rightarrow p_2$$
$$\boxed{\geq 0.5} : \neg p_1 \wedge p_2 \wedge \neg p_3 \rightarrow p_1$$

In the literature, given a signed formula of the form $\boxed{\geq i} : \phi$, *it is usual to represent it like* $(\phi; i)$. *We will follow this convention in the rest of the thesis. Hence, the previous formulas will be represented as follows:*

$$(p_1 \wedge \neg p_3 \rightarrow p_2; 0.3)$$
$$(\neg p_1 \wedge p_2 \wedge \neg p_3 \rightarrow p_1; 0.5)$$

Example 2.2 *Some examples of T-norms and their corresponding R-implications are shown below.*

$$T(a,b) = a \cdot b \qquad I_T(a,b) = \begin{cases} 1 & \textit{if } a \leq b \\ b/a & \textit{otherwise} \end{cases}$$

$$T(a,b) = max(0, a+b-1) \qquad I_T(a,b) = min(1, 1-a+b)$$

$$T(a,b) = min(a,b) \qquad I_T(a,b) = \begin{cases} 1 & \textit{if } a \leq b \\ b & \textit{otherwise} \end{cases}$$

Chapter 3

The Satisfiability Problem in Signed CNF Formulas

Abstract: In this chapter we investigate the SAT problem in both signed and regular CNF formulas. After reviewing the existing resolution calculi for signed, regular and monosigned CNF formulas, we introduce the Davis-Putnam (DP) procedure, which is our starting point for facing the SAT problem in multiple-valued logics. Then, we present the concept of maximal truth value set, which we use to define an optimized multiple-valued branching rule. Next, we extend the DP procedure to signed CNF formulas, prove its completeness and propose heuristics for choosing the propositional atom selected for doing branching. Finally, we describe a satisfiability checking procedure for regular CNF formulas that can be seen as a refinement of the former. We prove its completeness and analyze carefully those aspects that are able to improve the computational performance of satisfiability testing algorithms. In particular, we offer different options for doing branching, suitable data structures for representing formulas, heuristics for choosing the next literal to which the branching rule is applied and several deletion strategies.

3.1 Introduction

The propositional satisfiability (SAT) problem in classical logic (i.e. the problem of determining whether a two-valued formula is satisfiable) was the first problem shown to be NP-complete (Cook, 1971). In the last years, it has attracted the interest of several research communities (Artificial Intelligence, Complexity Theory, Logic, Operations Research, ...), and a wealth of papers reporting algorithms for solving that problem have been published. Because of the state of maturity reached, even competitions for evaluating and comparing satisfiability testing algorithms have been organized (Buro and Büning, 1993; Harche et al., 1994). For solving the SAT problem in classical logic, algorithms based on the Davis-Putnam (DP) procedure (Davis and Putnam, 1960) are among the most competitive ones. Actually, efficient implementations are variants of a modified version proposed two years later by Davis, Logemann and Loveland (Davis et al., 1962; Loveland, 1978). The main achievement of the latter is that it introduces the branching rule.[1]

In this chapter we investigate the SAT problem in both signed and regular CNF formulas taking the DP procedure as a starting point. We think that this is a promising approach since the semantics of signed CNF formulas is classical above the literal level. Therefore, classical proof procedures may be naturally generalized to work with signed CNF formulas provided that special care be taken at the literal level.

After reviewing the existing resolution calculi for signed, regular and monosigned CNF formulas, we present the DP procedure. Then, we define the concept of maximal truth value set. This new concept is aimed at formalizing the minimum number of assignments of truth values to a given propositional atom that a satisfiability checking procedure for signed CNF formulas needs to test. We use it to define an optimized multiple-valued branching rule. Next, we describe a DP-style procedure for signed CNF formulas and prove its completeness. We also propose heuristics for choosing the propositional atom involved in the branching rule, since it is a decisive factor to get small size proof trees.

The previous proof procedure turns out to be an appropriate framework for defining refinements for special classes of signed CNF formulas such as regular CNF formulas. In view of this, we give a DP-style procedure for regular CNF formulas, prove its completeness and analyze carefully those aspects that are able to improve its computational performance. In particular, we offer different options for doing branching, suitable data structures for representing formulas, heuristics for choosing the next literal to which the branching rule is applied and some deletion strategies.

This chapter is organized as follows. In Section 3.2 we review several resolution calculi for signed, regular and monosigned CNF formulas; in Section 3.3 we present the DP procedure; in Section 3.4 we define the concept of maximal truth value set and an optimized signed branching rule; in Section 3.5 and Section 3.6 we describe in detail satisfiability checking procedures for signed and regular CNF formulas, respectively.

3.2 Resolution-based proof methods

We have already discussed, in the previous chapters, the importance of signed CNF formulas for solving the SAT problem in multiple-valued logics. Our aim in this section is to review the existing resolution systems for signed, regular and monosigned CNF formulas.

We focus our attention on propositional logics because, as Hähnle (1996) points out, all deviations from classical logic occur on the ground level while

[1]The branching rule is also known as splitting rule. In the following when we say Davis-Putnam (DP) procedure we mean the version that incorporates the branching rule. In the literature, this version is usually known as the Davis-Putnam-Loveland procedure.

lifting is done exactly as in the classical case and presents no new insights. The concepts of proof and refutation for the calculi described below are defined in the usual way. Thus, we will only present the inference rules that lead to *refutation complete calculi.*[2]

The reader interested in an homogeneous exposition and the main references of earlier approaches to multiple-valued resolution not based on signed clauses can consult (Hähnle, 1993a; Chapter 8). Among the resolution systems examined there are those of Morgan (1976), Orłowska (1978) and Di Zenzo (1988) for Post logics, those of Lee and Chang (1971,1972) for fuzzy logic, that of Schmitt (1986) for a specific three-valued logic for use within a natural language dialogue system and those of Stachniak and O'Hearn (1990b,1990a,1988) for a wide class of multiple-valued logics.

We divide our exposition into three subsections. Section 3.2.1, Section 3.2.2 and Section 3.2.3 are devoted to signed, regular and monosigned resolution, respectively.

3.2.1 Signed resolution

In this section we present three refutation complete resolution calculi for signed CNF formulas that appeared in the literature. The first one was defined by Murray and Rosenthal (1993) and is one of the first resolution systems published.[3] It is formed by the following inference rules:[4]

signed parallel resolution

$$\frac{S_1\!:\!p \vee D_1 \quad \cdots \quad S_m\!:\!p \vee D_m}{(S_1 \cap \cdots \cap S_m)\!:\!p \vee D_1 \vee \cdots \vee D_m}$$

merging

$$\frac{S_1\!:\!p \vee S_2\!:\!p \vee D}{(S_1 \cup S_2)\!:\!p \vee D}$$

simplification

$$\frac{\emptyset\!:\!p \vee D}{D}$$

[2]A calculus is refutation complete if there exists a refutation for any unsatisfiable formula.

[3]The signed CNF formulas we are dealing with correspond to the atomic signed formulas of Murray and Rosenthal. In fact, they work with a more general notion of signed literal where the sign part is followed by any propositional formula; it is not necessarily followed by a propositional atom. Nevertheless, the calculus presented here is only complete for their atomic signed formulas.

[4]These inference rules also appeared in (Murray and Rosenthal, 1991b), but the truth value set was assumed to obey some restrictions.

Independently, Hähnle (1993b) defined a similar resolution calculus:

signed parallel resolution

$$\frac{S_1\!:\!p \vee D_1 \quad \cdots \quad S_m\!:\!p \vee D_m}{D_1 \vee \cdots \vee D_m} \quad \text{if } S_1 \cap \cdots \cap S_m = \emptyset$$

merging

$$\frac{S_1\!:\!p \vee \cdots \vee S_m\!:\!p \vee D}{(S_1 \cup \cdots \cup S_m)\!:\!p \vee D}$$

A sequential refutation complete resolution calculus for signed CNF formulas was defined in (Hähnle, 1994b):

signed binary resolution

$$\frac{S_1\!:\!p \vee D_1 \qquad S_2\!:\!p \vee D_2}{(S_1 \cap S_2)\!:\!p \vee D_1 \vee D_2}$$

simplification

$$\frac{\emptyset\!:\!p \vee D}{D}$$

Completeness of the previous calculi was proved using a straightforward generalization of classical semantic trees to the multiple-valued context. Hähnle (1996) pointed out that for obtaining refutation completeness the merging rule is not necessary. This can be proved using the excess literal technique of Anderson and Bledsoe (1970).

The previous calculi are a first step to resolution-based automated theorem proving for signed CNF formulas. Nevertheless, we believe that further research on resolution refinements for signed clauses has to be done before achieving competitive resolution-based automatic theorem provers.

3.2.2 Regular resolution

The fact of developing separately inference rules for regular clauses appears to be very beneficial for obtaining efficient proof procedures for regular CNF formulas. On the one hand, all regular literals have either positive or negative polarity; on the other hand, any unsatisfiable set of regular literals has an unsatisfiable subset of cardinality two, contrary to what happens with signed literals. These points enable one to define refinements of signed resolution calculi which are complete for regular CNF formulas and bear a close resemblance to classical resolution versions.

Hähnle (1994b) proved that the calculus formed by the following regular versions of the resolution and merging rules is refutation complete for regular CNF formulas:

regular resolution

$$\frac{\boxed{\geq i_1} : p \vee D_1 \quad \cdots \quad \boxed{\geq i_m} : p \vee D_m \quad \boxed{\leq j} : p \vee D}{D_1 \vee \cdots \vee D_m \vee D} \qquad \text{if } \max_{1 \leq k \leq m} i_k > j$$

merging

$$\frac{\boxed{\geq i_1} : p \vee \cdots \vee \boxed{\geq i_l} : p \vee \boxed{\leq j_1} : p \vee \cdots \vee \boxed{\leq j_m} : p \vee D}{\boxed{\geq \min_{1 \leq k \leq l} i_k} : p \vee \boxed{\leq \max_{1 \leq k' \leq m} j_{k'}} : p \vee D}$$

As already noted, the merging rule is not necessary for obtaining completeness. Later, Hähnle (1996) defined the following complete regular version of negative hyperresolution:

regular negative hyperresolution

$$\frac{\boxed{\leq i_1} : p_1 \vee D_1 \quad \cdots \quad \boxed{\leq i_m} : p_m \vee D_m \qquad \boxed{\geq j_1} : p_1 \vee \cdots \vee \boxed{\geq j_m} : p_m \vee E}{D_1 \vee \cdots \vee D_m \vee E}$$

provided $m \geq 1$, $i_l < j_l$ for all $1 \leq l \leq m$, $D_1, \ldots, D_m, E$ are negative.

Moreover, he claimed that the following regular unit resolution rule is complete for regular Horn formulas.

regular unit resolution

$$\frac{\begin{array}{l} S{:}p \\ S'{:}p \vee C \end{array}}{C} \qquad \text{if } S \cap S' = \emptyset$$

In Section 4.2 we prove that, for obtaining completeness, it is enough to resolve on regular positive unit clauses. Hence, the rule below constitutes a refutation complete calculus for regular Horn formulas.

regular positive unit resolution

$$\frac{\begin{array}{l} \boxed{\geq j} : p \\ \boxed{\leq i} : p \vee C \end{array}}{C} \qquad \text{if } i < j$$

The concept of regular resolution also appears in (Lu et al., 1993), although there the truth value set is a complete lattice whereas we assume that the truth value set is a totally ordered set. Therefore, their notion of regular sign is a little bit more general than ours. It is worthwhile to point out that with their more general regular signs the resulting resolution calculus is equivalent to the *p-resolution* calculus for the paraconsistent logics defined in (Kifer and Lozinskij, 1989; Lu et al., 1991).

3.2.3 Monosigned resolution

Baaz and Fermüller (1995) studied monosigned resolution and proved that only one inference rule is needed to determine the satisfiability of monosigned CNF formulas. This rule is very close to classical binary resolution:

$$\frac{\{v_1\}:p \vee D_1 \qquad \{v_2\}:p \vee D_2}{D_1 \vee D_2} \quad \text{if } v_1 \neq v_2;\ v_1, v_2 \in N$$

They also defined a first-order rule which is a straightforward generalization of the classical one. Such a rule needs to be enhanced with factoring in order to obtain completeness. Moreover, Baaz and Fermüller extended A-ordering resolution and hyperresolution to the monosigned setting.

3.3 The Davis-Putnam procedure

The Davis-Putnam (DP) procedure (Davis et al., 1962) is one of the fastest and most widely used methods for solving the SAT problem in classical logic. As our intention is to extend this proof procedure to the framework of signed CNF formulas, we will first discuss the details of the two-valued version. It is based on two inference rules known as unit resolution and branching rule:

unit resolution

$$\frac{\begin{array}{c} L \\ \overline{L} \vee C \end{array}}{C}$$

branching rule

$$\frac{}{p \mid \neg p}$$

The algorithm automates the application of the previous rules as follows: first, it applies unit resolution until either the empty clause is deduced or a saturation state is reached. Actually, given a CNF formula Γ that contains a unit clause $\{L\}$, the algorithm removes all clauses that contain the literal L from Γ, and then removes all occurrences of the literal $\overline{L}$ from the remaining clauses (this step is known as application of the *one-literal rule*). This way, after several applications of the one-literal rule, the algorithm produces a simpler CNF formula Γ', without unit clauses, from the CNF formula Γ. Second, it selects a literal L' occurring in Γ' and reduces the problem of testing the satisfiability of Γ' to two new subproblems: to determine whether $\Gamma' \cup \{L'\}$ is satisfiable or $\Gamma' \cup \{\overline{L'}\}$ is satisfiable (this step is known as application of the *branching rule*). As these subproblems by construction contain a unit clause, the one-literal rule can be applied again. These steps are repeated and the algorithm stops when the empty formula is derived in some subproblem (then, the input formula is satisfiable) or the empty clause is derived in all the subproblems (then, the input

```
function dp-sat ($\Gamma$: set of clause) : boolean
var L: literal
begin
  $\Gamma$ := unit-resolve($\Gamma$);
  if $\Gamma = \emptyset$ then return(true);
  if $\Box \in \Gamma$ then return(false);
  L := pick-literal($\Gamma$);
  if dp-sat ($\Gamma \cup \{L\}$) then
    return(true)
  else
    return(dp-sat($\Gamma \cup \{\overline{L}\}$))
  endif
end
```

```
function unit-resolve ($\Gamma$: set of clause) : set of clause
var L: literal
var C: clause
begin
  while $\exists\{L\} \in \Gamma$ and $\Box \notin \Gamma$ do
    $\Gamma := \{C \mid L \notin C \in \Gamma\}$;
    $\Gamma := \{C \setminus \{\overline{L}\} \mid C \in \Gamma\}$
  endwhile;
  return($\Gamma$)
end
```

Figure 3.1: The Davis-Putnam procedure

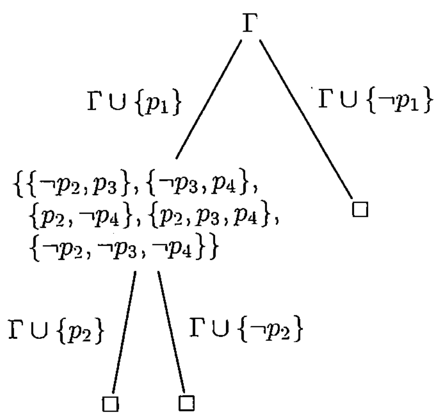

Γ
$\Gamma \cup \{p_1\}$
$\Gamma \cup \{\neg p_1\}$
$\{\{\neg p_2, p_3\}, \{\neg p_3, p_4\}, \{p_2, \neg p_4\}, \{p_2, p_3, p_4\}, \{\neg p_2, \neg p_3, \neg p_4\}\}$
$\Box$
$\Gamma \cup \{p_2\}$
$\Gamma \cup \{\neg p_2\}$
$\Box$
$\Box$

Figure 3.2: A DP proof tree for the formula Γ from Example 3.1

formula is unsatisfiable). This algorithm is a decision procedure for solving the SAT problem in classical propositional CNF formulas.

The pseudo-code of the DP procedure is shown in Figure 3.1. This procedure can also be viewed as the construction of a proof tree: nodes are labelled by formulas and the root node contains the input formula Γ. Before expanding a node, function unit-resolve applies repeatedly the one-literal rule to the formula of the node. Then, a new formula Γ' is obtained and pick-literal selects a literal L of Γ'. Two branches, labelled $\Gamma' \cup \{L\}$ and $\Gamma' \cup \{\overline{L}\}$, are created and their leaf nodes contain the formulas obtained after applying function unit-resolve to $\Gamma' \cup \{L\}$ and $\Gamma' \cup \{\overline{L}\}$, respectively. The proof tree is created using the depth-first strategy and the procedure backtracks when a node contains the empty clause. The search terminates when either the empty formula is deduced (in this case, the input formula Γ is satisfiable) or until the procedure backtracks to the root node (in this case, all the leaf nodes contain the empty clause and the input formula Γ is unsatisfiable).

Example 3.1 *Let Γ be the following CNF formula:*

$$\begin{aligned}\Gamma &= \{\{\neg p_1, \neg p_2, p_3\}, \{\neg p_1, \neg p_3, p_4\}, \{p_1, p_5\}, \{p_2, \neg p_4\}, \\ &\quad \{p_2, p_3, p_4\}, \{\neg p_2, \neg p_3, \neg p_4\}, \{p_1, \neg p_5\}\}\end{aligned}$$

Figure 3.2 shows the proof tree created by the DP procedure when the input CNF formula is Γ. The root node contains Γ and the remaining nodes contain the CNF formula obtained after applying unit-resolve to the formula selected for doing branching.

A concrete implementation of the DP procedure, in order to be efficient, should take into account the following points:

- The definition of suitable data structures for representing formulas is an important factor to get time efficient operations in satisfiability testing algorithms. In particular, a right choice of data structures enables one to implement function unit-resolve with a linear-time complexity in the worst case.

- A clever choice of the heuristic embodied in pick-literal, for selecting the next literal to which the branching rule is applied, can reduce the size of a DP proof tree considerably. From now on, we will refer to this kind of heuristics as *branching heuristics.*

- Deletion strategies (i.e. strategies for eliminating irrelevant and redundant clauses) simplify formulas. It is important to test if the inclusion of any of them in the current implementation leads to faster algorithms.

As already noted, the branching heuristic embodied in pick-literal is an important factor to reduce the size of a DP proof tree considerably. Two branching heuristics that have achieved a good performance in satisfiability competitions (Buro and Büning, 1993; Harche et al., 1994) are the following ones:

Two-sided Jeroslow-Wang rule (Jeroslow and Wang, 1990): Given a propositional CNF formula Γ, for each literal L that occurs in Γ the following function is defined:

$$J(L) = \sum_{L \in C \in \Gamma} 2^{-|C|},$$

where $|C|$ is the length of clause C. Then, pick-literal selects a literal that maximizes $J(L) + J(\overline{L})$.[5]

Hooker and Vinay (1995) offer a careful analysis of the family of branching heuristics known as Jeroslow-Wang rules. By constructing a Markov chain model and by experiments, they conclude that the success of the two-sided Jeroslow-Wang rule is explained because it selects a literal that allows to eliminate more literals and clauses during the execution of function unit-resolve in each one of the branches, so that it is more likely to resolve the SAT subproblems obtained without a great deal of branching.

Lexicographic heuristic (Böhm and Speckenmeyer, 1996): Given a propositional CNF formula Γ, for each atom p that occurs in Γ the following vector is defined:

$$(H_1(p), H_2(p), \ldots, H_k(p)),$$

where

$$H_i(p) = \alpha\, max(h_i(p), h_i(\neg p)) + \beta\, min(h_i(p), h_i(\neg p))$$

and $h_i(p)$ (resp. $h_i(\neg p)$) is the number of clauses of length i that contain the literal p (resp. $\neg p$). Pick-literal selects an atom with maximal vector under the lexicographic order. It was shown experimentally that, in order

[5]When we write $L \in C \in \Gamma$, we mean $L \in C$ and $C \in \Gamma$.

to get subproblems of about the same size, good values for the parameters α and β are 1 and 2, respectively.

The idea behind the lexicographic heuristic of Böhm-Speckenmeyer is to select a literal occurring as often as possible in the shortest clauses of the formula, so that after a few steps the shortest clauses become often unit clauses. This way, the formula collapses fast during the execution of function unit-resolve.

3.4 Maximal truth value set

Before extending the DP procedure to the framework of signed CNF formulas, we first analyze how to extend the branching rule in such a way that the number of branches for each node of the proof tree is as small as possible.

In the classical case, the branching rule states that a CNF formula Γ is satisfiable iff $\Gamma \cup \{p\}$ is satisfiable or $\Gamma \cup \{\neg p\}$ is satisfiable, where p is a propositional atom occurring in Γ. In the multiple-valued case, a straightforward generalization of the two-valued branching rule for a signed CNF formula Γ' would be as follows: Γ' is satisfiable iff $\Gamma' \cup \{\{0\}:p\}$ is satisfiable or $\Gamma' \cup \{\{\frac{1}{n-1}\}:p\}$ is satisfiable or $\ldots$ or $\Gamma' \cup \{\{\frac{n-2}{n-1}\}:p\}$ is satisfiable or $\Gamma' \cup \{\{1\}:p\}$ is satisfiable, where p is a propositional atom occurring in Γ' and $N = \{0, \frac{1}{n-1}, \ldots, \frac{n-2}{n-1}, 1\}$ is the truth value set.[6] Therefore, if we want to expand a node of a proof tree, we should develop $|N|$ branches, where $|N|$ denotes the cardinality of N. However, we show below that, in many cases, it is enough that the branching rule considers only a subset of N.

In the following we assume, without loss of generality, that a signed CNF formula has the property that a propositional atom occurs at most once in each clause. If this is not the case and we have a signed clause of the form $S_1:p\vee\cdots\vee S_k:p\vee D$, then we replace it with the signed clause $(S_1\cup\cdots\cup S_k):p\vee D$; i. e. we apply the merging rule defined on page 26. Observe that both clauses are logically equivalent.

From now on, we denote by N_Γ^p the set formed by the truth values of N that appear in a signed CNF formula Γ in literals of the form $S:p$.

Given a signed CNF formula Γ and a propositional atom p occurring in Γ, it can happen that some truth values $y_1, \ldots, y_k$ appear exactly in the same signs of literals of the form $S:p$. In this case, it suffices that the branching rule considers only one of the truth values from $y_1, \ldots, y_k$; if a model of Γ assigns to p one truth value from $y_1, \ldots, y_k$, the interpretation that assigns to p any other truth value from $y_1, \ldots, y_k$ and is identical for the remaining propositional atoms satisfies Γ as well. To identify the truth values that appear exactly in the same signs of literals of the form $S:p$ we define the following equivalence relation over N_Γ^p:

$$x \approx_p y \iff \forall S:p \text{ such that } S:p \in C \text{ and } C \in \Gamma,\ x \in S \text{ iff } y \in S \qquad (3.1)$$

[6] As already noted in the previous chapter, our results for signed CNF formulas are also valid for any other finite truth value set.

Next, we define the following partial order relation over the equivalence classes of $N^p_\Gamma / \approx_p$:

$$\overline{x} \preceq_p \overline{y} \Longleftrightarrow \forall S{:}p \text{ such that } S{:}p \in C \text{ and } C \in \Gamma, \text{ if } x \in S \text{ then } y \in S \qquad (3.2)$$

Observe that if $\overline{x} \preceq_p \overline{y}$, the elements in class $\overline{y}$ appear, among other signs, in all those signs in which the elements in class $\overline{x}$ appear. Therefore, the branching rule can ignore the elements of class $\overline{x}$; if a model of Γ assigns to p a truth value from class $\overline{x}$, the interpretation that assigns to p any truth value from class $\overline{y}$ and is identical for the remaining propositional atoms satisfies Γ as well. Then, we have that it suffices that the branching rule considers only the maximal elements of the relation $\preceq_p$. Even more, it is enough that it considers only one representative element of each one of the maximal elements of $\preceq_p$ because maximal elements are equivalence classes. This is the idea behind the concept of maximal truth value set.

Definition 3.1 maximal set of signs *Let Γ be a signed CNF formula, let p be a propositional atom occurring in Γ, let $\approx_p$ be the equivalence relation over N^p_Γ defined in (3.1), and let $\preceq_p$ be the partial order relation over $N^p_\Gamma / \approx_p$ defined in (3.2). The set formed by the maximal elements of $\preceq_p$ is the maximal set of signs of p in Γ.*

Definition 3.2 maximal truth value set *Let Γ be a signed CNF formula, let p be a propositional atom occurring in Γ, and let $\{\overline{x_1}, \dots, \overline{x_m}\}$ be the maximal set of signs of p in Γ. A set of the form $\{x_1, \dots, x_m\}$, where $x_1 \in \overline{x_1}, \dots, x_m \in \overline{x_m}$, is a maximal truth value set of p in Γ.*

In the following, when we say a maximal truth value we mean an element of a maximal truth value set.

Example 3.2 *Let $N = \{0, \frac{1}{6}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{5}{6}, 1\}$ and let*

$$\begin{array}{l} \{\frac{1}{6}, 1\}{:}p \lor D_1 \\ \{0, \frac{1}{3}, \frac{5}{6}\}{:}p \lor D_2 \\ \{0, \frac{1}{3}, \frac{1}{2}\}{:}p \lor D_3 \\ \{0, \frac{1}{6}, \frac{1}{3}, 1\}{:}p \lor D_4 \\ \{\frac{1}{6}, \frac{2}{3}, 1\}{:}p \lor D_5 \end{array}$$

be the clauses in a signed CNF formula Γ in which literals of the form $S{:}p$ occur. Then, the equivalence classes of $N^p_\Gamma / \approx_p$ are

$$\overline{\tfrac{1}{6}} = \{\tfrac{1}{6}, 1\} \quad \overline{\tfrac{1}{3}} = \{0, \tfrac{1}{3}\} \quad \overline{\tfrac{1}{2}} = \{\tfrac{1}{2}\} \quad \overline{\tfrac{2}{3}} = \{\tfrac{2}{3}\} \quad \overline{\tfrac{5}{6}} = \{\tfrac{5}{6}\}$$

The maximal elements of the partial order relation $\preceq_p$ are $\overline{\frac{1}{3}}$ and $\overline{\frac{1}{6}}$. A maximal truth value set of p in Γ is $\{\frac{1}{3}, \frac{1}{6}\}$.

Once we have introduced the concept of maximal truth value set, we define a branching rule that considers a maximal truth value set instead of the whole truth value set.

Proposition 3.1 signed branching rule *Let Γ be a signed CNF formula, let p be a propositional atom occurring in Γ, and let $\{x_1, \ldots, x_m\}$ be a maximal truth value set of p in Γ. Then, Γ is satisfiable iff there is an $i \in \{1, \ldots, m\}$ such that $\Gamma \cup \{\{x_i\}:p\}$ is satisfiable.*

Proof: Suppose that Γ is satisfiable. Let I be a model of Γ and let $I(p) = \alpha$. We distinguish two cases:

1. There are no occurrences of literals of the form $S:p$ in Γ such that $\alpha \in S$. Then, the value that I assigns to p has no effect on the satisfiability of Γ. We define an interpretation I' which assigns to p a maximal truth value of p in Γ, say x_i, and is identical for the remaining atoms. Then, we have that I' satisfies both Γ and $\Gamma \cup \{\{x_i\}:p\}$.

2. Γ contains literals $S_1 : p, \ldots, S_k : p$ such that $\alpha \in S_1, \ldots, \alpha \in S_k$. Thus, $\alpha \in S_1 \cap \cdots \cap S_k$. If α is a maximal truth value, then we are done. Otherwise, we have that there exists a maximal truth value x_i such that $x_i \in S_1 \cap \cdots \cap S_k$. The interpretation I' which assigns to p the truth value x_i and is identical for the remaining atoms satisfies both Γ and $\Gamma \cup \{\{x_i\}:p\}$.

Suppose that there exists an $i \in \{1, \ldots, m\}$ such that $\Gamma \cup \{\{x_i\}:p\}$ is satisfiable. Since Γ is a subset of $\Gamma \cup \{\{x_i\}:p\}$, it follows that Γ is satisfiable. ■

The branching rule of Proposition 3.1 can reduce the size of a proof tree considerably. For instance, when this branching rule is applied to the signed CNF formula from Example 3.2, the number of branches is reduced from seven to two.

Given a signed CNF formula Γ and a propositional atom p occurring in Γ, a maximal truth value set of p in Γ can be computed in polynomial time. The cost of calculating the equivalence classes of $N_\Gamma^p / \approx_p$ and calculating the maximal elements of $\preceq_p$ with a brute force search algorithm is in $\mathcal{O}(|N|^2\,|\Gamma|)$, where $|N|$ denotes the cardinality of the truth value set and $|\Gamma|$ denotes the length of Γ. Nevertheless, we believe that this complexity could be improved with an appropriate definition of data structures.

We claim that the usefulness of the concept of maximal truth value set is not confined to define optimized branching rules, it could be very beneficial in the multiple-valued setting when we work with enumerative proof procedures (i.e. proof procedures whose principle is to assign truth values to propositional atoms) because it allows to reduce the number of truth values that must be considered for a given propositional atom.

3.5 A satisfiability checking procedure for signed CNF formulas

In this section we describe a DP-style procedure for signed CNF formulas. As explained in Section 3.3, the DP procedure relies on the branching rule and the one-literal rule. Since we have already defined an optimized signed branching rule based on the concept of maximal truth value set, our next step is to define a one-literal rule for signed CNF formulas.

Proposition 3.2 signed one-literal rule *Let Γ be a signed CNF formula and let $\{S_1 : p\}, \ldots, \{S_l : p\}$ be signed unit clauses of Γ such that $S_1 \cap \cdots \cap S_l \neq \emptyset$. Let Γ' be obtained from Γ by first removing all clauses that contain a literal of the form $S : p$ such that $S_1 \cap \cdots \cap S_l \subseteq S$, and second by removing all occurrences of literals of the form $S' : p$ such that $S_1 \cap \cdots \cap S_l \cap S' = \emptyset$ from the remaining clauses. Let Γ'' be obtained from Γ' by replacing all occurrences of literals of the form $S'' : p$ with $(S_1 \cap \cdots \cap S_l \cap S'') : p$. Then, Γ is satisfiable iff Γ'' is satisfiable.*

Proof: Suppose that Γ is satisfiable. Since $\{S_1 : p\} \in \Gamma, \ldots, \{S_l : p\} \in \Gamma$, there exists a model I of Γ such that $I(p) \in S_1 \cap \cdots \cap S_l$. On the one hand, I is a model of any subset of Γ. On the other hand, if we remove the literals of the form $S' : p$ such that $S_1 \cap \cdots \cap S_l \cap S' = \emptyset$, I is also a model of Γ' because $I(p) \notin S'$. Since $I(p) \in S_1 \cap \cdots \cap S_l$, we can now replace in Γ' all occurrences of literals of the form $S'' : p$ with $(S_1 \cap \cdots \cap S_l \cap S'') : p$, and we have that I satisfies Γ''.

Suppose that Γ'' is satisfiable. By construction of Γ'', we have that all occurrences of literals of the form $S_i : p$ in Γ'' verify that $S_i \subseteq S_1 \cap \cdots \cap S_l$. Thus, there exists a model I of Γ'' such that $I(p) \in S_1 \cap \cdots \cap S_l$. For obtaining Γ' from Γ'' we add some truth values to the signs of the atom p, but anyway I continues being a model of Γ'. For obtaining Γ from Γ' we add new clauses to Γ' and literals to clauses of Γ'. The addition of literals has no effect on the satisfiability and the new clauses contain a literal $S : p$ such that $S_1 \cap \cdots \cap S_l \subseteq S$. Therefore, $I(p) \in S_1 \cap \cdots \cap S_l \subseteq S$ and I satisfies Γ. ■

The pseudo-code of a satisfiability checking procedure based on the notions of signed branching rule (cf. Proposition 3.1) and signed one-literal rule (cf. Proposition 3.2) is shown in Figure 3.3. The main function is signed-sat, which takes as input a signed CNF formula Γ with the property that a propositional atom occurs at most once in each clause. It returns true if Γ is satisfiable, and it returns false if Γ is unsatisfiable. Function signed-unit-resolve applies repeatedly the signed one-literal rule, function pick-atom selects, by applying an heuristic, the next propositional atom to which the signed branching rule is applied (this question is discussed in Section 3.5.1) and function maximal-truth-value-set(Γ, p) computes a maximal truth value set of the propositional atom p in the signed CNF formula Γ.

```
function signed-sat ($\Gamma$: set of clause) : boolean
var p: atom
var x: truth-value
var S: set of truth-value
begin
  $\Gamma$ := signed-unit-resolve($\Gamma$);
  if $\Gamma = \emptyset$ then return(true);
  if $\Box \in \Gamma$ then return(false);
  p := pick-atom($\Gamma$);
  S := maximal-truth-value-set($\Gamma, p$);
  if $\exists x \in$ S such that signed-sat($\Gamma \cup \{\{x\}:p\}$) then
      return(true)
    else
      return(false)
  endif
end
```

```
function signed-unit-resolve ($\Gamma$: set of clause) : set of clause
var p: atom
var C, D: clause
var $S, S_1, \ldots, S_l$: set of truth-value
begin
  while $\exists\{S_1:p\} \in \Gamma, \ldots, \{S_l:p\} \in \Gamma$ and $\Box \notin \Gamma$ do
    $\Gamma := \{C \mid \nexists S:p \in C \in \Gamma$ such that $S_1 \cap \cdots \cap S_l \subseteq S\}$;
    $\Gamma := \{C \setminus \{S:p \mid S:p \in C$ and $S_1 \cap \cdots \cap S_l \cap S = \emptyset\} \mid C \in \Gamma\}$;
    for each $C = S:p \vee D \in \Gamma$ do
      $C := (S_1 \cap \cdots \cap S_l \cap S):p \vee D$
    endfor
  endwhile;
  return($\Gamma$)
end
```

Figure 3.3: A DP-style procedure for signed CNF formulas

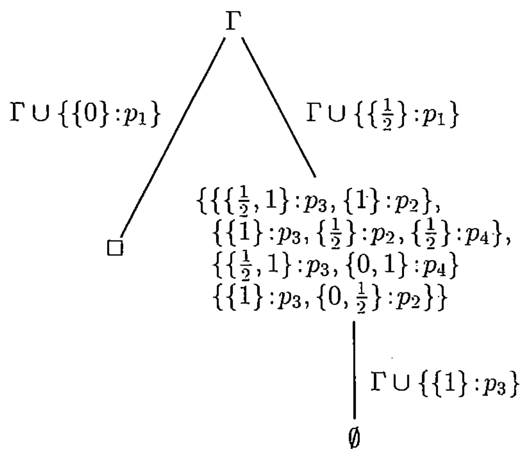

Γ
$\Gamma \cup \{\{0\}:p_1\}$
$\Gamma \cup \{\{\frac{1}{2}\}:p_1\}$
$\Box$
$\{\{\{\frac{1}{2},1\}:p_3,\{1\}:p_2\},$
$\{\{1\}:p_3,\{\frac{1}{2}\}:p_2,\{\frac{1}{2}\}:p_4\},$
$\{\{\frac{1}{2},1\}:p_3,\{0,1\}:p_4\}$
$\{\{1\}:p_3,\{0,\frac{1}{2}\}:p_2\}\}$
$\Gamma \cup \{\{1\}:p_3\}$
$\emptyset$

Figure 3.4: A signed-sat proof tree for the formula Γ from Example 3.3

Example 3.3 *Let $N = \{0, \frac{1}{2}, 1\}$ and let Γ be the following signed CNF formula:*

$$\begin{aligned}\Gamma = \; & \{\{\{\tfrac{1}{2},1\}:p_1,\{0,\tfrac{1}{2}\}:p_4\},\{\{0\}:p_1,\{\tfrac{1}{2},1\}:p_3,\{1\}:p_2\},\\ & \{\{0\}:p_1,\{1\}:p_3,\{\tfrac{1}{2}\}:p_2,\{\tfrac{1}{2}\}:p_4\},\{\{\tfrac{1}{2},1\}:p_1,\{1\}:p_4\},\\ & \{\{\tfrac{1}{2},1\}:p_3,\{0,1\}:p_4\},\{\{1\}:p_3,\{0,\tfrac{1}{2}\}:p_2\}\}\end{aligned}$$

Figure 3.4 shows the proof tree created by function signed-sat when the input signed CNF formula is Γ. The root node contains Γ and the remaining nodes contain the signed CNF formula obtained after applying signed-unit-resolve to the formula selected for doing branching.

Let us prove that signed-sat is a decision procedure for the SAT problem in signed CNF formulas.

Theorem 3.1 *Given a signed CNF formula Γ, function signed-sat terminates and returns either true or false. if it returns true, then Γ is satisfiable. If it returns false, then Γ is unsatisfiable.*

Proof: First, we show that if function signed-sat terminates then either some branch of the proof tree created by signed-sat contains the empty formula (i.e. returns true) or every branch of the proof tree contains the empty clause (i.e. returns false). Suppose we have a proof tree with no branch containing the empty formula and some leaf node of a branch with a signed CNF formula Γ' which does not contain the empty clause; we show that signed-sat does not terminate. If Γ' contains signed unit clauses, then signed-sat calls signed-unit-resolve and applies the signed one-literal rule; otherwise, signed-sat selects a propositional atom of Γ' and applies the signed branching rule. Either way, signed-sat does not terminate.

Next we show that every proof attempt must terminate. Function signed-sat applies the signed one-literal rule and the signed branching rule. On the one hand, the signed one-literal rule decreases the number of distinct signed literals occurring in the formula. On the other hand, after applying the signed branching rule we obtain new signed CNF formulas that contain a signed unit clause. Then, signed-sat applies the signed one-literal rule to such formulas and so decreases the number of distinct signed literals occurring in them. Since we began with a finite number of occurrences of distinct signed literals, these rules can only be applied a finite number of times.

As we have shown that the signed one-literal rule (see Proposition 3.2) and the signed branching rule (see Proposition 3.1) preserve satisfiability, it is clear that when signed-sat returns false (i.e all the branches of the proof tree contain the empty clause), the input signed CNF formula Γ is unsatisfiable, and when signed-sat returns true (i.e. there is a branch with the empty formula), Γ is satisfiable. ■

3.5.1 Branching heuristics

Function signed-sat creates a proof tree whose size depends on the propositional atom selected for doing branching. Consequently, the heuristic embodied in pick-atom is a key element for achieving a proof procedure that runs as fast as possible in a wide range of hard instances. Two factors that are relevant for selecting a good propositional atom in the signed case are the following ones:

- The branching factor that results of doing branching on the propositional atom selected by function pick-atom.
- For each maximal truth value x_i and each propositional atom p, the number of small size clauses that contain a signed literal $S{:}p$ such that $x_i \notin S$. As such signed clauses are shortened during the application of the signed one-literal rule, they may become unit clauses after a few steps.

For each propositional atom p that appears in a signed CNF formula Γ, we define its branching factor $bf(p)$ as follows:

$$bf(p) = |\{x_1, \ldots, x_m\}|,$$

where $\{x_1, \ldots, x_m\}$ is a maximal truth value set of p in Γ.

A simple branching heuristic is to choose a propositional atom with a low branching factor. This way, we may get smaller proof trees. It is expected that this branching criterion works better than a random choice of the propositional atom selected for doing branching. The branching factor is at most two in the classical case, whereas it can be as large as the cardinality of the truth value set in the signed case. It is also interesting to note that the branching factor of a propositional atom p may decrease during the execution of signed-sat. This follows from the fact that some signed literals of the form $S{:}p$ are eliminated and the sign of some signed literals of the form $S'{:}p$ is shortened during the application of the signed one-literal rule.

Besides obtaining a low branching factor, it is also important to get subproblems that give rise to small proof subtrees: on the one hand, we should take into account the some signed clauses are shortened during the application of the signed one-literal rule; on the other hand, subproblems should have about the same size in order to avoid to explore a large search space when the algorithm backtracks.

The rest of this section is devoted to a first approach to the definition of a branching heuristic that relates the branching factor and the number of small size signed clauses that are shortened during the application of the signed one-literal rule. We take the lexicographic heuristic as a starting point.

In the lexicographic heuristic, given a classical CNF formula Γ and a propositional atom p, we have that $h_i(p)$ measures the number of occurrences of the literal p that are eliminated in clauses of length i when the one-literal rule is applied to $\Gamma \cup \{\neg p\}$, and $h_i(\neg p)$ measures the number of occurrences of the literal $\neg p$ that are eliminated in clauses of length i when the one-literal rule is applied to $\Gamma \cup \{p\}$.

Let us extend function h_i to the signed setting. Let Γ be a signed CNF formula, let p be a propositional atom occurring in Γ, and let $\{x_1, \ldots, x_m\}$ be a maximal truth value set of p in Γ. For each propositional atom p and for each maximal truth value x_j, $1 \leq j \leq m$, we define $h_i(\{x_j\}:p)$ as follows:

$$h_i(\{x_j\}:p) = |\{C \mid S:p \in C,\ C \in \Gamma,\ x_j \notin S,\ |C| = i\}|$$

We have that $h_i(\{x_j\} : p)$ measures the number of clauses of length i that are shortened because a literal of the form $S:p$ is eliminated when the signed one-literal rule is applied to $\Gamma \cup \{\{x_j\}:p\}$.

Let $h_i(\{x_{l_1}\}:p), \ldots, h_i(\{x_{l_m}\}:p)$ be the sequence of the $h_i(\{x_j\}:p)$'s in increasing order of their values. Then, we define

$$H_i(p) = \alpha_1\, h_i(\{x_{l_1}\}:p) + \cdots + \alpha_m\, h_i(\{x_{l_m}\}:p)$$

Function H_i is a heuristic measure of the number of clauses of length i that are shortened. As it is important that the subproblems generated have a similar size we provide the adjustable parameters $\alpha_1, \ldots, \alpha_n$. Such parameters are similar to the parameters α and β of the classical lexicographic heuristic. The setting of such adjustable parameters should be done experimentally as in the classical case.

Now we are ready to define a branching heuristic combining the two factors mentioned as follows: the *signed lexicographic heuristic* selects a propositional atom p with maximal vector

$$\left(\frac{H_1(p)}{bf(p)^\gamma}, \frac{H_2(p)}{bf(p)^\gamma}, \ldots, \frac{H_k(p)}{bf(p)^\gamma}\right)$$

under the lexicographic order.

The idea behind the adjustable parameter γ is to give priority to the propositional atoms that have a low branching factor. In practice, we would choose a propositional atom p with maximal vector $\left(\frac{H_l(p)}{bf(p)^{\gamma}}, \frac{H_{l+1}(p)}{bf(p)^{\gamma}}\right)$, where l is the length of the shortest signed clause. Then, we should develop the proof tree using the depth-first strategy starting by the branch with fewer clauses.

We claim that the previous branching heuristic could be extended taking into account, for each propositional atom p and for each maximal truth value x_i, the number of clauses that are removed when the signed one-literal rule is applied to a signed CNF formula that contains a signed unit clause of the form $\{\{x_i\}:p\}$.

3.6 A satisfiability checking procedure for regular CNF formulas

In this section we describe a satisfiability checking procedure for regular CNF formulas that can be viewed as a refinement of the proof procedure for arbitrary signed CNF formulas we have presented in the previous section.

Hähnle (1996) gave the pseudo-code of an extension of the DP procedure for regular CNF formulas and, as far as we know, it is the only multiple-valued DP-style procedure published so far. Nevertheless, our proof procedure differs in several aspects: we define suitable data structures for representing formulas, different branching rules, deletion strategies, and a branching heuristic which is an extension of the lexicographic heuristic (an extension of the two-sided Jeroslow-Wang rule was proposed in (Hähnle, 1996)).

As our aim is to design a DP-style procedure for regular CNF formulas, we first define the concepts of regular branching rule and regular one-literal rule. The branching rule and the one-literal rule we give below are the ones described in (Hähnle, 1996). In Section 3.6.1 we define an improved version of Hähnle's branching rule and a regular version of the shortest positive clause rule (used, for instance, in (Gallo and Urbani, 1989)).

Proposition 3.3 regular branching rule *Let Γ be a regular CNF formula, let $S:p$ be a regular literal occurring in Γ and let N be the truth value set. Then, Γ is satisfiable iff $\Gamma \cup \{S:p\}$ is satisfiable or $\Gamma \cup \{(N \setminus S):p\}$ is satisfiable.*

Proof: Assume that Γ is satisfiable. By definition of interpretation, $I(p) \in N$ for any model I of Γ. If $I(p) \in S$, then $\Gamma \cup \{S:p\}$ is satisfiable. Otherwise, $I(p) \in (N \setminus S)$ and $\Gamma \cup \{(N \setminus S):p\}$ is satisfiable.

Assume that $\Gamma \cup \{S:p\}$ is satisfiable or $\Gamma \cup \{(N \setminus S):p\}$ is satisfiable. If $\Gamma \cup \{S:p\}$ is satisfiable, then Γ is satisfiable. If $\Gamma \cup \{(N \setminus S):p\}$ is satisfiable, then Γ is satisfiable. ■

Proposition 3.4 regular one-literal rule *Let Γ be a regular CNF formula that contains a regular unit clause $\{S:p\}$. Let Γ' be obtained from Γ by first removing all clauses that contain a literal of the form $S':p$ such that $S \subseteq S'$, and second by removing all occurrences of literals of the form $S'':p$ such that $S \cap S'' = \emptyset$ from the remaining clauses. Then, Γ is satisfiable iff Γ' is satisfiable.*

Proof: Suppose that Γ is satisfiable. Since $\{S:p\} \in \Gamma$, there exists a model I of Γ such that $I(p) \in S$. On the one hand, I is a model of any subset of Γ. On the other hand, if we remove the literals $S'':p$ such that $S \cap S'' = \emptyset$, then I continues being a model of Γ' because $I(p) \notin S''$.

Suppose that Γ' is satisfiable. One can realize that by construction of Γ' there exists a model I of Γ' such that $I(p) \in S$. To obtain Γ from Γ' we add new clauses to Γ' and literals to clauses of Γ'. The addition of literals has no effect on the satisfiability and all the new clauses contain a literal $S':p$ such that $S \subseteq S'$. But I also satisfies these new clauses because $S \subseteq S'$ and $I(p) \in S$. Thus, Γ is satisfiable. ■

The pseudo-code of a satisfiability checking procedure based on the concepts of regular branching (cf. Proposition 3.3) and regular one-literal rule (cf. Proposition 3.4) is shown in Figure 3.5. The main function is regular-sat: it returns true if the input regular CNF formula Γ is satisfiable, and it returns false if Γ is unsatisfiable. Function regular-unit-resolve applies repeatedly the regular one-literal rule and function pick-literal selects the next literal to which the regular branching rule is applied.

Example 3.4 *Let $N = \{0, \frac{1}{3}, \frac{2}{3}, 1\}$ and let Γ be the following regular CNF formula:*

$$\begin{aligned}\Gamma \;=\; \{&\{\boxed{\leq 0}:p_1, \boxed{\geq 1}:p_3, \boxed{\geq 1}:p_4\}, \{\boxed{\leq \tfrac{1}{3}}:p_1, \boxed{\leq \tfrac{1}{3}}:p_3, \boxed{\geq 1}:p_4\},\\ &\{\boxed{\leq 0}:p_1, \boxed{\leq \tfrac{1}{3}}:p_3, \boxed{\leq \tfrac{1}{3}}:p_4\}, \{\boxed{\geq 1}:p_1, \boxed{\leq 0}:p_5\},\\ &\{\boxed{\geq \tfrac{2}{3}}:p_1, \boxed{\geq \tfrac{1}{3}}:p_5\}, \{\boxed{\geq 1}:p_3, \boxed{\leq \tfrac{1}{3}}:p_4\}\}\end{aligned}$$

Figure 3.6 shows the proof tree created by function regular-sat when the input regular CNF formula is Γ. The root node contains Γ and the remaining nodes contain the regular CNF formula obtained after applying regular-unit-resolve to the formula selected for doing branching.

Let us prove that regular-sat is a decision procedure for the SAT problem in regular CNF formulas.

Theorem 3.2 *Given a regular CNF formula Γ, function regular-sat terminates and returns either true or false. If it returns true, then Γ is satisfiable. If it returns false, then Γ is unsatisfiable.*

Proof: First, we show that if function regular-sat terminates, then either some branch of the proof tree created by regular-sat contains the empty formula (i.e. returns true) or every branch of the proof tree contains the empty clause (i.e. returns false). Suppose we have a proof tree with no branch containing the empty formula and some leaf node of a branch with a regular CNF formula Γ' which does not contain the empty clause; we show that regular-sat does not terminate. If Γ' contains regular unit clauses, then regular-sat calls regular-unit-resolve and

```
function regular-sat ($\Gamma$: set of clause) : boolean
var L: literal
begin
   $\Gamma$ := regular-unit-resolve($\Gamma$);
   if $\Gamma = \emptyset$ then return(true);
   if $\Box \in \Gamma$ then return(false);
   L := pick-literal($\Gamma$);
   if regular-sat ($\Gamma \cup \{L\}$) then
      return(true)
   else
      return(regular-sat ($\Gamma \cup \{\overline{L}\}$))
   endif
end
```

```
function regular-unit-resolve ($\Gamma$: set of clause) : set of clause
var L: literal
var C: clause
begin
   while $\exists\{L\} \in \Gamma$ and $\Box \notin \Gamma$ do
      $\Gamma := \{C \mid \nexists L' \in C \in \Gamma$ such that $L \subseteq L'\}$;
      $\Gamma := \{C \setminus \{L' \mid L' \in C$ and $L' \subseteq \overline{L}\} \mid C \in \Gamma\}$
   endwhile;
   return($\Gamma$)
end
```

Figure 3.5: A DP-style procedure for regular CNF formulas

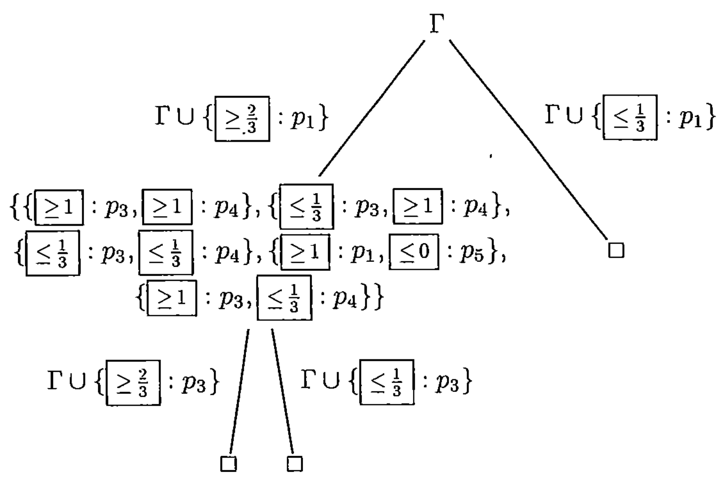

Γ
$\Gamma \cup \{\boxed{\geq \frac{2}{3}} : p_1\}$
$\Gamma \cup \{\boxed{\leq \frac{1}{3}} : p_1\}$
$\{\{\boxed{\geq 1} : p_3, \boxed{\geq 1} : p_4\}, \{\boxed{\leq \frac{1}{3}} : p_3, \boxed{\geq 1} : p_4\}, \{\boxed{\leq \frac{1}{3}} : p_3, \boxed{\leq \frac{1}{3}} : p_4\}, \{\boxed{\geq 1} : p_1, \boxed{\leq 0} : p_5\}, \{\boxed{\geq 1} : p_3, \boxed{\leq \frac{1}{3}} : p_4\}\}$
$\Gamma \cup \{\boxed{\geq \frac{2}{3}} : p_3\}$
$\Gamma \cup \{\boxed{\leq \frac{1}{3}} : p_3\}$
$\Box$
$\Box$
$\Box$

Figure 3.6: A regular-sat proof tree for the formula Γ from Example 3.4

applies the regular one-literal rule; otherwise, regular-sat selects a regular literal of Γ' and applies the regular branching rule. Either way, regular-sat does not terminate.

Next we show that every proof attempt must terminate. Function regular-sat applies the regular one-literal rule and the regular branching rule. On the one hand, the regular one-literal rule decreases the number of distinct regular literals occurring in the formula. On the other hand, after applying the regular branching rule we obtain two new formulas to which regular-sat applies the regular one-literal rule and so decreases the number of distinct regular literals occurring in them. Since we began with a finite number of occurrences of distinct literals, these rules can only be applied a finite number of times.

Proposition 3.4 and Proposition 3.3 state that Γ is satisfiable iff regular-unit-resolve($\Gamma \cup \{L\}$) is satisfiable or regular-unit-resolve($\Gamma \cup \{\overline{L}\}$) is satisfiable. Then, it is clear that when regular-sat returns false (i.e all the branches of the proof tree contain the empty clause), the input CNF formula Γ is unsatisfiable, and when regular-sat returns true (i.e. there is a branch with the empty formula), Γ is satisfiable. ■

The computational performance of an implementation of the previous satisfiability checking procedure depends on some aspects we list below:

- Hähnle's branching rule can be improved and other alternative branching criteria can be defined. This topic is considered in Section 3.6.1.

- The branching heuristic embodied in function pick-literal is an important factor to reduce the size of the proof tree created during the execution of function regular-sat. This question is discussed in Section 3.6.2.

- The computational complexity of the operations appearing in the proof procedure depends greatly on the data structures defined for representing formulas. In particular, function regular-unit-resolve can reach a linear-time complexity in the worst case. This point is treated in Section 3.6.3.
- Regular CNF formulas can be simplified using the deletion strategies defined in Section 3.6.4.

3.6.1 Alternative branching rules

As said before, the branching rule defined in Proposition 3.3 is not the only option we have of doing branching in the regular setting. In this section, we first define an improved version of Hähnle's branching rule and then a regular version of the two-valued shortest positive clause rule.

Proposition 3.5 *Let Γ be a regular CNF formula, let p be a propositional atom that occurs in Γ, and let $\top$ and $\bot$ denote the top and bottom elements of the truth value set. Then,*

1. $\Gamma \cup \{\boxed{\geq\alpha} : p\}$ *is satisfiable iff* $\Gamma \cup \{\boxed{\geq\gamma_1} : p\}$ *is satisfiable, where*

$$\gamma_1 = \begin{cases} \top \textit{ if } \nexists \boxed{\leq\beta} : p \in C \in \Gamma \textit{ such that } \beta \geq \alpha \\ \min\{\beta \mid \boxed{\leq\beta} : p \in C \in \Gamma, \beta \geq \alpha\}; \textit{ otherwise} \end{cases}$$

2. $\Gamma \cup \{\boxed{<\alpha} : p\}$ *is satisfiable*[7] *iff* $\Gamma \cup \{\boxed{\leq\gamma_2} : p\}$ *is satisfiable, where*

$$\gamma_2 = \begin{cases} \bot \textit{ if } \nexists \boxed{\geq\beta} : p \in C \in \Gamma \textit{ such that } \beta < \alpha \\ \max\{\beta \mid \boxed{\geq\beta} : p \in C \in \Gamma, \beta < \alpha\}; \textit{ otherwise} \end{cases}$$

Proof: We will only prove the first statement; the proof of the second one is similar. Suppose that $\Gamma \cup \{\boxed{\geq\alpha} : p\}$ is satisfiable. Then, there exists an interpretation I that satisfies $\Gamma \cup \{\boxed{\geq\alpha} : p\}$. It follows that $I(p) \geq \alpha$. We distinguish two cases:

1. There is no regular negative literal $\boxed{\leq\beta} : p$ in Γ such that $\beta \geq \alpha$: then, the interpretation I', obtained from I, which assigns to p the truth value $\top$ and is identical for the remaining propositional atoms satisfies $\Gamma \cup \{\boxed{\geq\alpha} : p\}$ and $\Gamma \cup \{\boxed{\geq\top} : p\}$. This is so since if there is a regular literal $S{:}p$ in $\Gamma \cup \{\boxed{\geq\alpha} : p\}$ whose sign S contains a truth value v such that $v > \alpha$, then it has positive polarity and $\top \in S$.

[7]When we say $\boxed{<\alpha}$ we mean the truth values which precede α on the chain of truth values. If α' is the greatest value of a regular sign, which appears in the regular CNF formula under consideration, such that $\alpha' < \alpha$, then $\boxed{<\alpha}$ means $\boxed{\leq\alpha'}$.

2. There is some regular negative literal $\boxed{\leq\beta}:p$ in Γ such that $\beta \geq \alpha$: let γ_1 be $\min\{\beta \,|\, \boxed{\leq\beta}:p \in C \in \Gamma, \beta \geq \alpha\}$. If $I(p) > \gamma_1$, then I also satisfies $\Gamma \cup \{\boxed{\geq\gamma_1}:p\}$. Otherwise, we have that $\alpha \leq I(p) \leq \gamma_1$. Then, the interpretation I' which assigns to p the truth value γ_1 and is identical for the remaining atoms satisfies $\Gamma \cup \{\boxed{\geq\alpha}:p\}$ and $\Gamma \cup \{\boxed{\geq\gamma_1}:p\}$. This is so since if there is a regular literal $S:p$ in $\Gamma \cup \{\boxed{\geq\alpha}:p\}$ whose sign contains a truth value v such that $\alpha \leq v \leq \gamma_1$, then $\gamma_1 \in S$, too.

Suppose that $\Gamma \cup \{\boxed{\geq\gamma_1}:p\}$ is satisfiable. As $\gamma_1 \geq \alpha$, it is clear that if $\Gamma \cup \{\boxed{\geq\gamma_1}:p\}$ is satisfiable, then $\Gamma \cup \{\boxed{\geq\alpha}:p\}$ is also satisfiable. ∎

Proposition 3.5 enables us to redefine Hähnle's branching rule as follows: instead of doing branching on $\Gamma \cup \{\boxed{\geq\alpha}:p\}$ and $\Gamma \cup \{\boxed{<\alpha}:p\}$, we now branch on $\Gamma \cup \{\boxed{\geq\gamma_1}:p\}$ and $\Gamma \cup \{\boxed{\leq\gamma_2}:p\}$, where γ_1 and γ_2 are defined as in Proposition 3.5. This way, the regular CNF formula obtained after the application of function regular-unit-resolve will usually have a smaller size than without this improvement. This is so since the new branching rule allows to remove a greater number of clauses during the execution of the regular one-literal rule.

Next, we present another another regular branching rule which can be viewed as a regular version of the two-valued shortest positive clause rule. As explained in (Hooker and Vinay, 1995), the two-valued shortest positive clause rule selects a shortest positive clause and creates as many branches as literals occur in the positive clause. Suppose we have the classical clause $p_1 \vee p_2 \vee p_3$, then three branches are created. The first one sets p_1 to true, the second one sets p_2 to true and p_1 to false (to avoid regenerating solutions in which p_1 is true), and the third one sets p_3 to true and p_2 and p_1 to false. By branching on positive clauses, it exploits the fact that there is no need to branch on any literal that never occurs in a positive clause. This is because if only such literals remain, the remaining clauses can always be satisfied by setting all propositional atoms to false.

Shortest positive clause branching is extended to the regular setting as follows: given a regular CNF formula Γ that contains a regular positive clause $(L_1 \vee L_2 \vee \cdots \vee L_k)$, we branch on $\Gamma \cup \{L_1\}$, $\Gamma \cup \{L_2\} \cup \{\overline{L_1}\}$, ..., $\Gamma \cup \{L_k\} \cup \{\overline{L_{n-1}}\} \cup \cdots \cup \{\overline{L_1}\}$. The main disadvantage of this approach is that the expansion of a node can create more than two branches. However, if the input formula has at most two regular literals per clause, then the number of branches is at most two. In Section 4.3.2 we define a satisfiability checking procedure for regular 2-CNF formulas which incorporates this branching criterion.

Finally, it is worth mentioning that the branching rule we have defined for arbitrary signed CNF formulas, which is based on the concept of maximal truth value set (cf. Proposition 3.1), is also valid for the subclass of regular CNF formulas. Contrary to the other branching rules, it has the advantage that it eliminates all the occurrences of regular literals of the form $S:p$ during the application of the one-literal rule if we branch on the propositional atom p.

3.6.2 Branching heuristics

Function regular-sat creates a proof tree whose size depends on the literal selected by the heuristic embodied in pick-literal. Notice that a regular branching heuristic has to choose both a propositional atom and a sign.

Hähnle (1996) defined a branching heuristic which is a regular version of the two-sided Jeroslow-Wang rule: given a regular CNF formula Γ, pick-literal selects a regular literal L occurring in Γ that maximizes $J(L) + J(\overline{L})$, where

$$J(L) = \sum_{\substack{\exists L' : L' \subseteq L \\ L' \in C' \in \Gamma}} 2^{-|C|} .$$

We will now define a refinement of the signed lexicographic heuristic given in Section 3.5.1. As the branching factor of the proof tree created by function regular-sat is at most two, we drop this factor from the branching heuristic defined and we get the following *regular lexicographic heuristic*:

We select a regular literal $S\!:\!p$ with maximal vector

$$(H_1(S\!:\!p), H_2(S\!:\!p), \ldots, H_k(S\!:\!p))$$

under the lexicographic order, where

$$H_i(S\!:\!p) = \alpha_1 \max(h_i(S\!:\!p), h_i((N \setminus S)\!:\!p)) + \alpha_2 \min(h_i(S\!:\!p), h_i((N \setminus S)\!:\!p))$$

and $h_i(S\!:\!p)$ is defined as follows:

$$h_i(S\!:\!p) = |\{C \mid S'\!:\!p \in C,\ C \in \Gamma,\ S' \cap S = \emptyset,\ |C| = i\}|$$

Observe that $h_i(S\!:\!p)$ measures the number of regular literals containing the propositional atom p which are removed in regular clauses of length i when we apply the regular one-literal rule to $\Gamma \cup \{S\!:\!p\}$. The meaning of the parameters α_1 and α_2 is like in the signed case.

In practice, we would choose a regular literal $S\!:\!p$ with maximal vector $(H_l(S\!:\!p), H_{l+1}(S\!:\!p))$, where l is the length of the shortest regular clause. Then, we should develop the branch with the fewest number of regular clauses.

As in the classical case, the idea behind the previous branching heuristics is to select regular literals occurring as often as possible in the shortest clauses of the formula, so that after a few steps the proof procedure generates as many regular unit clauses as possible in both branches of the proof tree.

3.6.3 Data Structures

In (Hähnle, 1996) there are no details about the underlying data structures used to represent formulas. This point, however, is significant because linear-time simplifications discussed below can only be achieved by choosing suitable data structures. In particular, our regular-unit-resolve has a linear-time complexity in the worst case in contrast with the quadratic complexity of (Hähnle, 1996).

The data structures we have developed to get time efficient operations in function regular-sat are the following ones:

- A regular CNF formula is represented as a doubly-linked list of its clauses, called regular clause list. A global counter of regular clauses is maintained and also a list of regular unit clauses.

- For each clause we maintain a doubly-linked list of its literals and each literal has a pointer to the head of the clause. The head of a clause contains a counter of its length.

- We maintain an array of all the different propositional atoms occurring in the formula. For each atom in the array we have a doubly-linked list of all its positive occurrences (called positive literal occurrence list) and a doubly-linked list of all its negative occurrences (called negative literal occurrence list). During the initialization phase, these lists are sorted in increasing order of the values of the regular signs. Thus, these lists are ordered. We have also pointers to the first and last elements of each list.

When we apply the regular one-literal rule to a regular CNF formula Γ that contains a regular unit clause $\{L\}$, we have to eliminate all the regular clauses that contain a literal L' such that $L \subseteq L'$ and delete all the occurrences of regular literals L'' such that $L'' \subseteq \overline{L}$. Observe that if the literal occurrence lists are ordered, these lists are traversed only once. For instance, if $L = \boxed{\geq 0.5} : p$ we have to eliminate all the clauses that contain a literal $\boxed{\geq i} : p$ such that $i \leq 0.5$. Then, we scan only the positive literal occurrence list from the first element until we reach a regular positive literal $\boxed{\geq i} : p$ such that $i > 0.5$ (these literals are removed from the positive occurrence list when the clause is deleted). Moreover, we have to delete all the occurrences of regular literals $\boxed{\leq i} : p$ such that $i < 0.5$. Then, we scan only the negative literal occurrence list from the first element until we reach a negative literal $\boxed{\leq i} : p$ such that $i \geq 0.5$ (these literals are removed from the positive occurrence list when the literal is deleted). When the regular unit clause $\{L\}$ contains a regular literal with negative polarity we start to traverse the literal occurrence lists from the last element instead of the first one.

Observe that if we do not sort the literal occurrence lists, they can be traversed, during the execution of regular-unit-resolve, as many times as regular literals they contain.

Example 3.5 *Figure 3.7 shows the data structure defined above for the following regular CNF formula:*

$$\{\{\boxed{\geq 0.5} : p_1, \boxed{\geq 0.3} : p_2, \boxed{\leq 0.8} : p_4\}, \{\boxed{\geq 0.7} : p_1, \boxed{\geq 0.4} : p_2, \boxed{\leq 0.1} : p_3\},$$
$$\{\boxed{\geq 0.6} : p_1\}, \{\boxed{\leq 0.8} : p_3, \boxed{\leq 0.9} : p_4\}, \{\boxed{\geq 0.9} : p_2, \boxed{\leq 0.4} : p_3\}\}$$

In the following we will examine the worst-case time complexity of the main operations appearing in function regular-sat.

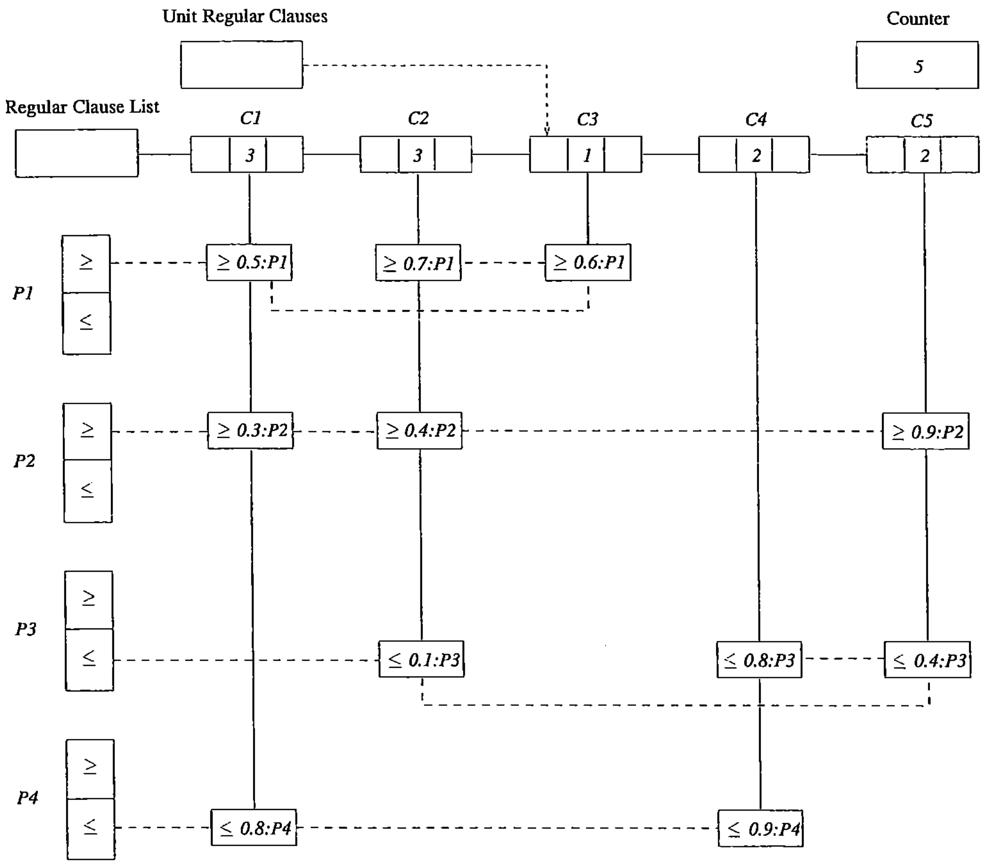


Figure 3.7: Data structures

operation	complexity
$\Gamma =^? \emptyset$	$\mathcal{O}(1)$
$\square \in^? \Gamma$	$\mathcal{O}(1)$
pick-literal(Γ)	$\mathcal{O}(\lvert\Gamma\rvert)$
regular-unit-resolve(Γ)	$\mathcal{O}(\lvert\Gamma\rvert)$

Since a global counter of clauses is maintained, the test $\Gamma = \emptyset$ needs constant time. Since we can detect in constant time if we delete a regular literal from a regular unit clause, the test $\square \in \Gamma$ needs constant time.

The steps and their corresponding worst-case running times of function pick-literal are the following ones: first, it traverses the regular clause list to find out the smaller clause length l and, as the head of each clause contains its length, this step is proportional to the number of clauses. Second, it creates a list of all the different atoms occurring in the regular clauses of length l. This step is in $\mathcal{O}(\Gamma)$. And third, for each sign S and for each atom p of the previous list, it calculates $H_l(S:p)$ and $h_{l+1}(S:p)$ (defined in Section 3.6.2) and chooses the most promising literal. This step is in $\mathcal{O}(\Gamma)$. Thus, the worst-case time complexity of pick-literal is in $\mathcal{O}(\Gamma)$.

The worst-case time complexity of regular-unit-resolve for a regular CNF formula Γ is in $\mathcal{O}(\lvert\Gamma\rvert)$ due to the following facts: detection of regular unit clauses is done in constant time, because a list of regular clauses of length 1 is maintained; direct access to the lists of all the regular literals and all the regular clauses that have to be eliminated by the regular one-literal rule is provided by the literal occurrence lists; deletion of a specific regular literal is done in constant time (we remove the literal in constant time because clauses are doubly-linked lists, we delete the literal from the literal occurrence list in constant time because literal occurrence lists are doubly-linked, and we decrement the counter of the corresponding clause in constant time because a pointer to the head of the clause is provided); and deletion of a specific regular clause is proportional to the length of the clause (we remove each literal in constant time and also decrement the global counter of regular clauses in constant time).

As already noted, literal occurrence lists are scanned only once since they are sorted during the initialization phase. The cost of sorting is in $\mathcal{O}(n \log n)$ for each list, where n is the length of the list. This cost has not been considered for obtaining the complexity of regular-unit-resolve. In the finitely-valued case, this sorting is not necessary as we will show in Section 4.3.2.

An implementation of the proof procedure should also maintain a stack containing the information removed during the application of the regular one-literal rule. This way, this information could be recovered when the algorithm does backtracking and it would be necessary to maintain just one regular CNF formula in the computer memory.

3.6.4 Deletion strategies

In this section we present several deletion strategies (i.e. elimination of irrelevant or redundant regular clauses for proving satisfiability). It is important to check if the inclusion of any of them improves the computational performance of particular satisfiability testing algorithms for regular CNF formulas.

Proposition 3.6 regular pure literal rule *Let Γ be a regular CNF formula, let p be a propositional atom appearing in Γ such that all its occurrences have the same polarity, and let $\Gamma' = \{C \in \Gamma \mid \nexists S{:}p \in C \in \Gamma\}$. Then, Γ is satisfiable iff Γ' is satisfiable.*

Proof: Suppose that Γ is satisfiable. Since Γ' is a subset of Γ, it follows that Γ' is satisfiable.

Suppose that Γ' is satisfiable and I is a model of Γ'. Observe that Γ' contains no occurrences of the propositional atom p. let $\top$ and $\bot$ denote the top and bottom elements of the truth value set. If all the occurrences of regular literals of the form $S{:}p$ in Γ have positive polarity, then the interpretation I' that assigns to p the value $\top$ and is identical to I for the remaining propositional atoms, satisfies Γ. If all the occurrences of literals of the form $S{:}p$ in Γ have negative polarity, then the interpretation I' that assigns to p the value $\bot$ and is identical to I for the remaining propositional atoms, satisfies Γ. ■

Proposition 3.6 states that as long as there are monotone regular literals (literals with the same polarity for a given propositional atom) in a regular CNF formula, we can remove the regular clauses that contain them, and satisfiability is preserved.

Proposition 3.7 *Let Γ be a regular CNF formula, let p be a propositional atom appearing in Γ such that $\max\{i \mid \boxed{\geq i}: p \in C \in \Gamma\} \leq \min\{j \mid \boxed{\leq j}: p \in C \in \Gamma\}$, and let $\Gamma' = \{C \in \Gamma \mid \nexists S{:}p \in C \in \Gamma\}$. Then, Γ is satisfiable iff Γ' is satisfiable.*

Proof: Suppose that Γ is satisfiable. Since Γ' is a subset of Γ, it follows that Γ' is satisfiable.

Suppose that Γ' is satisfiable and I is a model of Γ'. Observe that the intersection of all the regular signs occurring in Γ with the propositional atom p is non-empty and Γ' contains no occurrences of p. Thus, the interpretation I' that assigns to p an arbitrary value of such an intersection and is identical to I for the remaining propositional atoms satisfies Γ. ■

Given a regular CNF formula, Proposition 3.7 allows to remove all the regular clauses with a propositional atom p if the intersection of all the signs occurring in the formula in regular literals of the form $S{:}p$ is non-empty.

Given a signed CNF formulas Γ and a propositional atom p occurring in Γ, if the maximal truth value set of p in Γ is a singleton $\{x\}$, then the application of the signed one-literal rule to $\Gamma \cup \{\{x\}{:}p\}$ can be seen as an extension of Proposition 3.7 to arbitrary signed CNF formulas.

Finally, we consider the elimination of subsumed and tautology clauses.

Definition 3.3 subsumed regular clause *Let C_1 and C_2 be regular clauses. C_2 is subsumed by C_1 iff for each regular literal $S_1 : p \in C_1$ there is a literal $S_2{:}p \in C_2$ such that $S_1 \subseteq S_2$.*

It is clear that the elimination of subsumed regular clauses in a regular CNF formula has no effect on the satisfiability of the formula; if a regular clause C_2 is subsumed by a regular clause C_1, then C_2 is satisfiable if C_1 is satisfiable. The elimination of subsumed clauses defined for regular clauses is also valid for signed clauses, provided they are *merged* in the sense of the assumptions made on page 32.

On the other hand, signed and regular clauses can be eliminated if they contain literals of the form $S_1\!:\!p, \ldots, S_k\!:\!p$ such that $S_1 \cup \cdots \cup S_k = N$. Such clauses are tautologies.

Chapter 4

Polynomially Solvable Satisfiability Problems

Abstract: The SAT problem in classical logic is polynomially solvable for Horn and 2-CNF formulas. In this chapter we address the Horn SAT and 2-SAT problems in signed CNF formulas. Our objective is to investigate under which circumstances such signed SAT problems are polynomially solvable. Concerning the Horn SAT problem, we first describe an almost linear-time decision procedure for a subclass of infinitely-valued regular Horn formulas. Then, we extend this result to arbitrary regular Horn formulas and give a satisfiability checking procedure that reaches a linear-time complexity when the truth value set is finite. The complexity is almost linear in the infinitely-valued case. Concerning the 2-SAT problem, we start by proving that it is NP-complete in signed CNF formulas, contrary to what happens in classical logic. The 2-SAT problem, however, turns out to be polynomially solvable in regular and monosigned CNF formulas. We describe quadratic-time decision procedures for solving the 2-SAT problem in such subclasses of signed CNF formulas. To this end, we define appropriate calculi, design decision procedures equipped with suitable data structures, prove their completeness and analyze their time complexity.

4.1 Introduction

It is a well-known fact that there exist subclasses of classical formulas whose SAT problems are polynomially solvable. Horn and 2-CNF formulas are the two most representative examples. Nevertheless, little attention has been paid so far to identify similar subclasses of multiple-valued formulas whose SAT problems are also polynomially solvable. In view of this, our aim in this chapter is to look into this question and present some new results. More specifically, we focus on the Horn SAT and 2-SAT problems, whose classical counterparts admit linear-time algorithms.[1]

Our approach to investigate polynomially solvable SAT problems in signed CNF formulas starts by defining refinements of existing logic calculi for signed CNF formulas and proving that they are refutation complete for Horn formulas or 2-CNF formulas. We then define suitable data structures for representing formulas and design efficient satisfiability checking procedures. Finally, we analyze the computational complexity of the proof procedures and prove their completeness.

Concerning the Horn SAT problem, we define a unit resolution-style calculus and describe an almost linear-time decision procedure for a subclass of infinitely-valued regular Horn formulas we call Horn mv-formulas. Then, we extend all these results to arbitrary regular Horn formulas and give a satisfiability checking procedure that reaches a linear-time complexity when the truth value set is finite. The complexity is almost linear in the infinitely-valued case.

Concerning the 2-SAT problem, we begin by showing that it is NP-complete in signed CNF formulas. The 2-SAT problem, however, turns out to be polynomially solvable in regular and monosigned CNF formulas. Taking as a starting point a refinement of the DP-style procedures described in Chapter 3, we then design several quadratic-time decision procedures for solving the 2-SAT problem in these special, but important, subclasses of signed CNF formulas. In addition, we propose different alternative branching rules for the binary case.

This chapter is organized as follows. In Section 4.2 we define the logic of mv-formulas and a unit resolution-style calculus, and we describe an efficient satisfiability checking procedure for Horn mv-formulas. We then extend all these results to the framework of regular Horn formulas. In Section 4.3 we prove the NP-completeness of the 2-SAT problem in signed CNF formulas. Finally, we present a detailed description of quadratic-time decision procedures for solving the 2-SAT problem in regular and monosigned CNF formulas.

4.2 The Horn SAT problem

The linearity of the Horn SAT problem in classical logic was first proved by Dowling and Gallier (1984) and since then other linear-time algorithms have been published; e.g. (Minoux, 1988; Escalada-Imaz, 1989a; Scutellà, 1990; Ghallab and Escalada-Imaz, 1991).

As far as we know, the first paper that addressed the Horn SAT problem in the multiple-valued setting is due to Escalada-Imaz and Manyà (1994c). This paper presents an almost linear-time satisfiability checking procedure for a subclass of infinitely-valued regular Horn formulas we call mv-formulas. Later, Hähnle (1996) described a graph-based satisfiability testing algorithm for arbitrary regular Horn formulas. It has the same complexity in the infinitely-valued case and a linear-time complexity in the finitely-valued case.

[1] As already noted, the Horn SAT problem can only be considered in regular CNF formulas because other known subclasses of signed CNF formulas lack the concept of polarity of literals.

In this section, we first present a detailed description of a decision procedure for solving the Horn SAT problem in mv-formulas. Then, we extend our result to arbitrary regular Horn formulas. In both cases we define a unit resolution-style calculus, prove its refutation completeness, define suitable data structures for representing Horn formulas, describe a satisfiability checking procedure, and analyze the computational complexity.

4.2.1 The Horn SAT problem in mv-formulas

First of all, we make precise the logic of mv-formulas. Then, we define a calculus and prove its soundness and completeness in the Horn case.

Syntax and semantics

Definition 4.1 literal *A literal is either a propositional atom p, called positive literal, or the negation of a propositional atom $\neg p$, called negative literal.*

Definition 4.2 mv-clause *Let $L_1, \ldots, L_m$ be literals and let $\alpha \in [0,1]$. An expression of the form $(L_1 \vee \cdots \vee L_m; \alpha)$ is an mv-clause and α is its sign. An mv-clause $(S; \alpha)$ such that S has exactly one literal is a unit mv-clause. An mv-clause $(S; \alpha)$ such that S has at most one positive literal is a Horn mv-clause. An mv-clause $(S; \alpha)$ such that all the literals occurring in S are positive (are negative) is a positive (a negative) mv-clause.*

Definition 4.3 Horn mv-formula *An mv-formula is a finite set of mv-clauses. A Horn mv-formula is a finite set of Horn mv-clauses.*

Definition 4.4 interpretation *An interpretation I is a mapping that assigns to every propositional atom an element of the unit interval $[0,1]$. An interpretation I is extended to literals as follows:*

- *if $L = p$, then $I(L) = I(p)$;*
- *if $L = \neg p$, then $I(L) = 1 - I(p)$.*

Definition 4.5 satisfiability *An interpretation I satisfies an mv-clause $C = (L_1 \vee \cdots \vee L_m; \alpha)$, denoted by $I \models C$, iff $I(L_i) \geq \alpha$ for some literal L_i, $1 \leq i \leq m$. An mv-formula Γ is satisfiable iff there exists an interpretation that satisfies all the mv-clauses in Γ. An mv-formula that is not satisfiable is unsatisfiable. The empty mv-clause is always unsatisfiable and the empty mv-formula is always satisfiable.*

Looking at the previous definitions, it is easy to realize that an mv-clause of the form

$$(p_1 \vee \cdots \vee p_i \vee \neg p_{i+1} \vee \cdots \vee \neg p_m; \alpha)$$

is equivalent to the following regular clause:

$$\boxed{\geq \alpha} : p_1 \vee \cdots \vee \boxed{\geq \alpha} : p_i \vee \boxed{\leq 1 - \alpha} : p_{i+1} \vee \cdots \vee \boxed{\leq 1 - \alpha} : p_m .$$

Thus, mv-formulas can be considered to be a subclass of regular CNF formulas.[2]

Proposition 4.1 *Let Γ be an mv-formula. If Γ contains two mv-clauses $(\neg p; \alpha), (p; \beta)$ such that $\alpha + \beta > 1$, then Γ is unsatisfiable.*

Proof: Assume that there exists an interpretation I that satisfies Γ. Since I satisfies $(\neg p; \alpha)$ and $(p; \beta)$, it must be that $I(p) \leq 1 - \alpha$ and $I(p) \geq \beta$. But this is a contradiction because $\alpha + \beta > 1$. Therefore, Γ is unsatisfiable. ■

The satisfiability checking procedure for Horn mv-formulas we describe later on tries to derive two mv-clauses that fulfill the conditions of Proposition 4.1.

Logical inference

Next, we present a refutation complete calculus for Horn mv-formulas. Its only inference rule is called Multiple-valued Positive Unit Resolution (MPUR).

Definition 4.6 multiple-valued positive unit resolution *Let $(\neg p \vee S; \alpha)$ be a Horn mv-clause and let $(p; \beta)$ be a positive unit mv-clause. The multiple-valued positive unit resolution rule (MPUR) is defined as follows:*

$$\frac{(\neg p \vee S; \alpha) \quad (p; \beta)}{(S; \alpha)}$$

provided that $\alpha + \beta > 1$.

Definition 4.7 proof *A proof of an mv-clause C from a Horn mv-formula Γ, denoted by $\Gamma \vdash_{MPUR} C$, is a finite sequence of mv-clauses $C_1, \ldots, C_m$ such that $C_m = C$ and, for each k $(1 \leq k \leq m)$, either C_k is a clause of Γ or C_k is obtained from C_i and C_j $(k > i, j)$ applying the MPUR rule.*

The mv-clause derived from $(\neg p; \alpha)$ and $(p; \beta)$, provided that $\alpha + \beta > 1$, is the empty mv-clause, denoted by $\Box$. A proof of the empty mv-clause from a Horn mv-formula Γ is a *refutation* of Γ. Next, we show that the MPUR calculus is refutation complete for Horn mv-formulas.

Proposition 4.2 one-literal rule for mv-formulas *Let Γ be a Horn mv-formula that contains a positive unit mv-clause $(p; \alpha)$. Let Γ' be obtained from Γ by first removing all mv-clauses of the form $(S; \beta)$ such that S contains an occurrence of the literal p and $\alpha \geq \beta$, and second by removing all occurrences of the literal $\neg p$ in all mv-clauses of the form $(S'; \gamma)$ such that S' contains an occurrence of $\neg p$ and $\alpha + \gamma > 1$. Then, Γ is satisfiable iff Γ' is satisfiable.*

Proof: Suppose that Γ is satisfiable. Since $(p; \alpha) \in \Gamma$, there exists an interpretation I that satisfies Γ such that $I(p) \geq \alpha$. On the one hand, I satisfies any subset of Γ. On the other hand, if we delete the occurrences of the literal $\neg p$ in mv-clauses of the form $(S'; \gamma)$ such that S' contains an occurrence of $\neg p$

[2]This fact was already noted by Hähnle (1996).

and $\alpha + \gamma > 1$, the interpretation I also satisfies Γ' because $I(\neg p) \leq 1 - \alpha$ and $1 - \alpha < \gamma$. Thus, Γ' is satisfiable.

Suppose that Γ' is satisfiable. One can realize that by construction of Γ' there exists an interpretation I that satisfies Γ' such that $I(p) \geq \alpha$. To obtain Γ from Γ' we add new clauses to Γ' and literals to clauses of Γ'. The addition of literals has no effect on the satisfiability and the new clauses $(S; \alpha_i)$ we add contain an occurrence of the literal p. Since by construction of Γ' we have that $\alpha \geq \alpha_i$ and $I(p) \geq \alpha \geq \alpha_i$, the interpretation I also satisfies these new clauses. Thus, Γ is satisfiable. ■

Theorem 4.1 soundness and completeness *A Horn mv-formula* Γ *is unsatisfiable iff there exists a refutation of* Γ.

Proof: (Soundness) Since the empty mv-clause is unsatisfiable (cf. Proposition 4.1) and is obtained after a finite number of applications of the MPUR rule, it suffices to show that if there exists an interpretation that satisfies both $(\neg p \vee S; \alpha)$ and $(p; \beta)$, and $\alpha + \beta > 1$, then this interpretation satisfies $(S; \alpha)$ as well. Assume that $(\neg p \vee S; \alpha)$ and $(p; \beta)$ are satisfiable. Let I be an interpretation that satisfies both $(\neg p \vee S; \alpha)$ and $(p; \beta)$. Then, it must be that $I(p) \geq \beta$ and $I(\neg p) \leq 1 - \beta$. Since $I(\neg p) \leq 1 - \beta$ and $1 - \beta < \alpha$, the interpretation I satisfies $(S; \alpha)$ as well.

(Completeness) Suppose that Γ is unsatisfiable. We must show that there exists a refutation of Γ using the MPUR rule. We proceed by induction on l, where l is the length of Γ (i.e. the sum of the total number of literal occurrences in each mv-clause of Γ). If $l = 0$, then it must be that $\Gamma = \{\Box\}$ because Γ is unsatisfiable. Thus, there exists a refutation of Γ.

Suppose now that there exists a refutation for every unsatisfiable Horn mv-formula whose length is at most l, and suppose that the length of Γ is $l + 1$. Since Γ is unsatisfiable, Γ must contain at least one positive unit mv-clause (otherwise, the interpretation that assigns to every propositional atom the value 0 satisfies Γ). Let $(p; \alpha)$ be a positive unit mv-clause that appears in Γ and let Γ' be the Horn mv-formula obtained from Γ by applying the one-literal rule for mv-formulas using the unit mv-clause $(p; \alpha)$ (cf. Proposition 4.2). Then, the length of Γ' is at most l and, by Proposition 4.2, Γ' is unsatisfiable. By the induction hypothesis, there exists a refutation $\mathcal{R}$ of Γ'. If all the mv-clauses of Γ' appearing in $\mathcal{R}$ are also mv-clauses of Γ, then $\mathcal{R}$ is also a refutation of Γ and we are done. Otherwise, let $(S_1; \gamma_1), \ldots, (S_m; \gamma_m)$ be the mv-clauses of Γ' appearing in $\mathcal{R}$ such that the literal $\neg p$ was deleted during the application of the one-literal rule for mv-formulas. We put back the literal $\neg p$ at these mv-clauses, and we have that the mv-clauses so obtained belong to Γ. We now apply the MPUR rule to each one of these mv-clauses together with the unit mv-clause $(p; \alpha)$ used to create Γ'. Since by construction of Γ' we have that $\gamma_1 + \alpha > 1, \ldots, \gamma_m + \alpha > 1$, we obtain $(S_1; \gamma_1), \ldots, (S_m; \gamma_m)$ again. Adding these resolution steps to $\mathcal{R}$ we get a refutation of Γ. ■

```
procedure Hmv-sat-1 ($\Gamma$: Horn_mv-formula)
var p, $\neg$p: literal
var S: clause
var $\alpha, \beta$: truth_value
var $\Gamma'$: Horn_mv-formula
begin
  $\Gamma' := \Gamma$;
  while $\Gamma' \neq \emptyset$ and $\Box \notin \Gamma'$ do
    $\Gamma := \Gamma \cup \Gamma'$;
    $\Gamma' := \emptyset$;
    while $\exists(p; \alpha), (\neg p \vee S; \beta) \in \Gamma$ such that $\alpha + \beta > 1$ do
      if $(S; \beta) \notin \Gamma$ then $\Gamma' := \Gamma' \cup \{(S; \beta)\}$
    endwhile
  endwhile;
  if $\Box \in \Gamma'$ then
    return(unsatisfiable)
  else
    return(satisfiable)
  endif
end
```

Figure 4.1: A first satisfiability checking procedure for Horn mv-formulas

A satisfiability checking procedure for Horn mv-formulas

The remainder of this section is devoted to design an efficient satisfiability checking procedure for Horn mv-formulas based on the MPUR calculus. As it has some complicated data structures and operations, we present three descriptions of the procedure. The first description is shorter and more abstract; the others are longer and more detailed.

A first description of the procedure

The pseudo-code of the first proof procedure for checking the satisfiability of Horn mv-formulas is shown in Figure 4.1. The principle of this procedure consists in applying repeatedly the MPUR rule until either the empty mv-clause is derived (in this case, the input mv-formula is unsatisfiable) or a saturation state is reached (in this case, the input mv-formula is satisfiable provided that the empty mv-clause has not been derived). In the pseudo-code, Γ contains initially the input Horn mv-formula. In the i–th iteration, Γ contains both the input Horn mv-clauses and the resolvents derived in the $i-1$ previous iterations.

Example 4.1 *Let Γ be a Horn mv-formula that contains the following clauses:*

$$(\neg p_1, p_2; 0,7)$$
$$(\neg p_2, p_3; 0,9)$$
$$(\neg p_3, p_2; 0,5)$$
$$(\neg p_3, \neg p_1; 0,7)$$
$$(p_1; 0,4)$$

The different values that Γ takes in successive iterations are

$$\begin{array}{lll}\Gamma & \leftarrow & \Gamma \\ \Gamma & \leftarrow & \Gamma \cup \{(p_2;0,7),(\neg p_3;0,7)\} \\ \Gamma & \leftarrow & \Gamma \cup \{(p_2;0,7),(\neg p_3;0,7)\} \cup \{(p_3;0,9)\} \\ \Gamma & \leftarrow & \Gamma \cup \{(p_2;0,7),(\neg p_3;0,7)\} \cup \{(p_3;0,9)\} \cup \{(p_2;0,5),(\neg p_1;0,7),\Box\}\end{array}$$

Γ is unsatisfiable because we have derived the empty mv-clause.

Theorem 4.2 *Given a Horn mv-formula Γ, procedure Hmv-sat-1 terminates and returns either satisfiable or unsatisfiable. It returns satisfiable if the input Horn mv-formula Γ is satisfiable. It returns unsatisfiable if the input Horn mv-formula Γ is unsatisfiable.*

Proof: The procedure applies the MPUR rule to each pair of mv-clauses of the form $(p;\beta)$, $(\neg p \vee S;\alpha)$ such that $\alpha + \beta > 1$, eliminates the literal $\neg p$ and adds to the input formula the new mv-clauses so derived until either the empty mv-formula is derived or a saturation state is reached. Since the input Horn mv-formula contains a finite number of negative literals, only a finite number of negative literals can be eliminated. Hence, the procedure terminates and two final situations can arise:

1. The empty mv-clause is derived because of applying the MPUR rule to two mv-clauses of the form $(\neg p;\alpha)$ and $(p;\beta)$ such that $\alpha + \beta > 1$. Then, the input Horn mv-formula is unsatisfiable by virtue of the soundness of the MPUR calculus.

2. A state is reached in which the MPUR rule cannot produce new resolvents and the empty mv-clause has not been derived. Then, the input Horn mv-formula is satisfiable by virtue of the completeness of the MPUR calculus.

■

An improved procedure

Even though procedure Hmv-sat-1 is a decision procedure that allows us to solve the Horn SAT problem in mv-formulas in polynomial time, it is worthwhile to pursue a faster proof procedure taking into account the following points:

1. It is not necessary to save all the positive unit mv-clauses that have a same propositional atom and only differ in the value of their sign. For instance, if we have two positive unit mv-clauses $C_1 = (p; \alpha)$ and $C_2 = (p; \beta)$ such that $\alpha > \beta$, then it suffices to save C_1. This follows from the fact that the interpretations that satisfy both C_1 and C_2 are exactly the same interpretations that satisfy C_1. In other words, $C_1 \wedge C_2$ is logically equivalent to C_1. From a computational point of view, this is obtained with a data structure –val(p)– that is updated to the value α each time a new positive unit mv-clause $(p; \alpha)$ such that $\alpha >$ val(p) is derived. If a positive unit mv-clause $(p; \beta)$ such that val(p) $\geq \beta$ is derived, then it is not considered.

2. In order to determine the mv-clauses to which we can apply the MPUR rule in a given state, we must pick up a positive unit mv-clause $(p; \alpha)$ and a Horn mv-clause $(S; \beta)$ such that S contains an occurrence of $\neg p$ and $\alpha + \beta > 1$. This could be done as follows: we maintain one list –negative-clauses(p)– for each propositional atom p. Initially, negative-clauses(p) contains all the mv-clauses that have an occurrence of the literal $\neg p$. Each time a new positive unit mv-clause $(p; \alpha)$ is derived, the mv-clauses in negative-clauses(p) are scanned. For each Horn mv-clause $C = (S; \beta)$ such that S contains an occurrence of $\neg p$ and $\alpha + \beta > 1$, the literal $\neg p$ is removed from S and C is deleted from negative-clauses(p). If all the negative literals have been removed from S then the clause $(q; \beta)$ is derived, where q is the positive literal in S. If there is no positive literal in S, then the empty mv-clause is derived and so the input mv-formula Γ is unsatisfiable.

   Although this principle is simple, one can realize that to find an occurrence of a literal $\neg p$ in an mv-clause $(S; \beta)$, when its literals are represented as a list, is in $\mathcal{O}(|S|)$ and to remove all the negative literals occurring in $(S; \beta)$ is in $\mathcal{O}(|S|^2)$. Thus, instead of representing S as a list we associate with each clause $C = (S; \beta)$ a counter –counter(C)– that indicates the number of occurrences of negative literals $\neg p$ in S such that val(p) $+ \beta \leq 1$ (i.e. the number of negative literals in S that have not yet been removed by an application of the MPUR rule). With this data structure, to eliminate a negative literal of S is done in constant time and to remove all the negative literals of S is in $\mathcal{O}(|S|)$. When counter(C) $= 0$, the mv-clause $(q; \beta)$, where q is the positive literal in S, is derived. If S has no positive literal, then the empty mv-clause is derived.

Our next step is to incorporate the previous improvements into procedure Hmv-sat-1, as well as to define suitable data structures for representing Horn mv-formulas. First of all, we introduce the data structures that the new satisfiability checking procedure will use.

For each propositional atom p that occurs in the input Horn mv-formula Γ:

val(p) $= \alpha$, where α is the sign with maximum value that appears in the unit mv-clauses containing p that have been derived so far.

$negative\text{-}clauses(p) = \{(S; \alpha) \in \Gamma \mid \neg p \in S\}$

For each Horn mv-clause $C = (S, \alpha)$ in Γ:

$sign(C) = \alpha$, where α is the sign of C

$counter(C) = r$, where $r = |\{\neg p \in S \mid val(p) + sign(C) \leq 1\}|$

$positive(C) = p$, where p is the positive literal in S; otherwise it is *nil*.

The pseudo-code of the improved procedure is shown in Figure 4.2 and Figure 4.3. Procedure initialization takes as input a Horn mv-formula Γ and creates and updates the data structures we have defined. Procedure Hmv-sat-2 takes as input a Horn mv-formula Γ, and returns satisfiable if Γ is satisfiable and returns unsatisfiable if Γ is unsatisfiable.

Theorem 4.3 *Given a Horn mv-formula Γ, Hmv-sat-2(Γ) terminates and returns satisfiable iff Γ is satisfiable. The time complexity of Hmv-sat-2(Γ) is in $\mathcal{O}(|\Gamma| + c\,k)$, where $|\Gamma|$ is the length of Γ, c is the number of mv-clauses whose positive literal also occurs in another mv-clause of Γ and k is the maximum number of mv-clauses of Γ that contain an occurrence of a same negative literal.*

Proof: The completeness of procedure mvH-sat-2 is a straightforward consequence of Theorem 4.2 and the improvements discussed above. Let us analyze its complexity:

Procedure initialization is in $\mathcal{O}(|\Gamma|)$ due to the following facts:

- The for loop of line 8 has a time complexity that is bounded by $|\Gamma|$.
- The for loop of line 12 has a time complexity that is bounded by the number of mv-clauses in Γ.
- The for loop of line 16 is executed as many times as mv-clauses are in Γ and the time complexity of each iteration is the number of occurrences of literals in the mv-clause under consideration. Therefore, its time complexity is bounded by $|\Gamma|$.

Procedure mvH-sat-2 is in $\mathcal{O}(|\Gamma| + c\,k)$:

- Line 5 is in $\mathcal{O}(|\Gamma|)$.
- The for loop of line 10 is executed at most k times and if this loop is executed once for each distinct positive literal occurring in Γ, then the complexity of the while loop of line 6 is in $\mathcal{O}(|\Gamma|)$. But, as some positive literals p can be derived with different sign, negative-clauses(p) can be scanned as many times as mv-clauses exist containing an occurrence of the literal p. Therefore, the worst-case complexity of the while loop of line 6 is in $\mathcal{O}(|\Gamma| + c\,k)$.

■

```
procedure initialization ($\Gamma$: Horn_mv-formula)
var p, $\neg$p: literal
var S: clause
var $\alpha$: truth_value
var $C$: mv-clause
var $\Gamma_{unit}$: set of mv-clause
begin
  $\Gamma_{unit}$ := $\emptyset$;
  for each atom p in $\Gamma$ do
    val(p):= 0;
    negative-clauses(p):= $\emptyset$
  endfor;
  for each unit clause $(p; \alpha) \in \Gamma$ do
    if val(p)=0 then $\Gamma_{unit} := \Gamma_{unit} \cup \{p\}$;
    if $\alpha$ >val(p) then val(p):= $\alpha$
  endfor;
  for each clause $C = (S, \alpha) \in \Gamma$ do
    sign(C) := $\alpha$;
    if $\exists p \in S$ then
      positive(C) := p;
      counter(C) := $|S| - 1$
    else
      positive(C) := nil;
      counter(C) := $|S|$
    endif;
    for each $\neg p \in S$ do
      negative-clauses(p) := negative-clauses(p) $\cup\{C\}$
    endfor
  endfor
end
```

Figure 4.2: Initialization phase for Horn mv-formulas

```
procedure Hmv-sat-2 ($\Gamma$: Horn_mv-formula)
var p, $\neg$p: literal
var C: mv-clause
var $\Gamma_{unit}, \Gamma'_{unit}$: set of mv-clause
begin
  initialization($\Gamma$);
  while $\Gamma_{unit} \neq \emptyset$ do
    $\Gamma'_{unit} := \Gamma_{unit}$;
    $\Gamma_{unit} := \emptyset$;
    for each atom p in $\Gamma'_{unit}$ do
      for each $C \in$ negative-clauses($p$) do
        if $sign(C) + val(p) > 1$ then
          $decrement(counter(C))$;
          negative-clauses($p$) := negative-clauses($p$) $\setminus \{C\}$;
          if $counter(C) = 0$ then
            if $positive(C) = nil$ then return(unsatisfiable);
            if $sign(C) > val(positive(C))$ then
              $val(positive(C)) := sign(C)$;
              $\Gamma_{unit} := \Gamma_{unit} \cup \{positive(C)\}$
            endif
          endif
        endif
      endfor
    endfor
  endwhile;
  return(satisfiable)
end
```

Figure 4.3: A satisfiability checking procedure for Horn mv-formulas

The definitive proof procedure

In order to improve the time complexity of procedure initialization and procedure Hmv-sat-2, we can incorporate the following modifications:

- For each propositional atom p that appears in Γ, the list negative-clauses(p) is sorted. An ordered list $\{C_1, \ldots, C_i, \ldots, C_j, \ldots, C_n\}$ such that $sign(C_i) \geq sign(C_j)$ for all $i < j$ $(1 \leq i, j \leq n)$ is obtained.

- This ordering enables one to efficiently scan the lists negative-clauses(p): each time a new unit mv-clause of the form $(p; \alpha)$ is derived, only the mv-clauses C_i such that $sign(C_i) + \alpha > 1$ are scanned and removed from the list. This way, the lists negative-clauses(p) are scanned only once.

The previous modifications are incorporated into procedure initialization and procedure mvH-sat-2, giving raise to the definitive satisfiability checking procedure for Horn mv-formulas. Its pseudo-code is shown in Figure 4.4 and Figure 4.5.

Theorem 4.4 *Given a Horn mv-formula Γ, Hmv-sat(Γ) terminates and returns satisfiable iff Γ is satisfiable. The time complexity of Hmv-sat(Γ) is in $\mathcal{O}(|\Gamma| + m \log k)$, where $|\Gamma|$ is the length of Γ, m is the sum of the number of occurrences of negative literals in Γ and k is the maximum number of mv-clauses of Γ that contain an occurrence of a same negative literal.*

Proof: The completeness of procedure Hmv-sat is a straightforward consequence of Theorem 4.3 and the improvements discussed above. Let us analyze its complexity:

The new procedure initialization is in $\mathcal{O}(|\Gamma| + m \log k)$ due to the following facts:

- The for loops of lines 8, 12 and 16 have the time complexity calculated in the proof of Theorem 4.3.

- The for loop of line 29 is executed as many times as distinct propositional atoms occur in Γ. Let $p_1, \ldots, p_n$ be such propositional atoms. The cost of each iteration is $k_{p_i} \log k_{p_i}$, where k_{p_i} is the number of mv-clauses that contain an occurrence of the literal $\neg p_i$. Then, the sum $k_{p_1} \log k_{p_1} + \ldots + k_{p_n} \log k_{p_n}$ is bounded by $m \log k$.

Procedure Hmv-sat is in $\mathcal{O}(|\Gamma| + m \log k)$:

- Line 5 is in $\mathcal{O}(|\Gamma| + m \log k)$.

- Now, the total number of mv-clauses scanned and removed on the lists negative-clauses(p) is bounded by m. Therefore, the worst-case complexity of the while loop of line 6 is in $\mathcal{O}(m)$.

■

```
procedure initialization ($\Gamma$: Horn_mv-formula)
var p, ¬p: literal
var S: clause
var $\alpha$: truth_value
var $C$: mv-clause
var $\Gamma_{unit}$: set of mv-clause
begin
    $\Gamma_{unit} := \emptyset$;
    for each atom p in $\Gamma$ do
        val(p):= 0;
        negative-clauses(p):= $\emptyset$
    endfor;
    for each unit clause $(p;\alpha) \in \Gamma$ do
        if val(p)=0 then $\Gamma_{unit} := \Gamma_{unit} \cup \{p\}$;
        if $\alpha$ >val(p) then val(p):= $\alpha$
    endfor;
    for each clause $C = (S, \alpha) \in \Gamma$ do
        sign(C) := $\alpha$;
        if $\exists p \in S$ then
            positive(C) := p;
            counter(C) := $|S| - 1$
        else
            positive(C) := nil;
            counter(C) := $|S|$
        endif;
        for each $\neg p \in S$ do
            negative-clauses(p) := negative-clauses(p) $\cup \{C\}$
        endfor
    endfor;
    for each atom p in $\Gamma$ do
        negative-clauses(p):= sort(negative-clauses(p))
    endfor
end
```

Figure 4.4: Initialization phase for Horn mv-formulas

```
procedure Hmv-sat ($\Gamma$: Horn_mv-formula)
var p, $\neg$p: literal
var C: mv-clause
var $\Gamma_{unit}, \Gamma'_{unit}$: set of mv-clause
begin
  initialization($\Gamma$);
  while $\Gamma_{unit} \neq \emptyset$ do
    $\Gamma'_{unit} := \Gamma_{unit}$;
    $\Gamma_{unit} := \emptyset$;
    for each atom p in $\Gamma'_{unit}$ do
      $C$ := head(negative-clauses(p));
      while $C \neq nil$ and $sign(C) + val(p) > 1$ do
        $decrement(counter(C))$;
        negative-clauses(p) := tail(negative-clauses(p));
        if $counter(C) = 0$ then
          if $positive(C) = nil$ then return(unsatisfiable);
          if $sign(C) > val(positive(C))$ then
            $val(positive(C)) := sign(C)$;
            $\Gamma_{unit} := \Gamma_{unit} \cup \{positive(C)\}$
          endif
        endif;
        $C$ := head(negative-clauses(p))
      endwhile
    endfor
  endwhile;
  return(satisfiable)
end
```

Figure 4.5: The definitive version of a satisfiability checking procedure for Horn mv-formulas

4.2.2 The Horn SAT problem in regular CNF formulas

The aim of this section is to extend the results obtained for mv-formulas to arbitrary regular Horn formulas. Therefore, we begin by defining a unit resolution-style calculus and proving that it is refutation complete for regular Horn formulas. Then, we describe an efficient satisfiability checking procedure for regular Horn formulas. As such a proof procedure is similar to the procedure we have designed for mv-formulas, we only present the definitive version.

Logical inference

Unit resolution for regular clauses is known to be complete for determining the satisfiability of regular Horn formulas (Hähnle, 1996). The calculus we introduce here is a refinement of regular unit resolution since only regular positive unit clauses must be considered for deriving resolvents. We make precise our calculus in the following definitions and then prove its refutation completeness.

Definition 4.8 regular positive unit resolution (RPUR) *Let* $\boxed{\geq i} : p$ *be a regular positive unit clause and let* $\boxed{\leq j} : p \vee C$ *be a regular Horn clause. The regular positive unit resolution rule (RPUR) is defined as follows:*

$$\frac{\begin{array}{c}\boxed{\geq i} : p \\ \boxed{\leq j} : p \vee C\end{array}}{C}$$

provided that $i > j$.

Definition 4.9 proof *A proof of a regular clause* C *from a regular Horn formula* Γ*, denoted by* $\Gamma \vdash_{RPUR} C$*, is a finite sequence of regular Horn clauses* $C_1, \ldots, C_m$ *such that* $C_m = C$ *and, for each* k $(1 \leq k \leq m)$*, either* C_k *is a clause of* Γ *or* C_k *is obtained from* C_i *and* C_j $(k > i, j)$ *applying the RPUR rule.*

The regular clause derived from $\boxed{\geq i} : p$ and $\boxed{\leq j} : p$, provided that $i > j$, is the regular empty clause, denoted by $\Box$. A proof of the regular empty clause from a regular Horn formula Γ is a *refutation* of Γ.

Theorem 4.5 soundness and completeness *A regular Horn formula* Γ *is unsatisfiable iff there exists a refutation of* Γ.

Proof: (Soundness) Since the regular empty clause is unsatisfiable and is obtained after a finite number of applications of the RPUR rule, it suffices to show that if there exists an interpretation that satisfies both $\boxed{\geq j} : p$ and $\boxed{\leq i} : p \vee C$, and $i < j$, then this interpretation satisfies C as well. Assume that $\boxed{\geq j} : p$ and $\boxed{\leq i} : p \vee C$ are satisfiable. Let I be an interpretation that satisfies both $\boxed{\geq j} : p$ and $\boxed{\leq i} : p \vee C$. Then, it must be that $I(p) \geq j$. Since $i < j$, the interpretation I satisfies C as well.

(Completeness) Suppose that Γ is unsatisfiable. We must show that there exists a refutation of Γ using the RPUR rule. We proceed by induction on l, where l is the length of Γ. If $l = 0$, then it must be that $\Gamma = \{\Box\}$ because Γ is unsatisfiable. Thus, there exists a refutation of Γ.

Suppose now that there exists a refutation for every unsatisfiable regular Horn formula whose length is at most l, and suppose that the length of Γ is $l+1$. Since Γ is unsatisfiable, Γ must contain at least one regular positive unit clause (otherwise, Γ is clearly satisfiable). Let $\{\boxed{\geq j} : p\}$ be a regular positive unit clause that appears in Γ and let Γ' be the regular Horn formula obtained from Γ by applying the regular one-literal rule using the regular unit clause $\{\boxed{\geq j} : p\}$ (cf. Proposition 3.4). Then, the length of Γ' is at most l and, by Proposition 3.4, Γ' is unsatisfiable. By the induction hypothesis, there exists a refutation $\mathcal{R}$ of Γ'. If all the regular clauses of Γ' appearing in $\mathcal{R}$ are also regular clauses of Γ, then $\mathcal{R}$ is also a refutation of Γ and we are done. Otherwise, let $C_1, \ldots, C_m$ be the regular clauses of Γ' appearing in $\mathcal{R}$ such that the literals $\boxed{\leq i_1} : p, \ldots, \boxed{\leq i_m} : p$ were deleted during the application of the regular one-literal rule. We put back the eliminated literals $\boxed{\leq i_1} : p, \ldots, \boxed{\leq i_m} : p$ at these clauses, and we have that the regular clauses so obtained belong to Γ. We now apply the RPUR rule to each one of these clauses together with the regular unit clause $\{\boxed{\geq j} : p\}$ used to create Γ'. Since by construction of Γ' we have that $i_1 < j, \ldots, i_m < j$, we obtain $C_1, \ldots, C_m$ again. Adding these resolution steps to $\mathcal{R}$ we get a refutation of Γ. ■

A satisfiability checking procedure for regular Horn formulas

First of all, we define suitable data structures for representing regular Horn formulas. The differences between the data structures for Horn mv-formulas and the data structures for regular Horn formulas are: the lists negative-clauses(p) not only contain the regular clauses in which there is an occurrence of a regular negative literal containing the propositional atom p, but also contain the value of the sign of such regular literals; and as regular clauses contain a *local* sign for each literal instead of a *local* sign for each clause, the sign has been eliminated in the data structure used to represent clauses.

Data Structures:

For each propositional atom p that occurs in the input regular Horn formula Γ:

$val(p) = i$, where i is the greatest value of the regular sign that appears in the regular positive unit clauses containing p that have been derived so far.

$negative\text{-}clauses(p) = \{(C, i) \mid \boxed{\leq i} : p \in C \text{ and } C \in \Gamma\}$

For each regular Horn clause $C = (L_1 \vee \cdots \vee L_m)$ in Γ:

$counter(C) = r$, where $r = |\{L_i \in C \mid L_i = \boxed{\leq j} : p \text{ and } val(p) \leq j\}|$

positive$(C) = L$, where L is the positive literal in C; otherwise it is *nil*.

The pseudo-code of the satisfiability checking procedure for regular Horn formulas we propose is shown in Figure 4.6 and Figure 4.7. Procedure initialization takes as input a regular Horn formula Γ and creates and updates the data structures defined. Procedure Horn-sat takes as input a regular Horn formula Γ; it returns satisfiable if Γ is satisfiable and returns unsatisfiable if Γ is unsatisfiable.

Theorem 4.6 *Given a regular Horn formula Γ, Horn-sat(Γ) terminates and returns satisfiable iff Γ is satisfiable. The time complexity of Horn-sat(Γ) is in $\mathcal{O}(|\Gamma| + m\log k)$, where $|\Gamma|$ is the length of Γ, m is the sum of the number of occurrences of negative literals in Γ and k is the maximum number of regular clauses of Γ that contain an occurrence of a regular negative literal with the same propositional atom.*

Proof: The arguments used to establish the completeness of procedure Hmv-sat remain valid to establish the completeness of procedure Horn-sat taking into account that we now automate the application of the RPUR inference rule, which is refutation complete for regular Horn formulas. The time complexity of Horn-sat(Γ) is in $\mathcal{O}(|\Gamma|+m\log k)$. This complexity coincides with the complexity we have obtained for procedure Hmv-sat because we have only introduced minor changes which have no effect on the time complexity. ■

Procedure Horn-sat may reach a linear-time complexity, provided that the truth value set is fixed and finite, modifying slightly the data structure used to represent propositional atoms. Remember that we maintain, for each propositional atom p, the list negative-clauses(p) formed by all the regular clauses that contain an occurrence of a regular negative literal of the form $S\!:\!p$. Now, we should maintain, for each truth value k and atom p, the list clauses$\left(\boxed{\leq k} : p\right)$ formed by all the regular clauses that contain an occurrence of the regular literal $\boxed{\leq k} : p$. This way, we have that

- the lists clauses$\left(\boxed{\leq k} : p\right)$ do not need to be sorted during the initialization phase;
- if a regular positive unit clause of the form $\{\boxed{\geq i} : p\}$ is derived, we have to scan all the lists clauses$\left(\boxed{\leq j} : p\right)$ such that $i > j$. Observe that the lists clauses$\left(\boxed{\leq j} : p\right)$ are scanned only once during the execution of Horn-sat, since once they are scanned they become empty lists.

Therefore, it is easy to realize that if we incorporate the previous modifications into procedure Horn-sat we obtain a linear-time complexity when the truth value set is fixed and finite. In Section 4.3.2 we give more details about this question.

Hähnle (1996) obtained the same results on time complexity for his satisfiability checking procedure for regular Horn formulas, but with a different method.

```
procedure initialization-Horn ($\Gamma$: regular_Horn_formula)
var p: atom
var L: regular_literal
var $i, j, k$: truth_value
var $C$: regular_clause
var $\Gamma_{unit}$: set of regular_clause
begin
    $\Gamma_{unit} := \emptyset$;
    for each atom p in $\Gamma$ do
        val(p):= 0;
        negative-clauses(p):= $\emptyset$
    endfor;
    for each positive unit clause $\{\boxed{\geq i} : p\} \in \Gamma$ do
        if val(p)=0 then $\Gamma_{unit} := \Gamma_{unit} \cup \{p\}$;
        if $i$ >val(p) then val(p):= $i$
    endfor;
    for each clause $C = (L_1 \vee \cdots \vee L_m) \in \Gamma$ do
        if there is a positive literal $L_j \in S$ then
            positive(C) := $L_j$;
            counter(C) := $|C| - 1$
        else
            positive(C) := nil;
            counter(C) := $|C|$
        endif;
        for each negative literal $\boxed{\leq k} : p \in C$ do
            negative-clauses(p) := negative-clauses(p) $\cup\{(C, k)\}$
        endfor
    endfor;
    for each atom p in $\Gamma$ do
        negative-clauses(p):= sort(negative-clauses(p))
    endfor
end
```

Figure 4.6: Initialization phase for regular Horn formulas

```
procedure Horn-sat ($\Gamma$: regular_Horn_formula)
var p: atom
var C: regular_clause
var $\Gamma_{unit}, \Gamma'_{unit}$ : set of regular_clause
begin
  initialization-Horn ($\Gamma$);
  while $\Gamma_{unit} \neq \emptyset$ do
    $\Gamma'_{unit} := \Gamma_{unit}$;
    $\Gamma_{unit} := \emptyset$;
    for each atom $p$ in $\Gamma'_{unit}$ do
      $(C, i)$ := head(negative-clauses(p));
      while $(C, i) \neq nil$ and $val(p) > i$ do
        $decrement(counter(C))$;
        negative-clauses(p) := tail(negative-clauses(p));
        if $counter(C) = 0$ then
          if $positive(C) = nil$ then return(unsatisfiable);
          if $positive(C) = \boxed{\geq k} : p'$ and $k > val(p')$ then
            $val(p') := k$;
            $\Gamma_{unit} := \Gamma_{unit} \cup \{p'\}$
          endif
        endif;
        $(C, i)$ := head(negative-clauses(p))
      endwhile
    endfor
  endwhile;
  return(satisfiable)
end
```

Figure 4.7: A satisfiability checking procedure for regular Horn formulas

4.3 The 2-SAT problem

The 2-SAT problem in classical logic is polynomially solvable (Cook, 1971). Several efficient algorithms for solving this problem have been described in the literature: Even et al. (1976) reduce the 2-SAT problem to the timetable problem; Aspvall et al. (1979) and Escalada-Imaz (1989b) reduce the 2-SAT problem to the problem of finding strongly connected components in a digraph and describe algorithms with linear-time complexity in the worst case. It is also worth mentioning the parallel algorithms developed by Jones et al. (1976) and Cook and Luby (1988). Petreschi and Simeone (1991) give a comparison of the computational performance of different algorithms.

As far as we know, the 2-SAT problem in multiple-valued logics has not been considered before. In view of this, our first aim is to study the complexity class of this problem. In Section 4.3.1 we prove that, contrary to what happens in classical logic, the 2-SAT problem in signed CNF formulas is NP-complete. Since we are faced with an intractable problem in the sense of polynomially bounded time complexity, our next step is to identify subclasses of signed 2-CNF formulas whose SAT problems are polynomially solvable. In Section 4.3.2 and Section 4.3.3 we describe quadratic-time satisfiability checking procedures for regular 2-CNF formulas and monosigned 2-CNF formulas, respectively.

4.3.1 The 2-SAT problem in signed CNF formulas

Our aim in this section is to show the NP-completeness of the 2-SAT problem in signed CNF formulas (signed 2-SAT).

Theorem 4.7 *The 2-SAT problem in signed CNF formulas is NP-complete.*

Proof: We will show that (i) this problem belongs to NP and (ii) the 3-colourability problem is polynomially reducible to the signed 2-SAT problem.

The signed 2-SAT problem clearly belongs to NP: given a satisfiable signed 2-CNF formula, a nondeterministic algorithm can guess a satisfying interpretation and check that it satisfies the formula in polynomial time.

In the 3-colourability problem we are given an undirected graph $G = (V, E)$ and we are asked whether there is a function $c : V \to \{1,2,3\}$ such that for each edge $[u,v] \in E$ we have $c(u) \neq c(v)$. This problem is known to be NP-complete (Garey and Johnson, 1979). Given such a graph we construct an instance of the signed 2-SAT problem as follows: for each edge $[u,v] \in E$, we define three signed binary clauses

$$\{\{2,3\}{:}u, \{2,3\}{:}v\}, \{\{1,3\}{:}u, \{1,3\}{:}v\}, \{\{1,2\}{:}u, \{1,2\}{:}v\}$$

and we take as truth value set $N = \{1,2,3\}$. The intended meaning of the previous signed clauses is that there are no two adjacent vertices with the same colour. Observe that from the definition of interpretation we can ensure that every vertex is coloured with only one colour. This reduction can obviously be performed in polynomial time.

Let Γ be the signed 2-CNF formula obtained by reducing an instance of the 3-colourability problem for an arbitrary graph G to an instance of the signed 2-SAT problem. Observe that if V is empty, we obtain the signed empty formula. We claim that Γ is satisfiable iff the graph G is 3-colourable. On the one hand, if there exists an interpretation I that satisfies Γ, we define, for each vertex u, $c(u) = I(u)$. It is clear that this is a valid colouring. On the other hand, if G is 3-colourable then it is easy to check that, for each vertex u, the interpretation I such that $I(u) = c(u)$ satisfies Γ since adjacent vertices have different colour. ■

From the previous proof we get the following results as corollaries:

Corollary 4.1 *The SAT problem in signed CNF formulas (signed SAT) is NP-hard.*

Corollary 4.2 *The signed 2-SAT problem, when $|N| \geq 3$ and the length of signs is ≥ 2, is NP-hard.*

The 3-colourability problem can also be reduced to the classical SAT problem. Given an undirected graph $G = (V, E)$, we construct a classical CNF formula Γ as follows:

- For each vertex $u \in V$, the CNF formula Γ contains the following clauses:

$$\bigvee_{j \in \{1,2,3\}} u_j \wedge \bigwedge_{\substack{j,k \in \{1,2,3\} \\ j \neq k}} (\neg u_j \vee \neg u_k)$$

  The intended meaning of u_j is that vertex u is coloured with colour $j \in \{1, 2, 3\}$ and the intended meaning of the whole expression is that vertex u is coloured with only one colour.

- For each edge $[u, v] \in E$, the CNF formula Γ contains the following clauses:

$$\bigwedge_{j \in \{1,2,3\}} (\neg u_j \vee \neg v_j)$$

  The intended meaning of this expression is that the adjacent vertices u and v do not have the same colour.

Observe that the 3-colourability problem for an undirected graph can be represented in a more concise way if we use signed 2-CNF formulas instead of classical CNF formulas. This shows that signed CNF formulas can be used as a powerful knowledge representation language for certain kinds of problems. Moreover, as new satisfiability checking procedure for signed CNF formulas appear, we will need to evaluate and compare the computational performance of such procedures by performing experiments on both real-world and randomly generated problems. Thus, in this section, we have made a first step into this direction because the 3-colourability problem is frequently used as a benchmark in satisfiability competitions.

4.3.2 The 2-SAT problem in regular CNF formulas

In the preceding section we have shown that the signed 2-SAT problem is NP-complete. In the present section we examine this problem again, but we turn our attention to regular CNF formulas (regular 2-SAT problem). We show that, in this special but important case, there exist polynomial-time satisfiability testing algorithms.

Our approach to design efficient algorithms for solving the regular 2-SAT problem relies on an improvement of the classical DP procedure suggested by Rauzy (1995). Such an improvement is known as model separation and is implicit in other algorithms that solve the classical 2-SAT problem. The key idea is that when a pair variable-value is chosen in an enumerative algorithm, either this assignment is demonstrated unsatisfiable by applying repeatedly the one-literal rule or it can be definitively kept. In the regular setting, this amounts to the following proposition:

Proposition 4.3 *Let Γ be a regular 2-CNF formula, let L be a regular literal occurring in Γ and let Γ' be the formula obtained after applying regular-unit-resolve to $\Gamma \cup \{L\}$. Then, Γ' is satisfiable iff Γ is satisfiable and $\Box \notin \Gamma'$.*

Proof: Suppose that Γ' is satisfiable. As regular-unit-resolve applies a finite number of times the regular one-literal rule and this rule preserves satisfiability (cf. Proposition 3.4), it follows that Γ' is satisfiable iff $\Gamma \cup \{L\}$ is satisfiable. Since $\Gamma \cup \{L\}$ is satisfiable and $\Gamma \subseteq \Gamma \cup \{L\}$, it follows that Γ is satisfiable. As Γ' is satisfiable, it is clear that $\Box \notin \Gamma'$.

Suppose that Γ is satisfiable and $\Box \notin \Gamma'$. As the regular clauses of Γ are binary, regular-unit-resolve derives a regular unit clause when a literal from a clause of Γ is deleted. As $\Box \notin \Gamma'$, regular-unit-resolve cannot delete literals from regular unit clauses; it removes all such clauses. Therefore, the set of regular clauses returned by regular-unit-resolve($\Gamma \cup \{L\}$) is a subset of Γ. Since Γ is satisfiable and Γ' is a subset of Γ, it follows that Γ' is satisfiable. ■

Rauzy (1995) points out that the worst-case time complexity of the DP procedure is not polynomial when the input formula is a 2-CNF formula, contrary to what many authors have written. From the results of Proposition 4.3, we now describe a satisfiability checking procedure for regular 2-CNF formulas with quadratic-time worst-case complexity which can be viewed as a refinement of function regular-sat (cf. Section 3.6).

A satisfiability checking procedure for regular 2-CNF formulas

Proposition 4.3 states that, when the input formula to regular-sat is a regular 2-CNF formula, it suffices to expand only one node per level in the proof tree created by regular-sat. The node expanded does not have to contain the empty clause. This is the idea behind the procedure whose pseudo-code is shown in Figure 4.8. Function regular-2sat returns true if the input regular 2-CNF formula Γ is satisfiable, and it returns false if Γ is unsatisfiable, regular-unit-resolve

```
function regular-2sat ($\Gamma$: set of clause) : boolean
var L: literal
var $\Gamma'$: set of clause
begin
  if $\Gamma = \emptyset$ then return(true);
  if $\Box \in \Gamma$ then return(false);
  L := pick-literal($\Gamma$);
  $\Gamma'$ := regular-unit-resolve($\Gamma \cup \{L\}$);
  if $\Box \in \Gamma'$ then $\Gamma'$ := regular-unit-resolve($\Gamma \cup \{\overline{L}\}$);
  regular-2sat($\Gamma'$)
end
```

Figure 4.8: A proof procedure for regular 2-CNF formulas

applies repeatedly the regular one-literal rule (cf. Section 3.6) and pick-literal selects the next literal to which the branching rule is applied.

Theorem 4.8 *Let Γ be a regular 2-CNF formula. If Γ is satisfiable, then function regular-2sat terminates and returns true. If Γ is unsatisfiable, then function regular-2sat terminates and returns false.*

Proof: First, we show that if function regular-2sat terminates, then either some branch of the proof tree created by regular-2sat contains the empty formula (i.e. returns true) or every branch of the proof tree contains the empty clause (i.e. returns false). Suppose we have a proof tree with no branch containing the empty formula and some leaf node of a branch with a regular CNF formula Γ' which does not contain the empty clause; we show that regular-2sat does not terminate. If Γ' contains regular unit clauses, then regular-2sat calls regular-unit-resolve and applies the regular one-literal rule; otherwise, regular-2sat selects a regular literal of Γ' and applies the regular branching rule. Either way, regular-2sat does not terminate.

Next we show that every proof attempt must terminate. Function regular-2sat applies the regular one-literal rule and the regular branching rule. On the one hand, the regular one-literal rule decreases the number of distinct regular literals occurring in the formula. On the other hand, after applying the regular branching rule we obtain two new formulas to which regular-2sat can apply the regular one-literal rule and so decrease the number of distinct regular literals occurring in them. Since we began with a finite number of occurrences of distinct literals, these rules can only be applied a finite number of times.

Proposition 3.4 and Proposition 3.3 state that Γ is satisfiable iff regular-unit-resolve($\Gamma \cup \{L\}$) is satisfiable or regular-unit-resolve($\Gamma \cup \{\overline{L}\}$) is satisfiable. If $\Box \in$ regular-unit-resolve($\Gamma \cup \{L\}$), then Γ is satisfiable iff regular-unit-resolve($\Gamma \cup \{\overline{L}\}$) is satisfiable. Otherwise, by Proposition 4.3 we have that Γ is satisfiable iff regular-unit-resolve($\Gamma \cup \{L\}$) is satisfiable. Then, it is clear that when regular-2sat returns false (i.e all the branches of the proof tree contain the empty clause), the input formula Γ is unsatisfiable, and when regular-2sat returns true (i.e. there is a branch with the empty formula), Γ is satisfiable. ■

As our procedure expands only one node per level, the literal selected by pick-literal is not so critical for the computational performance of the algorithm and, if any branching heuristic is used, we should consider a trade-off between the cost of computing the heuristic and the benefits obtained. Actually, most algorithms developed for solving the classical 2-SAT problem do not incorporate any kind of heuristic. Thus, an option is to choose a random literal. Another option is to choose the most frequent literal. Our intention is to develop this topic in the future: on the one hand, implementing different algorithms for solving the regular 2-SAT problem; on the other hand, looking for both real-world and randomly generated instances for evaluating and comparing such algorithms.

Example 4.2 *Let $N = \{0, \frac{1}{2}, 1\}$ and let Γ be the following regular 2-CNF formula:*

$$\begin{aligned}\Gamma \;=\; & \{\{\boxed{\geq 1}:p_1, \boxed{\geq 1}:p_2\}, \{\boxed{\geq 1}:p_3, \boxed{\geq 1}:p_4\}, \{\boxed{\leq \tfrac{1}{2}}:p_1, \boxed{\geq \tfrac{1}{2}}:p_2\},\\ & \{\boxed{\leq \tfrac{1}{2}}:p_3, \boxed{\geq 1}:p_4\}, \{\boxed{\geq \tfrac{1}{2}}:p_1, \boxed{\geq \tfrac{1}{2}}:p_2\}, \{\boxed{\geq 1}:p_3, \boxed{\leq \tfrac{1}{2}}:p_4\},\\ & \{\boxed{\geq 1}:p_1, \boxed{\leq 0}:p_1\}, \{\boxed{\leq \tfrac{1}{2}}:p_1, \boxed{\geq 1}:p_2\}, \{\boxed{\leq \tfrac{1}{2}}:p_4, \boxed{\leq \tfrac{1}{2}}:p_3\}\}\end{aligned}$$

Figure 4.9 shows the proof tree created by function regular-sat when the input formula is Γ. Figure 4.10 shows the proof tree created by function regular-2sat when the input formula is Γ.

Data structures and complexity

The data structures we propose to get time efficient operations in procedure regular-2sat, provided we are working with a fixed and finite truth value set, are the following ones:

- Each regular clause has a head and a doubly-linked list of its regular literals.
- We maintain a global counter of regular clauses and a list of regular unit clauses.
- For each possible combination formed by a regular sign S and a propositional atom p occurring in the formula (i.e., for all the possible regular literals, where S can have either positive or negative polarity) we maintain a list of the regular clauses containing an occurrence of the regular literal $S:p$. These lists are called literal occurrence lists.

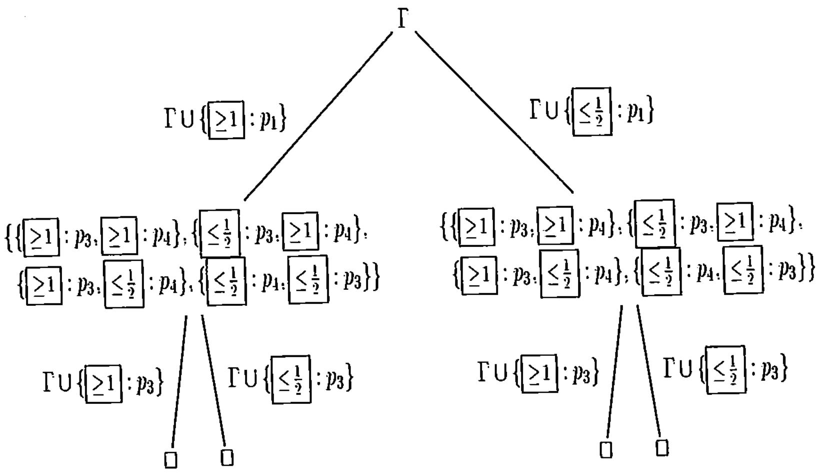

Γ
$\Gamma \cup \{\boxed{\geq 1} : p_1\}$
$\Gamma \cup \{\boxed{\leq \frac{1}{2}} : p_1\}$
$\{\{\boxed{\geq 1} : p_3, \boxed{\geq 1} : p_4\}, \{\boxed{\leq \frac{1}{2}} : p_3, \boxed{\geq 1} : p_4\},$
$\{\boxed{\geq 1} : p_3, \boxed{\leq \frac{1}{2}} : p_4\}, \{\boxed{\leq \frac{1}{2}} : p_4, \boxed{\leq \frac{1}{2}} : p_3\}\}$
$\Gamma \cup \{\boxed{\geq 1} : p_3\}$
$\Gamma \cup \{\boxed{\leq \frac{1}{2}} : p_3\}$
$\Box$
$\Box$
$\Box$
$\Box$

Figure 4.9: A regular-sat proof tree for the formula Γ from Example 4.2

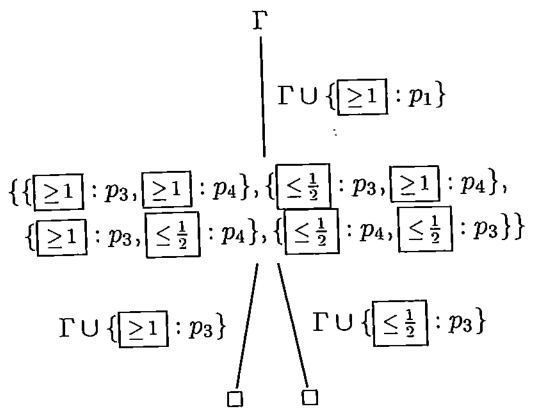

Γ
$\Gamma \cup \{\boxed{\geq 1} : p_1\}$
$\{\{\boxed{\geq 1} : p_3, \boxed{\geq 1} : p_4\}, \{\boxed{\leq \frac{1}{2}} : p_3, \boxed{\geq 1} : p_4\},$
$\{\boxed{\geq 1} : p_3, \boxed{\leq \frac{1}{2}} : p_4\}, \{\boxed{\leq \frac{1}{2}} : p_4, \boxed{\leq \frac{1}{2}} : p_3\}\}$
$\Gamma \cup \{\boxed{\geq 1} : p_3\}$
$\Gamma \cup \{\boxed{\leq \frac{1}{2}} : p_3\}$
$\Box$
$\Box$

Figure 4.10: A regular-2sat proof tree for the formula Γ from Example 4.2

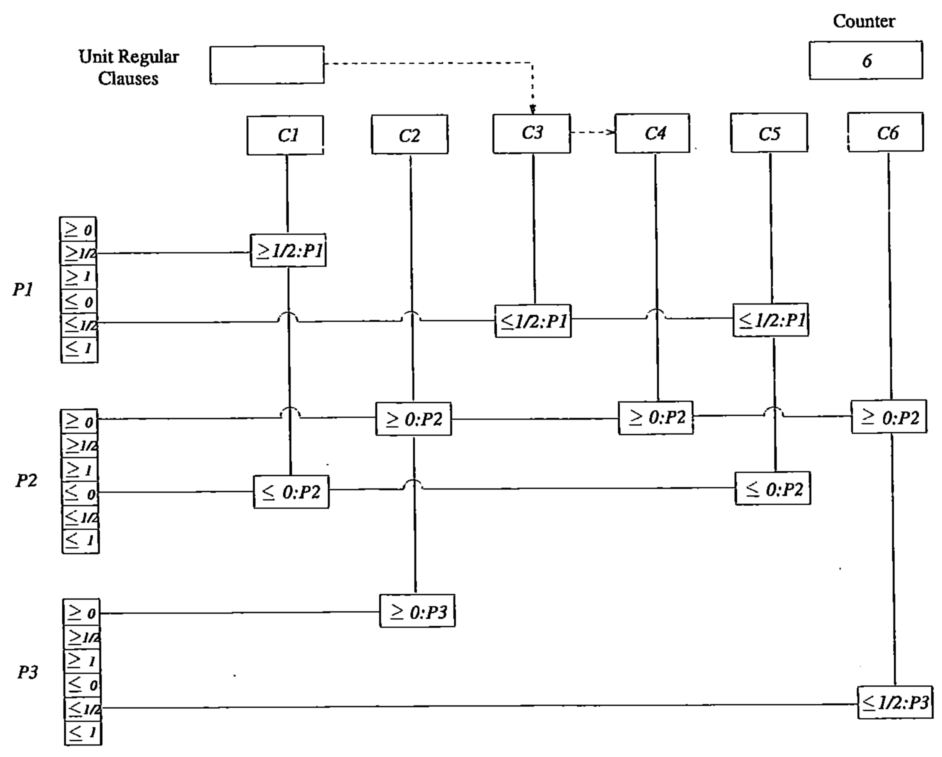


Figure 4.11: Data structures

Example 4.3 *Figure 4.11 shows the data structure defined above for the following regular 2-CNF formula:*

$$\{\{\boxed{\geq \tfrac{1}{2}} : p_1, \boxed{\leq 0} : p_2\}, \{\boxed{\geq 0} : p_2, \boxed{\geq 0} : p_3\}, \{\boxed{\leq \tfrac{1}{2}} : p_1\},$$
$$\{\boxed{\geq 0} : p_2\}, \{\boxed{\leq \tfrac{1}{2}} : p_1, \boxed{\leq 0} : p_2\}, \{\boxed{\geq 0} : p_2, \boxed{\leq \tfrac{1}{2}} : p_3\}\}$$

Theorem 4.9 complexity *Let Γ be a regular 2-CNF formula and let m be the number of regular clauses in Γ. The worst-case time complexity of regular-2sat for Γ is in $O(m \cdot |\Gamma|)$ when the truth value set N is fixed and finite.*

Proof: The complexity of regular-2sat follows from the following facts:

- Function regular-2sat generates a proof tree such that the number of levels is bounded by m, the number of clauses in Γ.
- Function regular-2sat generates at most two new nodes per level.
- Function regular-2sat expands at most one node per level.

- Checking whether $\Gamma = \emptyset$ needs constant time because a global counter of regular clauses is maintained.
- The test $\Box \in \Gamma$ needs constant time because we can check whether $\Box \in \Gamma$ when we eliminate a literal from a regular unit clause.
- The worst-case time complexity of regular-unit-resolve is in $\mathcal{O}(|\Gamma|)$: detection of regular unit clauses can be done in constant time, since we maintain a list of regular unit clauses. Given a regular literal L, direct access to all the clauses containing this literal is provided. On the one hand, we have to remove all the clauses containing a regular literal L' such that $L \subseteq L'$. These clauses can be found immediately looking up all the literal occurrence lists of regular literals L''s ($L \subseteq L'$) and each clause can be eliminated in constant time, since each clause contains at most two literals. On the other hand, we have to shorten all the clauses containing a a regular literal L'' such that $L'' \subseteq \overline{L}$. These clauses can be found immediately looking up all the literal occurrence lists of regular literals L'''s ($L'' \subseteq \overline{L}$) and a literal can be removed in constant time.

■

When the truth value set N is infinite, then the only difference is that the worst-case time complexity of regular-unit-resolve is almost linear instead of linear. The trick for obtaining a low complexity is the same as that used in Section 3.6.3 for arbitrary infinitely-valued regular formulas: we now have to maintain, for each propositional atom p, a list of all the occurrences of regular literals $S:p$ with negative polarity and a list of all the occurrences of regular literals $S':p$ with positive polarity. Then, these lists must be sorted, during the initialization phase, in such a way that they are scanned only once during the execution of the procedure.

Another satisfiability checking procedure for regular 2-CNF formulas

We now present another proof procedure for solving the regular 2-SAT problem which can be viewed as a variant of the previous procedure. It incorporates a refinement of the branching rule that we have defined as regular shortest positive clause branching in Section 3.6.1.

Proposition 4.4 regular shortest positive clause branching *Let Γ be a regular 2-CNF formula and let $\{L_1, L_2\}$ be a regular clause of Γ. Then, Γ is satisfiable iff $\Gamma \cup \{L_1\} \cup \{\overline{L_2}\}$ is satisfiable or $\Gamma \cup \{L_2\}$ is satisfiable.*

Proof: Suppose that Γ is satisfiable. For any interpretation I that satisfies Γ, it holds that I satisfies $\{L_1, L_2\}$. If I satisfies L_1, then we distinguish two cases: (i) I does not satisfy L_2; in this case, I satisfies $\Gamma \cup \{L_1\} \cup \{\overline{L_2}\}$; and (ii) I satisfies L_2; in this case, I satisfies $\Gamma \cup \{L_2\}$. If I satisfies L_2, then I satisfies $\Gamma \cup \{L_2\}$.

Suppose that $\Gamma \cup \{L_1\} \cup \{\overline{L_2}\}$ is satisfiable or $\Gamma \cup \{L_2\}$ is satisfiable. Since Γ is a subset of both formulas, it follows that Γ is satisfiable. ■

```
function regular-2sat' ($\Gamma$: set of clause) : boolean
var $L_1, L_2$: literal
var $\Gamma'$: set of clause
begin
  if $\Gamma = \emptyset$ or $\Gamma$ contains no positive clauses then return(true);
  if $\Box \in \Gamma$ then return(false);
  Let $\{L_1, L_2\}$ be a positive clause of $\Gamma$;
  $\Gamma'$ := regular-unit-resolve($\Gamma \cup \{L_1\} \cup \{\overline{L_2}\}$);
  if $\Box \in \Gamma'$ then $\Gamma'$ := regular-unit-resolve($\Gamma \cup \{L_2\}$);
  regular-2sat'($\Gamma'$)
end
```

Figure 4.12: A proof procedure for regular 2-CNF formulas with positive clause branching

Observe that clause branching with binary clauses can be viewed as a refinement of the β-rule with local lemma of semantic tableaux: given a β-formula ($\beta = \beta_1 \operatorname{op} \beta_2$), two expansions are created; the first one contains β_1 and the second one contains β_2 and $\neg\beta_1$.

Figure 4.12 shows the pseudo-code of a satisfiability checking procedure for regular 2-CNF formulas that incorporates the results of Proposition 4.3 and the branching rule defined in Proposition 4.4. Function regular-2sat$'$ returns true if the input regular 2-CNF formula Γ is satisfiable and it returns false if Γ is unsatisfiable. Function regular-unit-resolve applies repeatedly the regular one-literal rule.

Theorem 4.10 *Let Γ be a regular 2-CNF formula. If Γ is satisfiable, then function regular-2sat$'$ terminates and returns true. If Γ is unsatisfiable, then function regular-2sat$'$ terminates and returns false.*

Proof: The same arguments used in Theorem 4.8 for proving completeness are valid for function regular-2sat$'$ because the new branching rule also preserves satisfiability.

■

Theorem 4.11 complexity *Let Γ a regular 2-CNF formula and let m' be the number of regular positive clauses in Γ. The worst-case time complexity of regular-2sat$'$ for Γ is in $O(m' \cdot |\Gamma|)$ when the truth value set N is fixed and finite.*

Proof: The complexity of regular-2sat$'$ follows from the following facts:

- Function regular-2sat$'$ generates a proof tree such that the number of levels is bounded by m', the number of positive clauses in Γ.

- Function regular-2sat′ generates at most two children per level.
- Function regular-2sat′ expands at most one node per level.
- The worst-case time complexity of regular-unit-resolve is in $\mathcal{O}(|\Gamma|)$, using similar data structures to those defined for function regular-2sat.

■

As explained above, if the truth value set N is infinite, then the only difference is that the worst-case time complexity of regular-unit-resolve is almost linear instead of linear.

4.3.3 The 2-SAT problem in monosigned CNF formulas

We have shown that the 2-SAT problem in signed CNF formulas is NP-complete, whereas the 2-SAT problem in regular CNF formulas is polynomially solvable. In this section we investigate the 2-SAT problem in monosigned CNF formulas and describe a satisfiability checking procedure with a quadratic-time complexity in the worst case.

Baaz and Fermüller (1995) proved that the below resolution rule is refutation complete for monosigned CNF formulas.

$$\frac{\{v_1\}:p \vee D_1 \qquad \{v_2\}:p \vee D_2}{D_1 \vee D_2} \quad \text{if } v_1 \neq v_2;\ v_1, v_2 \in N$$

Observe that (i) resolvents have at most two literals when this rule is applied to monosigned binary clauses and (ii) the number of possible resolvents for a given monosigned 2-CNF formula Γ is polynomial in the number of distinct literals occurring in Γ. Therefore, the monosigned 2-SAT problem is polynomially solvable. This fact was not noticed in (Baaz and Fermüller, 1995).

Next, we design a DP-style procedure for monosigned 2-CNF formulas. To this end, we first define a branching rule and a one-literal rule for monosigned CNF formulas.

Proposition 4.5 monosigned clause branching rule *Let Γ be a monosigned 2-CNF formula and let $\{L_1, L_2\}$ be a monosigned binary clause of Γ. Then, Γ is satisfiable iff $\Gamma \cup \{L_1\}$ is satisfiable or $\Gamma \cup \{L_2\}$ is satisfiable.*

Proof: Assume that Γ is satisfiable. For any model I of Γ, I satisfies $\{L_1, L_2\}$. If I satisfies L_1, then I satisfies $\Gamma \cup \{L_1\}$. If I satisfies L_2, then I satisfies $\Gamma \cup \{L_2\}$.

Assume that $\Gamma \cup \{L_1\}$ is satisfiable or $\Gamma \cup \{L_2\}$ is satisfiable. If $\Gamma \cup \{L_1\}$ is satisfiable, then Γ is satisfiable. If $\Gamma \cup \{L_2\}$ is satisfiable, then Γ is satisfiable. ■

This is not the only possible branching rule we can define. For instance, the branching rule defined in Proposition 3.1 is another alternative.

Proposition 4.6 monosigned one-literal rule *Let Γ be a monosigned 2-CNF formula that contains a monosigned unit clause $\{\{x\}:p\}$. Let Γ' be obtained from Γ by first removing all clauses that contain an occurrence of the literal $\{x\}:p$, and second by removing all occurrences of literals of the form $\{y\}:p$ such that $x \neq y$ from the remaining clauses. Then, Γ is satisfiable iff Γ' is satisfiable.*

Proof: Suppose that Γ is satisfiable. Since $\{\{x\}:p\} \in \Gamma$, every model of Γ must assign to p the value x. Let I be an arbitrary model of Γ. On the one hand, I satisfies any subset of Γ. On the other hand, if we remove the literals $\{y\}:p$ such that $x \neq y$, then I also satisfies the formula so obtained because $I(p) = x$. Therefore, Γ' is satisfiable.

Suppose that Γ' is satisfiable. Let I be a model of Γ'. To obtain Γ from Γ' we add literals to clauses of Γ' and new clauses that contain the literal $\{x\}:p$. Since Γ' contains no occurrences of p, the interpretation I' such that $I'(p) = x$ and is identical to I for the remaining propositional atoms is a model of Γ. Therefore, Γ is satisfiable. ∎

The results obtained in Proposition 4.3 for regular 2-CNF formulas can be moved to monosigned 2-CNF formulas as the following proposition states:

Proposition 4.7 *Let Γ be a monosigned 2-CNF formula, let $\{x\}:p$ be a monosigned literal occurring in Γ and let Γ' be the formula obtained after applying monosigned-unit-resolve to $\Gamma \cup \{\{x\}:p\}$. Then, Γ' is satisfiable iff Γ is satisfiable and $\Box \notin \Gamma'$.*

Proof: Suppose that Γ' is satisfiable. As monosigned-unit-resolve applies a finite number of times the monosigned one-literal rule and this rule preserves satisfiability (cf. Proposition 4.6), it follows that Γ' is satisfiable iff $\Gamma \cup \{\{x\}:p\}$ is satisfiable. Since $\Gamma \cup \{\{x\}:p\}$ is satisfiable and $\Gamma \subseteq \Gamma \cup \{\{x\}:p\}$, it follows that Γ is satisfiable. As Γ' is satisfiable, it is clear that $\Box \notin \Gamma'$.

Suppose that Γ is satisfiable and $\Box \notin \Gamma'$. As the monosigned clauses of Γ are binary, monosigned-unit-resolve derives a monosigned unit clause when a literal from a clause of Γ is deleted. As $\Box \notin \Gamma'$, monosigned-unit-resolve cannot eliminate literals from monosigned unit clauses; it removes all such clauses. Therefore, the set of monosigned clauses returned by monosigned-unit-resolve($\Gamma \cup \{\{x\}:p\}$) is a subset of Γ. Since Γ is satisfiable and Γ' is a subset of Γ, it follows that Γ' is satisfiable. ∎

The results of the previous propositions give raise to the satisfiability checking procedure shown in Figure 4.13. Function monosigned-2sat returns true if the input monosigned 2-CNF formula Γ is satisfiable, and it returns false if Γ is unsatisfiable. Function monosigned-unit-resolve applies repeatedly the monosigned one-literal rule.

Theorem 4.12 *Let Γ be a monosigned 2-CNF formula. If Γ is satisfiable, then function monosigned-2sat terminates and returns true. If Γ is unsatisfiable, then function monosigned-2sat terminates and returns false.*

```
function monosigned-2sat ($\Gamma$: set of clause) : boolean
var $L_1, L_2$: literal
var $\Gamma'$: set of clause
begin
   if $\Gamma = \emptyset$ then return(true);
   if $\Box \in \Gamma$ then return(false);
   Let $\{L_1, L_2\}$ be a clause of $\Gamma$;
   $\Gamma'$ := monosigned-unit-resolve($\Gamma \cup \{L_1\}$);
   if $\Box \in \Gamma'$ then $\Gamma'$ := monosigned-unit-resolve($\Gamma \cup \{L_2\}$);
   monosigned-2sat($\Gamma'$)
end
```

```
function monosigned-unit-resolve ($\Gamma$: set of clause) : set of clause
var p: atom
var x,y: truth_value
var C: clause
begin
   while $\exists\{\{x\}:p\} \in \Gamma$ and $\Box \notin \Gamma$ do
      $\Gamma := \{C \mid \nexists\{x\}:p \in C \in \Gamma\}$;
      $\Gamma := \{C - \{\{y\}:p \mid \{y\}:p \in C \text{ and } x \neq y\} \mid C \in \Gamma\}$
   endwhile;
   return($\Gamma$)
end
```

Figure 4.13: A proof procedure for monosigned 2-CNF formulas

Proof: First, we show that if function monosigned-2sat terminates then either some branch of the proof tree created by monosigned-2sat contains the empty formula (i.e. returns true) or every branch of the proof tree contains the empty clause (i.e. returns false). Suppose we have a proof tree with no branch containing the empty formula and some leaf node of a branch with a monosigned formula Γ' which does not contain the empty clause; we show that monosigned-2sat does not terminate. If Γ' contains monosigned unit clauses, then monosigned-2sat calls monosigned-unit-resolve and applies the monosigned one-literal rule; otherwise, monosigned-2sat selects a monosigned clause $\{L_1, L_2\}$ of Γ' and applies the monosigned clause branching rule. Either way, monosigned-2sat does not terminate.

Next we show that every proof attempt must terminate. Function monosigned-2sat applies the monosigned one-literal rule and the monosigned clause branching rule. On the one hand, the monosigned one-literal rule decreases the number of distinct propositional atoms occurring in the formula. On the other hand, after applying the monosigned clause branching rule we obtain two new formulas to which monosigned-2sat can apply the monosigned one-literal rule and so decrease the number of distinct propositional atoms occurring in them. Since we began with a finite number of distinct propositional atoms, these rules can only be applied a finite number of times.

By Proposition 4.5 and Proposition 4.6 we have that Γ is satisfiable iff monosigned-unit-resolve($\Gamma \cup \{L_1\}$) is satisfiable or monosigned-unit-resolve($\Gamma \cup \{L_2\}$) is satisfiable. If the empty clause is contained in monosigned-unit-resolve($\Gamma \cup \{L_1\}$), then Γ is satisfiable iff monosigned-unit-resolve($\Gamma \cup \{L_2\}$) is satisfiable. Otherwise, by Proposition 4.7 we have that Γ is satisfiable iff monosigned-unit-resolve($\Gamma \cup \{L_1\}$) is satisfiable. Then, it is clear that when monosigned-2sat returns false (i.e all the branches of the proof tree contain the empty clause), the input formula Γ is unsatisfiable, and when monosigned-2sat returns true (i.e. there is a branch with the empty formula), Γ is satisfiable.

■

Example 4.4 *Let $N = \{0, \frac{1}{2}, 1\}$ and let Γ be the following monosigned 2-CNF formula:*

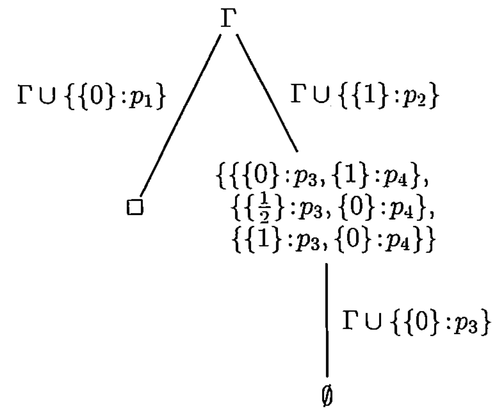
$$\begin{aligned}\Gamma = {} & \{\{\{0\}:p_1, \{1\}:p_2\}, \{\{0\}:p_3, \{1\}:p_4\}, \{\{\tfrac{1}{2}\}:p_1, \{1\}:p_2\}, \\ & \{\{\tfrac{1}{2}\}:p_3, \{0\}:p_4\}, \{\{1\}:p_3, \{0\}:p_4\}, \{\{\tfrac{1}{2}\}:p_1, \{\tfrac{1}{2}\}:p_2\}\}\end{aligned}$$

Figure 4.14 shows the proof tree created by monosigned-2sat when the input formula is Γ.

Data structures and complexity

The data structures we propose to get time efficient operations in function monosigned-2sat are the following ones:

- Each monosigned clause has a head and a doubly-linked list of its monosigned literals.



Γ
$\Gamma \cup \{\{0\}:p_1\}$
$\Gamma \cup \{\{1\}:p_2\}$
$\Box$
$\{\{\{0\}:p_3, \{1\}:p_4\}, \{\{\frac{1}{2}\}:p_3, \{0\}:p_4\}, \{\{1\}:p_3, \{0\}:p_4\}\}$
$\Gamma \cup \{\{0\}:p_3\}$
$\emptyset$

Figure 4.14: A monosigned-2sat proof tree for the formula Γ from Example 4.4

- We maintain a global counter of monosigned clauses and a list of monosigned unit clauses.
- For each possible combination formed by a propositional atom p and a sign $\{x\}$ (i.e., for all the possible monosigned literals) we maintain a list of the monosigned clauses containing an occurrence of the monosigned literal $\{x\}:p$. These lists are called literal occurrence lists.

Example 4.5 *Figure 4.15 shows the data structure defined above for the following monosigned formula:*

$$\{\{\{0\}:p_1\}, \{\{1\}:p_1, \{\tfrac{1}{2}\}:p_3\}, \{\{\tfrac{1}{2}\}:p_2\},$$
$$\{\{0\}:p_1, \{1\}:p_2\}, \{\{\tfrac{1}{2}\}:p_2, \{\tfrac{1}{2}\}:p_3\}\}$$

Theorem 4.13 complexity *Let Γ be a monosigned 2-CNF formula and let m be the number of monosigned clauses in Γ. The worst-case time complexity of monosigned-2sat for Γ is in $\mathcal{O}(m \cdot |\Gamma|)$.*

Proof: The complexity of monosigned-2sat follows from the following facts:

- Function monosigned-2sat generates a proof tree such that the number of levels is bounded by m, the number of clauses in Γ.
- Function monosigned-2sat generates at most two new nodes per level.
- Function monosigned-2sat expands at most one node per level.
- Checking whether $\Gamma = \emptyset$ needs constant time because a global counter of clauses is maintained.
- The test $\Box \in \Gamma$ needs constant time because we can check whether $\Box \in \Gamma$ when we eliminate a literal from a monosigned unit clause.

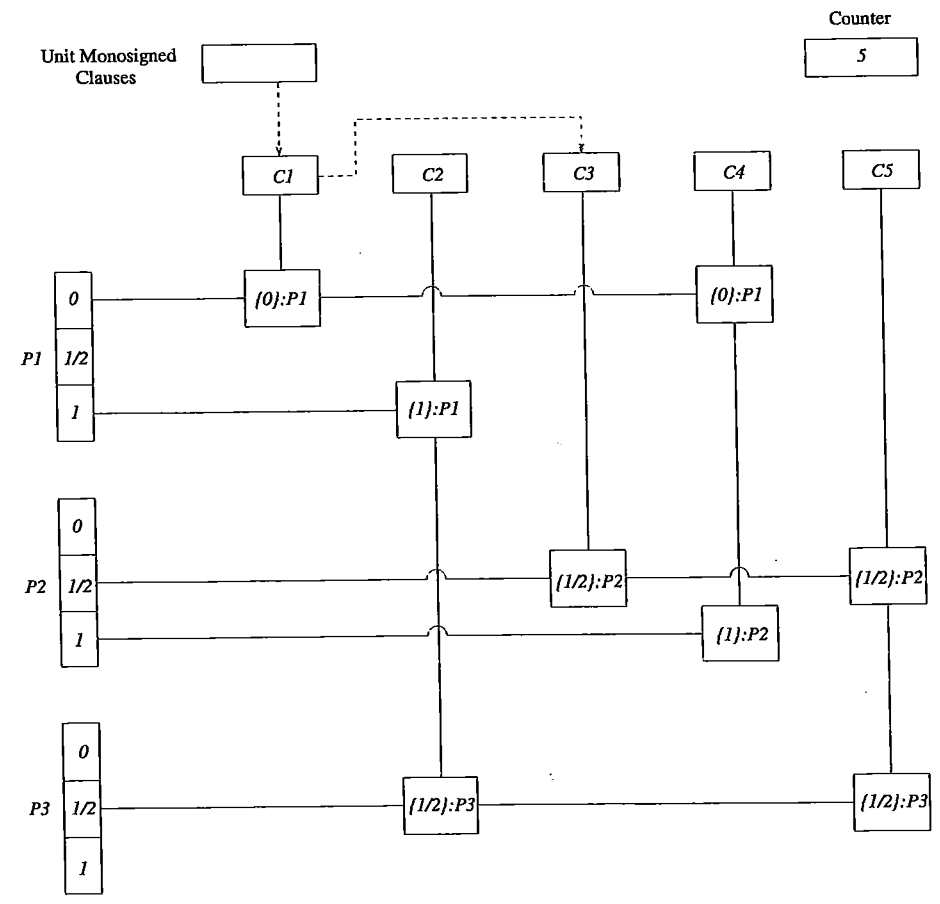


Figure 4.15: Data structures

- The worst-case time complexity of monosigned-unit-resolve is in $\mathcal{O}(|\Gamma|)$: detection of unit clauses can be done in constant time, since we maintain a list of monosigned unit clauses. Direct access to all the clauses containing a literal $\{x\}:p$ and all the clauses containing a literal $\{y\}:p$ $(x \neq y)$ is provided. On the one hand, we have to remove all the clauses containing $\{x\}:p$. These clauses can be found immediately looking up the literal occurrence list of $\{x\}:p$ and each clause can be eliminated in constant time, since they have at most two literals. On the other hand, we have to shorten all the clauses containing a literal $\{y\}:p$ $(x \neq y)$. These clauses can be found immediately looking up the literal occurrence list of $\{y\}:p$ and a literal can be removed in constant time.

■

Chapter 5

An Interpreter for Multiple-valued Propositional Logic Programs

Abstract: In recent years, new logic programming languages have been developed for reasoning and representing knowledge in situations where there is vague, incomplete or imprecise information. Some of such languages rely on multiple-valued logics and demand fast deduction procedures. In this chapter, we describe an efficient interpreter of logic programs that can deal with a wide family of infinitely-valued propositional logics. To this end, we first define a logic programming language, a complete calculus that contains a modus ponens-style inference rule, suitable data structures for representing logic programs and strategies for pruning the proof search space. Then, we define a negation as failure rule and a cut operator adapted to our multiple-valued setting. Finally, we describe an interpreter whose worst-case time complexity is linear in the total number of occurrences of propositional atoms in the input logic program.

5.1 Introduction

Logic programming languages have been applied to a wide range of areas such as Artificial Intelligence and Deductive Databases. Among these languages, Prolog is the most representative, but it is not powerful enough for representing knowledge and reasoning in situations where there is vague, incomplete or imprecise information. To overcome this problem, new logic programming languages have been developed. They are based on a variety of non-standard logics such as multiple-valued logics (Ishizuka and Kanai, 1985; Mukaidono et al., 1989; Martin et al., 1987; Li and Liu, 1990; Alsinet and Manyà, 1996), possibilistic logic (Dubois et al., 1990), evidential logic (Baldwin et al., 1995; Baldwin, 1987) and fuzzy operator logic (Weigert et al., 1993). Depending on the underlying logic some systems are more suitable for dealing with vague information while others are more appropriate for processing incomplete or imprecise information. In (Escalada-Imaz et al., 1996c), there is a uniform exposition and discussion of the capabilities of these systems.

In this chapter we turn our attention to the family of infinitely-valued propositional logics defined in Section 2.6. More specifically, we focus on the logic programming language formed by their corresponding infinitely-valued facts and rules. The èlection of such facts and rules is motivated by the following points:

- They have been previously used as a powerful knowledge representation language in expert systems for diagnosis applications (Sierra, 1989; Puyol-Gruart, 1994).
- Their underlying logics have been carefully investigated by Pavelka (1979), Valverde and Trillas (1985) and Godo (1990), among others. In particular, a sound modus ponens-style inference rule has been defined for this family of logics.
- The design of efficient deduction algorithms for these facts and rules has not been investigated so far despite their significance in real-world applications.

Our first objective in this chapter is to design an efficient proof procedure for these infinitely-valued facts and rules. Given an infinitely-valued logic program P (i.e. a set of facts and rules) and a propositional atom q (goal), such a proof procedure should determine the greatest α such that $P \models (q; \alpha)$; i.e. the maximum degree of logical consequence of q in P. To this end, we first define a complete calculus that contains the modus ponens-style inference rule described in (Godo, 1990), and then suitable data structures for representing logic programs and strategies for pruning the proof search space. Next, we design a proof procedure whose worst-case time complexity is linear in the total number of occurrences of propositional atoms in the input logic program. This proof procedure could be incorporated into the inference engine of knowledge-based systems such as Milord II (Puyol-Gruart, 1994).

Our second objective is to design an efficient interpreter for infinitely-valued propositional logic programs. To this end, we first define a negation as failure rule and a cut operator adapted to our multiple-valued setting. Then, we modify the proof procedure we have designed for dealing with this extended programming language, and this way we obtain an interpreter of infinitely-valued propositional logic programs. Finally, we prove that this interpreter of logic programs also reaches a linear-time complexity in the worst case.

This chapter is organized as follows. In Section 5.2 we define the syntax, semantics and logical inference of the infinitely-valued logic programs the interpreter can deal with. In Section 5.3 we present a detailed description of the interpreter. In Section 5.4 and Section 5.5 we define a negation as failure rule and a cut operator adapted to our multiple-valued context, respectively. In Section 5.6 we outline the implementation of a programming environment based on the interpreter.

5.2 The logic of programs

5.2.1 Syntax

Definition 5.1 logic program *An infinitely-valued fact is an ordered pair* $(p; \alpha)$*, where* p *is a propositional atom and* $\alpha \in [0,1]$ *is a truth value. An infinitely-valued rule is an ordered pair* $(q \leftarrow p_1, \ldots, p_k; \alpha')$*, where* $q \leftarrow p_1, \ldots, p_k$ *is a propositional rule and* $\alpha' \in [0,1]$ *is a truth value. An infinitely-valued logic program is a finite set of infinitely-valued facts and rules.*

In the following when we say fact (rule, logic program) we mean infinitely-valued fact (rule, logic program). From a syntactic point of view, the only difference between classical and infinitely-valued logic programs is that facts and rules are followed by a positive regular sign.

Example 5.1 *We show below an example of logic program. In Section 5.3.1 we describe how the interpreter determines the maximum degree of logical consequence of the goal* q= *"renal failure because hypovolemia secondary to bleeding" in this logic program.*

R1: (renal failure ← high serum creatinin; 1)
R2: (renal failure because hypovolemia ←
  hypovolemia, arterial hypotension, oliguria; 1)
R3: (renal failure because hypovolemia secondary to bleeding ←
  renal failure, hypovolemia, drainage bleeding > 2 ml/kg/hour; 1)
R4: (renal failure because hypovolemia secondary to bleeding ←
  drainage bleeding > 2 ml/kg/hour, tachycardia,
  renal failure because hypovolemia; 1)
R5: (hypovolemia ← tachycardia, low CVP; 0.6)
R6: (hypovolemia ← weak arterial pulse, delayed capillary refill, low CVP; 0.8)
F1: (low CVP; 0.8)
F2: (oliguria; 1)
F3: (tachycardia; 0.5)
F4: (high serum creatinin; 0.9)
F5: (weak arterial pulse ; 0.8)
F6: (arterial hypotension; 0.9)
F7: (delayed capillary refill; 1)
F8: (drainage bleeding > 2 ml/kg/hour; 1)

5.2.2 Semantics

Given a classical logic program P and a propositional atom q, called goal, the objective of a classical propositional interpreter is to determine whether all the models of P are also models of q. If so, we say that q is a logical consequence of P. Given an infinitely-valued logic program P' and a goal q', each model of P' may assign to q' a different truth value. This fact leads us to define the notion of maximum degree of logical consequence, i.e. the greatest α such that $P' \models (q', \alpha)$. The objective of our interpreter will be to find the maximum degree of logical consequence of a goal in a program.

Definition 5.2 interpretation *Let T be a triangular norm and let I_T be the implication defined by residuation with respect to T. An interpretation I for a logic program P is a mapping that assigns to every propositional atom occurring in P a truth value from the interval $[0,1]$. An interpretation I is extended to the first components of the rules of P as follows:*

- $I(q \leftarrow p) = I_T(I(p), I(q))$;
- $I(q \leftarrow p_1, \dots, p_n) = I_T(T(I(p_1), \dots, I(p_n)), I(q))$.

Observe that the the truth value that an interpretation assigns to a rule can be computed once the interpretation of the propositional atoms and the functions I_T and T are provided.

Definition 5.3 satisfiability *Let $(S;\alpha)$ be either a fact or a rule. An interpretation I satisfies $(S;\alpha)$ iff $I(S) \geq \alpha$. A interpretation I satisfies a logic program P, denoted by $I \models P$, iff I satisfies all the facts and rules of P. We say then that I is a model of P.*

Observe that an interpretation I not only satisfies a fact $(p;\alpha)$ (a rule $(q \leftarrow p_1, \dots, p_k;\alpha)$) when I assigns to p (to $q \leftarrow p_1, \dots, p_k$) the truth value α. The interpretation I satisfies the fact (rule) when the value assigned by I is any truth value from the interval $[\alpha, 1]$. The main reason for considering such intervals is to guarantee that there exists an interpretation that satisfies the facts and rules of a logic program. The interpretation that assigns to every propositional atom the value 1 is a model of any logic program.

From the previous definitions one can realize that the logic of the infinitely-valued logic programs we have defined is a sub-logic of the logics defined in Section 2.6.

Definition 5.4 logical consequence *A fact $(p;\alpha)$ is a logical consequence of a logic program P, denoted by $P \models (p;\alpha)$, iff all the interpretations that satisfy P also satisfy $(p;\alpha)$.*

Definition 5.5 maximum degree of logical consequence *The maximum degree of logical consequence of a goal q in a logic program P is the greatest truth value α such that $P \models (q;\alpha)$.*

The following proposition states that the maximum degree of logical consequence of q in P (i.e. $Sup\{\alpha \mid P \models (q;\alpha)\}$) is the least truth value that the models of P assign to q (i.e. $Inf\{I(q) \mid I \models P\}$).

Proposition 5.1 *Let P be a program, let q be a goal and let $\alpha \in [0,1]$ be a truth value. Then,*

$$Sup\{\alpha \,|\, P \models (q;\alpha)\} = Inf\{I(q) \,|\, I \models P\}.$$

Proof: We define $\alpha_1 = Sup\{\alpha \,|\, P \models (q;\alpha)\}$ and $\alpha_2 = Inf\{I(q) \,|\, I \models P\}$.

1. $\alpha_2 \geq \alpha_1$:

   As $\alpha_1 = Sup\{\alpha \,|\, P \models (q;\alpha)\}$ we have that $P \models (q;\alpha)$, for all $\alpha < \alpha_1$. Then, for every model I of P we have that $I(q) \geq \alpha$ for all $\alpha < \alpha_1$. Thus, $\alpha_2 = Inf\{I(q) \,|\, I \models P\} \geq \alpha$, for all $\alpha < \alpha_1$, hence, $\alpha_2 \geq \alpha_1$.

2. $\alpha_2 \leq \alpha_1$:

   As $\alpha_2 = Inf\{I(q) \,|\, I \models P\}$ we have that $I(q) \geq \alpha_2$ for all I such that $I \models P$, that is, $P \models (q,\alpha_2)$, and thus $\alpha_2 \leq Sup\{\alpha \,|\, P \models (q;\alpha)\} = \alpha_1$.

■

5.2.3 Logical inference

In this section we define a calculus and then prove that it is sound and complete for determining the maximum degree of logical consequence of a goal in a logic program. The calculus has the following inference rule and axiom:

Multiple-valued modus ponens:

$$\frac{\begin{array}{c}(p_1;\alpha_1)\\ \cdots\\ (p_n;\alpha_n)\\ (q \leftarrow p_1,\ldots,p_n;\alpha_R)\end{array}}{(q;T(\alpha_R,\alpha_1,\ldots,\alpha_n))}$$

Axiom:

$$\frac{}{(p;0)}$$

Godo (1990) proved that multiple-valued modus ponens is sound.

Definition 5.6 degree of deduction *A goal q is deduced with a degree of deduction α from a logic program P, denoted by $P \vdash (q;\alpha)$, iff there exists a finite sequence of facts and rules $C_1,\ldots,C_m$ such that $C_m = (q;\alpha)$ and, for each $k \in \{1,\ldots,m\}$, it holds that C_k is a fact or rule of P, C_k is an instance of the axiom or C_k is obtained by applying the multiple-valued modus ponens to previous facts and rules in the sequence.*

Next, we define the syntactic counterpart of maximum degree of logical consequence. We call it maximum degree of deduction.

Definition 5.7 maximum degree of deduction *The maximum degree of deduction of a goal q in a logic program P is the greatest α such that $P \vdash (q;\alpha)$.*

Restriction: In the logic programs considered in the present chapter we will assume that we do not have the *recursivity problem*; i.e. it cannot happen that the value of a propositional atom p depends on itself. For instance, a logic program cannot contain two rules of the form $(q \leftarrow p; \alpha_1)$ and $(p \leftarrow q; \alpha_2)$.

Justification: The deduction of a goal q from a logic program P can be graphically represented by an AND/OR graph which is defined as follows:

- for each propositional atom p occurring in P, there is a node labelled p;
- for each rule $(q \leftarrow p_1, \ldots, p_k; \alpha)$ occurring in P, there is a connector $(q, (p_1, .., p_k))$ formed by k arcs (q, p_i), $1 \leq i \leq k$.

An AND/OR graph representing the deduction of a goal from a logic program without indirect recursivity has no cycles. It is known that algorithms for classical propositional logic dealing with cycles are fairly difficult (Dowling and Gallier, 1984; Ghallab and Escalada-Imaz, 1991). Moreover, cycles are not included in any solution and complicate the search in the propositional case.

Figure 5.1 shows the AND/OR graph corresponding to the deduction of the goal "renal failure because hypovolemia secondary to bleeding" from the program from Example 5.1.

Proposition 5.2 *(Godo, 1990) Let I be an interpretation and let P be a logic program of the form*

$$\begin{array}{c}(p_1; \alpha_1)\\ \cdots\cdots\\ (p_n; \alpha_n)\\ (q \leftarrow p_1, \ldots, p_n; \alpha_R)\end{array}$$

Then, $Inf\{I(q) \,|\, I \models P\} = T(\alpha_R, \alpha_1, \ldots, \alpha_n)$.

Theorem 5.1 soundness and completeness *Let P be a logic program and let q be a goal. If α is the maximum degree of logical consequence of q in P and β is the maximum degree of deduction of q in P, then $\alpha = \beta$.*

Proof: We must prove that $Sup\{\alpha \,|\, P \models (q;\alpha)\} = Sup\{\beta \,|\, P \vdash (q;\beta)\}$. We proceed by induction on l, where l is the number of facts and rules of P.

If $l = 1$, then it must be that the program P contains only either one fact, say $(p;\gamma)$, or one rule, say $(p \leftarrow p_1, \ldots, p_n; \gamma)$. We assume that q occurs in P; otherwise, we have $Sup\{\alpha \,|\, P \models (q;\alpha)\} = Sup\{\beta \,|\, P \vdash (q;\beta)\} = 0$. Suppose that P contains only the fact $(q;\gamma)$. Then, it is clear that $Sup\{\alpha \,|\, P \models (q;\alpha)\} = Sup\{\beta \,|\, P \vdash (q;\beta)\} = \gamma$. Suppose that P contains only the rule $(p \leftarrow p_1, \ldots, p_n; \gamma)$. Let I be an interpretation such that $I(p_i) = 0$ $(1 \leq i \leq n)$ and $I(p) = 0$. By definition of R-implication (cf. Section 2.6), $I_T(a,b) = 1$ iff $a \leq b$. Therefore, $I(q \leftarrow p_1, \ldots, p_n) = 1 \geq \gamma$ and $I \models P$. Thus, $Inf\{I(p) \,|\, I \models P\} = 0$ and $Inf\{I(p_i) \,|\, I \models P\} = 0$ $(1 \leq i \leq n)$. By Proposition 5.1, it follows that $Sup\{\alpha \,|\, P \models (p;\alpha)\} = 0$ and $Sup\{\alpha \,|\, P \models (p_i;\alpha)\} = 0$

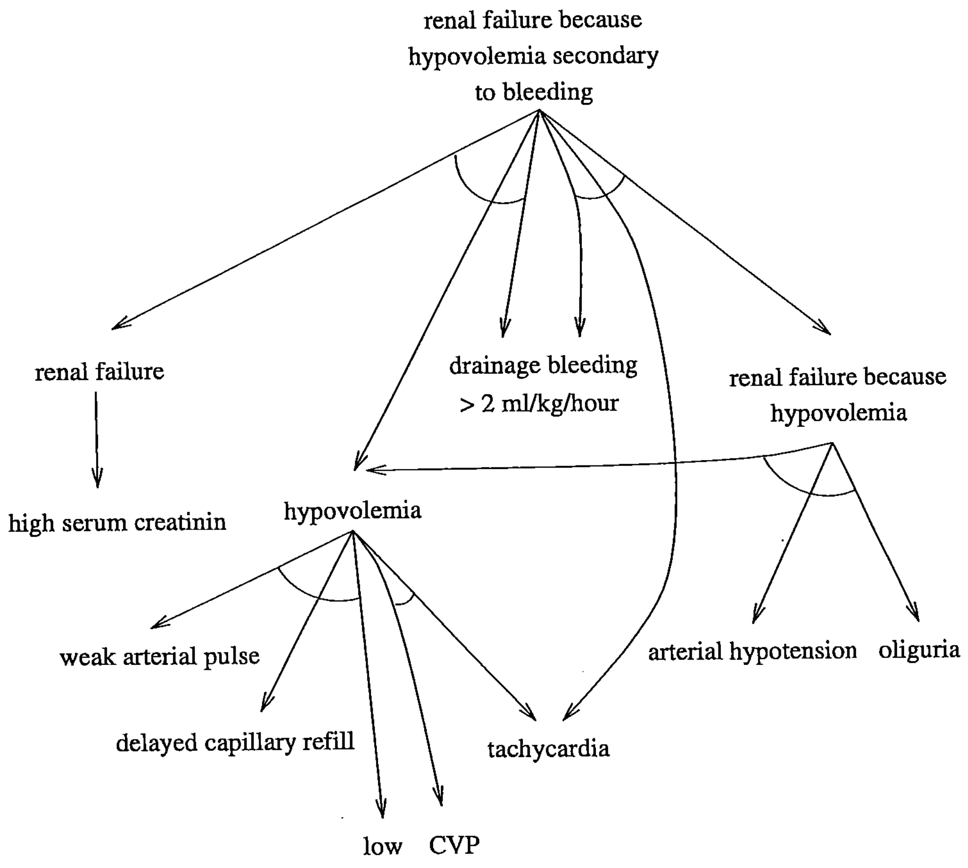


Figure 5.1: AND/OR graph

$(1 \leq i \leq n)$. On the other hand, $Sup\{\beta \,|\, P \vdash (p;\beta)\} = T(0,\ldots,0,\gamma) = 0$ and $Sup\{\beta \,|\, P \vdash (p_i;\beta)\} = 0$ $(1 \leq i \leq n)$. So, if $p = q$ or $p_i = q$, for some i $(1 \leq i \leq n)$, we have that $Sup\{\alpha \,|\, P \models (q;\alpha)\} = Sup\{\beta \,|\, P \vdash (q;\beta)\}$.

Suppose now that for any program P' that contains l facts and rules it holds that $Sup\{\alpha' \,|\, P' \models (q;\alpha')\} = Sup\{\beta' \,|\, P' \vdash (q;\beta')\}$, and suppose that the program P contains $l+1$ facts and rules. We distinguish three cases:

Case 1: q occurs in a fact; i.e. $P = P' \cup \{(q;\gamma)\}$. Then, it is clear that $Sup\{\alpha \,|\, P \models (q;\alpha)\} = \max(\gamma, Sup\{\alpha' \,|\, P' \models (q;\alpha')\}) = Sup\{\alpha \,|\, P \vdash (q;\alpha)\}$.

Case 2: q is the consequent of a rule; i.e. $P = P' \cup \{(q \leftarrow p_1,\ldots,p_n;\gamma)\}$. By the induction hypothesis, for any propositional atom p it holds that $Sup\{\alpha' \,|\, P' \models (p;\alpha')\} = Sup\{\beta' \,|\, P' \vdash (p;\beta')\}$. Let $\alpha_i = Sup\{\alpha' \,|\, P' \models (p_i;\alpha')\}$ $(1 \leq i \leq n)$ and $\alpha_q = Sup\{\alpha' \,|\, P' \models (q;\alpha')\}$. By Proposition 5.2, it holds that $T(\alpha_1,\ldots,\alpha_n,\gamma) = Inf\{I(q) \,|\, I \models (p_1,\alpha_1),\ldots,I \models (p_n,\alpha_n), I \models (q \leftarrow p_1,\ldots,p_n;\gamma)\}$. Therefore, $\max(T(\alpha_1,\ldots,\alpha_n,\gamma),\alpha_q) = Sup\{\alpha \,|\, P \models (q;\alpha)\} = Sup\{\beta \,|\, P \vdash (q;\beta)\}$.

Case 3: q is an antecedent of a rule; i.e. $P = P' \cup \{(p \leftarrow p_1,\ldots,q,\ldots,p_n;\gamma)\}$. In this case, $Sup\{\alpha' \,|\, P' \models (q;\alpha')\} = Sup\{\alpha \,|\, P \models (q;\alpha)\} = Sup\{\beta' \,|\, P' \vdash (q;\beta')\} = Sup\{\beta \,|\, P \vdash (q;\beta)\}$.

■

In (Puyol et al., 1992), there is a similar result of soundness and completeness for the finitely-valued case. Nevertheless, they do not consider the concept of maximum degree of deduction.

5.3 The interpreter

Our primary objective in this section is to define a linear-time procedure for obtaining the maximum degree of deduction of a goal in a program. So, we will define suitable data structures for representing logic programs and techniques for reducing the search space to be explored. In order to facilitate its comprehension, we present first a simple version. Then, we complete the algorithm in a second stage.

The interpreter we will describe applies multiple-valued modus ponens in a reverse way as it is done in classical backward inference systems; i.e. rather than applying the inference rule to $r \leftarrow p,q$ once p and q are deduced, it checks first whether r can be deduced which in turn leads to check whether p and q can be deduced.

5.3.1 A simple version

We have seen that the problem of determining the maximum degree of deduction of a goal q in a logic program P can be represented by an AND/OR graph. The principle of our procedure relies on a depth-first search through the graph for establishing the maximum degree of deduction.

In a brute force search through the AND/OR graph, redundant computations can appear because of examining certain parts of the graph more than once; a node p can have several connectors as successor nodes. To overcome this problem we will use a marking technique for nodes when scanning the graph: the first time an atom node is visited, all the possible deduction paths concluding this atom are explored and the maximum degree of deduction obtained by this exploration will be assigned to the atom node. This way, further visits to an already visited node will retrieve the previous computed maximum degree of deduction, avoiding so redundant expansions of a same node.

The data structures we define for representing the AND/OR graph and getting time efficient operations are the following ones:

Data Structures

For each propositional atom p:

$val(p)$ represents the maximum degree of deduction of p in a given state of the algorithm. Initially, if $(p;\alpha_1),\ldots,(p;\alpha_n)$ are the facts that appear in P containing p, then $val(p) = \max(\alpha_1,\ldots,\alpha_n)$. Otherwise, $val(p) = 0$.

$visit(p) \in \{\text{true}, \text{false}\}$, where $visit(p) = \text{true}$ iff p has already been expanded. Initially, $visit(p) = \text{false}$ for all the atoms.

$rules\text{–}conseq(p) = \{R \mid p \text{ is the consequent of rule } R\}$, the sequence of rules concluding p.

For each rule $R = (q \leftarrow p_1,\ldots,p_n;\alpha_R)$:

$val(R) = \alpha_R$.

$antecedents(R) = \{p_1,\ldots,p_n\}$, the sequence of propositional atoms which are antecedents of rule R.

Figure 5.2 shows the pseudo-code of the first version of the interpreter. It has two functions called or-extension and and-extension. Function or-extension scans all the rules $R : (p \leftarrow p_1,\ldots,p_n;\alpha_R)$ whose consequent is a given atom p. To do it, or-extension calls and-extension that visits recursively, calling or-extension, the successor nodes $p_1,\ldots,p_n$ appearing in the antecedents of each rule R. Given the value α_R of a rule R and after obtaining the maximum degree of deduction α_i of each successor node p_i, and-extension computes a deduction degree α for p using rule R. This computation applies multiple-valued modus ponens; i.e. $\alpha = T(\alpha_R,\alpha_1,\ldots,\alpha_n)$. The scanning of all the rules by or-extension enables to determine the maximum degree of deduction of p among the different deduction paths defined by the different possible applications of the rules. This maximum deduction degree is given back to the corresponding and-extension function performing thus a bottom-up propagation of the maximum degree of deduction from the facts up to the goal.

```
function interpreter (P: program, q: goal): truth_value
var $p$: atom
var R: rule
begin
    for each atom $p$ in P do
        if $P$ contains $(p;\alpha_1),\ldots,(p;\alpha_n)$ then
            val($p$) := max($\alpha_1,\ldots,\alpha_n$)
        else
            val($p$) := 0
        endif;
        visit($p$) := false
    endfor;
    for each rule $R = (p \leftarrow p_1,\ldots,p_n;\alpha)$ in P do
        $val(R)$ := $\alpha$;
        antecedents($R$) := $\{p_1,\ldots,p_n\}$;
        rules-conseq($p$) := rules-conseq($p$) $\cup$ $R$
    endfor;
    return(or-extension($q$))
end
```

```
function or-extension (p: atom): truth_value
var R: rule
var value: truth_value
begin
    if visit($p$) then return(val($p$));
    visit($p$) := true;
    for each rule $R$ in rules-conseq($p$) do
        value := and-extension($R$);
        if value > val($p$) then val($p$) := value
    endfor;
    return(val($p$))
end
```

```
function and-extension (R: rule): truth_value
var p: atom
var value: truth_value
begin
   value := val($R$);
   for each atom $p$ in antecedents($R$) do
      value := $T$(value, or-extension($p$))
   endfor;
   return(value)
end
```

Figure 5.2: A first version of the interpreter

Example 5.2 Let us consider the program P from Example 5.1 and its AND/OR graph (cf. Figure 5.1) for the goal q="renal failure because hypovolemia secondary to bleeding". The interpreter computes the maximum degree of deduction of q in P as follows: it starts by visiting the root node "renal failure because hypovolemia secondary to bleeding" and, following a depth-first strategy, visits "renal failure" and then "high serum creatinin". If we assume that $T(x,y) = x{\cdot}y$, then "renal failure" is established with maximum degree of deduction $0.9 = T(0.9, 1)$. The next arc of the first connector outgoing from the root is traversed and "hypovolemia" is established with maximum degree of deduction $0.51 = T(0.8, 1, 0.8, 0.8)$, corresponding to the value of its first connector, because the second one deduces "hypovolemia" with degree of deduction 0.24. As "drainage bleeding > 2 ml/kg/hour" only appears in the program as a fact with value 1, we have that the first connector outgoing from the root enables to deduce the goal "renal failure because hypovolemia secondary to bleeding" with degree of deduction $0.46 = 1 \cdot 0.9 \cdot 0.51 \cdot 1$. Repeating the same process for the second connector outgoing from the root, the interpreter determines that "renal failure because hypovolemia secondary to bleeding" is deduced with a smaller degree of deduction (0.18). Therefore, the interpreter returns 0.46 as the maximum degree of deduction of q in P.

Theorem 5.2 correctness *Function interpreter(P, q) returns α iff α is the maximum degree of deduction of the goal q in the program P.*

Proof: We will prove by induction on rank(p) that

$$\text{or-extension}(p) \text{ returns } Sup\{\alpha \mid P \vdash (p; \alpha)\},$$

where rank(p) is defined as follows:

$$\text{rank}(p) = \begin{cases} 0 \text{ if there is no rule whose consequent is } p \\ 1 + max\{\text{rank}(p') \mid p' \text{occurs in a rule with consequent } p\}, \textit{otherwise.} \end{cases}$$

Induction base: rank$(p) = 0$. In this case, as there are no rules whose consequent is p we can only deduce the facts $(p; \alpha)$ appearing in P. Thus, or-extension$(p) = Sup\{\alpha \,|\, (p; \alpha) \in P\} = Sup\{\alpha \,|\, P \vdash (p; \alpha)\}$. If there is no fact $(p; \alpha)$ in P, then or-extension(p) returns 0.
Induction step: rank$(p) = k + 1$. Suppose that for each atom p_i such that rank$(p_i) \leq k$ it holds that or-extension(p_i) returns $Sup\{\alpha \,|\, P \vdash (p_i; \alpha)\}$. In this case, as well as deducing the facts $(p; \alpha)$ appearing in P, we can deduce the conclusion of the rules $p \leftarrow p_1, \ldots, p_n$ appearing in P by applying multiple-valued modus ponens. The atoms p_i appearing as antecedents in the rules that conclude p verify that rank$(p_i) \leq k$. By induction hypothesis, we can ensure that the value of p_i, $val(p_i)$, used by the interpreter is $Sup\{\alpha \,|\, P \vdash (p_i; \alpha)\}$. Since T-norms are non-decreasing in the first argument (i.e. $T(a, b) \leq a$), it is clear that or-extension(p) returns $Sup\{\alpha \,|\, P \vdash (p; \alpha)\}$.

■

Theorem 5.3 complexity *Let P be a logic program, let q be a goal and let m be the total number of occurrences of propositional atoms in P. Then, the worst-case time complexity of function interpreter(P, q) is in $\mathcal{O}(m)$.*

Proof: The time complexity of the algorithm follows from the following facts:

1. During the initialization phase of the interpreter are executed three for loops: the first one is bounded by the number of distinct propositional atoms, the second one is bounded by the number of facts and the third one is bounded by the number of rules. Therefore, the initialization phase is clearly in $\mathcal{O}(m)$.

2. The algorithm explores an AND/OR graph using a depth-first strategy. The number of nodes of the graph is bounded by the number of different atoms, the number of or-connectors is bounded by the number of rules and the total number of arcs of the and-connectors is bounded by the total number of atoms.

3. Function or-extension is executed at most once for each different atom p: if p is the goal, it is called once by function interpreter; otherwise, it is called once by and-extension if visit(p) is false. Observe that the first time or-extension(p) is called, visit(p) is marked to avoid subsequent calls.

4. The number of iterations of the for loop of or-extension(p) is bounded by the number of rules that conclude p. In each iteration, and-extension is called for each one of those rules. Taking into account that and-extension needs constant time to process each node, the total computational cost of or-extension and and-extension is in $\mathcal{O}(m)$.

■

5.3.2 An improved version

The previous proof procedure, in order to determine the maximum degree of deduction of a goal in a logic program, explores all the nodes of the AND/OR graph generated. Nevertheless, some portions of the graph could be pruned by virtue of the following fact: given a triangular norm T and truth values $x_1, x_2, x_3, \dots, x_k \in [0,1]$, it holds that

$$T(x_1, x_2) \geq T(x_1, x_2, x_3) \geq \dots \geq T(x_1, x_2, x_3, \dots, x_k).$$

Given a node of the graph containing an atom p with several connectors, the interpreter starts by exploring the first connector of node p, then continues by the second connector and so on. Suppose that the interpreter has obtained a degree of deduction α for the first connector and (we assume that the second connector is $(p, (p_1, .., p_m); \alpha_R)$) has established that $\alpha_1, \dots, \alpha_m$ are the maximum degrees of deduction of $p_1, \dots, p_m$, respectively. Then, the degree of deduction for the second connector is $T(\alpha_R, \alpha_1, \dots, \alpha_m)$. Using the above property of triangular norms, we have that

$$T(\alpha_R, \alpha_1) \geq \dots \geq T(\alpha_R, \alpha_1, \dots, \alpha_i) \geq \dots \geq T(\alpha_R, \alpha_1, \dots, \alpha_m);\ i < m.$$

As this function is calculated incrementally, as soon as the interpreter reaches an arc containing an atom p_i $(i < m)$ such that

$$T(\alpha_R, \alpha_1, \dots, \alpha_i) \leq \alpha$$

we can prune the remaining search space defined by the subgraphs whose root nodes are the nodes $p_{i+1}, \dots, p_m$.

We have confined our explanation to the first and second connectors of a node, but it is straightforward to generalize this result to all the connectors outgoing from a given node.

In order to incorporate this improvement into the previous interpreter, we have to modify function or-extension and function and-extension. The new pseudo-code for such functions is shown in Figure 5.3.

Theorem 5.4 correctness *Function interpreter(P, q) returns α iff α is the maximum degree of deduction of q in P.*

Proof: The correctness follows from Theorem 5.2 and from the fact that the pruning strategy only eliminates some portions of the graph which are irrelevant for computing the maximum degree of deduction. ■

Theorem 5.5 complexity *Let P be a logic program, let q be a goal and let m be the total number of occurrences of propositional atoms in P. Then, the worst-case time complexity of function interpreter(P, q) is in $O(m)$.*

Proof: It is clear that if t is the number of execution steps of function interpreter(P, q) of the former version and t' is the number of execution steps of the latter version, then $t \geq t'$. ■

```
**function** or-extension (p: atom): truth_value
**var** R: rule
**var** value: truth_value
**begin**
    **if** visit($p$) **then** **return**(val($p$));
    visit($p$) := true;
    **for each** rule $R$ in rules-conseq($p$) **do**
        value := and-extension($R$,val($p$));
        **if** value $>$ val($p$) **then** val($p$) := value
    **endfor**;
    **return**(val($p$))
**end**
```

```
**function** and-extension (R: rule, max_value: truth_value): truth_value
**var** p: atom
**var** value: truth_value
**begin**
    **if** val($R$) $\leq$ max_value **then** **return**(val($R$));
    value := val($R$);
    **for each** atom $p$ in antecedents($R$) **do**
        value := $T$(value, or-extension($p$));
        **if** value $\leq$ val($p$) **then** **return**(value)
    **endfor**;
    **return**(value)
**end**
```

Figure 5.3: An improved version of the interpreter

5.4 Negation

The negation as failure rule is generally used to deduce negative information in logic programming. Before incorporating such a rule into the interpreter, we will first review how it works in classical logic.

Assume that we have a classical propositional rule $p \leftarrow p_1, \ldots, \neg p_i, \ldots, p_k$ with a negative literal $\neg p_i$ in the body. The negation as failure rule works as follows: if $P \not\vdash p_i$ then $P \vdash \neg p_i$.[1]

From a semantical point of view, $P \not\models p_i$ implies that p_i is false (or 0) in some models of P and is possibly true (or 1) in others. In this case, the negation as failure rule establishes that we restrict the models of P to those models I of P such that $I(p_i) = 0$ $(I(\neg p_i) = 1)$.

A possible extension of the negation as failure rule to our multiple-valued setting could be as follows: if the maximum degree of logical consequence of a propositional atom p_i is 0, then we apply the negation as failure rule considering only the interpretations $I(p_i) = 0$ and so $I(\neg p_i) = 1$. In a more general case, if $P \models (p_i; \alpha)$, where α is the maximum degree of logical consequence, then we consider only the models I of P such that $I(p_i) = \alpha$. This means that the negation as failure rule in the multiple-valued case considers only those models of P where $I(\neg p_i) \geq 1 - \alpha$, namely in the interval $[1 - \alpha, 1]$.

Example 5.3 *Given the goal q and the following logic program*

$$\begin{array}{l} (q \leftarrow p_1, \neg p_2, p_3; \alpha) \\ (p_1; \alpha_1) \\ (p_2; \alpha_2) \\ (p_3; \alpha_3) \end{array}$$

the maximum degree of deduction of p_2 is α_2 and the maximum degree of deduction of $\neg p_2$ is $1 - \alpha_2$. Therefore, the maximum degree of deduction of q is $T(\alpha, \alpha_1, 1 - \alpha_2, \alpha_3)$.

Remark: Inconsistencies can be derived when negative literals occur in the body of a rule (Lloyd, 1987). Given the program formed by $(p \leftarrow \neg p; 1)$ and $(p; 0)$, p and $\neg p$ are deduced with maximum degree of deduction 1. It should be noticed that this also happens in classical propositional logic. Some conditions under which the consistency of a program is guaranteed could be studied (Escalada-Imaz, 1989a), but they are beyond the scope of the present thesis.

In order to incorporate the negation as failure rule into our interpreter, we assume that negated atoms appear only in the body of rules and are stored in the field antecedents of the data structure used to represent rules. We also need to modify function and-extension; the new pseudo-code is shown in Figure 5.4.

Theorem 5.6 correctness *Function interpreter(P, q) returns α iff $P \vdash (q; \alpha)$, where α is the maximum degree of deduction of q using the deduction relation and the negation as failure rule.*

[1] In the propositional case, the negation as failure rule coincides with the closed world assumption.

```
function and-extension (R: rule, max_value: truth_value): truth_value
var $p$: atom
var L: literal
var value: truth_value
begin
   if val($R$) $\leq$ max_value then return(val($R$));
   value := val($R$);
   for each L in antecedents($R$) do
      if L = $p$ then
         value := $T$(value, or-extension($p$))
      else   /* L = $\neg p$ */
         value := $T$(value, 1 − or-extension($p$))
      endif;
      if value $\leq$ val($p$) then return(value)
   endfor;
   return(value)
end
```

Figure 5.4: Function and-extension

Theorem 5.7 complexity *Let P be a logic program, let q be a goal and let m be the total number of occurrences of propositional atoms in P. Then, the worst-case time complexity of function interpreter(P, q) is in $O(m)$.*

5.5 Cut

In classical logic programming, the cut operator, denoted by "/", is a control facility used to prune some portions of the search space. For example, assume that we have a rule $q \leftarrow p_1, p_2, /, p_3$. Once the cut is executed, it prunes the possible alternatives for all the literals on its left in the body of the rule as well as the alternatives to satisfy the consequent.

If we are dealing with propositional logic, the cut influences only the search space of the consequent of the rule because there are no variables to be instantiated; i.e. the alternatives for q, p_1 and p_2 are not considered if the cut is executed. So, in our multiple-valued propositional context, we could define a multiple-valued cut as follows:

Definition 5.8 multiple-valued cut $(/, \alpha)$ *If the interpreter finds a rule of the form $q \leftarrow p_1, p_2, \ldots, p_i, (/, \alpha), p_{i+1}, \ldots, p_k$ then if the maximum degree of deduction α_{1i} of the conjunction $p_1 \wedge \ldots \wedge p_i$ verifies that $\alpha > \alpha_{1i}$ (that $\alpha \leq \alpha_{1i}$) then the remaining alternatives for q are (are not) considered.*

Note that in our multiple-valued propositional context it makes no sense to consider the alternatives of satisfying the literals in the body which are on the left of the cut. Before arriving at the cut, the maximum degrees of deduction of such literals have been computed and so the search space involving such literals has been explored.

Example 5.4 *Let $T(x,y) = x{\cdot}y$. Given the following portion of a logic program:*

$$\begin{array}{l}(q \leftarrow p_1, p_2, (/, 0.6), p_3, p_4; 1)\\ (q \leftarrow p_4, p_5; 1)\\ (q \leftarrow p_5, p_6, p_7; 1)\\ \dots\dots\dots\dots\\ (p_1; 0.8)\\ (p_2; 0.8)\end{array}$$

the maximum deduction degree of $p_1 \wedge p_2$ is 0.64 and hence only the first rule should be considered among those concluding q. If we assume that the first rule is $(q \leftarrow p_1, p_2, (/, 0.8), p_3, p_4; 1)$, then the maximum degree of deduction of q is obtained considering the remaining rules with consequent q.

A proof procedure for logic programs incorporating the multiple-valued negation as failure rule and the cut operator is shown in Figure 5.5. We add the field *cut* to the data structure used to represent rules; $cut(R)$ = true iff rule R contains a cut. If a rule R' contains a cut, then such a cut is stored as a literal in $antecedents(R')$. These changes are incorporated into function interpreter. Function or-extension scans the rules whose consequent is a propositional atom p, but this scanning is stopped if a rule $(p \leftarrow p1, \dots, p_i, (/, \alpha), p_{i+1}, \dots; p_n; \alpha_R)$ such that $T(val(p_1), \dots, val(p_i)) \geq \alpha$ is found. To this end, function and-extension returns a boolean variable (prune). It holds that prune=true iff there is a rule $(p \leftarrow p1, \dots, p_i, (/\alpha), p_{i+1}, \dots, p_n; \alpha_R)$ such that $T(val(p_1, \dots, val(p_i) \geq \alpha$.

Theorem 5.8 correctness *Function interpreter(P, q) returns α iff $P \vdash (q; \alpha)$, where α is the maximum degree of deduction of q using the deduction relation, the negation as failure rule and the cut operator.*

Theorem 5.9 complexity *Let P be a logic program, let q be a goal and let m be the total number of occurrences of propositional atoms in P. Then, the worst-case time complexity of function interpreter(P, q) is in $\mathcal{O}(m)$.*

The proof of the previous theorems is a straightforward consequence of the correctness and complexity stated before.

5.6 Implementation

Béjar (1993) implemented, in C++, a programming environment for infinitely-valued propositional logic programs. The main features of this environment are the following ones:

```
function interpreter (P: program, q: goal): truth_value
var $p$: atom
var R: rule
begin
  for each atom $p$ in $P$ do
    if $P$ contains $(p;\alpha_1),\ldots,(p;\alpha_n)$ then
        $val(p) := \max(\alpha_1,\ldots,\alpha_n)$
    else
        $val(p) := 0$
    endif;
    $visit(p) :=$ false
  endfor;
  for each rule $R = (p \leftarrow p_1,\ldots,p_n;\alpha)$ in P do
    $val(R) := \alpha$;
    antecedents$(R) := \{p_1,\ldots,p_n\}$;
    rules-conseq$(p)$ := rules-conseq$(p) \cup R$;
    if R contains a cut $(/,\beta)$ then
        cut(R) := true
    else
        $cut(R) :=$ false
    endif
  endfor;
  return(or-extension($q$))
end
```

```
function or-extension (p: atom): truth_value
var R: rule
var prune: boolean
var value: truth_value
begin
    if visit($p$) then return(val($p$));
    visit($p$) := true;
    prune := false;
    for each rule R in rules-conseq($p$) do
        if prune then return(val($p$));
        (value, prune) := and-extension(R,val($p$));
        if value > val($p$) then val($p$) := value
    endfor;
    return(val($p$))
end
```

```
function and-extension (R: rule, max_value: truth_value):
                              (truth_value, boolean)
var p: atom
var L: literal
var prune: boolean
var value: truth_value
begin
  if val($R$) $\leq$ max_value and not cut($R$) then
    return(val($R$), false)
  endif;
  value := val($R$);
  prune := false;
  for each L in antecedents($R$) do
    case L of
        $p$ :   value := $T$(value, or-extension($p$));
       $\neg p$ :   value := $T$(value, 1 − or-extension($p$));
      $(/,\beta)$:  if $\beta \leq$ value then prune := true
    endcase;
    if value $\leq$ max_value and not cut($R$) then
      return(value, prune)
    endif
  endfor;
  return(value, prune)
end
```

Figure 5.5: The final version of the interpreter

- It has a compiler that from the source program creates the data structures for representing facts and rules. This compiler is formed by a lexical analyzer and a syntactical analyzer which were implemented with Lex and Yacc, respectively. The compiler has facilities for handling lexical and syntactical errors.
- The T-norm used by the program can be modified. Some T-norms are predefined, but the user can define his/her own T-norms.
- Programs can be executed in debug mode. The user can know the state of the execution when entering and leaving functions and-extension and or-extension. He/she can also select a particular list of atoms to be executed in debug mode.
- It has user-friendly interface implemented using Object Windows Borland Library 2.0 for IBM PC-like computers.

Chapter 6

Conclusions

In the preceding chapters we have been concerned with the design of proof procedures for solving both satisfiability and deduction problems in multiple-valued logics. As our ultimate goal was to obtain competitive algorithms from a computational point of view, our approach has not been confined to present complete calculi and sketch pseudo-code of algorithms that automate their application. We have also paid special attention to the definition of suitable data structures for representing formulas, heuristics for exploring the proof search space and techniques for avoiding redundant and useless computations.

Our next step should be to implement the proof procedures designed and incorporate them into some automated theorem prover, as well as to examine their behaviour in real-world applications. This way, we could put our ideas into practice and, from this experience, gain new insights for developing other proof methods and improving existing ones.

In Chapter 1, we have already given in detail the main contributions of this thesis. Here, we summarize them briefly:

Regarding satisfiability problems we have focused on signed CNF formulas, because they are an appropriate formalism for representing and solving the SAT problem in any finitely-valued propositional logic. We have described DP-style decision procedures for both signed and regular CNF formulas. To this end, we have extended the one-literal rule to the signed setting, introduced the new concept of maximal truth value set and defined original branching rules and heuristics. For the subclass of regular CNF formulas, we have also defined suitable data structures for representing formulas and some deletion strategies for eliminating redundant and irrelevant clauses.

Starting out from the fact that there exist polynomial-time algorithms for testing the satisfiability of classical Horn and 2-CNF formulas, we have then investigated what happens with these particular subclasses of formulas in the logic of signed CNF formulas. First, we have developed a complete unit resolution calculus for regular Horn formulas. Then, we have designed efficient Horn satisfiability checking procedures with linear-time complexity when the truth value set is finite (if infinite, the complexity is almost linear). Second, we have demonstrated that the SAT problem in arbitrary signed 2-CNF formulas is NP-complete. Third, we have shown that this problem is polynomially solvable when the formulas are regular or monosigned. We have described efficient satisfiability checking procedures for these subclasses of signed 2-CNF formulas. These procedures reach a quadratic-time complexity in the worst case.

Regarding deduction problems we have focused on a wide and important family of infinitely-valued logics, including Łukasiewicz logic. First, we have defined a propositional logic programming language for this family of logics and proved that a modus ponens-style inference rule is complete for determining the maximum degree of logical consequence of a goal in a given infinitely-valued logic program. Second, we have defined a multiple-valued negation as failure rule and a cut operator. Third, we have defined suitable data structures for representing logic programs and described an interpreter with a linear-time complexity in the worst case.

Apart from the implementation of the proof procedures we have introduced in this thesis, we would like to point out some other future research perspectives:

- To create a test-bed, formed by a collection of both real-world and randomly generated instances, for evaluating and comparing satisfiability testing algorithms for signed CNF formulas.
- To define new branching heuristics for the signed DP-style procedures described in this thesis and perform an experimental comparison.
- To develop new concepts, such as maximal truth value set, that allow one to avoid redundant and useless computations during the exploration of the proof search space.
- To investigate the use of stochastic local search algorithms, e.g. GSAT (Selman et al., 1992), for finding satisfying interpretations in signed CNF formulas.
- To identify new subclasses of multiple-valued formulas whose satisfiability and deduction problems are polynomially solvable, and equip them with complete calculi and efficient proof procedures.
- To find further application areas for the the infinitely-valued logic programming language defined, extend it to the first-order case and implement a declarative programming environment.

References

Ackermann, R. (1967). *Introduction to Many-Valued Logics.* Routledge & Kegan, London.

Alsinet, T. and Manyà, F. (1996). A declarative programming environment for infinitely-valued logics. In *Proc. International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, IPMU'96, Granada, Spain*, volume 3, pages 1205–1210.

Anderson, R. and Bledsoe, W. (1970). A linear format for resolution with merging and a new technique for establishing completeness. *JACM*, 17:525–534.

Aspvall, R., Plass, M. and Tarjan, R. (1979). A linear time algorithm for testing the truth of certain quantified boolean formulae. *Information Processing Letters*, 8(3):121–123.

Baaz, M., Fermüller, C. G., Ovrutcki, A. and Zach, R. (1993). MULTLOG: A system for axiomatising many-valued logics. In Voronkov, A., editor, *Proc 4th Int. Conf. on Logic Programming and Automated Theorem Proving LPAR, St. Petersburg, Russia*, pages 345–347. Springer, LNAI 698.

Baaz, M. and Fermüller, C. G. (1995). Resolution-based theorem proving for many-valued logics. *Journal of Symbolic Computation*, 19:353–391.

Baaz, M., Fermüller, C. G., Salzer, G. and Zach, R. (1996). MUltlog 1.0: Towards an expert system for many-valued logics. In *Proc 13th Int. Conf. on Automated Deduction CADE'96, New Brunswick/NJ, USA*, pages 226–230. Springer, LNAI 1104.

Baldwin, J. (1987). Evidential support logic programming. *Fuzzy Sets and Systems*, 24:1–26.

Baldwin, J., Martin, T. and Pilsworth, B. (1995). *Fril: Fuzzy and Evidential Reasoning in Artificial Intelligence.* John Whiley & Sons Inc.

Beckert, B., Hähnle, R., Oel, P. and Sulzmann, M. (1996). The many-valued tableau-based theorem prover $_3T^AP$: Version 4.0. In *Proc 13th Int. Conf. on Automated Deduction CADE'96, New Brunswick/NJ, USA*, pages 303-307. Springer, LNAI 1104.

Béjar, R. (1993). *Implementación de un intérprete proposicional y de un intérprete de primer orden para programación lógica multivaluada.* EUP-Universitat de Lleida. Master's Thesis.

Böhm, M. and Speckenmeyer, E. (1996). A fast parallel SAT-solver - efficient workload balancing programming. *Annals of Mathematics and Artificial Intelligence*, 5(17):381–400.

Bolc, L. and Borowik, P. (1992). *Many-Valued Logics. 1: Theoretical Foundations.* Springer Verlag.

Buro, M. and Büning, H. K. (1993). Report on a SAT competition. *EATCS Bulletin*, 1(49):143–151.

Carnielli, W. A. (1987). Systematization of finite many-valued logics through the method of tableaux. *Journal of Symbolic Logic*, 52(2):473–493.

Church, A. (1936). An unsolvable problem of elementary number theory. *Americal Journal of Mathematics*, 58:345–363.

Cook, S. (1971). The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158.

Cook, S. and Luby, M. (1988). A simple parallel algorithm for finding a satisfying truth assignment to a 2-CNF formula. *Information Processing Letters*, 27:141–145.

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *JACM*, 7(3):201–215.

Davis, M., Logemann, G. and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5:394–397.

Davis, M. (1983a). A computer program for Presburger's algorithm. In Siekmann, J. and Wrightson, G., editors, *Automation of Reasoning: Classical Papers in Computational Logic 1957–1966*, volume 1, pages 41–48. Springer-Verlag.

Davis, M. (1983b). The prehistory and early history of automated deduction. In Siekmann, J. and Wrightson, G., editors, *Automation of Reasoning: Classical Papers in Computational Logic 1957–1966*, volume 1, pages 1–28. Springer-Verlag.

Di Zenzo, S. (1988). A many-valued logic for approximate reasoning. *IBM Journal of Research and Development*, 32(4):552–565.

Dowling, W. and Gallier, J. (1984). Linear-time algorithms for testing the satisfiability of propositional Horn formulæ. *Journal of Logic Programming*, 3:267–284.

Dubois, D., Lang, J. and Prade, H. (1990). Poslog, an inference system based on possibilistic logic. *Proceedings North American Fuzzy Information Processing Society Congress*, pages 177–180.

Escalada-Imaz, G. (1989a). *Optimisation d'Algorithmes d'Inference Monotone en Logique des Propositions et du Premier Ordre.* PhD thesis, Université Paul Sabatier, Toulouse.

Escalada-Imaz, G. (1989b). Un algorithme de complexité quadratique et un algorithme de complexité lineaire pour la 2-satisfiabilité. Technical Report 89378, LAAS, Toulouse.

Escalada-Imaz, G. and Manyà, F. (1993a). An interpreter of logic programs in multivalued propositional logic. In Agustí, J. and García, P., editors, *Proc. Segundo Congreso de Programación Declarativa PRODE'93, Blanes, Spain*, pages 245–259.

Escalada-Imaz, G. and Manyà, F. (1993b). Testing the satisfiability of multivalued Horn formulæ. In *Proc. V Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA'93, Madrid, Spain*, pages 226–235.

Escalada-Imaz, G. and Manyà, F. (1994a). El problema de la satisfactibilidad en cláusulas de Horn multivaluadas. *Novatica*, 108:11–15.

Escalada-Imaz, G. and Manyà, F. (1994b). A linear interpreter for logic programming in multiple-valued propositional logic. In *Proc. International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, IPMU'94, Paris*, volume 2, pages 943–949.

Escalada-Imaz, G. and Manyà, F. (1994c). The satisfiability problem for multiple-valued Horn formulæ. In *Proc. International Symposium on Multiple-Valued Logics, ISMVL'94, Boston/MA, USA*, pages 250–256. IEEE Press, Los Alamitos.

Escalada-Imaz, G. and Manyà, F. (1995). Efficient interpretation of propositional multiple-valued logic programs. In B. Bouchon-Meunier, R. Y. and Zadeh, L., editors, *Advances in Intelligent Computing*, pages 428–439. Springer Verlag, LNCS 945.

Escalada-Imaz, G. and Manyà, F. (1996a). On multiple-valued logic programming. In Dahl, V. and Sobrino, A., editors, *Estudios sobre Programación Lógica y sus Aplicaciones*, pages 387–419. Servicio de Publicaciones de la Universidad de Santiago de Compostela.

Escalada-Imaz, G. and Manyà, F. (1996b). On the 2-SAT problem for signed formulas. In *Proc. Workshop/Conference on Many-Valued Logics for Computer Science Applications, COST Action 15, Barcelona, Spain.*

Escalada-Imaz, G., Manyà, F. and Sobrino, A. (1996c). Principios de programación lógica con información incierta. Descripción de algunos de los sistemas más relevantes. *Theoria*, 11(27):123–148.

Even, S., Itai, A. and Shamir, A. (1976). On the complexity of timetable and multicommodity flow problems. *SIAM J. Computing*, 5:691–703.