Essays on Logic Programming

Renato Bruni

Dissertation submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
in
Operations Research

Supervisor Antonio Sassano

Department of Computer and Systems Science
University of Roma
"La Sapienza"

© Copyright 2000 by Renato Bruni

Contents

Acknowledgments **vii**

Preface **ix**

1 Introduction to Logic and Inference **1**

1.1 Logic 1
1.2 Propositional Logic 3
1.3 Normal Forms 5
1.4 Boolean and Pseudo-Boolean Functions 8
1.5 First-Order Logic 9
1.6 Implication and Inference 11
1.7 The Role of Satisfiability 14
1.8 Constraint Satisfaction Problems 15
1.9 Combinatorial Optimization Models 18

2 Algorithms for SAT **23**

2.1 An Overview 23
2.2 Resolution 25
2.2.1 Set of Support 26
2.2.2 P-Resolution and N-Resolution 27
2.2.3 Linear Resolution 27
2.2.4 Regular Resolution 27
2.2.5 Davis-Putnam Resolution 27
2.2.6 Extended Resolution 28
2.2.7 Subsumption 28
2.2.8 Pure Literals 28
2.3 Enumeration trough Splitting and Backtrack 28
2.3.1 Simple Backtracking 29
2.3.2 Unit Clause Backtracking 29

2.3.3 Clause Order Backtracking . . . 29
2.3.4 Probe Order Backtracking . . . 30
2.3.5 Shortest Clause Backtracking . . . 30
2.3.6 Jeroslow-Wang . . . 30
2.3.7 Backtracking and Resolution . . . 31
2.3.8 The Pure Literal Rule Algorithm . . . 31
2.3.9 Clause Area . . . 31
2.3.10 Branch Merging . . . 32
2.3.11 Search Rearrangement . . . 32
2.3.12 2-SAT Relaxation . . . 33
2.3.13 Horn Relaxation . . . 33
2.3.14 Backmarking and Backjump . . . 33
2.3.15 Conflict Directed Backjumping . . . 34
2.3.16 Backtracking with Lookahead . . . 34
2.3.17 Backtrack with Lookback . . . 34
2.3.18 Unit propagation with delayed unit subsumption . . . 35
2.3.19 Backtracking for Proving Non-Existence . . . 35
2.3.20 Intelligent Backtracking . . . 35
2.3.21 Macro Expansion . . . 35
2.3.22 Backtrack Programming . . . 36
2.3.23 Special-Purpose Architectures . . . 36
2.3.24 Branch and bound . . . 36
2.3.25 Some Remarks on Complexity . . . 36
2.4 Integer Programming Methods . . . 37
2.4.1 Linear Program Relaxation . . . 37
2.4.2 Branch and Bound Method . . . 38
2.4.3 Cutting Plane Method . . . 39
2.4.4 Branch and Cut Method . . . 39
2.4.5 Interior Point Method . . . 40
2.4.6 Improved Interior Point Method . . . 40
2.4.7 Integer Programming Heuristics . . . 41
2.5 Local Search . . . 41
2.5.1 The Min-Conflicts Heuristics . . . 43
2.5.2 The Best-Neighbor Heuristics . . . 44
2.5.3 The Random Value/Variable Heuristics . . . 44
2.5.4 Random Flip Heuristic . . . 44
2.5.5 Random Value (Assignment) Heuristics . . . 44
2.5.6 Random Variable (Selection) Heuristics . . . 44
2.5.7 Partial Random Variable Selection Heuristics . . . 45
2.5.8 The Trap Handling Heuristics . . . 46

2.5.9 Boolean Local Relaxation 47
2.5.10 Simulated Annealing Algorithm 47
2.5.11 Randomized Local Search 47
2.5.12 Tunneling Heuristic 48
2.5.13 Parallel Local Search 49
2.5.14 Complete Local Search 49
2.5.15 Greedy Local Search 50
2.5.16 Tabu Local Search 50
2.6 Global Optimization 50
2.6.1 Universal SAT Input Models 51
2.6.2 Complete Global Optimization Algorithms 53
2.6.3 Continuous Lagrangian Based Constrained Optimization Algorithms 53
2.6.4 Discrete Lagrangian Based Constrained Optimization Algorithms 56
2.7 Advanced Techniques 59
2.7.1 Analysis and Decomposition 59
2.7.2 General Boolean Representations 60
2.7.3 Binary Decision Diagram 61
2.7.4 The Unison Algorithms 62
2.7.5 Multispace Search 63
2.7.6 Partitioning to Reduce Input Size 64
2.7.7 Partitioning Variable Domains 65
2.7.8 Parallel SAT Algorithms and Architectures 66
2.8 Special Subclasses of SAT 67
2.8.1 2-SAT 67
2.8.2 Horn and Extended Horn Formulas 67
2.8.3 Formulas from Balanced Matrices 68
2.8.4 Single-Lookahead Unit Resolution 69
2.8.5 q-Horn Formulas 70
2.8.6 Renamable Formulas 72
2.8.7 Formula Hierarchies 72
2.8.8 Pure Implication Formulas 73
2.8.9 Easy class for Nonlinear Formulations 73
2.8.10 Nested and Extended Nested Satisfiability 73
2.9 Probabilistic and Average-Case Analysis 74

3 A Complete Adaptive Solver **75**
3.1 Introduction 75
3.2 Individuation of hard clauses 76
3.3 Clause based Branching Tree 79
3.4 Minimally Unsatisfiable Subformulas 80
3.5 Adaptive core search 81
3.6 Computational results 83
3.6.1 The series *ii32* 84
3.6.2 The series *par16* 85
3.6.3 The series *aim100* 86
3.6.4 The series *jnh* 87
3.6.5 The series *ssa* 89
3.6.6 The series *des* 90
3.7 MUS Selection 90
3.8 Conclusions 92

4 Orthogonalization of a Logic Formula **95**
4.1 Introduction 95
4.2 Notation 96
4.3 The Orthogonal form 98
4.4 Basic Orthogonalization operation 100
4.5 Improvements 104
4.5.1 Recognition of Internally Orthogonal Sets of Terms . . 104
4.5.2 Absorption 105
4.5.3 Synthesis Resolution 105
4.6 Complete Orthogonalization Operation 105
4.7 Testing 106
4.8 Conclusions 107

5 Logic for Error Free Data Collecting **109**
5.1 Data Collecting through Questionnaires 110
5.2 A Logical Representation of the Set of Edits 113
5.2.1 Identification of the domains for the fields 115
5.2.2 Identification of the breakpoints for the fields 116
5.2.3 Individuation of equivalent classes of subsets 117
5.2.4 Definition of logic variables 119
5.2.5 Encoding of edits as clauses 121
5.2.6 Identification of congruency clauses 122
5.3 Edits Validation 124
5.3.1 Complete Inconsistency in the Set of Edits 125

5.3.2 Partial Inconsistency in the Set of Edits 126
5.3.3 Redundancy in the set of edits 128
5.4 Individuation of Erroneous Records 129
5.5 The Problem of Imputation 129
5.5.1 Error Localization 131
5.5.2 Imputation through a Donor 132
5.6 A Set Covering Formulation 134
5.6.1 Error Localization 137
5.6.2 Imputation through a Donor 138
5.7 Implementation 140
5.8 Results 141
5.8.1 Edit Validation 142
5.8.2 Data Imputation 143
5.9 Conclusions 149

References **151**

Acknowledgments

During the course of my Ph.D., I felt many times grateful to people who helped me. On the other hand, like everyone else, I also had my troubles (the great troubles of life: luggage lost at the airport, traffic jam and parking fines, slow or no connection to the Internet, old computers, too slow refunds, etc.), and then I felt ungrateful to people who thwarted me. Now I think, and I hope I do not have to change my mind in a near future, that, since improvements come from difficulties, I should be grateful to them as well, at least for making me understand how much time is so incomparably precious. Anyway, I will not mention them, and therefore all the names below are people in the first category.

I thank my supervisor, Antonio Sassano, who, although sometimes too busy even to breath, gave me many precious advice and the example of working fast and well. I thank my father, Carlo Bruni, who gave me the sake for scientific research. I thank Peter Hammer, who trusted me and gave me the opportunity to work with him for some time in Rutgers. I thank Carlo Mannino, who gave me some of his time to help me solving some of my problems when I encountered them.

I thank, moreover, many good researchers and nice people which I met during these years, mostly in conferences around the world, either for interesting exchange of ideas or for being a good company. They include Sorin Alexe, Edoardo Amaldi, Claudio Arbib, Francisco Barahona, Luca Basile, Sonia Benzi, Dimitri Bertsekas, Florin Bidian, Fabiana Birarelli, Stefano Bistarelli, Endre Boros, Thomas Davoine, John Franco, Jarl Friis, Michel Goemans, Georg Gottlob, Andrea Grosso, Pierre Hansen, Giuseppe Italiano, Etienne de Klerk, Hans Van Maaren, Joao Marquis Silva, Andrea Lodi, Simeone Marino, Fabio Massacci, Carlo Meloni, Michela Milano, Luciano Nieddu, Paolo Nobili, Giampaolo Oriolo, Jacopo Patrizi, David Pisinger, Daniele Pretolani, Alessandra Reale, Line Reinhardt, Fabrizio Rossi, Laurent Simon, Roman Slowinski, Stefano Smriglio, Stefan Szeider, Csaba Toth, Klaus Truemper, Joost Warners, and of course all the friends at D.I.S.

Above all, following the better tradition in acknowledgments, I thank three nice girls, in strictly chronological order: Ursula, Ioana and, especially, Claudia. They know the reasons.

Preface

Optimization models with binary variables and a linear structure possess a high expressive power, and are expecially suitable to represent logic and decision problems allowing the representation of options in the cases where indivisibility is required or where there is not a continuum of solutions. Many real problems have in fact such characteristics, being, at the very essential level, a choice.

Binary problems are hard. The finite number of alternatives is in fact not a simplification, with respect to the case of a continuum of possibilities, because of the impressive number of such alternatives in real world problems. However, recent years have witnessed a huge amount of research in this field, and consequent decisive algorithmic improvements. We are now closer than ever before to realizing the ancient vision of using logic for reducing delicate and complex choice problems to computations.

This work is organized as follows.

Chapter 1 constitutes a brief overview of logic. Some introductory elements of propositional logic, Boolean functions, propositional satisfiability and constraint satisfaction problems are given.

Chapter 2 is a more detailed review of algorithms for satisfiability problems, focused in particular on branching search frameworks. A unified perspective of various algorithms is attempted.

Chapter 3 presents a new search framework for propositional satisfiability, which makes use of an adaptive strategy. The concept of minimal unsatisfiable subformula is discussed, and a technique for its selection is developed. Computational results and comparisons are provided.

Chapter 4 treats further logic problems, introducing the concept of orthogonal Boolean formulae, and presenting a procedure to transform any Boolean formula into an orthogonal Boolean formula. Test results are reported.

Finally, chapter 5 is an application of the above concepts to computationally demanding real world problems. A case of data collecting is considered. The problems arising from automatic error detection and correction are respectively modeled as satisfiability and set covering problems. The satisfiability problems are solved by using solver presented in chapter 3. The set covering problems are solved by using state-of-the-art solvers. Results are reported and discussed.

Roma, Italy, December 2000. RENATO BRUNI

Chapter 1

Introduction to Logic, Inference, Satisfiability and Constraint Satisfaction

1.1 Logic

Logic is the science of correct reasoning. It can be viewed as the system of principles underlying any science. Logic describes relationship among propositions, with the purpose of providing instruments to check the correctness of an argument, to deduce conclusions from premises, and therefore to establish the consistency of a theory. In order to reach this aim, logic must have descriptive symbolic tools to represent propositions, and deterministic and formal rules to work on them. In the classical symbolic approach, both of the above elements must be merely mechanical and completely independent from the meaning of the considered propositions.

This independence from the contents was already clear to the Greek philosophers, in particular to Aristotle (384-322 B.C.) with the development of syllogism.

The results formulated at that time stood almost unchanged, through medieval logicians, up to the time of G.W. Leibniz (1646-1716), who developed several versions of a calculus for logic. His definitions and terminology had some difficulties and defects, but, nevertheless, they lead him to imagine an encoding of human knowledge and a computational process that would reduce rational discussions to mere calculations.

After one hundred and fifty years, G. Boole (1815-1864) created a calculus for logic that, in today's terms, uses the logical *and*, the exclusive *or*,

and a certain inverse of that *or*. Such system made logic calculation possible, and, with some simplifications, is known today as *Boolean algebra* or *Propositional Logic*.

After that, G. Frege (1848-1925) created a complete notation for mathematical logic. Today, Frege's formulae would be called *parse trees* of logic expressions. Frege envisioned that mathematics could be based on logic. He set out to prove that conjecture in an extraordinary effort spanning many years. His work contains many novel ideas, but, later, B. Russel (1872-1970) showed that the assumptions made by Frege contained a flaw.

A.N. Whitehead (1861-1947) and Russel successfully carried out much of Frege's plan, and argued in a three volume treatise called *Principia Mathematica* that all of mathematics can be derived from logic. At that time, it seemed that, in principle, one could derive all theorems of mathematics by computation.

Subsequently, D. Hilbert (1862-1943) defined the axiomatic method, introduced *meta-mathematics*, and hoped that, with these tools, one could carry out a program that eventually would establish the consistency of most, if not all, mathematical systems.

In 1931 [106], K. Gödel (1906-1978) proved that the construction of Whitehead and Russel cannot build all of mathematics, and that Hilbert's program generally cannot be carried out. That result, called the *Incompleteness Theorem*, implies that one cannot systematically compute all theorems of mathematics starting from some given axioms.

The availability of computers in the second half of the XX century raised the hope that computations in logic could be used to solve real-world problems. Loveland gives a concise account of the developments since the 1950s in [176]. Logic and Mathematics are nowadays always more two aspects of a same stream of science. Just to mention some remarkable examples, Hammer always pursued this idea with the theory of boolean functions [124]. Chandru and Hooker stress and enhance the above interconnections in [43]. Truemper developed a whole new mathematical method for computation in propositional logic [250]. Extensive references can be found it the bibliography of these three books.

The vision of logic computation is therefore gradually being translated into reality. Logic is nowadays used in a variety of fields. Examples are deductive databases, automatic theorem proving, Artificial Intelligence, integrated logic circuit design and test, etc. We give an overview of the various approaches later, once we have defined the logic problems to be considered.

In the field of logic we can define several formal systems. Mainly used are propositional logic, first-order predicate logic, probabilistic and related

logic, etc. In particular, the last several years have seen orders-of-magnitude improvements in inference algorithm for propositional logic. In this work we mainly deal with propositional logic. Throughout the rest of this chapter we give an introduction of propositional logic, Boolean and pseudo-Boolean functions, first-order logic, satisfiability and constraint satisfaction.

1.2 Propositional Logic

Propositional logic is the simplest formal system where we are interested in solving inference problems. Propositional logic, sometimes called sentential logic, may be viewed as a grammar for exploring the construction of complex sentences (propositions) from *atomic statements*, using the logical connectives. In prepositional logic we consider formulas (sentences, propositions) that are built up from atomic propositions that are unanalyzed. In a specific application, the meaning of these atomic propositions will be known.

The traditional (symbolic) approach to prepositional logic is based on a clear separation of the syntactical and semantical functions. The syntactics deals with the laws that govern the construction of logical formulas from the atomic propositions and with the structure of proofs. Semantics, on the other hand, is concerned with the interpretation and meaning associated with the syntactical objects. Prepositional calculus is based on purely syntactic and mechanical transformations of formulas leading to inference.

Propositional formulae are syntactically built by using an alphabet over the two following sets:

- The set of primary logic connectives $\{\neg, \vee, \wedge\}$, together with the brackets () to distinguish start and end of the field of a logic connective.
- The set of proposition symbols, such like $x_1, x_2, \ldots, x_n$.

The only significant sequences of the above symbols are the well-formed formulas (wffs). An inductive definition is the following:

- A propositional symbol x or its negation $\neg x$.
- Other wffs connected by binary logic connectives and surrounded, in case, by brackets.

Both propositional symbols and negated propositional symbols are called literals. Propositional symbols represent atomic (i.e. not divisible) propositions, sometimes called atoms. An example of wff is the following.

$$(\neg x_1 \vee (x_1 \wedge x_3)) \wedge ((\neg(x_2 \wedge x_1)) \vee x_3) \tag{1.1}$$

A formula is a wff if and only if there is no conflict in the definition of the fields of the connectives. Thus a string of atomic propositions and primitive connectives, punctuated with parentheses, can be recognized as a well-formed formula by a simple linear-time algorithm. We scan the string from left to right while checking to ensure that the parentheses are nested and that each field is associated with a single connective. Incidentally, in order to avoid the use of the awkward abbreviation "wffs," we will henceforth just call them propositions or formulas and assume they are well formed unless otherwise noted.

The calculus of propositional logic can be developed using only the three primary logic connectives above. However, it is often convenient to permit the use of certain additional connectives, such as $\Rightarrow$ which is called *implies*. They are essentially abbreviations that have equivalent formulas using only the primary connectives. In fact, if S_1 and S_2 are formulas, we have:

$$(S_1 \Rightarrow S_2) \quad \text{is equivalent to} \quad (\neg S_1 \vee S_2)$$

The elements of the set $B = \{T, F\}$ (or equivalently $\{1, 0\}$) are called truth values with T denoting $True$ and F denoting $False$. The truth or falsehood of a formula is a semantic interpretation that depends on the values of the atomic propositions and the structure of the formula. In order to examine the above, we need to establish a correspondence between propositional symbols and the elements of our Domain. This provides a truth assignment, which is the assignment of values T or F to all the atomic propositions. For this reason, propositions are often, although slightly improperly, called binary variables, but no connection with the concept of variable such as in the case of first-order logic exists.

To evaluate a formula we interpret the logic connectives, with their appropriate meaning of "not," "or," and "and". As an illustration, consider the formula (1.1). Let us start with an assignment of true (T) for all three atomic propositions x_1, x_2, x_3. At the next level, of subformulas, we have $\neg x_1$ evaluates to F, $(x_1 \wedge x_3)$ evaluates to T, $(x_2 \wedge x_1)$ evaluates to T, and x_3 is T. The third level has $(\neg x_1 \vee (x_1 \wedge x_3))$ evaluating to T and $((\neg(x_2 \wedge x_1)) \vee x_3)$ also evaluating to T. The entire formula is the "and" of two propositions both of which are true, leading to the conclusion that the formula evaluates to T. This process is simply the inductive application of the rules:

- S is T if and only if $\neg S$ is F.
- $(S_1 \vee S_2)$ is F if and only if both S_1 and S_2 are F.

- $(S_1 \wedge S_2)$ is T if and only if both S_1 and S_2 are T.

The assignment of truth values to atomic propositions and the evaluation of truth/falsehood of formulas is the essence of the semantics of this logic. We now introduce a variety of inference questions related to the truth or falsehood of propositions.

A truth assignment for which a formula evaluates to T is called a *model* for the formula. A formula is said to be *satisfied* by a model. A formula is *satisfiable* if has at least one model. A formula with no model is called *unsatisfiable*. While a model is proof of satisfiability, a proof of unsatisfiability is called a *refutation*. A formula for which every truth assignment is a model is called a *tautology*. The formula $(x_1 \vee \neg x_1)$ is a tautology. A formula for which no truth assignment is a model is called a *contradiction*.

A formula S_1 is said to *imply* another formula S_2, defined on the same set of atomic propositions as S_1, if every model of the former is also a model of the latter. This concept of logic implication is fundamental for the definition of the *Inference* problem (see section 1.6). Two formulas are said to be semantically *equivalent* if they share the same set of models, or if their models are equivalent under some mapping.

1.3 Normal Forms

The normal forms of formulas are special representations that make evident the duality of the binary connectives: *conjunction* $\wedge$ and *disjunction* $\vee$. Every propositional formula can be transformed in normal form. To begin with, we need to get rid of the unary *negation* operator $\neg$. This is done by expanding the set of atomic propositions $\{x_1, x_2, \ldots, x_n\}$ into the set of literals $\{x_1, x_2, \ldots, x_n; \neg x_1, \neg x_2, \ldots, \neg x_n\}$. We subsequently need to absorb the negation connective $\neg$ into the literals.

This is achieved by repeated application of the De Morgan's laws,

$$\neg(S_1 \vee S_2) \quad \text{is equivalent to} \quad (\neg S_1 \wedge \neg S_2) \tag{1.2}$$

$$\neg(S_1 \wedge S_2) \quad \text{is equivalent to} \quad (\neg S_1 \vee \neg S_2) \tag{1.3}$$

along with the involutory property of negation,

$$\neg\neg S = S \tag{1.4}$$

and the distributive laws

$$S_1 \vee (S_2 \wedge S_3) \quad \text{is equivalent to} \quad (S_1 \vee S_2) \wedge (S_1 \vee S_3) \tag{1.5}$$

$$S_1 \wedge (S_2 \vee S_3) \quad \text{is equivalent to} \quad (S_1 \wedge S_2) \vee (S_1 \wedge S_3) \tag{1.6}$$

We apply the above rules recursively until, $\neg$ operates only on atomic propositions. Thus we only have formulas with literals connected by disjunctions and conjunctions. A subformula that is a pure disjunction of literals is called a *clause*. A formula is said to be in *conjunctive normal form*, or CNF, if it is the pure conjunction of a set of clauses. Such a formula would have the appearance

$$(x_{i_1} \vee ... \vee x_{j_1} \vee \neg x_{k_1} \vee ... \vee \neg x_{n_1}) \wedge \ldots \wedge (x_{i_m} \vee ... \vee x_{j_m} \vee \neg x_{k_m} \vee ... \vee \neg x_{n_m})$$

A formula is said to be in *disjunctive normal form*, or DNF, if it is the pure disjunction of terms, each of which is a pure conjunction of literals. Such formulas have the appearance

$$(x_{i_1} \wedge ... \wedge x_{j_1} \wedge \neg x_{k_1} \wedge ... \wedge \neg x_{n_1}) \vee \ldots \vee (x_{i_m} \wedge ... \wedge x_{j_m} \wedge \neg x_{k_m} \wedge ... \wedge \neg x_{n_m})$$

The two normal forms CNF and DNF are dual representations with symmetric properties. Although there are some applications of propositional logic for which the DNF may be the more accepted normal form, we will stay with CNF as the standard form for propositions in this work.

We now have a complete procedure for transforming a given formula into CNF.

1. Use the transformation rules of De Morgan's laws and involution of negation to absorb all $\neg$ into the literals.

2. Use the distributive law to move the conjunctions out of the subformulas until each subformula is a clause of pure disjunctions.

As an illustration, let us consider the formula (2.1). The step by step transformations on this sample are depicted below.

$$\begin{array}{rcl} (\neg x_1 \vee (x_1 \wedge x_3)) & \wedge & ((\neg(x_2 \wedge \neg x_1)) \vee x_3) \\ (\neg x_1 \vee (x_1 \wedge x_3)) & \wedge & ((\neg x_2 \vee x_1) \vee x_3) \\ (\neg x_1 \vee x_1) \wedge (\neg x_1 \vee x_3) & \wedge & (x_1 \vee \neg x_2 \vee x_3) \\ (\neg x_1 \vee x_3) & \wedge & (x_1 \vee \neg x_2 \vee x_3) \end{array} \tag{1.7}$$

In the last step we removed the clause $(x_1 \wedge \neg x_1)$ since this is a tautology whose truth is invariant. In general, however, this simple procedure for reducing formulas to CNF runs into trouble. The main difficulty is that there may be an explosive growth in the length of the formula. The length of a formula is measured by the total number of literals in the description

of the formula. Consider the action of the above procedure on the family of DNF formulas

$$(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \ldots \vee (x_{2n-1} \wedge x_{2n}) \tag{1.8}$$

It is not difficult to see that the CNF formula produced is made up of the 2^n clauses. We therefore have that, in the worst case, the above procedure may generate a CNF formula whose length is exponentially related to the length of the input formula. Such CNF do not contain redundant clauses, since it can be proved [42] that no clause is implied by others.

On the other hand, there is another more complex procedure of conversion into CNF. Such procedure allows us to obtain succinct CNF representations, by using *rewriting* techniques. Rewriting techniques are essentially based on the introduction of additional atomic propositions corresponding to subformulas of the given formula, and on writing new clauses that represent the structure of the given formula.

By using such techniques, the two formulas will be equivalent in the sense that every model for the original formula can be extended to a model for the rewritten one (with appropriate truth values for the new atomic propositions). Conversely, every model of the rewritten formula, when restricted to the original atomic propositions (projected), corresponds to a model of the original formula. This result can be generalized for all formulas in propositional logic with the procedure described in [251, 268].

The result is a CNF formula that is satisfiable if and only if the input formula is. Further, it is easy to prove by induction that the length of the clause sets constructed is, in each case, bounded by a constant factor of the length of the corresponding subformula. Hence we have the following remarkable *rewriting theorem*.

Theorem 1.1 *Any formula in propositional logic can be rewritten to an equivalent CNF formula whose length is linearly related to the length of the original formula.*

It is possible to show that the CNF formula produced by the first procedure is the geometric projection of the formula produced by this second procedure [20]. The transformation into normal forms has several advantages, as explained in more detail in chapter 4.

1.4 Boolean and Pseudo-Boolean Functions

Instead of defining the formal system of propositional logic starting by its grammar, we could consider a formula as the definition of a boolean function [124]. Defining, as above, $B = \{T, F\}$, or, equivalently, $\{0, 1\}$, a Boolean function is a function $f_B(x_1, x_2, ..., x_n)$ from the Boolean hypercube B^n to the Boolean set B. Instead, a pseudo-Boolean function is a function $f_{pB}(x_1, x_2, ..., x_n)$ from the Boolean hypercube B^n to the set of real numbers $\mathbb{R}$.

Since every $V = (v_1, v_2, \dots, v_n)$ in B^n is the characteristic vector of the subset $\{i : v_i = 1\}$ of $\{1, 2, \dots, n\}$, a Boolean function is essentially a boolean-valued set function defined on the set of all subsets of an n-element set. A pseudo-Boolean function is instead a real-valued set function defined on the above set. These functions appear in numerous areas of discrete optimization, computer science, reliability theory, data analysis, graph theory, as well as in many interdisciplinary models of VLSI design, physics, telecommunications, etc.

Boolean and pseudo-Boolean functions can be defined in many different ways. For example, they can be specified by a truth table. On the left we have a Boolean function, on the right a pseudo-Boolean one.

x_1, x_2	$f_B(x_1, x_2)$
0, 0	1
0, 1	1
1, 0	0
1, 1	1

x_1, x_2	$f_{pB}(x_1, x_2)$
0, 0	4
0, 1	0
1, 0	-2
1, 1	1

It can be seen that pseudo-Boolean functions are a generalization of Boolean ones. We will therefore present some general results on pseudo-Boolean functions (pBfs) .

It was seen in [124] that every pBf has a polynomial representation. For instance, the functions defined above have the polynomial expression

$$f_B(x_1, x_2) = 1 - x_1 + x_1 x_2$$

$$f_{pB}(x_1, x_2) = 4 - 6x_1 - 4x_2 + 7x_1 x_2$$

By using both the original variables x_j , and their complements $\bar{x}_j = 1 - x_j$, every pBf can be represented as an (*additive*) *posiform*, i.e. a polynomial depending on the original variables and their complements, and having only positive coefficients, with the possible exception of the constant term [82].

Alternatively, a Boolean function can be represented using the primary logic operators $\{\neg, \vee, \wedge\}$. The polynomial expression representation and the propositional logic representation are closely related [82], and each one of them can be obtained from the other. A natural conversion into DNF is obtained by substituting every complemented variable $1 - x_i$ with the negated proposition $\neg x_i$, every addition $(+)$ with a disjunction $(\vee)$, and every multiplication $(.)$ with a conjunction $(\wedge)$. A propositional representation of the above f_B for instance is the following:

$$\neg x_1 \vee (x_1 \wedge x_2)$$

Propositional logic formulas are therefore representation of Boolean functions, and many results obtained for the latter ones hold of course for the formers.

1.5 First-Order Logic

We give here a very introductory mention of First-order predicate logic. More accurate treatments can be found in [102, 175]. Predicate logic is a refinement of propositional logic that brings substantially more expressive power. This comes at the price of a much more difficult inference problem. The modeling power of predicate logic derives from two features. It further analyzes atomic formulas of propositional logic by viewing them as attributing properties to individual objects, or stating that individual objects stand in certain relations to each other. It uses variables that range over objects, as well as constants that refer to particular objects. The syntactical limit of first-order logic is that a variable cannot represent a predicate.

First-order formulae are syntactically built by using an alphabet over the following sets:

- The set of primary logic connectives $\{\neg, \vee, \wedge\}$ and of the quantifiers $\{\forall, \exists, \}$, together with the brackets ().
- The set of constants symbols, such like $a, b, \ldots$
- The set of variables symbols, such like $x, y, \ldots$
- The set of function symbols, such like $f(\ ), g(\ ,\ ) \ldots$
- The set of predicate symbols, such like $P(\ ), Q(\ ,\ ), \ldots$

A constant is a fixed object of our *universe*. A variable can be any object of our universe. A function takes a number of objects of the universe as argument, and returns one object of the universe. A predicate takes a number of objects of the universe as argument, and returns a truth value in $\{T, F\}$.

We can congruently define well-formed formulas [197]. The *universal quantifier* $\forall$ means 'for every'. The *existential quantifier* $\exists$ means 'there exists'. Variables are said to be *bound* if they are in the field of action of a quantifier. Variables not bounded by quantifiers are said *free*.

A formula without variables can be view as a propositional logic formula, and is said to be at *ground level*. A formula F obtained from another formula G by substituting every variable with some different symbols if said to be an *instance* of G. G is a *generalization* of F.

Any first-order formula can be converted in *Skolem normal form*, that is a conjunction of universally quantified clauses, and such clauses have no variables in common. The original formula is satisfiable if and only if the converted formula is satisfiable.

A basic result of first-order logic shows how to write for any first order formula a propositional formula, called the Herbrand extension, that is satisfiable if and only if the original formula is satisfiable. The Herbrand extension may be infinite in length, but when it is not, it allows one to check the satisfiability of a first-order formula using methods designed for propositional logic.

Unfortunately, the propositional formula, even when finite, becomes rapidly more complex for larger problems. This difficulty can be alleviated somewhat by using a *partial instantiation* technique introduced by Jeroslow [147]. Two general approaches to partial instantiation exist. They are called primal and dual. The primal approach begins with a short propositional formula that is too weak, and adds clauses to it until it is unsatisfiable or the original first order formula is proved satisfiable. The dual approach begins with a propositional formula that is too strong, and weakens it until it is satisfiable or the original first-order formula is proved unsatisfiable.

On the other hand, the best-known method for checking satisfiability directly in first-order logic probably is the resolution method [223]. By thinking of all its computational drawbacks, we can understand the complexity of such task.

By imposing syntactical limitations, we can consider subsets, called *fragments*, of first-order logic. Partial instantiation methods can be adapted to take advantage of the structure of such fragments.

Because full first-order logic is undecidable [48], no finite algorithm can

always settle the satisfiability issue. In fact, first-order logic can be given a *complete axiomatization*, [105], i.e. there is a finite set of axioms from which every valid formula can be deduced. The point is that there is no finite algorithm that will always determine whether such a deduction exists. However, first-order logic is *semidecidable*, meaning that there is a procedure that will always terminate with a proof of unsatisfiability if a formula is unsatisfiable. If a formula is satisfiable, the procedure may never terminate with the indication of satisfiability. However, termination is guaranteed when the formula belongs to a decidable fragment of first-order logic, such as the Löwenheim-Skolem or the $\exists\forall$ fragments.

1.6 Implication and Inference

Given two formulas S_1 and S_2 defined on the same set of atomic propositions, if every model of S_1 is also a model of S_2, S_1 logically *implies* S_2, or, equivalently, S_2 is logically *implied* by S_1, or, moreover, S_2 is an *implication* of S_1. So far, in all the cases when we have an interpretation were S_1 is $True$, we know that S_2 also is $True$, without the need to study truth value of S_2 by starting from its atoms. Unfortunately, this definition does not give a practical way to determine whether a formula implies another. The idea of checking all truth assignments would be hopeless.

The fundamental problem of inference in logic is therefore to ascertain if the truth of one formula implies the truth of another. We need some inference method to detect implication. An inference method is a procedure which, given a set of formulas called *axioms*, is able to derive a set of new formulas called *theorems*. An inference method is sound if all the theorems that can derive are logically implied by the axioms. An inference method is complete if all the theorems which are logically implied by the axioms can be derived.

A clause, in a CNF formula, is called redundant if it is implied by the remaining clauses. A single clause C *absorbs* clause D if every literal in C appears also in D. Further, a clause D is logically implied by a single clause C if and only if C absorbs D or if D is a tautology. In a CNF formula therefore a clause will be redundant if it is absorbed by another. What makes inference so complex is the fact that redundancy can be deeply embedded. Fortunately, there is a simple principle that, on repeated application, reveals the embedded structure. This is the *principle of resolution* [223, 21].

In any CNF formula, suppose there are two clauses C and D with exactly one atomic proposition x_j appearing negated in C and posited in D. It is

then possible to resolve C and D on x_j. We call C and D the parents of the resolvent clause, which is formed by the disjunction of literals in C and D, excluding x_j and $\neg x_j$. The resolvent is an implication of the parent clauses.

As an illustration of this principle consider the two clauses.

$$\begin{array}{c} C = (\neg x_1 \vee x_3) \\ \underline{D = (x_1 \vee \neg x_2 \vee x_3)} \\ (\neg x_2 \vee x_3) \end{array}$$

The resolvent of C and D is the clause below the line. The resolution principle is a manifestation of case analysis, since if C and D have to be satisfied and (case 1) x_1 is true, then x_3 must be true; or if (case 2) x_1 is false then $(\neg x_2 \vee x_3)$ must be true. Hence every model for C and D is also a model for the resolvent.

A *prime implication* of a CNF formula S, is a clause that is an implication of S and is not absorbed by any clause that is an implication of S, except by itself. In the above example, the prime implications are

$$\{(\neg x_1 \vee x_3), (\neg x_2 \vee x_3)\}$$

A simple procedure for deriving all prime implications of S is to repeatedly apply the resolution principle while deleting all absorbed clauses. If we are left with an empty clause, the formula S has no model. Otherwise, we will be left with all the prime implications.

Resolution is a complete inference method for propositional calculus, in the sense that any clause implied by S is absorbed by one of the clauses obtained in the resolution process. This was originally proved in the case of DNF, where the analogous of resolution is called *consensus* (see [220]). We therefore have a complete inference mechanism for propositional logic based on purely syntactic techniques. However, the principle of resolution is not a computationally viable inference method for all but small examples, in the general case.

More computationally attractive is a weakened form of resolution, called *unit resolution*, in which one of the parent clauses is always taken to be a clause with exactly one literal. This weaker principle is complete only on special classes of propositions called *Horn* formulas. Horn formulas are defined to be made by Horn clauses, or *nearly-negative* clauses, which are clauses with any number of negated literals but at most one positive literal.

An alternate syntax for propositional logic that has been popularized in recent years by the Artificial Intelligence community is the syntax of rule-based systems. A rule system, or rule-based system, is a type of propositional logic in which all the formulas are rules. A rule is an if-then statement

of the form

$$(x_1 \wedge \ldots \wedge x_j) \Rightarrow (y_1 \vee \ldots \vee y_k)$$

where $\Rightarrow$ means *implies*, $x_1, \ldots, x_j$ are the antecedents, and $y_1, \ldots, y_k$ is the consequent. The above rule is equivalent to the clause

$$\neg x_1 \vee \ldots \vee \neg x_j \vee y_1 \vee \ldots \vee y_k$$

Thus if there are no antecedents ($j = 0$), the rule simply asserts the consequent. If there is no consequent ($k = 0$), the rule denies the conjunction of the antecedents. Also, it is clear that any clause can be written as a rule, just by putting the positive literals in the consequent and treating the negative literals (stripped of their negations) as antecedents. If there is at most one disjunct in the consequent, the rule is called Horn, and in fact the corresponding clause is a Horn clause.

Rules are often called "inference rules" in the AI literature, but they are not inference rules in the classical sense. They are simply formulas in a logical system, to which an inference rule may be applied to derive other formulas.

In rule systems the most common inference rule is *modus ponens*, also known as *forward chaining*:

$$\begin{array}{rcl} (x_1 \wedge \ldots \wedge x_j) & \Rightarrow & (y_1 \vee \ldots \vee y_k) \\ & \Rightarrow & x_1 \\ & \vdots & \\ & \Rightarrow & x_j \\ \hline \multicolumn{3}{c}{(y_1 \vee \ldots \vee y_k)} \end{array}$$

This means that one can infer the conclusion below the line from the premises above the line.

Another popular inference rule is *modus tollens*, also called *backward chaining*:

$$\begin{array}{rcl} (x_1 \wedge \ldots \wedge x_j) & \Rightarrow & (y_1 \vee \ldots \vee y_k) \\ y_1 & \Rightarrow & \\ & \vdots & \\ y_k & \Rightarrow & \\ \hline \multicolumn{3}{c}{(x_1 \vee \ldots \vee x_j) \Rightarrow} \end{array}$$

Modus ponens is obviously the same as a series of unit resolutions in

which the unit clauses are positive. For example, the inference

$$\begin{array}{rcl} (x_1 \vee x_2) & \Rightarrow & y \\ & \Rightarrow & x_1 \\ & \Rightarrow & x_2 \\ \hline & y & \end{array}$$

can be accomplished by resolving $\neg x_1 \vee \neg x_2 \vee y$ with x_1 to obtain $\neg x_2 \vee y$, and resolving the latter with x_2 to obtain y. Similarly, modus tollens is the same as a series of unit resolutions in which the unit clauses are negated.

Since unit resolution is not a complete inference method for propositional logic, forward and backward chaining are not complete for a rule system. The following valid inference, for example, cannot be obtained using these two inference rules:

$$\begin{array}{rcl} x & \Rightarrow & (y_1 \vee y_2) \\ y_1 & \Rightarrow & z \\ y_2 & \Rightarrow & z \\ \hline & x \Rightarrow z & \end{array}$$

However, forward and backward chaining are a complete refutation method for a Horn rule system, since unit refutation is complete for Horn clauses.

1.7 The Role of Satisfiability

We have now defined the following three basic inference problems in propositional logic.

- Is a given formula satisfiable?
- Is a given formula a tautology?
- Does one formula imply another?

All of three can be solved in several ways, for example by enumeration. Since we have 2^n truth assignments for n propositions, this is obviously impracticable. However, the fact that one method can solve all three problems is indicative of a possible relationship between them. In fact, we have

Proposition 1.1 *The following statements are equivalent:*

1. *S_1 implies S_2.*
2. *$(\neg S_1 \vee S_2)$ is a tautology.*

3. $(S_1 \wedge \neg S_2)$ *is unsatisfiable.*

All of the three problems above can therefore be reduced to the problem of determining whether a given formula evaluates to $True$ (is satisfied) for some assignment of truth values to the atomic propositions. This is called the *satisfiability* problem.

We therefore understand that the satisfiability problem is the central problem of inference in propositional logic. For this reason, when above we showed the syntactical transformation rules to obtain normal form, our care was to modify a formula without affecting its models.

The propositional satisfiability problem, SAT for short, was the first problem shown to be NP-complete [55, 155]. The problem is therefore *hard*, in the sense that no known algorithm can always solve it in an amount of time that is a polynomial function of the problem size [98]. However, clever algorithms can rapidly solve many SAT formulas of practical interest. There has been great interest in designing efficient algorithms to solve most SAT formulas.

Moreover, the large family of computationally intractable NP-complete problems have been identified as central to a number of areas in computing theory and engineering. This means that, at least theoretically, all of those problems can be polynomially reduced to a SAT problem, hence solved by solving SAT. In practice, SAT is fundamental in solving many problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design automation, computer architecture design, and computer network design. Therefore, methods to solve SAT formulas play an important role in the development of efficient computing systems.

In the next chapter we analyze in greater detail many solutions methods for such problem.

1.8 Constraint Satisfaction Problems

Logic programming, in the sense of application of optimization methods to logic deduction, can be extended to constraint programming, which deals with constraint satisfaction problems. A *constraint satisfaction* problem, CSP for short, can be view as a generalization of SAT.

In the mid-1980's, constraint programming was developed as a computer science technique, combining developments in Artificial Intelligence community with the development of new computer programming languages. For

those familiar with mathematical programming, one confusions comes from the fact that the word *programming* is differently used in the two field of Computer Science and Operation Research.

On one hand, the field of mathematical programming arose from its roots in linear programming. In Dantzig's seminal textbook [62], linear programming is introduced by describing some different planning problems. Such problems were programming problems in the sense of program, or schedule, of activities. Therefore, the term *program*, previously used to describe a plan of activities, became associated with a specific mathematical problem in the operations research literature.

On the other hand, in constraint programming (often called constraint logic programming), the word *programming* refers to computer programming [159]. A computer program can be viewed as a plan of action of operations of the computer, and hence the common concept of a plan is shared with the origins of linear programming.

Hence, a constraint program is not a statement of a problem as in mathematical programming, but is rather a computer program that indicates a method for solving a particular problem.

A constraint satisfaction problem [133], CSP for short, consists in a set of n decision variables

$$x_1, x_2, \ldots, x_n$$

Each one of those x_i has its domain D_i of allowable values. The domain of a decision variable can be any possible set of symbols, for example, even numbers, names, etc. There is no restriction on type of each decision variable, and hence decision variables can take on integer values, real values, set elements, or even subsets of sets.

A constraint $c(x_1, x_2, \ldots, x_n)$ is a mathematical relation, i.e. a subset S of the set $D_1 \times D_2 \times \ldots \times D_n$ such that if $(x_1, x_2, \ldots, x_n) \in S$, then the constraint is said to be satisfied. A constraint is therefore represented by mathematical function

$$f : D_1 \times D_2 \times \ldots \times D_n \rightarrow \{0, 1\}$$

such that $f(x_1, x_2, \ldots, x_n) = 1$ if and only if $c(x_1, x_2, \ldots, x_n)$ is satisfied.

Using functional notation, we can define a constraint satisfaction problem (CSP) as: Given a set of n domains $\{D_1, D_2, \ldots, D_n\}$ and a set of m constraints $\{f_1, f_2, \ldots, f_m\}$, find a set of n values $\{x_1, x_2, \ldots, x_n\}$ such that

$$\begin{array}{ll} f_j(x_1, x_2, \ldots, x_n) = 1 & j = 1, \ldots, m \\ x_i \in D_i & i = 1, \ldots, n \end{array}$$

Note that this problem is only a feasibility problem, and that no objective function is defined. The functions f_i do not necessarily have closed mathematical forms, and can simply be defined by providing the set S described above. In other words, a solution to a CSP is simply a set of values of the variables such that the values are in the domains of the variables, and all of the constraints are satisfied.

Each constraint must have a form that is easy to evaluate, so any difficulty in solving such a problem comes from the interaction between the constraints and the need to find a setting for the variables that simultaneously satisfies all the constraints [217]. In a SAT instance, each constraint is expressed as a clause, making SAT a special case of the constraint satisfaction problem (see Figure 1).

An order-l constraint indicates the compatibility (i.e., consistency or inconsistency or conflicting measure) among l variables for a given variable assignment. The variables *conflict* if their values do not satisfy the constraint. In practice, two frequently used constraints are *unary constraints* imposed on a single variable and *binary constraints* imposed on a pair of variables.

Constraint satisfaction problems are extremely common. Most NP-complete problems are originally stated as constraint satisfaction problems. Indeed, the proof that a problem is NP-complete implies an efficient way to transform the problem into a constraint satisfaction problem.

For a few special forms of the constraint satisfaction problem there exist algorithms that solve such formulas in polynomial worstcase time. When no polynomialtime algorithm is known for a particular form of constraint satisfaction problem, it is common practice to solve such formulas with a *search* algorithm.

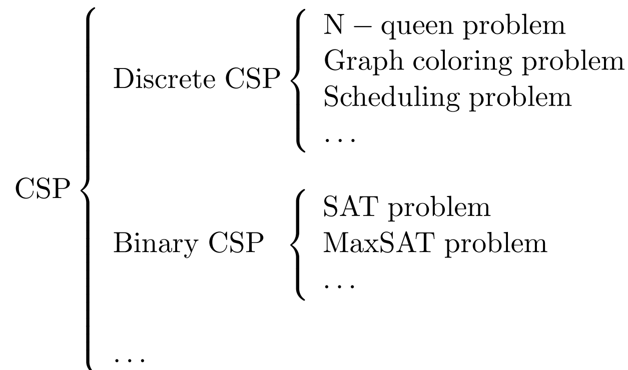

Figure 1. Some examples of the constraint satisfaction problem (CSP).

1.9 Combinatorial Optimization Models

In order to apply the paradigms of mathematical programming to inference in logic, we can bridge the different worlds to which these subjects belong. In the previous sections, we formulated the basic inference problems of propositional logic using symbolic valuations of propositions as either $True$ or $False$. Mathematical programming, however, works with numerical valuations. These "formulations" of inference in propositional logic are what we explore in this section.

The solubility of systems of linear inequalities over the real (rational) numbers is closely related to the problem of linear programming. A standard approach is to deal with solubility as a special case of optimization over linear inequalities. Integer programming deals with the linear programming problems with the added restriction that some (possibly all) variables may take values restricted to the integers. By introducing suitable bounds on the variables, it is possible to restrict variables to only take values in the nonnegative integers or even just values of 0 and 1.

This "boolean" restriction captures the semantics of propositional logic since the values of 0 and 1 may be naturally associated with $False$ and $True$. In integer programming, all the inequality constraints have to be satisfied simultaneously (in conjunction) by any feasible solution. It is natural therefore to attempt to formulate satisfiability of CNF formulas as integer programming problems with clauses represented by constraints and atomic propositions represented by 0–1 variables [43].

A positive atom (x_i) corresponds to a binary variable (x_i), and a negative atom ($\neg x_i$) corresponds to the complement of a binary variable ($1 - x_i$). Consider, for example, the single clause

$$x_2 \vee \neg x_3 \vee x_4$$

The satisfiability of this clause corresponds to the solubility of the following inequality over (0,1) variables.

$$x_2 + (1 - x_3) + x_4 \geq 1$$

The satisfiability of the formula

$$(x_1) \wedge (x_2 \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_5)$$

is equivalent to the existence of a solution to the system:

$$\begin{aligned} x_1 &\geq 1 \\ x_2 + (1 - x_3) + x_4 &\geq 1 \\ (1 - x_1) + (1 - x_4) &\geq 1 \\ (1 - x_2) + x_3 + x_5 &\geq 1 \\ x_1, \ldots, x_5 &\in \{0, 1\} \end{aligned}$$

It is conventional in mathematical programming to clear all the constants to the right-hand side of a constraint. Thus a clause C_j is represented by $a_j x \geq b_j$, where for each i, a_{ji} is $+1$ if x_i is a positive literal in C_j, is -1 if $\neg x_i$ is a negative literal in C_j, and is 0 otherwise. Also, b_j equals $1 - n(C_j)$, where $n(C_j)$ is the number of negative literals in C_j. We shall refer to such inequalities as clausal. So the linear inequalities converted to clausal form are given by

$$\begin{aligned} x_1 &\geq 1 \\ x_2 - x_3 + x_4 &\geq 0 \\ -x_1 - x_4 &\geq -1 \\ -x_2 + x_3 + x_5 &\geq 0 \\ x_1, \ldots, x_5 &\in \{0, 1\} \end{aligned}$$

In general, satisfiability in propositional logic is equivalent to solubility of

$$Ax \geq b, \qquad x \in \{0, 1\}^n$$

where the inequalities of $Ax \geq b$ are clausal. Notice that A is a matrix over $\{0, \pm 1\}$, and each b_j equals 1 minus the number of -1's in row j of the matrix A.

We are therefore looking for an extreme point of the unit hypercube in $\mathbb{R}^n$ that is contained in all the half–spaces defined by the clausal inequalities. This is a spatial or geometric embedding of inference.

The intersection of clausal half-spaces defines a convex polyhedron. If the box constraints $0 \leq x_i \leq 1,\ (i = 1, 2, \ldots, n)$ are added, we obtain a bounded polyhedron or a polytope. Thus satisfiability is a special case of checking for a integer lattice (Z^n) point in a polytope defined implicitly by linear inequalities. The latter is exactly the feasibility problem of integer (linear) programming. In optimizing a linear objective function over constraints, defined by linear inequalities and integer restrictions on variables, one must certainly also contend with checking the feasibility of the constraints. Thus it is natural to consider the optimization version of integer programming as a more general model than the feasibility version.

In clausal inequalities the right–hand side in any case (i.e. also when the inequalities are not satisfied) can not be greater than 1 plus the value of the left–hand side. We can therefore guarantee the feasibility of the above system by introducing an artificial binary variable x_0. This allows to pose the feasibility version of integer programming as a special case of optimization. Satisfiability of the above formula can also be tested by solving the optimization problem

$$\begin{array}{lll} \min & x_0 & \\ \text{s.t.} & x_0 + x_1 & \geq \ 1 \\ & x_0 + x_2 - x_3 + x_4 & \geq \ 0 \\ & x_0 - x_1 - x_4 & \geq \ -1 \\ & x_0 - x_2 + x_3 + x_5 & \geq \ 0 \\ & x_i \in \{0,1\}, \qquad i = & 0,1,\ldots,5 \end{array}$$

The original formula (2.15) is satisfiable if and only if the optimization problem above is solved with x_0 at 0. The variable x_0 is called the artificial variable and the optimization problem a phase I construction in integer programming terminology. The optimization formulation is useful since the objective function provides a mechanism for directed search of truth assignments (0-1 values for $x_1, \ldots, x_n$) that satisfy the formula. In general, the phase I construction is of the form

$$\begin{array}{ll} \min & x_0 \\ \text{s.t.} & x_0 e + Ax \geq b \\ & x_i \in \{0,1\}, \quad i = 0,1,\ldots,n \end{array}$$

where $Ax \geq b$ represents the original clausal inequalities and e is a column of 1's. This problem is said to be in the form of *generalized set covering* problem.

Inference in the form of implications can also be modeled as integer programming problems. An indirect route would be to transform the implication question to a satisfiability problem and then follow the Phase I construction discussed above. However, direct formulations are also possible [42]. Let us consider an implication problem of the form: Does a formula S_1 imply another formula S_2? For simplicity let us first assume that S_1 is in CNF and S_2 is given by a single clause C. The corresponding optimization model for testing the implication is

$$\begin{array}{ll} \min & cx \\ \text{s.t.} & Ax \geq b \\ & x_i \in \{0,1\}, \quad i = 1,\ldots,n \end{array}$$

where c is the incidence vector of clause C, and $Ax \geq b$ are the clausal inequalities representing S_1. If this optimization yields a minimum value $1 - n(C)$ or larger, we have established the valid implication of S_2 by S_1. Otherwise the implication does not hold.

Now, let us consider a situation where S_1 represents a consistent knowledge base in CNF with clausal inequalities $Ax \geq b$ and $\neg S_2$ a general formula in CNF with clausal inequalities $Bx \geq d$. To test if S_2 is logically implied by S_1 we solve

$$\begin{array}{ll} \min & x_0 \\ \text{s.t.} & x_0 e + Bx \geq d \\ & Ax \geq b \\ & x_i \in \{0,1\}, \quad i = 0,1,\ldots,n \end{array}$$

S_1 implies S_2 if and only if the minimum value of x_0 is 1. Notice that the artificial x_0 is not associated with $Ax \geq b$ since we have assumed that S_1 is consistent (satisfiable).

If we are interested not only in finding if the set of clauses is all simultaneously satisfiable, but also in finding the maximum number of clauses that we can simultaneously satisfy, we have a problem called the maximum satisfiability problem, max-SAT for short.

This can be modeled by adding, instead of one single artificial variable x_0, an artificial binary variable y_j for each clause. We obtain the following, where y is the vector $\{y_1, \ldots, y_m\}$ and I is the identity matrix.

$$\begin{array}{ll} \min & \sum_{j=1}^{m} y_j \\ \text{s.t.} & Iy + Ax \geq b \\ & x_i \in \{0,1\}, \quad i = 1,\ldots,n \\ & y_j \in \{0,1\}, \quad j = 1,\ldots,m \end{array}$$

When we need to put an artificial variable y_j at 1, this means that the corresponding clause C_j was not satisfied. Note that max-SAT is a generalization of SAT, since the solution of max-SAT gives also the solution of SAT. If the number of simultaneously satisfiable clauses is equal to the number of clauses in the instance, the problem is satisfiable. Otherwise, it is unsatisfiable.

So far, we can apply many integer programming technique to solve inference problems, the relaxation of integrality to begin with, as covered in detail in [43]. It becomes moreover possible to find tight correspondences between easy cases in propositional logic and polyhedral properties in combinatorics.

Chapter 2

Algorithms for Propositional Satisfiability

2.1 An Overview

Due to the motivations exposed in chapter 1, SAT has historically been considered as a sort of training field for algorithms designers. There have been in fact innumerable SAT solvers proposed, based on the most different techniques. We will try to give here an overview of the various approach, with, of course, some simplifications and omissions.

As told, an instance of the CNF satisfiability (SAT) problem [55, 98] consist in:

- A set of n propositions (or variables): $x_1, x_2, \ldots, x_n$.
- A set of m clauses: $C_1, C_2, \ldots, C_m$. Each clause consists of only propositions (x_i) or negated propositions ($\neg x_i$) combined by just logical *or* ($\vee$) connectives. All the clauses connected by just logical *and* ($\wedge$) connectives form the CNF formula.

The goal is to determine whether there exists an assignment of truth values to variables (and, if yes, which is) that makes the CNF formula $True$.

An example of satisfiable instance can be the following, which is $True$ for ($x_1 = True, x_2 = False$).

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$$

An example of unsatisfiable instance can be the following. No truth assignment can make it $True$.

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$$

In order to find the solution of such problem we perform a *search.* Search is the process of systematic exploration of a set (or space) of possible solutions. A search algorithm is said to be complete if it is guaranteed (given enough time) to find a solution if it exists, or report lack of solution otherwise. In addition, there are various schemes of incomplete search. These algorithms usually scale better than complete ones on large problems, but they run the risk of getting stuck in local optima and can therefore not guarantee to ever find the globally best solution.

Some complete algorithms are based on *resolution*, such as the precursor algorithm of Davis–Putnam [65]. They work by reducing the original CNF formula to one with fewer variables that is satisfiable if and only if this is so for the original CNF. To eliminate one variable x_i, we replace all clauses containing x_i with all possible clauses derivable by resolutions from the clauses containing x_i. The process is repeated to exhaustion or until an empty clause is generated. The original formula is not satisfiable if and only if an empty clause is a resolvent. The total number of steps (or resolvents) can be extremely large compared to the number of clauses in the original formula.

Many other complete methods are *enumeration* algorithms, such as the so-called Davis-Putnam-Loveland, or Davis-Logemann-Loveland [64]. They are based on tree search and splitting, that is, they implicitly define a tree where internal nodes correspond to partial solutions, branches are choices (partitioning the search space), and leaf nodes are complete solutions. At every internal node, a variable x_i is selected from a formula, and the formula is replaced by one subformula for each of two possible truth assignments to x_i (*splitting*). Each subformula has all the clauses of the original except those satisfied by the assignment to x_i and otherwise all the literals of the original formula except those falsified by the assignment. The original formula has a satisfying truth assignment if and only if either subformula has one. If a subformula is proved unsatisfiable, *backtrack* is performed.

The third family of complete algorithms uses *integer programming* methods, such as Branch-and-Cut procedures [43]. Such methods are based on the transformation of the satisfiability problem in the problem of the feasibility of a system of linear inequalities with binary variables, and then in the transformation of the latter in a binary optimization problem.

More complete algorithms exists, which do not stay in the above categories. The following are just two noteworthy examples. Binary Decision Diagrams [35], and their variants, which allows to represent the set of all solutions by means of a directed acyclic graph. The Analysis algorithm [250], which is able, after an analysis of the instance, of providing a tailored

solution algorithm together with a performance guaranteed.

Moreover, there are various schemes of incomplete search, of which *local search* techniques are examples. These iterative procedures maintain one or more complete solutions which are gradually improved by minor changes. Some of these methods may in addition use *Global optimization* techniques in order to escape from local minima.

Of course, such classification cannot take into account all of the features of all the various algorithms. Alternative classifications could consider discrete as opposed to continuous models, or sequential as opposed to parallel algorithms, etc.

Anyway, we will analyze in further detail the six following classes. This classification is inspired by [118] and [250]:

- Resolution
- Enumeration (Splitting and Backtracking)
- Integer Programming Techniques
- Local Search
- Global Optimization
- Other Techniques

Note, however, that state of the art solvers often combine more than one of the above techniques. Moreover, there are special subclasses of SAT which are solvable in polynomial time, by means of specialized algorithms.

2.2 Resolution

Recursive replacement of a formula by one or more other formulas, the solution of which implies the solution of the original formula, is an effective paradigm for solving CNF formulas. Recursion continues until one or more primitive formulas have been generated and solved to determine the satisfiability of the original. Most recursive SAT algorithms use the following primitive conditions to stop the recursion:

1. formulas with an empty clause have no solution.
2. formulas with no clauses have a solution.

3. formulas with no variables (i.e., all variables have been assigned values) are trivial.

One way to achieve this is through splitting (see below), while another effective paradigm is based on resolution [223].

Given two clauses $C_1 = (v \vee x_1 \vee \ldots \vee x_j)$ and $C_2 = (\neg v \vee y_1 \vee \ldots \vee y_k)$, the resolvent of C_1 and C_2 is the clause $(x_1 \vee \ldots \vee x_j \vee y_1 \vee \ldots \vee y_k)$, that is, the disjunction of C_1 and C_2 without v or $\neg v$. The resolvent is a logical consequence of the logical and of the pair of clauses. Resolution is the process of repeatedly generating resolvents from original clauses and previously generated resolvents until either the null clause is derived or until no more resolvents can be created [223]. In the former case (a refutation) the formula is unsatisfiable and in the latter case it is satisfiable. For some formulas the order in which clauses are resolved can have a big effect on how much effort is needed to solve it. The worstcase associated with the best possible order (the order is selected after the formula is given) has received considerable study [94, 251, 123, 253]. These studies all used unsatisfiable formulas, but where this is not obvious to the resolution algorithm. Eventually a much stronger result was shown: nearly all random l–SAT formulas need exponential time when the ratio of clauses to variables is above a constant (whose value depends on l) [50]. The constant is such that nearly all of the formulas in this set have no solution.

Although resolution can be applied to SAT, the main reason for interest in resolution is that it can be applied to the more difficult problem of solving sentences of first order predicate logic. There is a vast literature on that subject [17].

A number of restrictions and at least one extension to resolution have been proposed and applied to CNF formulas. Restrictions aim to shorten the amount of time needed to compute a resolution derivation by limiting the number of possible resolvents to choose from at each resolution step. The extension aims to provide shorter derivations than possible for resolution alone by adding equivalences which offer more clauses on which to resolve. A treatment of these refinements can be found in [38], Chapter 4. We mention here a few of these.

2.2.1 Set of Support

Split a given formula into two sets of clauses $\mathcal{F}_1$ and $\mathcal{F}_s$ such that $\mathcal{F}_1$ is satisfiable. Permit only resolutions involving one clause either in $\mathcal{F}_s$ or an appropriate previous resolvent. Set $\mathcal{F}_s$ is called the support set [271]. This

restriction can be useful if a large portion of the given formula is easily determined to be satisfiable.

2.2.2 P-Resolution and N-Resolution

If one of the two clauses being resolved has all positive literals (resp. negative literals), then the resolution step can be called a P-resolution (resp. N-resolution) step. In P-resolution (resp. N-resolution) only P-resolution (resp. N-resolution) steps are used. Clearly there is great potential gain in this restriction due to the usually low number of possible resolvents to consider at each step. However, it has been shown that some formulas solved in polynomial time with general resolution require exponential time with N-resolution.

2.2.3 Linear Resolution

We have linear resolution if every resolution step except the first involves the most recently generated resolvent (the other clause can be a previous resolvent or a clause in the given formula). Depending on the choice of initial clause and previous resolvents it is possible not to complete a refutation.

2.2.4 Regular Resolution

Regular resolution [251] proceeds by constructing a resolution tree where, in every path, no variable is eliminated more than once.

2.2.5 Davis-Putnam Resolution

Once all the resolvents with respect to a particular variable have been formed, the clauses of the original formula containing that variable can be dropped [65]. Doing this does not change the satisfiability of the given formula, but it does change the set of solutions to the extent that the value of that variable is no longer relevant. When dropping clauses, it is natural to first form all the resolvents for one variable, then all the resolvents for a second variable, and so on. When doing resolution in this way, it is easy to find one satisfying assignment if the formula is satisfiable. At the next to last step the formula has just one variable, so each value can be tested to see which one satisfies the formula (perhaps both will). Pick a satisfying value and plug it into the formula from previous step, converting it into a one variable formula. Solve that formula and proceed backwards in this manner until an assignment for all variables is found.

2.2.6 Extended Resolution

For any pair of variables x, y in a given formula $\mathcal{F}$, create a variable z not in $\mathcal{F}$ and append the following expression to $\mathcal{F}$

$$(z \vee x) \wedge (z \vee y) \wedge (\neg z \vee \neg x \vee \neg y)$$

Judicious use of such extensions can result in polynomial size refutations for problems that have no polynomial size refutations without extension [251].

2.2.7 Subsumption

Subsumption, or absorption, is a strategy which help reducing the time to compute a resolution derivation. If the literals in one clause are a subset of those in another clause, then the smaller clause is said to subsume the larger one. Any assignment of values to variables that satisfies the smaller clause also satisfies the larger one, so the larger one can be dropped without changing the set of solutions. Subsumption is of particular importance in resolution algorithms because resolution tends to produce large clauses.

2.2.8 Pure Literals

A literal is pure if all its occurrences are either all positive or all negative. No resolvents can be generated by resolving on a pure literal, but all clauses containing a pure literal can be removed without loss. An important improvement to the basic resolution algorithm is to first remove clauses containing pure literals (before resolving on non-pure literals) [65].

2.3 Enumeration trough Splitting and Backtrack

Backtracking algorithms are based on splitting. During each iteration, the procedure selects a variable and generates two subformulas by assigning the two values, true and false, to the selected variable. In each subformula, those clauses containing the literal which is true for the variable assignment are erased from the formula, and those clauses which contain the literal which is false have that literal removed. Backtrack algorithms differ in the way they select which variable to set at each iteration. The unit clause rule, the pure literal rule, and the smallest clause rule, are three most common ones. We state each algorithm informally.

The flow of control in splitting-based algorithms is often represented by a search tree. The root of the tree corresponds to the initial formula.

The internal nodes of the tree correspond to subformulas that cannot be solved directly, whereas the leaf nodes correspond to subformulas that can be solved directly. The nodes are connected with arcs that can be labeled with variable assignments.

2.3.1 Simple Backtracking

If the formula has an empty clause (a clause which always has value false) then exit and report that the formula has no solution. If the formula has no variables, then exit and report that the formula has a solution. (The current assignment of values to variables is a solution to the original formula.) Otherwise, select the first variable that does not yet have a value. Generate two subformulas by assigning each possible value to the selected variable. Solve the sub formulas recursively [29]. Report a solution if any subformula has a solution, otherwise report no solution.

2.3.2 Unit Clause Backtracking

This algorithm [214] is the same as simple backtracking except for how variables are selected. If some clause contains only one of the unset variables then select that variable and assign it a value that satisfies the clause containing it; otherwise, select the first unset variable.

In practice, this improved variable selection often results in much faster backtracking [19].

2.3.3 Clause Order Backtracking

This algorithm is the same as simple backtracking except for how variables are selected [39]. If this setting does not solve the formula, then select the first clause that can evaluate to both true and false depending on the setting of the unset variables. Select variables from this clause until its value is determined.

By setting only those variables that affect the value of clauses, this algorithm sometimes avoids the need to assign values to all the variables. The algorithm as stated finds all the solutions, but in a compressed form. The solutions come in cylinders, where some variables have the value *don't care*. Thus, a single solution with unset variables represents the set of solutions obtained by making each possible assignment to the unset variables.

2.3.4 Probe Order Backtracking

This algorithm [217] is the same as simple backtracking except for how clauses are selected. Temporarily set all the unset variables to some predetermined value. Select the first clause that evaluates to false with this setting. Return previously unset variables back to unset and continue as in clause order backtracking.

For practical formulas one should consider adding the following five refinements to probe order backtracking: stop the search as soon as one solution is found, carefully choose the probing sequence instead of just setting all variables to a fixed value [173, 238], probe with several sequences at one time [40], carefully select which variable to set, use resolution when it does not increase the input size [84].

Franco [83] noticed that a random assignment solves a nonzero fraction of the formulas in the average l–SAT model when pn is large compared to $\ln m$. Simple uses of that idea does not lead to good average time [217], but combining the idea with clause order backtracking leads to probe order backtracking, which is fast when pn is above $\ln m$. Probe order backtracking appears to have some similarities to one method that humans use in problem solving in that it focuses the algorithm's attention onto aspects of the problem that are causing difficulty, i.e., setting variables that are causing certain clauses to evaluate to false.

2.3.5 Shortest Clause Backtracking

This algorithm is the same as clause order backtracking except for the clause selected. In this case, select the shortest clause. The corresponding idea for constraint satisfaction is to first set a variable in the most constraining relation. This idea is quite important in practice [19].

2.3.6 Jeroslow-Wang

A backtrack search can sometimes be terminated early by checking whether the remaining clauses can be solved by a Linear Programming relaxation. An implementation of this idea can be expensive. Jeroslow and Wang have proposed a simpler and effective technique that is similar in spirit [146]. The idea is, before splitting, to apply a procedure that iteratively chooses the variable and value which, in some sense, maximizes the chance of satisfying the remaining clauses. The procedure does not backtrack and is, therefore, reasonably fast. Assignments determined by the procedure are temporarily added to the current partial truth assignment. If the procedure succeeds in

eliminating all clauses then the search is terminated and the given formula is satisfiable. Otherwise, the procedure fails, control is passed to the split, temporary assignments are undone, and backtracking resumes.

Such temporary assignment (i.e. fixing a variable to a truth value) corresponds in choosing a literal. At each iteration, we choose literal l which, for all the clauses C_j belonging to the current subformula $\mathcal{S}$, maximizes the weight function

$$w(\mathcal{S}, l) = \sum_{j:\, l \in C_j} 2^{-|C_j|}$$

The length of clause C_j, denoted $|C_j|$ above, is the number of literals that are not falsified by the current partial assignment and the sum is over clauses that are not satisfied by the current partial assignment.

The weight given above may be compared to that given by Johnson in [148], and to Mom's heuristic (prefer the proposition(s) having Maximum Occurrences in clauses of Minimum Size) [211].

2.3.7 Backtracking and Resolution

Some algorithms have adapted ideas inspired by resolution to splitting algorithms. For example, from the resolution view point, pure literals are interesting in that they lead to a single subformula that is no more complex than the original formula, while from the perspective of splitting, pure literals lead to two subformulas, but the solutions to the subformula where the literal has the value false are a subset of the one where the literal has the value true. Therefore, the original formula has a solution if and only if the formula associated with the true literal does.

2.3.8 The Pure Literal Rule Algorithm

Select the first variable that does not have a value. (If all variables have values, then the current setting is a solution if it satisfies all the clauses.) If some value of the selected variable results in all clauses that depend on that variable having the value true, then generate one subformula by assigning the selected variable the value that makes its literals true. Otherwise, generate a subformula for both values of the selected variable. Solve the one or two subformulas [108].

2.3.9 Clause Area

A clause with l distinct literals leads to the fraction $1/2^l$ of the possible variable settings not being solutions. One can think of the clause as blocking

out area $1/2^l$ on the Venn diagram for the formula. Iwama showed that combining this idea with inclusion-exclusion and careful programming leads to an algorithm which runs in polynomial average time [144] when

$$p > \sqrt{(\ln m)/n}$$

Where n and m are, as usual, number of variables and clauses, and p is a parameter used to generate the instances. If the sum of the area of all clauses is less than 1, then some variable setting leads to a solution. This idea works particularly well with shortest-clause backtracking since that algorithm tends to eliminate short clauses. See [88] for a probabilistic analysis of this idea. No average-time analysis has been done.

2.3.10 Branch Merging

This is complementary to preclusion. Backtracking is frequently used on problems such as the n-queens problem where there is a known symmetry group for the set of solutions. In such cases many search trees possess equivalent branches which can be merged to reduce search effort [19, 266]. The use of the symmetry group can greatly speed up finding the solutions. See [41] for examples from the field of group theory. Brown, Finklestein, and Purdom [28] gave additional problems that arise in making the backtracking techniques work with a backtracking algorithm which needs to set variables in different orders on different branches of the search tree.

2.3.11 Search Rearrangement

This is also known as most-constrained search or non-lexicographic ordering search. When faced with several choices of extending a partial solution, it is more efficient to choose the one that offers the fewest alternatives. That is, nodes with fewer successors should be generated early in the search tree, and nodes with more successors should be considered later. The vertical (variable) ordering and horizontal (value) ordering are special cases of search rearrangement [91, 214, 243]. The rule used to determine which variable to select next is often called the branching rule.

Many researchers are actively investigating the selection of branching variables in the DP procedures. Hooker studied the branching rule and its effect with respect to particular problem instances [135]. Böhm and Speckenmeyer experimented with branching effect with a parallel DP procedure implemented on an MIMD machine [22]. Boros, Hammer, and Kogan

developed branching rules that aim at the fastest achievement of q-Horn structures [26].

2.3.12 2-SAT Relaxation

In SAT problem formulation, very frequently in practical applications, many of the constraints will be coded as 2-SAT clauses, i.e. clauses with only 2 literals. Since 2-SAT is an easy class which can be solved with fast polynomial time algorithms, an important heuristic to SAT problem solving is to first solve 2-SAT clauses. This fast operation can significantly reduce the search space. The truth assignment to the rest of the variables can be handled with a DP procedure. This idea has been used in SAT solver Stamm [40], Gallo and Pretolani's 2-SAT relaxation [211], and Larrabee's algorithm [167]. Similar ideas to solving 2-SAT clauses were developed. Eisele's SAT solver uses a weighted number of occurrences whereas occurrences in 2-SAT clauses count more than other occurrences [40]. Dörre further added a limited amount of forward checking to quickly determine 2-SAT formulas in [40].

2.3.13 Horn Relaxation

Horn clauses, or nearly negative clauses, are clauses with any number of negated literals but no more than one positive literal. Since Horn-SAT is an easy class which can be solved with fast polynomial time algorithms, techniques were developed that use Horn-SAT relaxation in satisfiability testing [59, 96]. In Crawford's Tableau, Horn clauses are separated from non Horn clauses. Based on the DPL procedure, Tableau applies in priority the unit clause rule and if necessary branches on a variable selected in the non Horn clauses using three successive heuristics.

2.3.14 Backmarking and Backjump

When a failure is observed or detected, the algorithm simply records the source of failure and jumps back to the source of failure while skipping many irrelevant levels on the search tree [99]. The more effective one's search rearrangement is, the less need there is for backjumping. Good search orders tend to be associated with the source of failure being one level back.

2.3.15 Conflict Directed Backjumping

Instead of a simple backtracking to the last element in the stack of open splittings, conflict directed backjumping [212] backs up through this abstract stack in a nonsequential manner, skipping stack frames where possible for efficiency's sake. Its mechanics involve examining all assignments made in open splittings, not just last assignments, so it is more complicated than usual backtrack.

2.3.16 Backtracking with Lookahead

A lookahead processor is a preprocessing filter that prunes the search space by inconsistency checking [121, 120, 180, 195]. Backtracking with lookahead processing is performed by interplaying a depth-first tree traversal and a lookahead tree pruning processor that deletes nodes on the search tree whose value assignments are inconsistent with those of the partial search path. Techniques in this class include partial lookahead, full lookahead [121], forward checking [129], networkbased heuristics [180, 69], and discrete relaxation [120].

2.3.17 Backtrack with Lookback

When ever a contradiction is derived, we know some variable has both truth values excluded. In order to avoid the same contradiction for the remainder of the search, the procedure learns a new clause by resolving the two respective clauses containing the above variable; then it backs up to the most recently assigned variable in such new clause (backjumping). This new clause is called *nogood*, or *working reason* for the failure. Its generation is called *learning*, and corresponds to a lookback process.

Unrestricted learning records every derived clause exactly as if it was a clause from the underlying instance, allowing it to be used for the remainder of the search. Because the overhead of unrestricted learning is high, there exist restricted learning schemes [14]. *Size bounded* learning of order i retains indefinitely only those derived clauses containing i or fewer variables. *Relevance bounded* learning of order i maintains any clause that contains at most i variables whose assignments have changed since the reason was derived.

2.3.18 Unit propagation with delayed unit subsumption

At every splitting, we usually simplify the formula by performing unit propagation. This consists in unit resolution and unit subsumption. According to [278], the procedure is faster when unit subsumption is not performed at every splitting, but delayed and performed only under special conditions. Different kinds of delaying for subsumption can be performed, see for instance [33].

2.3.19 Backtracking for Proving Non-Existence

Dubois, Andre, Boufkhad, and Carlier proposed a complete SAT algorithm, C-SAT [76]. The C-SAT was developed for the proof of the non-existence of a solution. The algorithm uses a simple branching rule and a local processing at the nodes of search trees (to detect further search path consistency and make search decision). It performed efficiently on some DIMACS benchmarks.

2.3.20 Intelligent Backtracking

This is performed directly to the variable that causes the failure, reducing the effect of thrashing behavior. Methods in this category include dependency directed backtracking [241, 70], revised dependency directed backtracking [205], simple intelligent backtracking [92], and a number of simplifications [67]. Freeman [90] present an intelligent backtracking algorithm, POSIT, for PrOpositional SatIstiability Testbed. In this algorithm he used Mom's heuristic (prefer the proposition(s) having Maximum Occurrences in clauses of Minimum Size), detecting failed literals, and minimizing constant factors to speed up backtracking search.

2.3.21 Macro Expansion

In some applications of backtracking that require relatively little storage, this method can be used to decrease the running time of the program by increasing its storage requirements. The idea is to use macros in assembly language in such a way that some work is done at assembly time instead of many times at run time. This increases the speed at which nodes are processed in the tree [19].

2.3.22 Backtrack Programming

Much work has focused on developing a new programming language for backtracking search. This includes the sequential Prolog programming language [52, 242], Prolog with intelligent backtracking scheme [34], and logic programming [133].

2.3.23 Special-Purpose Architectures

Special-purpose hardware machines were built to prune search space [189], perform backtracking search, and do AI computations [257].

2.3.24 Branch and bound

Also known as ordered depth-first search. Select a variable. For each possible value of the variable generate a subformula and compute some quick to compute upper bound on the quality of the solution of the sub formula. Solve recursively all subformulas except those that have a cost above that of the best solution that has been found so far. Branch and bound is recognized as a generalization of many heuristic search procedures such as [2, 264, 167].

2.3.25 Some Remarks on Complexity

The worstcase time for all known SAT algorithms is exponential in the first power of the input size. The naive algorithm that tries every variable setting requires time 2^n for n variable formulas. For l–SAT, the best known bound on worstcase complexity has been worked down from 1.618^n [194] to slightly below 1.5^n obtained by Schiermeyer [230]. Other work on the topic is given in [100]. As with other NP-complete problems there are no exponential lower bound results for SAT. However, it has been proven that all resolution algorithms need time that is exponential in the first power of the input size [123, 50, 253]. No such lower bound analysis have been done on splitting-based algorithms. For a comprehensive treatment of the complexity of propositional proofs, see a recent survey by Urquhart [255].

2.4 Integer Programming Methods

In order to represent SAT in the framework of mathematical programming, we convert it into the following optimization problem:

$$\begin{aligned} \min \quad & x_0 \\ \text{s.t.} \quad & x_0 e + Ax \geq b \\ & x_j \in \{0, 1\}, \quad j = 0, 1, \ldots, n \end{aligned}$$

While the Simplex method is effective for solving linear programs (LP), there is no single technique that is fast for solving integer programs. Integer optimization problems belong to the class of NP-hard problems. It is well-known that the optimal integer programming solution is usually not obtained by rounding the linear programming solution, although this is possible in certain cases. The closest point to the optimal linear program may not even be feasible. In some cases, the nearest feasible integer point to the linear program solution is far removed from the optimal integer point. Thus, when using an integer linear program to solve the integer program for SAT, it is not sufficient simply to round linear programming solutions.

In the following subsections, we describe existing integer programming methods to solve SAT by briefly and informally recalling the general procedures.

2.4.1 Linear Program Relaxation

A basic method to solve an integer program is the linear program relaxation. In this approach, the LP relaxation is achieved by replacing $x_i \in \{0, 1\}$ with $0 \leq x_i \leq 1$. Linear programming problems are far easier to handle, both mathematically and computationally, than integer programming problems. The linear programming relaxations of satisfiability problems are particularly easy to solve because unit resolution can be adapted to design a complete solution method for these special linear programs [42]. The LP relaxation can be solved efficiently with some sophisticated implementations of Dantzig's Simplex method, or some variations of Karmarkar's interior point method [154].

Moreover, the linear programming relaxation of some special cases of SAT has remarkably properties. Such cases are Horn formulas, Q-Horn formulas, generalized Horn formulas. For instance, the linear program corresponding to a Horn formula has an integral least element that corresponds to the unique minimal model of the Horn formula (see [42]). This leads to

the development of particularly efficient solution algorithms based on integer programming techniques .

Hooker early reported that by solving a linear programming of SAT, one frequently produces an integer solution [138]. Kamath et al. used MINOS 5.1 to solve linear programming relaxation [153]. They tried some small SAT inputs and found that the Simplex method failed to find integral solutions to the linear programming relaxations in majority of instances tested.

2.4.2 Branch and Bound Method

Branch-and-bound (B&B) is essentially a strategy of *divide and conquer*. It is a straightforward and the most successful way to solve the integer programming problem. The idea is to systematically partition the linear programming feasible region into manageable subdivisions and make assessments of the integer programming problem based on these subdivisions. When moving from a region to one of its subdivisions, we add one constraint that is not satisfied by the optimal linear programming solution over the parent region. So the linear programs corresponding to the subdivisions can be solved efficiently. In general, there are a number of ways to divide the feasible region, and as a consequence there are a number of branch-and-bound algorithms.

We show the basic procedures of B&B with a simple example. The method starts with the fractional solution given by its corresponding LP relaxation. Then a variable of fractional solution is selected. For example, let x_1 be a variable, branch on x_1 and set $x_1 \leq 0$ as the additional constant. Resolve the LP relaxation with this augmented constraint. If it still produces a noninteger solution, branch on another noninteger variable, say x_2 , first with constraint $x_2 \leq 0$, and resolve the LP with extra constraint $x_1 \leq 0$ and $x_2 \leq 0$. This process continues until solving the augmented LP yields an integer solution, i.e., an incumbent solution, so there is no need to branch further at that node. Since we do not know this to be optimal, a backtracking procedure is required to search other subregions by adding the extra constraints $x_1 \leq 0$ and $x_2 > 0$, then $x_1 > 0$ and $x_2 \leq 0$, then $x_1 > 0$ and $x_2 > 0$, etc. We iterate the process until the optimal integer solution is obtained.

The above process produces a binary tree. In this way, we implicitly exhaust all possibilities and conclude with an optimal solution. Note that each time we obtain an incumbent solution we get a new upper bound on the minimum value of the objective function. It at the same node the LP yields an objective function with value that exceeds the best upper bound

obtained so far, then we can fathom that node, since any solution obtained at its successors can only be worse.

2.4.3 Cutting Plane Method

Unlike partitioning the feasible region into subdivisions, as in branch-and-bound approaches, the cutting plane algorithm solves integer programs by finding a sequence of linear programming solutions until an integer solution is obtained. It works with a single linear program, which it refines by adding new constraints. The new constraints successively reduce the feasible region until an integer optimal solution is found.

The idea of the cutting plane method can be illustrated from a simple geometric interpretation. The feasible region for the integer program, i.e., an integer polytope, consists of those integer lattice points satisfying all constraints. A *cut* is an inequality satisfied by all the feasible solutions of the integer program. A *cutting plane* is a hyperplane defined by that inequality and it conflicts with the solution x^* of the linear programming relaxation. The cutting plane passes between x^* and the integer polytope and cuts off a part of the relaxed polytope containing the optimal linear programming solution x^* without excluding any feasible integer points. After the cut, the resulting linear program is solved again. If the optimal values for the decision variables in the linear program are all integer, they are optimal; otherwise, a new cut is derived from the new optimal linear programming solution and appended to the constraints.

Historically, it was the first algorithm developed for integer programming that could be proven to converge in a finite number of steps.

2.4.4 Branch and Cut Method

Within a Branch-and-Bound framework, one could think of adding cuts in order to prune some branches of the search tree. Such procedures are called branch-and-cut algorithms. Cuts can be generated using different procedures, one of which is a special form of resolution called *separating resolution* [139]. The procedure generates by resolution a new clause, such that the corresponding clausal inequality cuts off the current fractionary solution. Since cut generation is a relatively costing operation, it is preferable to perform it at the first branching tree levels only, where its effect is greater. Branch and cut methods reveal their efficiency when instance size increases.

2.4.5 Interior Point Method

A very important advance in linear programming solution techniques was introduced by Karmarkar [154]. While the Simplex method jumps from a corner point to another corner point of the LP polytope until the optimal solution is found, Karmarkar's algorithm constructs an ellipsoid inside the polytope and uses nonlinear transformations to project better solution guesses in the interior of the polytope. Unlike the Simplex method which approaches the optimal solution indeed by step-by-step searching and has an exponential worstcase complexity, Karmarkar's algorithm has been proven to be a polynomial time algorithm.

To apply Karmarkar's algorithm on integer programming, first the 0-1 integer program is transformed to a ± 1 integer program. Then a potential function is chosen, such that the optimal integer solution to the original IP problem is at the point where the potential function achieves a maximum. However, using Karmarkar's algorithm on integer programming may get stuck at a local minimum, i.e., it does not guarantee to find the optimal solution by projection. Therefore, it is an incomplete algorithm.

2.4.6 Improved Interior Point Method

It is expected that a sequence of interior points

$$w^{k+1} = w^k + \alpha \Delta w^*$$

is generated such that the potential function in Karmarkar's algorithm is minimized. It is crucial to determine the descent direction Δw^* of the potential function around w^k and the step size α.

In the original Karmarkar's algorithm, the step size α is assumed with (0,1]. They used $\alpha = 0.5$ in their experiments to solve SAT inputs. If the potential function is well represented by the quadratic approximation around the given point, then if we move along the Newton direction and have the appropriate values for certain parameters, we will reach the minimum; otherwise, recall that the step size is chosen so that it reaches a minimum of the objective function on that line of the given descent direction. So there is no reason to restrict α within (0,1].

This suggests the necessity to use line search to choose optimal step size. Following this idea, Shi, Vannelli, and Vlach have recently given an improved interior point algorithm [237]. In their algorithm, the step size α is determined by a golden section search [177]. Experiments show significant improvements on Karmarkar's algorithm.

2.4.7 Integer Programming Heuristics

Because of the computational difficulty of integer programming, a number of heuristics have been developed. An heuristic is a techniques which finds near-optimal solutions at a reasonable computational cost, without guarantee either feasibility or optimality. Since they find approximate solutions, they are more suitable to solve the Max-SAT version of the problem. Two effective techniques used for this aim are the Lagrangian relaxation together with a subgradient-based algorithm, and semidefinite programming.

Nobili and Sassano propose in [198] an algorithm for strengthening the generalized set covering formulation of Max-SAT. This approach does not rely on the solution of the linear relaxation, but computes a Lagrangian bound and uses the Lagrangian multipliers to guide the generation of cutting planes. Moreover, a recent approximation algorithm using the Lagrangian relaxation that has been proved very effective for the (non-generalized) set covering problem is the Volume Algorithm [11].

A semidefinite program is the problem of optimizing a linear function of a symmetric matrix subject to linear equality constraints, and the constraint that the matrix be positive semidefinite. Such problem can be solved within an additive error of ε in polynomial time. For this reason, several integer problems are approached by transforming them into semidefinite programs. A randomized approximation algorithm for Max-2-SAT using semidefinite programming is proposed by Goemans and Williamson in [107]. They obtain a .7584-approximation algorithm for the overall Max-SAT problem.

De Klerk, Van Maaren and Warners in [158] derive a semidefinite programming relaxation for SAT. They develop an incomplete algorithm for detecting satisfiability. It is based on elliptic approximations of propositional formulas (see [178, 179]), which can be used to derive sufficient conditions for unsatisfiability. This condition is expressed in terms of an eigenvalue optimization problem, which in turn can be cast as a semidefinite program, which is solved in polynomial time (to a given accuracy). Pigeon hole problems and mutilated chessboard problems are solved in polynomial time.

2.5 Local Search

Local search is a major class of discrete, unconstrained optimization procedures that can be applied to a discrete search space. Such procedures can be used to solve SAT by introducing an objective function that counts the number of unsatisfiable clauses and solving to minimize the value of this function [110, 111, 200, 234].

Local search, or local optimization, is one of the primitive forms of continuous optimization applied to a discrete search space. It was one of the early techniques proposed to cope with the overwhelming computational intractability of NP-hard combinatorial optimization problems. There have been two major periods for the development of local search. Early greedy local search method was able to solve some small size, unconstrained path finding problems such as TSP [174]. During the middle and late eighties, more powerful techniques for randomized local search were developed. These randomized local search algorithm can handle large size, constrained problems such as CSP and SAT problems efficiently [239].

Given a minimization problem with objective function f and feasible region R, a typical local search procedure requires that, with each solution point $x^k \in R$, there is a predefined neighborhood $N(x^k) \subset R$. Given a current solution point $x^k \in R$, the set $N(x^k)$ is searched for a point x^{k+1} with $f(x^{k+1}) < f(xk)$. If such a point exists, it becomes the new current solution point, and the process is iterated. Otherwise, x^k is retained as a local optimum with respect to $N(x^k)$. Then, a set of feasible solution points is generated, and each of them is "locally" improved within its neighborhood. To apply local search to a particular problem, one needs only to specify the neighborhood and the procedure for obtaining a feasible starting solution.

Local search can be efficient for two reasons. First, at the beginning of search, a full assignment is assigned to all the variables in the search space. Search efforts are focused on a single path in the search space. Second, local search refines for improvement within its local neighborhood using a testing for improvement and, if there is any improvement, takes an action for improvement. Since the objective function has a polynomial number of input numbers, both testing and action can be done efficiently. Little effort is needed to generate the next solution point. A major weakness of local search is that the algorithm has a tendency to get stuck at a locally optimum configuration, i.e., a local minimum.

Greedy local search pursues only paths where every step leads to an improvement, but this leads to a procedure that becomes stuck much more often than the randomized local search. Greedy local search procedure gets stuck in flat places as well as at local minima.

Many search techniques, such as simulated annealing [156], stochastic evolution [226], conflict minimization [110], are either local search or variations of local search.

A search space can be roughly viewed in three levels: top level, middle level, and bottom level. The top level is the upper open portion of the search space with smoothing edges. Most optimization algorithms can descend

quickly in the top level and thus perform quite well. The middle level is the middle portion of the search space where there are relatively "big mountain peaks". During the descent, the search process may encounter problems and it may have to use some tunneling and random heuristics to proceed. The bottom level is the bottom portion of the valleys (particular the lowest valley) where there are many traps. The most difficult situation is a trap where a group of local minima is confined in a 'well'. The search process walks around the set of local minima periodically and cannot get away without special mechanism. In general there may be many traps in a search problem. When local search falls into a trap it may become locked into a loop of local minima. Most algorithms do not succeed in this stage and have difficulty continuing.

For the SAT problem, with high probability, a greedy local search will fall into a trap much more easily. In this case some variables are updated very quickly. The related clauses oscillate between the sat and unsat states. The search is limited to these states. Without any help, there is little chance of getting out to explore other states.

The above observations suggest to use multiphase search to handle the NP-hard problems [117]. That is we may use an open search in the top level, a peak search for searching "coarse" peak structures in the middle level, and a trap search for tracking "fine" rugged trap surface structures in the valleys.

Four components are crucial to the development of an efficient local search algorithm for SAT. They are: (1) the min-conflict heuristics, (2) the best-neighbor heuristics, (3) the random value/variable selection heuristics, and (4) the trap handling heuristics.

2.5.1 The Min-Conflicts Heuristics

Different forms of min-conflict heuristics were proposed for solving the SAT and CSP problems [112]. The min-conflict heuristics aim at performing local conflict minimization in Boolean, discrete, and real spaces [225]. Min-conflict heuristics are important to handle constraints in a constrained optimization problem.

Using inconsistency as objective, the objective function for the SAT problem gives the number of unsatisfied clauses. A CNF is true if and only if the objective function takes the global minimum value 0 on the corresponding solution point. This objective function is the basis of the design of the SAT1, SAT2, SAT3, and G-SAT algorithms [116, 234].

2.5.2 The Best-Neighbor Heuristics

A greedy algorithm selects the best neighbor that yields the minimum value to the objective function and takes this best neighbor direction as the descent direction of the objective function [114]. In a real search space, continuous optimization algorithms can find the best neighbor feasible solution efficiently. A number of local and global optimization algorithms have been developed to solve the SAT problem [114]. The first version of the G-SAT algorithm was proposed as a greedy local search algorithm.

A greedy local search alone may become stuck at local minima much more often and therefore may not be efficient in practice. Therefore, the best neighbor heuristic should be used in conjunction with random value/variable selection and trap handling heuristics described next.

2.5.3 The Random Value/Variable Heuristics

Random value assignment and random variable selection techniques are fundamental to the design of an effective local search algorithm for NP-hard problems [111].

2.5.4 Random Flip Heuristic

Randomly flip the truth values of $1 \leq k \leq n$ variables in the SAT formula [111]. This simple heuristic has been proven to be effective in improving the performance of greedy local search algorithms. During 1988 to 1990 a similar heuristic, random swap, was used to develop local search algorithms for the CSP (e.g., n-queen) problems. It showed significant performance improvement for solving large size n-queen problems [239].

2.5.5 Random Value (Assignment) Heuristics

These include: randomly select a value that generates the minimum number of conflicts; randomly select a value if there is a symmetry (i.e., more than one value producing the same performance); and randomly select a value for conflict minimization when local minima are encountered [116, 174, 199]. A simple random value assignment heuristic, random disturbance, was early used in solving the TSP problem.

2.5.6 Random Variable (Selection) Heuristics

There are two important heuristics [116]: (1) Any Variable Heuristic: select any variable randomly. (2) Bad Variable Heuristic: select a variable from

the set of conflicting variables randomly [239].

The random variable selection heuristic is one of the most important heuristics in the design of local search algorithms for NP-hard problems. It was first used in the local search solution for the SAT problem [110] and then used for the local search solution for the CSP (e.g., n-queen) problems [238]. Conflicting variables in the SAT problem contribute to the unsatisfied clauses.

The bad variable heuristic was first implemented to solve the large size n-queen problems [238]. The bad variable heuristic was independently developed by Papadimitriou for the 2-SAT problem in 1991 [200] and was used in the W-SAT algorithm by Selman et al. in 1994 [236].

2.5.7 Partial Random Variable Selection Heuristics

Partial variable random selection makes use of partial or alternating variable selection techniques [112]. Variants of partial random selection include partial and alternating selection of conflicting and nonconflicting variables, a combination of partial deterministic and partial random variable selection, partial interleaved selection of the different search phases, and partial random selection with metaheuristic control.

The simplest selection strategies include: select a variable deterministically (randomly) and select another variable randomly for conflict minimization; select a variable deterministically (randomly) from the set of conflicting variables and select another variable randomly for conflict minimization; select a variable deterministically and select another variable randomly from the set of conflicting variables for conflict minimization; during certain periods of search, select a variable deterministically (randomly) and select another variable randomly for conflict minimization; during certain periods of search, select a variable deterministically (randomly) from the set of conflicting variables and select another variable randomly for conflict minimization; during certain periods of search, select a variable deterministically and select another variable randomly from the set of conflicting variables for conflict minimization.

In the case of the SAT problem, a variable may be selected from the unsatisfied clauses in a random, partially alternating, partially periodic, or partially interleaving order. The partial and pre random variable selection heuristics were implemented in the SAT3 algorithm in 1990 [112] and were used to solve the large size n-queen problems. A similar heuristic to the partial random variable selection, random walk, was developed by Selman, Kautz, and Cohen independently in 1994 [236].

Random and partial variable selection heuristics were introduced in the design of SAT 1, QS2, QS3, and QS4 algorithms [112, 239]. They can overcome the weakness of the greedy local search algorithms. Selman et al. have recently developed and applied a number of random variable selection heuristics to improve the performance of the greedy G-SAT algorithm [236].

2.5.8 The Trap Handling Heuristics

The search is a process of combating local minima. When the search process is approaching the final search stage, trap handling heuristics are needed to cope with local minima and traps. Examples are the following.

A tunneling Heuristic for the SAT Problem [116] is to flip the truth value of a variable if it does not change the value of the objective function.

Local tracking heuristics [117] are used to track and break local loops (a periodic occurrence of a set of local minima). Several frequently used heuristics include: track local loop(s) when falling into a trap; give low priority to flip to variables in a local minimum loop; give high priority to flip to variables that lead to a new descending direction; lock and release trapping variables periodically, adaptively, or statistically; move gently in a trap to handle fine local structures; move strongly in a trap to handle coarse local structures; jump out of a trap if walking inside it sufficiently long.

Multispace search heuristics [115]. Structural multispace operations have been developed that empower a search process with an information flux which is derived from a sequence of stepwise structural transformations. These include multispace scrambling, extradimension transition, search space smoothing, multiphase search, local to global passage, tabu search, and perturbations. They can disturb the environment of forming local minima and facilitate efficient local search when there are many local minima.

Multiphase heuristics [111, 239, 218] are a part of multispace search heuristics. They have been developed to adapt to the different phases of a search process: perform a poor initial search and then a serious local search for conflict minimization; perform a good initial search and then a serious local search for conflict minimization; perform a good initial search, then a rough local search, and a serious local search for conflict minimization; perform an initial search, and then a rough local search and a serious local search alternatively for conflict minimization; perform a rough initial search, then a coarse local search, and finally, a fine local search for conflict minimization.

2.5.9 Boolean Local Relaxation

Boolean local relaxation may be viewed as a deterministic local search. It was an early inconsistency relaxation technique developed for solving the constraint satisfaction and satisfiability problems. For a variable having m values, m Boolean labels are used to indicate the variables' instantiation to the particular Boolean values. The conflicts produced by an assignment are coded in a set of Boolean objective functions (one for each label). The Boolean relaxation is a local conflict minimization process [109]. During each iteration, the algorithm checks each variable for every label and iteratively minimizes the objective functions by flipping bits (truth values) assigned to the labels: If the objective function does not change, keep it; If the objective function can be reduced, keep the best (i.e., update the label), and then report the inconsistency status. The iteration will terminate once the inconsistency signal turns off.

The Boolean local relaxation algorithm can be combined with backtracking search for CSP/SAT applications. Because of its iterative local conflict minimization and its direct applications to SAT/CSP, Boolean local search made itself a predecessor of several early local search algorithms for CSP and SAT problems.

2.5.10 Simulated Annealing Algorithm

Motivated by the method of simulated annealing, Hansen and Jaumard [127] proposed a steepest ascent, mildest descent algorithm for the maximum satisfiability (Max-SAT) problem. In this approach, Hansen and Jaumard focused on a local change and defined an objective function based on a switching variable and its related clauses. The objective function maximizes local compensation for each variable which can be used for solving the Max-SAT problem. The objective function can not be used for the SAT problem unless another objective function whose global minimum corresponds to a solution of the SAT problem is given. Furthermore, Hansen and Jaumard used local optima checking to handle the local optimum and found it by providing additional guidance to the search direction.

2.5.11 Randomized Local Search

A basic local search consists of an initialization stage and a search stage. At the beginning of search, an initial random solution is chosen. The number of unsatisfiable clauses is computed and is assigned as the value of the objective function. During each iteration, if the objective function can increase, a

flip operation is performed. The procedure terminates when the objective function is reduced to zero, i.e., a solution to the given SAT instance is found. In practice, before the objective function reduces to zero, the procedure may become stuck at local minima.

If the local search procedure becomes stuck at a local minimum, further progress may be achieved by using a noise perturbation to change its location in the search space. The effectiveness with which local minima are handled significantly affects the performance of a local search algorithm. Researchers have proposed a number of techniques such as jumping, climbing, annealing, and indexing to handle local minima [110].

In simulated annealing, a search process occasionally moves up rather than down in the search space, with large uphill moves being less likely than small ones. The probability of large uphill moves is gradually reduced as the search progresses.

A variety of local handlers have been designed for use in the local search algorithms. The basic idea is to generate random exchanges in some current solution points when the search is stuck at a local minimum. The search accepts a modified point as a new current solution not only when the value of the objective function is better but also when it is worse [116] (Traditional local search such as G-SAT used the greedy local descent and restart [234]).

A local handler and its activating condition(s) have significant effect on the performance (running time and average running time) of a local search algorithm for the SAT problem. The conditions for activating local handlers differ from algorithm to algorithm [116]. The local handler can be called if the objective function is not zero (an aggressive strategy). The local handler can be called if the objective function does not increase. The local handler can be called if the objective function does not increase or the objective function is greater than zero for some iterations. In the last two algorithms, the condition 'objective function does not increase' means that the objective value is either reduced (local descent) or remained unchanged (tunneling heuristic).

2.5.12 Tunneling Heuristic

Instead of making a random swing in the vertical direction in the search space, whenever a local minimum is encountered, one can tunnel through the rugged terrain structure in a horizontal direction, moving from one local basin to another local basin in an attempt to locate a better locally optimal solution. A tunnel can be thought of as a shortcut passing through a mountain separating points of equal elevation. Whenever a local minimum

is encountered, a tunnel is made through a mountain to a neighboring basin as long as this does not change/increase the objective function. Tunneling can be used to search a region with local minima effectively. The behavior of local search with tunneling illustrates the fact that seemingly innocuous changes in an optimization routine can have a surprisingly large effect on its performance.

2.5.13 Parallel Local Search

Several parallel algorithms and VLSI architectures have been developed to accelerate CSP and the SAT problems [112]. Depending on implementations, there are several ways of grouping variables or clauses together in parallel so they can be evaluated simultaneously.

A computer word has 32 or 64 bits (such as the DEC Alpha machine). The number of literals in a clause of most practical CNF formulas is much less than 32. In a local search algorithm, therefore, one can pack all the literals in a clause into the bits of a computer word and then evaluate all the literals in one clause in parallel. For m clauses, instead of $O(ml)$, it will take $O(m)$ time to evaluate and update the objective function. If, occasionally, a clause has more than 32 literals, they can be packed in several computer words and all of them can be evaluated simultaneously.

2.5.14 Complete Local Search

Local search algorithms are incomplete, i.e., they can find some solutions for certain CNF formulas and give no answer if the CNF formula is not satisfiable. To overcome this problem, researchers developed complete local search algorithms to test satisfiability as well as unsatisfiability. The basic idea in the SAT1.11 and SAT1.13 algorithms [110, 112] was to combine local search with a systematic search procedure, keeping local search's efficiency while maintaining search completeness by the systematic search method. If at a node of the search tree a solution point is found unsatisfiable, then the algorithm backtracks and continues searching until a solution is found or unsatisfiability is proven. Probe order backtracking is a simplified version of complete local search. Crawford studied a complete local search algorithm [60]. He used weights assigned to clauses to help choose branch variables. Variables occurring in heavily weighted clauses were given precedence.

2.5.15 Greedy Local Search

Traditional local search proceeds by taking a feasible solution point that reduces the value of the objective function. Among many neighboring solution points, local search does not evaluate its neighbors' relative performance with respect to the objective function. A greedy algorithm selects the best neighbor that yields the minimum value of the objective function and takes this best neighbor direction as the descent direction of the objective function. In a real search space, continuous optimization algorithms can find the best neighbor solution efficiently. Unconstrained local and global optimization algorithms have been developed for solving the SAT problem (see [111]). In the discrete search space, a greedy local search algorithm searches for the best neighbor solution. This requires that during each iteration the algorithm examine all the possible moves and select one with maximum descent.

2.5.16 Tabu Local Search

Mazure, Sais, and Gregoire proposed a tabu search algorithm, TSAT, for satisfiability problem [188]. The basic idea behind the TSAT is to avoid using randomness in local search algorithm design. TSAT makes a systematic use of a tabu list of variables in order to avoid recurrent flips and thus escape from local minima. The tabu list is updated each time a flip is made. TSAT keeps a fixed length chronologically ordered FIFO list of flipped variables and prevents any of the variables in the list from being flipped again during a given amount of time.

In this study, Mazure et al. found that the optimal length of the tabu list is crucial to the algorithm's performance. They showed that, for random 3SAT instances, the optimal length of the tabu list $L(n)$ for TSAT is:

$$L(n) = 0.01875n + 2.8125$$

2.6 Global Optimization

A local search procedure may get stuck in a local minimum or a basin. To escape from such local minima, global search strategies need to be developed.

Global optimization is concerned with the characterization and computation of global minima and maxima of constrained or unconstrained nonlinear problems [81, 141]. Global optimization problems belong to the class of NP-hard problems. Most global optimization algorithms are designed as an iterative refinement process.

There are three aspects in designing global search strategies to solve SAT: problem formulations and transformations, strategies to select a direction to move, strategies to help escape from local minima.

Since a search trajectory lacks global information in a search space, strategies to select a direction to move are either steepest descent or hill climbing. A steepest descent approach chooses the direction with the maximum gradient. A hill climbing approach, on the other hand, chooses the first point in the neighborhood of the current point that reduces the objective function. For large formulas, hill climbing methods are much faster than steepest descent because they descend in the first direction that leads to improvement, rather than the best one.

2.6.1 Universal SAT Input Models

In UniSAT models, we extend binary search space into real space. Subsequently, the SAT formula is transformed into an instance of an unconstrained global optimization problem on $\mathbb{R}^n$. All Boolean $\wedge$ and $\vee$ connectives in CNF formulas are transformed (using De Morgan laws) into $+$ and $\times$ of ordinary addition and multiplication operations, respectively. The true value of the CNF formula is converted to the 0 value of the objective function. Given a CNF formula $\mathcal{F}$ from $\{0,1\}^n$ to $\{0,1\}$ with m clauses $C_1, \ldots, C_m$, we define a real function $f(y)$ from $\mathbb{R}^n$ to $\mathbb{R}$ that transforms the SAT into an unconstrained global optimization problem:

$$\min_{y \in \mathbb{R}^n} \sum_{i=1}^{m} c_i(y)$$

where a clause function $c_i(y)$ is a product of n literal functions $q_{i,j}(y_j) \quad (1 \leq j \leq n)$

$$c_i(y) = \prod_{j=1}^{n} q_{i,j}(y_j)$$

In the UniSAT5 model [110]

$$q_{i,j}(y_j) = \begin{cases} |y_j - 1| & \text{if literal } x_j \in C_i \\ |y_j + 1| & \text{if literal } \neg x_j \in C_i \\ 1 & \text{if neither } x_j \text{ nor } \neg x_j \in C_i \end{cases}$$

and in the UniSAT7 model

$$q_{i,j}(y_j) = \begin{cases} (y_j - 1)^2 & \text{if literal } x_j \in C_i \\ (y_j + 1)^2 & \text{if literal } \neg x_j \in C_i \\ 1 & \text{if neither } x_j \text{ nor } \neg x_j \in C_i \end{cases}$$

The correspondence between the binary variables x_i and the continuous variables y_i is

$$x_i = \begin{cases} 1 & \text{if } \ y_i = 1 \\ 0 & \text{if } \ y_i = -1 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Clearly, $\mathcal{F}$ has value true iff $f(y) = 0$ on the corresponding $y \in \{-1, 1\}^n$. A model similar to UniSAT5 was proposed independently in the neural network area [291].

The UniSAT models transform SAT from a discrete, constrained decision problem into an unconstrained global optimization problem [111]. A good property of the transformation is that UniSAT models establish a correspondence between the global minimum points of the objective function and the solutions of the original SAT formula. For instance, the formula

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

is translated into

$$f(y) = |y_1 - 1||y_2 + 1| + |y_1 + 1||y_2 - 1||y_3 - 1|$$

or

$$f(y) = (y_1 - 1)^2 (y_2 + 1)^2 + (y_1 + 1)^2 (y_2 - 1)^2 (y_3 - 1)^2$$

As the iterative improvement progresses, a global minimum point may be approached gradually. The closeness between the present solution point and the global minimum solution point can be tested by solution point testing or objective value testing. In practice, the search process could be stuck at a locally optimum point. To improve the convergence performance of the algorithm, one or more local handlers may be added.

Any existing unconstrained global optimization methods can be used to solve the UniSAT problems (see literature). So far many global optimization algorithms have been developed [110]. These include the basic algorithms, steepest descent methods, modified steepest descent methods, Newton's methods, quasi-Newton methods, descent methods, cutting-plane methods, conjugate direction methods, ellipsoid methods, homotopy methods, and linear programming methods. In each algorithm family, different approaches and heuristics can be used to design objective functions, select initial points, scramble the search space, formulate higher-order local handlers, deflect descent directions, utilize parallelism, and implement hardware architectures to speed up computations.

2.6.2 Complete Global Optimization Algorithms

In order to achieve a complete algorithms, we combine global optimization algorithms with backtracking/resolution procedures [110]. Therefore, these algorithms are able to verify satisfiability as well as unsatisfiability. For small and medium size problems, backtracking is able to verify unsatisfiability quickly for certain classes of formulas but is slow when it comes to verifying satisfiability, as all possible resolutions need to be tried out before concluding that the inference relation holds or that the input formula is satisfiable. Recently some researchers investigated the number of solutions of SAT formulas. Extending Iwama's work [144], Dubois gave a combinatorial formula computing the number of solutions of a set of any clauses [73]. He and Carlier also studied the mathematical expectation of the number of solutions for a probabilistic model [74]. For an incomplete SAT algorithm, the number of solutions can have a strong effect on its computing efficiency. For a complete SAT algorithm, however, the number of search levels plays a crucial role.

2.6.3 Continuous Lagrangian Based Constrained Optimization Algorithms

To avoid getting trapped in local minima, algorithms for solving these problems must have strategies to escape from local minima. Some of these strategies, such as random restarts and tunneling, move the search to a new starting point and start over. In the process of doing so, vital information obtained during the descent to the current local minimum may be lost.

One way to bring a search out of a local minimum is to formulate a SAT problem as a constrained optimization problem. By using the force provided by the violated constraints, the search trajectory can be brought out of a local minimum. One way to implement this idea is compute the sum of the constraints weighted by penalties and to update the penalties continuously during the search. The difficulties with this approach lies in the choice of the proper penalties.

A more systematic approach is to use a Lagrangian formulation. In this and the next subsections, we show two Lagrangian formulations of SAT problems, one in the continuous space and the other in the discrete space.

In order to use a Lagrangian based algorithm, a SAT problem can first be transformed into a continuous constrained optimization problem. The obtained problem has a scalar differentiable objective function (similar to

the above continuous unconstrained problem) which takes the value 0 when the formula is satisfied, and constraint that may look redundant imposing that all clauses must be satisfied.

$$\min_{y \in D \subseteq \mathbb{R}^n} F(y) = \sum_{i=1}^{m} c_i(y) \tag{2.1}$$
$$\text{s.t. } c_i(y) = 0 \qquad \forall i \in \{1, 2, \ldots, m\}$$

where $y = (y_1, y_2, \ldots, y_n)$, and $c_i(y)$ is defined as follows.

$$c_i(y) = \prod_{j=1}^{n} q_{i,j}(y_j)$$

$$q_{i,j}(y_j) = \begin{cases} (1 - y_j)^2 & \text{if literal } x_j \in C_i \\ y_j^2 & \text{if literal } \neg x_j \in C_i \\ 1 & \text{if neither } x_j \text{ nor } \neg x_j \in C_i \end{cases}$$

Here the correspondence between the binary variables x_i and the continuous variables y_i is

$$x_i = \begin{cases} 1 & \text{if } y_i = 1 \\ 0 & \text{if } y_i = 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

All feasible points for the above problem are optima, but this is exactly the concept of SAT. There are two advantages in reformulating the unconstrained problem into a continuous constrained problem. First, a continuous objective value can indicate how close the constraints are being satisfied, hence providing additional guidance in leading to a satisfiable assignment. Second, when the search is stuck in a local minimum and some of the constraints are violated, the violated constraints can provide a force to lead the search out of the local minimum. This is more effective than restarting from a new starting point, as local information observed during the search can be preserved.

Active research in the past two decades has produced a variety of methods for finding global solutions to nonconvex nonlinear optimization problems [141, 81, 125]. In general, transformational and non-transformational methods are two approaches in solving these problems. Non-transformational approaches include discarding methods, back-to-feasible-region methods, and enumerative methods. Discarding methods [191] drop solutions once they were found to be infeasible, and back-to-feasible-region methods [151] attempt to maintain feasibility by reflecting moves from boundaries if such

moves went off the current feasible region. Both of these methods have been combined with global search and do not involve transformation to relax constraints. Last, enumerative methods are generally too expensive to apply except for problems with linear objectives and constraints, and for bilinear programming problems [15].

Transformational approaches, on the other hand, convert a problem into another form before solving it. Well known methods include penalty, barrier, and Lagrange-multiplier methods [177]. Penalty methods incorporate constraints into part of the objective function and require tuning penalty coefficients either before or during the search. Barrier methods are similar except that barriers are set up to avoid solutions from going out of feasible regions. Both methods have difficulties when they start from an infeasible region and when feasible solutions are hard to find. However, they can be combined with other methods to improve their solution quality.

In Lagrangian methods, Lagrange variables are introduced to gradually resolve constraints through iterative updates. They are exact methods that optimize the objective using Lagrange multipliers to meet the Kuhn-Tucker conditions [177]. The above problem can be reformulated using Lagrange multipliers into the following unconstrained problem.

$$L(y, \lambda) = F(y) + \lambda^T c(y) \qquad \text{(Lagrangian function)} \tag{2.2}$$

$$\mathcal{L}(y, \lambda) = F(y) + ||c(y)||_2^2 + \lambda^T c(y) \quad \text{(An augmented Lagrangian func.)} \tag{2.3}$$

where $c = (c_1(y), c_2(y), \ldots, c_m(y))$, and λ^T is the transpose of the set of Lagrange multipliers. The augmented Lagrangian formulation is often preferred because it provides better numerical stability.

According to classical optimization theory, all the extrema of (2.3), whether local or global, are roots of the following sets of equations.

$$\bigtriangledown_y \mathcal{L}(y, \lambda) = 0 \quad \text{and} \quad \bigtriangledown_\lambda \mathcal{L}(y, \lambda) = 0 \tag{2.4}$$

These conditions are necessary to guarantee the (local) optimality to the solution of (2.1). Search methods for solving (2.3) can be classified into local and global algorithms. Local minimization algorithms, such as gradient descent and Newton's methods, find local minima efficiently and work best in unimodal problems. Global methods, in contrast, employ heuristic strategies to look for global minima and do not stop after finding a local minimum [201, 177]. Note that gradients and Hessians can be used in both local and global methods.

Local search methods can be used to solve (2.4) by forming a Lagrangian dynamic system that includes a set of dynamic equations to seek equilibrium points along a gradient path. These equilibrium points are called saddle-points of (2.4), which correspond to the constrained minima of (2.1). The Lagrangian dynamic system of equations are as follows.

$$dy(t)/dt = -\bigtriangledown_y \mathcal{L}(y(t), \lambda(t)) \quad \text{and} \quad d\lambda(t)/dt = \bigtriangledown_\lambda \mathcal{L}(y(t), \lambda(t)) \qquad (2.5)$$

Optimal solutions to the continuous formulation are governed by the Saddle Point Theorem which states that y^* is a local minimum to the original problem defined in (2.1) if and only if there exists λ^* such that (y^*, λ^*) constitutes a saddle point of the associated Lagrangian function $F(y, \lambda)$. Here, a saddlepoint (y^*, λ^*) of Lagrangian function $F(y, \lambda)$ is defined as one that satisfies the following condition.

$$F(y^*, \lambda) \leq F(y^*, \lambda^*) \leq F(y, \lambda^*) \qquad (2.6)$$

for all (y^*, λ) and all (y, λ^*) sufficiently close to (y^*, λ^*).

Note, however, that a Lagrangian search modeled by (2.5) is incomplete: if it does not find a solution in a finite amount of time, it does not prove whether the original SAT problem is satisfiable or not. Hence, infinite time will be required to prove unsatisfiability. Consequently, continuous formulations are not promising in solving large SAT problems. In the next subsection, we extend continuous Lagrangian methods to discrete Lagrangian methods. Surprisingly, discrete methods work much better and can solve some benchmark problems that cannot be solved by other local/global search algorithms.

2.6.4 Discrete Lagrangian Based Constrained Optimization Algorithms

The discrete Lagrangian method is extended from the theory of continuous Lagrangian methods. We have the following discrete constrained formulation of a SAT problem.

$$\min_{y \in \{0,1\}^n} N(y) = \sum_{i=1}^{m} U_i(y) \qquad (2.7)$$

$$\text{s.t. } U_i(y) = 0 \qquad \forall i \in \{1, 2, \ldots, m\}$$

Without going into all the details [263], the continuous Lagrangian method can be extended to work on discrete problems. The discrete Lagrangian

function is defined as follows.

$$L(y, \lambda) = N(y) + \lambda^T U(y) \tag{2.8}$$

where $y \in \{0,1\}^n, U(y) = (U_1(y), \ldots, U_m(y)) \in \{0,1\}^m$, and λ^T is the transpose of $\lambda = (\lambda_1, \ldots, \lambda_m)$ that denotes the Lagrange multipliers. Note that λ_i can be continuous variables.

A saddle point (y^*, λ^*) of $L(y, \lambda)$ in (2.8) is defined as one that satisfies the following condition.

$$L(y^*, \lambda) \leq L(y^*, \lambda^*) \leq L(y, \lambda^*) \tag{2.9}$$

for all λ sufficiently close to λ^* and for all y whose Hamming distance between y^* and y is 1.

The Discrete Lagrangian Method (DLM) for solving SAT problems can be defined as a set of difference equations,

$$y^{k+1} = y^k - \Delta_y L(y^k, \lambda^k) \tag{2.10}$$

$$\lambda^{k+1} = \lambda^k + U(y^k) \tag{2.11}$$

where $\Delta_y L(y, \lambda)$ is the discrete gradient operator with respect to y such that $\Delta_y L(y, \lambda) = (\delta_1, \ldots, \delta_n) \in \{-1, 0, 1\}^n, \sum_{i=1}^n |\delta_i| = 1$, and $(y - \Delta_y L(y, \lambda) \in \{0,1\}^n$. Informally, Δ_y represents all the neighboring points of y.

A discrete Lagrangian algorithm (DLM) implementing (2.10) and (2.11) performs descents in the original variable space of y and ascents in the Lagrange multiplier space of λ. In discrete space, $\Delta_y L(y, \lambda)$ is used in place of the gradient function in continuous space. DLM is started from either the origin or from a random initial point. Further, λ is always set to zero.

As for descent and ascent strategies, there are two ways to calculate $\Delta_y L(y, \lambda)$: greedy and hill-climbing, each involving a search in the range of Hamming distance 1 from the current y (assignments with one variable flipped from the current assignment y). In a greedy strategy, the assignment leading to the maximum decrease in the Lagrangian function value is selected to update the current assignment. Therefore, all assignments in the vicinity need to be searched every time, leading to computation complexity of $O(m)$, where m is the number of variables in the SAT problem. In hill-climbing, the first assignment leading to a decrease in the Lagrangian function value is selected to update the current assignment. Depending on the order of search and the number of assignments that can be improved, hillclimbing

strategies are generally less computationally expensive than greedy strategies. A comparison of the two strategies show that hill-climbing is orders of magnitude faster and results in solutions of comparable quality.

The frequency in which λ is updated affects the performance of a search. The considerations here are different from those of continuous problems. In a discrete problem, descents based on discrete gradients usually make small changes in $L(y, \lambda)$ in each update of y because only one variable changes. Hence, λ should not be updated in each iteration of the search to avoid biasing the search in the Lagrange multiplier space of λ over the original variable space of y.

In binary problems like SAT, a search may find a very small subset of variables that can lead to no degradation in the objective function. Flipping variables in this small subset successively may likely lead to a cycle in the search space. To avoid such an undesirable situation, variables that have been flipped in the recent past can be stored in a tabu list [104, 127] and will not be flipped until they are out of the list.

Further, for large SAT problems formulated as discrete optimization problems, the search may encounter large and flat, but suboptimal, basins. Here, gradients in all directions are the same and the search may wander forever. The discrete gradient operator $\Delta_y L(y, \lambda)$ may have difficulties in basins/plateaus because it only examines adjacent points of $L(y, \lambda)$ that differ in one dimension. Hence, it may not be able to distinguish a plateau from a local minimum.

One way to escape is to allow uphill moves. For instance, in G-SAT's random walk strategy [236], uphill walks are allowed based on probability p. However, the chance of getting a sequence of uphill moves to get out a deep basin is small since each walk is independent.

There are two effective strategies that allow a plateau to be searched.

(a) Flatmove strategy. We need to determine the time to change λ when the search reaches a plateau. As indicated earlier, updating λ when the search is in a plateau changes the surface of the plateau and may make it more difficult for the search to find a local minimum somewhere inside the plateau. To avoid this situation, a strategy called *flat move* [263] can be employed. This allows the search to continue for some time in the plateau without changing λ, so that the search can traverse states with the same Lagrangian function value. How long should flat moves be allowed is heuristic and possibly problem dependent. Note that this strategy is similar to Selman's "sidewaymove" strategy [236].

(b) Tabu list. This search strategy aims to avoid revisiting the same set of states in a plateau. In general, it is impractical to remember every state

the search visits in a plateau due to the large storage and computational overheads. A tabu list [104, 127] can be kept to maintain the set of variables flipped in the recent past and to avoid flipping a variable if it is in the tabu list.

An analysis of convergence property and average time complexity can be found in [177].

2.7 Advanced Techniques

In this section, we describe a number of advanced optimization techniques for satisfiability testing which do not belong to previously discussed techniques. They have been used in practical engineering applications and have proven to be effective for certain classes of SAT.

2.7.1 Analysis and Decomposition

Truemper recently presented an impressive scheme based on the investigation of the structure of a SAT instance, called the *analysis algorithm* [250]. According to the structure of the $\{0, \pm 1\}$ matrix A of the system of linear inequalities representing a SAT instance (see chapter 1), a SAT instance defines a class $\bar{A}$. Based on the insight gained into the structure of A, the analysis algorithm assembles a solution algorithm that correctly processes all the instances of class $\bar{A}$. The analysis algorithm also computes an upper bound to the run time of the solution algorithm. That bound is valid regardless of which case within $\bar{A}$ is being solved. This therefore gives a performance guarantee for the solution algorithm.

In the language of computer science, one may call the analysis algorithm a *compiler* that accepts A as input and that outputs a solution algorithm, together with a performance guarantee.

The analysis algorithm carries out the following two groups of subroutines. The subroutines of the first group determine whether a given A belongs to an *easily* solvable case (see next section). Specifically, it is tested whether the given matrix A is 2-SAT, is (hidden) nearly negative, or is balanced. Note that the cited properties are inherited under submatrix taking. So, if A has one of the properties, then this is so for all instances of the class defined by A.

The subroutines of the second group carry out five decompositions that break down a given A. The components of each decompositions are obtained from A by the deletion of some non-zero entry or by submatrix taking. the latter step may be followed by the adjoining of some rows and columns.

If A does not have one of the special properties, then the analysis algorithm recursively searches for a decomposition of A into components which in turn are processed analogously to A. For this reason the approach is called *decomposition*.

Validity of the subroutines of the analysis algorithm is proved with a new algebra called system $\mathbb{B}$ that is an extension of propositional logic. Such algebra uses three binary operators $\odot$ ($\mathbb{B}$-multiplication), $\oplus$ ($\mathbb{B}$-addition), and $\ominus$ ($\mathbb{B}$-subtraction). This lets represent matrix inequalities in the form of $A \odot s \geq b$, with s is a $\{0, \pm 1\}$ vector, and b a $\{0, 1\}$ vector. This representation of SAT supports the analysis of the instances with certain combinatorics tools. The underlying notion is to analyze the matrix A using the new concepts of *Boolean independence*, *Boolean rank*, *Boolean basis* that are adaptations of familiar concepts of linear algebra.

2.7.2 General Boolean Representations

In practice, many problems in integrated circuit design, such as logic verification, test pattern generation, asynchronous circuit design, logic optimization, sequential machine reduction, and symbolic simulation, can be expressed as Boolean satisfiability problems with arbitrary Boolean functions. Many different representations have been proposed for manipulating Boolean functions. However, many Boolean functions derived from practical circuit design problems suffer from an exponential size in their representations, making satisfiability testing infeasible.

Most SAT algorithms work on conjunctive normal form (CNF) formulas. Sometimes the CNF formula is not an efficient representation in practical application problems. Many real engineering design problems use non-clausal representations rather than the CNF formula. Algorithms in this category may be regarded as non-clausal inference algorithms for satisfiability testing. Compared to CNF formulas, a non-clausal, general Boolean representation is much more compact and efficient, although the transformation of an arbitrary non-clausal expression into CNF can be done in polynomial time by introducing new variables. This will result in clause-form representation of substantially larger sizes [100]. While this is not critical in complexity theory, it will have serious impact in solving practical application problems.

In practice, a SAT algorithm can be made much more efficient if it works directly on problems represented in a compact number of general Boolean formulas rather than a large collection of CNF clauses. For a non-clausal SAT algorithm, the evaluation of arbitrarily large, complex Boolean functions is a key to its efficiency.

2.7.3 Binary Decision Diagram

Ordered Binary Decision Diagrams (OBDDs) [35, 36] is an efficient representation and manipulation method for arbitrary Boolean functions. This representation is defined by imposing restrictions on the Binary Decision Diagram (BDD) representation introduced by Lee [169] and Akers [4], such that the resulting form is canonical. The OBDD representation and its manipulation method are an extremely powerful technique in various practical applications. It is particularly useful with formulas where one needs to consider every solution, such as cases where one must search for optimal solutions.

Although the OBDD representation of a function may have size exponential in the number of variables, many useful functions have more compact representations in practice. A BDD gives a graphical representation of Boolean functions. It is a directed acyclic graph with two types of leaf nodes, 0 and 1. Each nonleaf node is labeled with a Boolean variable v and has two outgoing edges labeled 0 (the left edge) and 1 (the right edge). A BDD can be utilized to determine the output value of the function by examining the input values. Every path in a BDD is unique, i.e., no path contains nodes with the same variables. This means that if we arbitrarily trace out a path from the function node to the leaf node 1, then we have automatically found a value assignment to function variables for which function will be 1 regardless of the values of the other variables.

Given a simple example Boolean function $F = (a + b)(a + c)$, the BDD of function F can be constructed to determine its binary value, given the binary values of variables a, b, and c. At the root node of BDD, we begin at the value of variable a. If $a = 1$, then $F = 1$ and we are finished. If $a = 0$, we look at b. If $b = 0$, then $F = 0$ and again we are finished. Otherwise, we look at c, its value will be the value of F .

It is well known that the BDD size for a given function depends on the variable order chosen for the function (e.g., a,b,c). Since the early introduction of BDDs, several extensions have been proposed to reduce BDD sizes in practical applications. In an ordered BDD, the input variables are ordered, and every path from the root node to the leaf node visits the input variables in an ascending order. In practice, a simple topological based ordering heuristic [363] yields small size BDDs for practical Boolean instances. A reduced ordered BDD is an ordered BDD where each node represents a unique logic function. Bryant showed that the reduced ordered BDD of a Boolean function is well-defined and is a canonical representation of the function; i.e., two functions are equivalent if their reduced ordered BDDs are isomorphic.

The DBDD is efficient to search for optimal solutions for arbitrarily complicated Boolean expressions. In VLSI circuit design, many practical problems require the enumeration of all possible assignments for a given Boolean formula. The best assignment that yields the minimum cost (e.g., minimal circuit structure, minimum chip area, and maximum circuit speed) is then selected from these possible assignments. Since most algorithms for satisfiability testing are designed for finding one truth assignment, they are impractical for selecting an optimal assignment. BDDs are very useful in such situations, since a simple and incremental enumeration of all possible paths from the root node to the leaf node 1 yields all the truth assignments. Thus, once the BDD for a Boolean function has been constructed, it is straightforward to enumerate all assignments or find an optimal solution.

The BDD method can effectively handle small and medium size formulas. For larger size formulas, a partitioning into a set of smaller subformulas before applying the BDD algorithms has been suggested. This approach works well for asynchronous computer circuit design problems [119].

2.7.4 The Unison Algorithms

The Unison algorithm [240] is implemented by using a network of multiple universal Boolean elements (UBEs). The topology of the Unison network specifies the structure of Boolean functions. By dynamically reconfiguring the UBE's functionality, Unison is adaptable to evaluate general Boolean functions representing the SAT/CSP problems.

The total differential, dF , of a Boolean function F represents the difference in the function value due to the difference in input values. For a Boolean function $F(x, y)$ of two variables, x and y, the total differential is calculated from differences in input, dx and dy, as:

$$dF = F_x dx \oplus F_y dy \oplus F_{xy} dx dy$$

where $\oplus$ is the Exclusive-OR operation. Let $F(x, y)$ be a Boolean function of two dependent variables x and y; i.e., $x = G(x_1, x_2)$ and $y = H(y_1, y_2)$. In this case, the total differential dF is:

$$dF(G(x), H(y)) = F_x dG(x_1, x_2) \oplus F_y dH(y_1, y_2) \oplus F_{xy} dG(x_1, x_2) dH(y_1, y_2)$$

It can be observed that the value of dF depends on total differentials dG and dH, rather than the function values $G(x_1, x_2)$ and $H(y_1, y_2)$. By recursively applying the above, the total differential dF can be evaluated based on only total differentials of the independent variables.

The Unison algorithm works in two phases: initialization and evaluation. The initialization phase computes partial derivatives that determine the function to be evaluated in the evaluation phase. The partial derivatives are constant during the evaluation phase. The evaluation phase reads input values and computes the final results. The calculation is performed in a bottom–up fashion, starting from the independent variables.

Combined with parallel evaluation, partial evaluation, and incremental evaluation techniques, Unison can be incorporated into a variety of search and optimization algorithms for satisfiability testing. It is especially important in realtime applications where hardware processing with different Boolean functions is required. It provides an efficient approach for fast non-clausal processing of SAT inputs.

2.7.5 Multispace Search

Many search and optimization methods have been developed in combinatorial optimization, operations research, artificial intelligence, neural networks, genetic algorithms, and evolution programming. An optimization algorithm seeks a value assignment to variables such that all the constraints are satisfied and the performance objective is optimized. The algorithm operates by changing values to the variables in the value space. Because value changing does not affect the formula structure and the search space, it is difficult for a value search algorithm to handle the pathological behavior of local minima.

Multispace search is a new optimization approach developed in recent years [115]. The idea of multispace search was derived from principles of non-equilibrium thermodynamic evolution that structural changes are more fundamental than quantitative changes, and that evolution depends on the growth of new structure in biological system rather than just information transmission. A search process resembles the evolution process, and structural operations are important to improve the performance of traditional value search methods.

In multispace search, any active component related to the given input structure can be manipulated, and thus, be formulated as an independent search space. For a given optimization problem, for its variables, values, constraints, objective functions, and key parameters (that affect the input structure), we define the variable space, the value space (i.e., the traditional search space), the constraint space, the objective function space, the parameter space, and other search spaces, respectively. The totality of all the search spaces constitutes a *multispace*.

The basic idea of multispace search is simple. Instead of being restricted

in the value space, the multispace is taken as the search space. In the multispace, components other than value can be manipulated and optimized as well. During the search, a multispace search algorithm not only alters values in the value space; it also walks across the variable space and other active spaces, changes dynamically the input structure in terms of variables, parameters, and other components, and constructs systematically a sequence of structured, intermediate instances.

Each intermediate instance is solved by an optimization algorithm, and the solution found is used as the initial solution to the next intermediate instance. By interplaying value optimization with structured operations, multispace search incrementally constructs the final solution to the search instance through a sequence of structured intermediate instances. Only at the last moment of the search, the reconstructed instance structure approaches the original instance structure, and thus the final value assignment represents the solution of the given search input.

The major structural operations in multispace search [115] include multispace scrambling [115], extradimension transition (e.g., air bridge, real dimension, and extra dimension) [112], search space smoothing [120], multiphase search [112, 117], local to global passage [111], tabu search [104], and perturbations (e.g., jumping, tunneling, climbing, and annealing) [112]. In the next two subsections we describe two preprocessing methods for satisfiability testing in multispace search: partitioning input size and partitioning variable domain.

2.7.6 Partitioning to Reduce Input Size

Partitioning a large input into a set of smaller subinstances may permit efficient solution of the input. There are two partitioning methods, each consisting of a partitioning, a conquer, and an integration procedure. For constructive partitioning (e.g., divide and conquer), partitioning, conquer, and integration procedures are well defined and easy to implement. For destructive partitioning, it is difficult to design the partitioning and integration procedures.

Such procedure was developed to solve problems of automated design and synthesis of asynchronous circuits [168]. The design of asynchronous control and interface circuits, however, has proven to be an extremely complex and error-prone task. The core problem in asynchronous circuit synthesis can be formulated as an instance of SAT to satisfy the complete state coding (CSC) constraints, i.e., the SAT-Circuit problem. In this practical application problem, an optimal solution with minimal circuit layout area

is sought.

The partitioning preprocessor [119], at the beginning, decomposes a large size SAT formula that represent the given asynchronous circuit design into a number of smaller, disjoint SAT formulas. Each small size SAT formula can be solved efficiently. Eventually, the results of these subformulas are integrated together and contribute to the solution of the original formula. This preprocessor avoids the problem of solving very large SAT formulas and guarantees to finding one best solution in practice. This partitioning preprocessing is destructive since, during the search, extra variables are introduced to resolve the critical CSC constraints.

2.7.7 Partitioning Variable Domains

A variable domain contains values to be assigned to variables. The size of a variable domain, along with the number of variables, determine the computational complexity of an optimization algorithm. From a theoretical point of view, even a small reduction in the variable domain would result in significant improvements in computing efficiency. It is, however, difficult to make use of variable-domain reduction techniques in solving optimization problems. Recently, Wang and Rushforth have studied mobile cellular network structures and developed a novel variable-domain reduction technique for channel assignment in these networks [265].

The rapid growth of mobile cellular communication services has created a direct conflict to the limited frequency spectrum. Channel assignment is an important technique to the efficient utilization of frequency resource for mobile cellular communications. Among several channel assignment problems, the fixed channel assignment (FCA) is essential to the design and operation of cellular radio networks. An FCA algorithm assigns frequency channels to calls such that the frequency separation constraints are satisfied and the total bandwidth required by the system is minimized. By encoding the constraints into clauses, the problem becomes an instance of SAT. For a given cellular communication system, there are numerous ways to assign a channel to a call request. An optimal channel assignment decision can significantly improve the cellular system capacity without requiring extra cost. For a fixed mobile cellular system, the capacity of the cellular system is mainly determined by the performance of the channel assignment algorithms.

Wang and Rushforth's channel assignment algorithm was developed based on the structure of cellular frequency reuse patterns. Using their variable domain partitioning technique, they partition a mobile cellular network with

larger variable domain into two networks: a minimum network with a fixed and small variable domain (due to the known frequency reuse patterns) and a difference network with an even smaller variable domain [265]. Channels are assigned separately to the minimum network and to the difference network, and the superposition of these two assignments constitutes an assignment to the original network.

2.7.8 Parallel SAT Algorithms and Architectures

Many parallel SAT and CSP algorithms have been developed to speed up search computation [257, 171, 228].

However, algorithms running on loosely-coupled, multiprocessor parallel computers offer limited performance improvements for solving SAT.

First, in the worst case, a SAT algorithm may suffer from the exponential growth in computing time. In order to solve a SAT formulas effectively, we will need a computer that has much larger speedup than what is available today. This computer will require the integration of at least a few million processors in a tightly-coupled manner. This is infeasible in the current computer system integration technology.

Second, as the processor gets much faster, the communication overhead among processors in a parallel machine becomes a bottleneck, which may often take 70% to even 90% of the total computing time [148]. Ideally one would expect the speedup on a parallel computer to increase linearly with increasing number of processors. Due to serious off-processor communication delays, after certain saturation point, adding processors does not increase speedup on a looselycoupled parallel machine. Processor communication delay also makes process creation, process synchronization, and remote memory access very expensive in a looselycoupled multiprocessor system. For this reason, the speedup on a multiprocessor is normally less than the number of processors used. With simple SAT algorithms, however, speedup is sometimes greater than the number of processors.

In order to use a tightly-coupled parallel architecture for SAT computation, one must map a computing structure to the input structure and must reduce the total number of sequential computing steps through a large number of symmetrical interactions among simple processors [109]. Several different approaches, e.g., special-purpose parallel VLSI architectures [120], bitparallel programming on sequential machines [112, 240], and tight programming on parallel computer systems, are promising alternatives in this direction. These approaches are capable of providing a tight mapping between a formula structure and a computing structure, resulting in faster

computation. Parallel processing does not change the worstcase complexity of a SAT algorithm unless one has an exponential number of processors. Parallel processing, however, does delay the effect of exponential growth of computing time, allowing one to solve larger size instance of SAT.

2.8 Special Subclasses of SAT

Certain subclasses of SAT that are known to be solved in polynomial time have been identified and explored. A given formula can be preprocessed and examined to determine whether it is a member of a polynomialtime solvable subclass of SAT. If so, a special, fast algorithm can be brought to bear on the formula. Otherwise, a portion of a given formula may a member of such a subclass and its solution may make solving the given formula easier.

2.8.1 2-SAT

A CNF formula containing clauses of one or two literals only is called 2-SAT formula, and is solved in linear time by applying unit resolution [8, 77].

2.8.2 Horn and Extended Horn Formulas

A CNF formula is Horn if every clause in it has at most one positive literal. This class is widely studied, in part because of its close association with Logic Programming. Horn formulas can be solved in linear time using unit resolution [71, 143, 233].

The class of extended Horn formulas was introduced by Chandru and Hooker [42] who were looking for conditions under which a Linear Programming relaxation could be used to find solutions to propositional formulas. A theorem of Chandrasekaran [46] characterizes sets of linear inequalities for which 0-1 solutions can always be found (if one exists) by rounding a real solution obtained using an LP relaxation. Extended Horn formulas can be expressed as linear inequalities that be long to this family of 0-1 problems. The following graph-theoretic characterization, taken from [244], is simpler than the LP characterization.

Let C be a clause constructed from a variable set V , and let R be a rooted directed tree with root s (i.e., a directed tree with all edges directed away from s) and with edges uniquely labeled with variables in V. Then C is extended Horn w.r.t. R if the positive literals of C label a (possibly empty) dipath P of R, and the set of negative literals in C label an edge-

disjoint union of dipaths $Q_1, Q_2, \ldots, Q_t$ of R with exactly one of the following conditions satisfied:

1. $Q_1, Q_2, \ldots, Q_t$ start at the root s.
2. $Q_1, Q_2, \ldots, Q_{t-1}$ start at the root s and Q_t and P start at a vertex $q \neq s$ (if P is empty, Q_t can start from any vertex).

A clause is *simple extended Horn* w.r.t. R if it is extended Horn w.r.t. R and only condition 1 above is satisfied. A CNF formula is (simple) extended Horn w.r.t. R if each of its clauses is (simple) extended Horn w.r.t. R. A formula is (simple) extended Horn if it is (simple) extended Horn w.r.t. some such rooted directed tree R.

One tree R for a given Horn formula is a star (one root and all leaves with an edge for each variable in the formula). Hence, the class of extended Horn formulas is a generalization of the class of Horn formulas.

Chandru and Hooker [42] showed that unit resolution alone can determine whether or not a given extended Horn formula is satisfiable. A satisfying truth assignment for a satisfiable formula may be found by applying unit resolution, setting values of unassigned variables to 1/2 when no unit clauses remain, and rounding the result by a matrix multiplication. This algorithm cannot, however, be reliably applied unless it is known that a given formula is extended Horn. Unfortunately, the problem of recognizing extended Horn formulas is not known to be solved in polynomial time.

2.8.3 Formulas from Balanced Matrices

The class of formulas from balanced $(0, \pm 1)$ matrices, which we call balanced formulas here, has been studied by several researchers (see [54] for a detailed account of balanced matrices and a description of balanced formulas). The motivation for this class is the question, for SAT, when do Linear Programming relaxations have integer solutions?

Express a CNF formula of m clauses and n variables as an $m \times n\{0, \pm 1\}$ matrix M of the system of linear inequalities representing a SAT instance (see chapter 1). A CNF formula is a *balanced formula* if in every submatrix of M with exactly two nonzero entries per row and per column, the sum of the entries is a multiple of four [247].

Let a CNF formula be cast, in standard fashion, as a linear programming problem of the form $\{x : Mx \geq 1 - n(M), 0 \leq x \leq 1\}$ where $n(M)$ is the number of negated literals in every clause. If M is balanced, then for every submatrix S of M, the solution to $\{x : Sx \geq 1 - n(S), 0 \leq x \leq 1\}$ is integral.

From this follows that balanced formulas may be solved in polynomial time using linear programming.

Balanced formulas have the property that, if every clause contains more than one literal, then for every variable x there are two satisfying truth assignments, one with $x = True$ and one with $x = False$. Thus, the following is a simple linear time algorithm for finding solutions to known balanced formulas [54]. Apply unit resolution to the given formula. If a clause is falsified, the formula is unsatisfiable. Otherwise, repeat the following as long as possible: choose a variable and set its value to true, then apply unit resolution. If a clause becomes falsified, then the formula is unsatisfiable, otherwise all clauses have been satisfied by the assignment resulting from the variable choices and unit resolution. Unlike extended Horn formulas, balanced formulas are known to be recognized in polynomial time [54].

2.8.4 Single-Lookahead Unit Resolution

This class was developed as a generalization of other classes including Horn, extended Horn, simple extended Horn, and balanced formulas [231]. It is peculiar in that it is defined based on an algorithm rather than on properties of formulas. The algorithm, called SLUR, selects variables sequentially and arbitrarily and considers both possible values for each selected variable. If, after a value is assigned to a variable, unit resolution does not result in a clause that is falsified, the assignment is made permanent and variable selection continues. If all clauses are satisfied after a value is assigned to a variable (and unit resolution is applied), the algorithm returns a satisfying assignment. If unit resolution, applied to the given formula or to both subformulas created from assigning values to the selected variable on the first iteration, results in a clause that is falsified, the algorithm reports that the formula is unsatisfiable. If unit resolution results in falsified clauses as a consequence of both assignments of values to a selected variable on any iteration except the first, the algorithm reports that it has given up.

A formula is in the class SLUR if, for all possible sequences of selected variables, SLUR does not give up on that formula. SLUR takes linear time with the modification, due to Truemper [247], that unit resolution be applied simultaneously to both branches of a selected variable, abandoning one branch if the other finishes first without falsifying a clause. Note that due to the definition of this class, the question of class recognition is avoided.

All Horn, extended Horn, and balanced formulas are in the class SLUR. Thus, an important outcome of the results on SLUR is the observation that no special preprocessing or testing is needed for a variety of special subclasses

of SAT when using a reasonable variant of the DPL algorithm.

A limitation of all the classes above is that they do not represent many interesting unsatisfiable formulas. There are several possible extensions to SLUR which improve the situation. One is to add a 2-SAT solver to the unit resolution step. This extension is at least able to handle all 2-SAT formulas which is something SLUR cannot do. This extension can be elegantly incorporated into SLUR due to an observation of Truemper: "Whenever SLUR completes a sequence of unit resolutions, and if at that time the remaining clauses are nothing but a subset of the original clauses (which they would have to be if all clauses have at most two literals), then effectively the SLUR algorithm can start all over. That is, if fixing of a variable to both values leads to an empty clause, then the formula has been proved to be unsatisfiable. Thus, one need not augment SLUR by the 2-SAT algorithm, because the 2-SAT algorithm (at least one version of it) does exactly what the modified SLUR does."

Another extension of SLUR is to allow a polynomial number of backtracks, giving up if at least one branch of the DPL tree does not terminate at a leaf where a clause is falsified. Thus, unsatisfiable formulas with short DPL trees can be solved. However, such formulas are uncommon.

2.8.5 q-Horn Formulas

This class of propositional formulas was developed by Boros, Crama, Hammer, Saks, and Sun in [24, 27]. We choose to characterize the class of q-Horn formulas as a special case of monotone decomposition of matrices [248, 250].

Express a CNF formula of m clauses and n variables as an $m \times n\{0, \pm 1\}$ matrix M of the system of linear inequalities representing a SAT instance (see chapter 1). Consider the monotone decomposition of M, where columns are scaled by -1 and the rows and columns are partitioned as follows.

$$\left(\begin{array}{c|c} A^1 & E \\ \hline D & A^2 \end{array}\right)$$

A^1 has at most one +1 entry per row, D contains only -1 or 0 entries, A^2 has no restrictions other than the three values of -1, 0, +1 for each entry, and E has only 0 entries.

If the monotone decomposition of M is such that A^2 has no more than two non-zero entries per row, then the formula represented by M is *q-Horn*.

A recent result by Truemper [250] can be used to find a monotone decomposition for a matrix associated with a q-Horn formula in linear time.

Once a q-Horn formula is in its decomposed form it can be solved in linear time as follows. Treat submatrix A_1 as a Horn formula and solve it in linear time using a method such as in [71, 143, 233] which returns a minimum, unique truth assignment for the formula with respect to true. If the Horn formula is unsatisfiable then the q-Horn formula is unsatisfiable. Otherwise, the returned assignment satisfies A_1 and some or all rows of D. The set of true variables in every truth assignment satisfying A_1 contains the set of variables true in the returned minimum, unique truth assignment. Therefore, since elements of D are either 0 or -1, no truth assignment satisfying A^1 can satisfy any rows of D that are not satisfied by the minimum unique truth assignment.

Hence, the only way A^1 and D both can be satisfied is if A^2, minus the rows collinear with those of D that are satisfied by the minimum, unique truth assignment, can be satisfied. Since A^2 represents a 2-SAT formula, any subset is also 2-SAT and can be solved in linear time. If the answer is unsatisfiable then the q-Horn formula is unsatisfiable; if the answer is satisfiable then such a satisfying assignment plus the minimum, unique truth assignment returned earlier are a solution to the q-Horn formula.

The developers of the class q-Horn also offer a linear-time solution to formulas in this class. The main result of [27] is that a q-Horn formula can be recognized in linear time. See [23] for a linear-time algorithm for solving q-Horn formulas.

Formulas in the class q-Horn are thought to be close to what might be regarded as the largest easily definable class of polynomially solvable propositional formulas because of a result due to Boros, Crama, Hammer, and Saks [24]. Let $\{v_1, v_2, \ldots v_n\}$ be a set of Boolean variables, and P_k and N_k, $P_k \cap N_k = \phi$ be subsets of $\{1, 2, \ldots, n\}$ such that the k-th clause in a CNF formula is given by

$$\bigvee_{i \in P_k} v_i \bigvee_{i \in N_k} \neg v_i$$

Construct the following system of inequalities:

$$\sum_{i \in P_k} \alpha_i + \sum_{i \in N_k} (1 - \alpha_i) \leq Z \qquad k = 1, 2, \ldots, m, \;\; 0 \leq \alpha_i \leq 1 \quad i = 1, 2, \ldots, n$$

where $Z \in \mathbb{R}_+$. If all these constraints are satisfied with $Z \leq 1$ then the formula is q-Horn. On the other hand, the class of formulas such that the minimum Z required to satisfy these constraints is at least $1 + 1/n^{\varepsilon}$, for any fixed $\varepsilon < 1$, is NP-complete. For more information on the subject of q-Horn formulas see [250].

2.8.6 Renamable Formulas

Suppose clauses of a CNF formula $\mathcal{F}$ are constructed from a set V of variables and let $V' \subset V$. Define $switch(\mathcal{F}, V')$ to be the formula obtained as follows: for every $v \in V'$, reverse all unnegated occurrences of v to negated occurrences and all negated occurrences of v to unnegated occurrences. For a given formula $\mathcal{F}$, if there exists a $V' \subset V$ such that $switch(\mathcal{F}, V')$ is Horn, extended Horn, etc., then the formula is said to be renamable Horn, renamable extended Horn, etc., respectively.

The algorithms given above work even if a given formula is renamable to a formula in the class for which they apply. Additional classic references to Horn renamability are [170] and [9]. It is interesting to note that there exist formulas in the class of SLUR formulas that are not members of either renamable extended Horn formulas or balanced formulas [231].

2.8.7 Formula Hierarchies

Some sets of clauses not falling into one of the polynomially solvable classes defined above may be reduced to "equivalent" formulas that are members of at least one of these classes. If such reductions are efficient, these sets can be solved in polynomial time. Such reductions can take place in stages where each stage represents a class of polynomially solved formulas, and lower stages represent classes of perhaps lower time complexity than classes represented by higher stages. The lowest stage is a polynomially solved base class, such as one of the classes above.

An example of such a hierarchy is found in [95]. The base class, at stage 0, is Horn. Consider a stage 1 formula that is not Horn. By definition of the hierarchy, there is a variable v which, if set to true, leaves a set of non-satisfied clauses and non-falsified literals that is Horn. If this Horn formula is found to be satisfiable, we can conclude the original formula is. Otherwise, setting v to false leaves a set of clauses that is a stage 1 formula (empty formulas are considered to belong to every stage). Thus, the above process can be repeated (on stage 1 formulas) to exhaustion. Since it takes linear time to solve Horn formulas and in the worstcase a linear number of Horn systems must be considered, the process for solving formulas at stage 1 has quadratic complexity.

The above concept can be expanded to higher stages to form a hierarchy: at stage i, when setting v to true, a subformula is at stage $i-1$, and when setting v to false, a sub-formula is at stage i. Thus, solutions to stage i formulas are carried out recursively leading to a time complexity that is

bounded by m^i. An alternative way to solve formulas at stage i in the hierarchy is to use i-resolution (resolution is not applied unless at least one clause has at most i literals) [37].

The only remaining question is to determine whether a given formula is a stage i formula. This can be done with a bottom-up approach described in [95]. For other information on such Hierarchies see, for example, [61, 97].

2.8.8 Pure Implication Formulas

Pure implication formulas are defined recursively as follows:

- A variable is a pure implication formula.
- If F_1 and F_2 are pure implication formulas then $(F_1 \Rightarrow F_2)$ is a pure implication formula.

Eliminating parentheses on right to left associativity, a pure implication formula can be written $F_1 \Rightarrow F_2 \Rightarrow \ldots \Rightarrow F_p \Rightarrow z$, where z is a variable. We call the z variable of a formula the *right-end* variable.

The satisfiability problem is trivial for a pure implication formula but the problem of falsifiability is NP-complete even if all variables except the rightend variable occur at most twice in the formula. Furthermore, the complexity of determining falsifiability seems to increase at least exponentially with the number of occurrences of the rightend variable [134]; this yields a hierarchy of classes starting from linear time solvability and going through NP-completeness. This is possibly due to the fact that the expressive power of pure implication formulas at the lower levels of the hierarchy is extremely limited. Despite this lack of expressibility, it seems that the lower levels of the hierarchy are incomparable with other special polynomial timesolvable classes such as 2-SAT and SLUR. More details can be found in [118].

2.8.9 Easy class for Nonlinear Formulations

In some restricted cases, nonlinear methods have polynomial time complexity. Thus, the rich literature on this subject can be carried over to the domain of propositional satisfiability to provide low complexity algorithms for SAT under corresponding conditions. More details can be found in [118].

2.8.10 Nested and Extended Nested Satisfiability

The complexity of nested satisfiability has been studied in [161]. That study was inspired by Lichtenstein's theorem of planar satisfiability [172]. Index

all variables in a CNF formula. A clause C_1 straddles a clause C_2 if the index of a literal of C_2 is strictly between two indices of literals of C_1. Two clauses overlap if they straddle each other. A formula is nested if no two clauses overlap. The problem of determining satisfiability for nested formulas, the clauses ordered so that clause C_i does not straddle clause C_j when $i < j$, can be solved in linear time [161].

An extension to nested satisfiability has been proposed in [128]. This extension can be recognized and solved in linear time.

2.9 Probabilistic and Average-Case Analysis

Probabilistic and average-case analysis can give useful insight into the question of what SAT algorithms might be effective under certain circumstances. Sometimes, one or more structural properties shared by each of a collection of formulas may be exploited to solve such formulas efficiently; or structural properties might force a class of algorithms to require superpolynomial time. Such properties may be identified and then, using probabilistic analysis, one may hope to argue that these properties are so common for a particular class of formulas that the performance of an algorithm or class of algorithms can be predicted for most of the formulas in the class.

Algorithms can be developed based on probabilistic analysis. We mention the two best positive results to date and one negative result. The first algorithm, called SC for Short Clause, iteratively selects a variable and assigns it a value until either a solution is found or it gives up because it has reached a dead end. Such an assignment may satisfy some clauses and falsify some literals. There is no backtracking in SC. Variables are selected as follows: if there is a clause with one non-falsified literal, choose the variable and value that satisfies that clause; otherwise, if there is a clause with two non-falsified literals, choose one of the variables and value that satisfies that clause; otherwise, choose the variable arbitrarily. This algorithm is a restricted version of GUC [45] (Generalized Unit Clause) that always chooses a variable and value that satisfies a clause with the fewest number of nonfalsified literals. The analysis of SC is given in [51]. By adding a limited amount of backtracking to GUC, Frieze and Suen get an algorithm for 3-SAT, called GUCB, that finds a satisfying assignment, in probability, when $m/n < 3.003$ [93].

Finally, we mention the important result in [50] that resolution proofs must be exponentially large, in probability, for random unsatisfiable SAT formulas generated with fixed clause length and fixed ratio m/n.

Chapter 3

A Complete Adaptive Solver

3.1 Introduction

We describe here a complete approach to propositional satisfiability which makes use of a new adaptive branching rule. Moreover, we use a new search framework to speed-up the procedure while preserving completeness. In order to introduce it, let us recall from chapter 2 some element characterizing enumeration procedures. Branching techniques, sometimes called Davis-Putnam-Loveland [64, 86] variants, are the most used among complete methods, and they result to be the more reliable in solving hard instances.

They have the following general structure:

DPL scheme

1. Choose a variable according to some branching rule, e.g. [18, 64, 101, 146]. Every reasonable branching rule gives priority to variables appearing in unit clauses (i.e. clauses containing only one literal).

2. Fix that variable to a truth value and cancel from the formula all satisfied clauses and all falsified literals, because they would not be able to satisfy the clauses in which they appear.

3. If an empty clause is obtained, i.e. every literal is deleted from a clause which is still not satisfied, that clause would be impossible to satisfy. We therefore need to backtrack and change former choices. Usually, we change the last truth assignment, by switching its truth value, or, if both of them are already tried, the last but one, and so on. This means a depth-first exploration of the search tree.

The above is repeated until one of the two following conditions is reached:

- a satisfying solution is found: the formula is satisfiable.
- an empty clause is obtained and every truth assignment has been tried, i.e. the branching tree is completely explored: the formula is unsatisfiable.

A crucial choice is the adopted branching rule. In fact, although it does not affect complexity of the worst case, it shows its importance in the average case, which is the one we have to deal with in real world.

We propose a technique to detect hard subsets of clauses. Evaluation of clause hardness is based on the history of the search, and keeps improving throughout the computation, as illustrated in section 2. Our branching rule consists in trying to satisfy at first such hard sets of clauses, while visiting a clause-based branching tree [39, 135], as showed in section 3. Moreover, we develop a search technique that can speed-up enumeration, as explained in section 4. It is essentially based on the idea of considering only a hard subset of clauses (a *core*, as introduced in [184]), and solve it without propagating assignments to clauses out of this subset. Subsequently, we extend such partial solution to a bigger subset of clauses, until solving the whole formula. The proposed procedure is tested on a set of artificially generated hard problems from the Dimacs collection. Results are in section 5.

3.2 Individuation of hard clauses

Although a truth assignment S satisfies a formula $\mathcal{F}$ only when all C_j are satisfied, there are subsets $\mathcal{P} \subset \mathcal{F}$ of clauses which are more *difficult* to satisfy, i.e. which have a small number of satisfying truth assignment S, and subsets which are rather *easy* to satisfy, i.e. which have a large number of satisfying truth assignment S. In fact, every clause C_j actually *forbids* some of the 2^n possible truth assignments.

Hardness of $\mathcal{F}$ is typically not due to a single clause in itself, but to a combination of several, or, in other words, to the combinations of any generic clause with the rest of the clauses in $\mathcal{F}$. Therefore, we will speak about the difficulty of a clause when being part of the particular instance we are solving, and this would often be implicit. The same clause can, in fact, make difficult an instance $\mathcal{A}$, because it combines in an *unfortunate* way with other clauses, while let another instance $\mathcal{B}$ be easy.

The following is an example of a $\mathcal{P} \subset \mathcal{F}$ constituted by short clauses containing always the same variables:

$$\ldots \quad C_p = (\alpha_1 \vee \alpha_2), \quad C_q = (\neg\alpha_1 \vee \neg\alpha_2), \quad C_r = (\alpha_1 \vee \neg\alpha_2), \quad \ldots$$

$\mathcal{P}$ restricts the set of satisfying assignment for $\mathcal{F}$ to those which have $\alpha_1 = True$ and $\alpha_2 = False$. Hence, $\mathcal{P}$ has the falsifying assignments:

$$S_1 = \{\alpha_1 = False,\ \alpha_2 = False, \ldots\}$$

$$S_2 = \{\alpha_1 = True,\ \ \alpha_2 = True, \ldots\}$$

$$S_3 = \{\alpha_1 = False,\ \alpha_2 = True,\ \ldots\}$$

Each S_i identifies 2^{n-2} (2 elements are fixed) different points of the solution space. Thus, we forbid $3(2^{n-2})$ points. This number is as much as three forth of the number 2^n of points in the solution space.

On the contrary, an example of $\mathcal{P} \subset \mathcal{F}$ constituted by long clauses containing different variables is:

$$\ldots \quad C_p = (\alpha_1 \vee \neg\alpha_2 \vee \alpha_3), \quad C_q = (\alpha_4 \vee \neg\alpha_5 \vee \alpha_6), \quad C_r = (\alpha_7 \vee \neg\alpha_8 \vee \alpha_9), \quad \ldots$$

In this latter case, $\mathcal{P}$ has the falsifying assignments:

$$S_1 = \{\alpha_1 = False,\ \alpha_2 = True,\ \alpha_3 = False, \ldots\ldots\ldots\}$$

$$S_2 = \{\ldots, \alpha_4 = False,\ \alpha_5 = True,\ \alpha_6 = False, \ldots\ldots\}$$

$$S_3 = \{\ldots\ldots, \alpha_7 = False,\ \alpha_8 = True,\ \alpha_9 = False, \ldots\}$$

Each S_i identifies 2^{n-3} (3 elements are fixed) points of the solution space, but this time the S_i are not pairwise disjoint. 2^{n-6} of them falsifies 2 clauses at the same time (6 elements are fixed), and 2^{n-9} falsifies 3 clauses at the same time (9 elements are fixed). Thus, we forbid $3(2^{n-3}) - 3(2^{n-6}) + (2^{n-9})$ assignments. This number, for values of n we deal with, is much less then before. Hence, this $\mathcal{P}$ does not restrict too much the set of satisfying assignment for $\mathcal{F}$.

Starting assignment by satisfying the more difficult clauses, i.e. those which admit very few satisfying truth assignments, or, in other words, represent the more constraining relations, is known to be very helpful in reducing backtracks [19, 135]. This holds because, if such clauses are considered somewhere deep in the branching tree, where many possible truth assignments are already dropped, they would probably result impossible to satisfy, and

would cause to backtrack far. If, on the contrary, such clauses are considered at the beginning of the branching tree, they would cause to drop a lot of truth assignments, but they would be satisfied earlier, or, if this is not possible (because they are an unsatisfiable set), unsatisfiability would be detected faster. Indeed, there would be no need to backtrack far. As for clauses considered deep in the branching tree, they should be the easier ones, which would probably not cause any backtrack.

The point is how to find the hardest clauses. An *a priori* parameter is the length, which is quite inexpensive to calculate. In fact, unit clauses are universally recognized to be hard, and the procedure of unit propagation, which is universally performed, satisfies them at first. Other *a priori* parameters could be the observations made before, not exactly formalized, but probably quite expensive to compute. Remember also that hardness is due both to the clause itself and to the rest of the instance. For the above reasons, a merely *a priori* evaluation is not easy to carry on.

We say that a clause C_j is *visited* during the exploration of the tree if we make a truth assignment aimed at satisfying C_j. The technique we used to evaluate the difficulty of a clause C_j when appearing in the particular instance $\mathcal{F}$, is to count how many times C_j is visited during the exploration of the tree, and how many times the enumeration fails on C_j. Failures can be either because an empty clause is generated due to truth assignment made on C_j, or because C_j itself becomes empty. Visiting C_j many times shows that C_j is difficult, and failing on it shows even more clearly that C_j is difficult. Counting visits and failures has the important feature of requiring very little overhead.

Clause hardness adaptive evaluation

Let v_j be the number of visits of clause C_j, f_j the number of failures due to C_j, p the penalty considered for failures, and l_j the length of C_j. An hardness evaluation of C_j in $\mathcal{F}$ is given by

$$\varphi(C_j) = (v_j + pf_j) \;/\; l_j$$

Therefore, during the elaborations performed by a DPL-style procedure, we can evaluate the value of branching in order to satisfy a generic clause C_j by calculating the above fitness function $\varphi(C_j)$. Moreover, as widely recognized, unit clauses should be satisfied as soon as we have them in the formula, by performing all unit resolutions. Altogether, we use the following branching rule:

Adaptive clause selection

1. *Perform all unit resolutions.*

2. *When no unit clauses are present, make a truth assignment satisfying the clause:*

$$C_{max} = \arg \max_{\substack{C_j \in \mathcal{F} \\ C_j \text{ still unsat.}}} \varphi(C_j)$$

The variable assignment will be illustrated in next section, after introduction of a not binary tree search paradigm. Due to the above adaptive features, the proposed procedure can perform good on problems which are difficult for algorithms using *static* branching rules.

3.3 Clause based Branching Tree

Being our aim to satisfy C_{max}, the choice is restricted to the variables which are present in C_{max}, one of which must be assigned to the unique value that satisfies the clause. A variable appearing positive must be fixed at $True$, and a variable appearing negative must be fixed at $False$ [39].

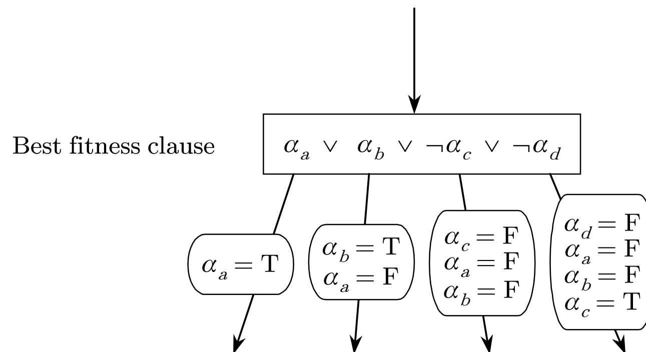

Figure 3.1: Branching node structure. An example of selected clause appears in the rectangle, and the consistent branching possibilities appear in the ellipses

If such a truth assignment causes a failure, i.e. generates an empty clause, and thus we need to backtrack and change it, the next assignment would not be, as usual [176], the opposite truth value for the same variable,

because it would not permit to satisfy C_{max}. Instead, we backtrack and select another variable in C_{max}. Moreover, since the former truth assignment was not successful, we can also fix the opposite truth value for that variable. The resulting node structure is shown in figure 1. If we have no more free variables in C_{max}, or if we tried all of them without success, we backtrack again to the truth assignments made to satisfy the previous clause, until we have another choice.

The above is a complete scheme: if a satisfying truth assignment exists, it will be reached, and, if the search tree is completely explored, the instance is unsatisfiable. Completeness is guaranteed by being this a Branch and Bound scheme. Completeness would be guaranteed even in the case of branching only on all-positive clauses [135] (or on all-negative). However, being our aim to select a set of hard clauses, as explained below, this could not be reached by selecting only all-positive clauses.

This scheme leads to explore a branching tree that is not, in general, binary: every node has as many successors as the number of unassigned variables appearing in C_{max}. In practical case, however, very few of this successors need to be explored. On the other hand, this scheme allows to avoid even to try some truth assignments: the useless ones, namely those containing values which do not satisfy any still unsatisfied cause.

As usual in branching techniques, the solution that satisfies the entire set of the clauses may contain some variables that are still *free*, i.e. not assigned. This happens when such variables were not used to satisfy clauses, so their value can be called "don't care". If d is the number of variables put to "don't care", the number of satisfying solutions trivially is 2^d. They are explicitly obtainable by substitution of each "don't care" with $True$ and $False$. At present, variable assignment order is just their original order within C_{max}, because reordering seems not to improve computational times.

3.4 Minimally Unsatisfiable Subformulas

In the case of unsatisfiable instances, one or more subsets of clauses (within the unsatisfiable instance) are still unsatisfiable. Such subsets can be either proper or not. We can call them unsatisfiable subformulas.

By considering the cardinality of an unsatisfiable subformula, we can define a minimal unsatisfiable subformula. We call minimally unsatisfiable subformula (MUS) any collection $\mathcal{G} \subseteq \mathcal{F}$ of clauses of the original instance having the following properties:

1. A MUS is a subformula (proper or not) of the original formula.
2. A MUS is unsatisfiable.
3. Every proper subset of a MUS is satisfiable

Of course, if any subformula is unsatisfiable, the whole problem is. On the other hand, an unsatisfiable formula always contains a MUS. In the general case they can be more than one. Given any two MUS, they can be either disjoint, or overlapping, but they cannot be contained one in another.

There are tight connections between the concept of MUS and that of Irreducible Infeasible Systems (IIS) in the case of systems of linear inequalities.

There are procedures that, given a set of clauses, recognize whether is a MUS or not in polynomial time [79] [165]. The key point is how to select a MUS. We propose a procedure to rapidly select a good approximation of a MUS, that means an unsatisfiable set of clauses having almost as few clauses as the smallest MUS.

3.5 Adaptive core search

The adaptive branching scheme presented above can be modified in order to speed-up the entire procedure. Roughly speaking, the idea is that, when we have a hard subset of clauses, that we call a *core*, we can at first work on it, just ignoring other clauses. After solving such core, if that is unsatisfiable, the whole formula is unsatisfiable. Conversely, if the core admits a satisfying solution, we try to extend such solution to a bigger subset of clauses, until solving the whole formula. Selection of hardest clauses within a clause-set of cardinality m is always intended as the selection of the top $c \cdot m$ values for φ, with $0 < c < 1$.

The algorithm works as follows:

Adaptive core search

0. (Preprocessing) *Perform p branching iterations using just shortest clause rule. If the instance is already solved, Stop.*

1. (Base) *Select an initial collection of hardest clauses $\mathcal{C}_1$. This is the first core. Remaining clauses form $\mathcal{O}_1$.*

k. (Iteration) *Perform b branching iteration on $\mathcal{C}_k$, ignoring $\mathcal{O}_k$, using adaptive clause rule. We have one of the following:*

k.1. *$\mathcal{C}_k$ is unsatisfiable $\Rightarrow \mathcal{F}$ is unsatisfiable, then Stop.*

k.2. *No answer after b iteration $\Rightarrow$ select a new collection of hardest clauses $\mathcal{C}_{k+1}$ within $\mathcal{C}_k$, put $k := k+1$, goto **k.***

k.3. *$\mathcal{C}_k$ is satisfied by solution $S_k \Rightarrow$ try S_k on $\mathcal{O}_k$. One of the following:*

k.3.a *All clauses are satisfied $\Rightarrow \mathcal{F}$ is satisfied, then Stop.*

k.3.b *There is a set $\mathcal{T}_k$ of falsified clauses $\Rightarrow$ add them to the core: put $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{T}_k$, $k := k+1$, goto **k.***

k.3.c *No clauses are falsified, but there is a set $\mathcal{V}_k$ of still not satisfied clauses $\Rightarrow$ select a collection $\mathcal{C}'_k$ of hardest clauses in $\mathcal{V}_k$, put $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{C}'_k$, $k := k+1$, goto **k.***

The preprocessing step has the aim to give initial values of visits and failures, in order to compute φ. After that, we select the clauses that resulted hard during this branching phase, and try to solve them as if they were our entire instance. If they really are an unsatisfiable instance, we have done. If, after b branching iterations we cannot solve them, our instance is still too big, and it must be reduced more. Finally, if we find a satisfying solution for them, we try to extend it to the rest of the clauses. If some clauses are falsified, this means that they are difficult (together with the clauses of the core), and therefore they should be added to the core. In this case, since the current solution falsifies some clauses now in the core, it results faster to rebuilt it completely. The iteration step is repeatedly applied to instances until their solution.

In order to ensure termination to the above procedure, solution rebuilding is allowed only a finite number of times. After that, the solution is not

entirely rebuilt, but modified by performing backtrack. This choice makes the above algorithm a complete one.

Core Search has the important feature of solving, in average case, smaller subproblems at the nodes of the search tree, hence the operation performed, such like unit propagation consequent to any truth assignment, are performed only on the current $\mathcal{C}_k$. Such idea of delaying (at least partially) the unit propagation subsequent to any variable fixing is recently recognized to be successful [278], and nowadays state of the art solvers (as Sato [277]) try in different ways to incorporate it.

Moreover, when an instance is declared unsatisfiable, the current core selection is an unsatisfiable subset. Core selection is performed in order to contain a small number of hard clauses, and, typically, when the core becomes unsatisfiable, it was satisfiable in the previous iteration. This means that we have an unsatisfiable set of clauses which was satisfiable before adding a small number of clauses. Such set of clauses is therefore an approximation of a MUS, in the sense that it may contain some clauses more than a MUS. On the other hand, such set should contain the hardest clauses of the instance, hence it should be an approximation of a small MUS.

3.6 Computational results

The algorithm was coded in C++. The following results are obtained on a Pentium II 450 MHz processor running MS Windows NT operating system. In the tables, columns labeled n and m shows respectively number of variables and number of clauses. Column labeled *literals* shows the number of all literals appearing in the formula, hence the sum of the lengths of the clauses. Column labeled *sol* reports if satisfiable or unsatisfiable. Column labeled ACS reports times for solving the instance by Adaptive Core Search. Other table specific columns are described in following subsection. Times are in CPU seconds. We set a time limit of 600 sec. When this is exceeded, we report > 600. When a running time is not available, we report n.a.

Computational tree size was not considered because the different solvers compared here do not perform similar node processing, hence times to perform such node processing can greatly vary. It would not help to know that a procedure needs to explore only a small number of nodes if their exploration requires a very long time. Therefore, for the following comparisons, we consider meaningful only computational time.

Parameter p appearing in hardness evaluation function φ was set at 10. During our experiments, in fact, such choice seems to give better and more

uniform results.

We choose the test problems available from the DIMACS [1], since they are widely-known, and the test instances [2], together with computational results, are easily available. Some problems are randomly generated instances, such like the series *aim, jnh*, while some other are encoding of real logic problems, such like the series *ii, par, ssa*. In addition, we solved some real-life problems arisen from a cryptography application, the *des* series.

Running times of Adaptive Core Search are compared with those of other complete algorithms. In such comparisons, either we could make the algorithms run on our machine, or we considered times reported in literature but, when possible, normalized as if they were run on our machine. In order to compare times taking into account such machine performance, we measure it by using the DIMACS benchmark *dfmax* [3], although it had to be slightly modified to be compiled with our compiler. The measure of our machine performance in CPU seconds is therefore:

r100.5.b = 0.01   r200.5.b = 0.42   r300.5.b = 3.57   r400.5.b = 22.21   r.500.b = 86.63

3.6.1 The series *ii32*

The series *ii32* is constituted by instances encoding inductive inference problems, contributed from M.G.C. Resende [153]. They essentially contain two kind of clauses: a set of binary clauses and a set of long clauses. Their size is quite big. On this problem we compare the algorithm of Adaptive Core Search with two simpler branching algorithm: Adaptive Branching Rule and Shortest Clause Branching Rule. Adaptive Branching Rule is a branching algorithm which does not use core search, but uses the adaptive branching rule based on φ. Its times are in column labeled ABR. Shortest Clause Branching Rule is a branching algorithm which does not use core search, and just uses shortest-clause-first branching rule. Its times are in column labeled SCBR. Results on this set are in table 1.

[1]NFS Science and Technology Center in Discrete Mathematics and Theoretical Computer Science - A consortium of Rutgers University, Princeton University, AT&T Bell Labs, Bellcore.

[2]Available from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`

[3]Available from `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/volume/Machine/`.

Problem	n	m	*literals*	*sol*	ACS	ABR	SCBR
ii32a1	459	9212	33003	SAT	0.02	475.57	> 600
ii32b1	228	1374	6180	SAT	0.00	20.65	356.74
ii32b2	261	2558	12069	SAT	0.03	36.56	> 600
ii32b3	348	5734	29340	SAT	0.03	108.57	> 600
ii32b4	381	6918	35229	SAT	1.53	311.62	> 600
ii32c1	225	1280	6081	SAT	0.00	2.67	1.75
ii32c2	249	2182	11673	SAT	0.00	27.29	0.02
ii32c3	279	3272	17463	SAT	2.84	57.03	> 600
ii32c4	759	20862	114903	SAT	5.07	> 600	> 600
ii32d1	332	2730	9164	SAT	0.01	409.21	> 600
ii32d2	404	5153	17940	SAT	0.76	> 600	> 600
ii32d3	824	19478	70200	SAT	7.49	> 600	> 600
ii32e1	222	1186	5982	SAT	0.00	1.24	0.01
ii32e2	267	2746	12267	SAT	0.01	82.13	> 600
ii32e3	330	5020	23946	SAT	0.08	131.38	> 600
ii32e4	387	7106	35427	SAT	0.02	312.28	> 600
ii32e5	522	11636	49482	SAT	1.03	382.36	> 600

Table 1: Results of ACS on the *ii32* series: inductive inference problems. From M.G.C. Resende

ACS distinctly is the fastest, and solves all problems in remarkably short times. ABR is generally faster than SCBR, although not always. The very simple SCBR is sometimes quite fast, but its results are very changeable, and in most of the cases exceeds the time limit.

3.6.2 The series *par16*

The series *par16* is constituted by instances arisen from the problem of learning the parity function, for a parity problem on 16 bits. Contributed from J. Crawford. They contain clauses of different length: unit, binary and ternary. Their size is sometimes remarkably big. par16-x-c denotes an instance which represent a problem equivalent to the corresponding par16-x, except that the first instance have been expressed in a *compressed* form. For this set, we compare with the latest version (3.2) [4] of the state-of-the-art sat solver *Sato* [277]. Results are in table 2. They are extremely encouraging. We can observe a sort of complementarity in computational time results: *ACS* is fast on the *compressed* versions of the problems, where *Sato* is slow. The converse happen on the *expanded* versions. Our hypothesis is that ACS is faster when it can take advantage of the identification of the hard part of the instances, but, due to an implementation and a data structure still

[4] Available from `ftp.cs.uiowa.edu/pub/sato/`.

not refined as Sato's ones, has more difficulties on bigger instances. On the contrary, due to its very carefully implementation, which has been improved for several years, Sato 3.2 can handle more efficiently bigger instances, but on smaller and harder instances, cannot compensate the advantages of adaptive branching and core search.

Problem	n	m	*literals*	*sol*	ACS 1.0	Sato 3.2
par16-1	1015	3310	8788	SAT	10.10	24.16
par16-1-c	317	1264	3670	SAT	11.36	2.62
par16-2	1015	3374	9044	SAT	52.36	49.22
par16-2-c	349	1392	4054	SAT	100.73	128.15
par16-3	1015	3344	8924	SAT	103.92	40.81
par16-3-c	334	1332	3874	SAT	8.19	78.91
par16-4	1015	3324	8844	SAT	70.82	1.51
par16-4-c	324	1292	3754	SAT	5.10	133.07
par16-5	1015	3358	8980	SAT	224.84	4.92
par16-5-c	341	1360	3958	SAT	72.29	196.33

Table 2: Results of ACS and $Sato$ 3.2 on the *par16* series: instances arisen from the problem of learning the parity function. From J. Crawford.

3.6.3 The series *aim100*

The series *aim100* is constituted by 3-SAT instances artificially generated by K. Iwama, E. Miyano and Y. Asahiro [6], and have the peculiarity that the satisfiable ones admit only one satisfying truth assignment. Such instances are not big in size, but can be very difficult. Results on these sets are reported in table 3.

Some instances from this set were used in the test set of the Second DIMACS Implementation Challenge [149]. We also report the results of the four faster complete algorithms of that challenge, normalizing their times according to the results with *dfmax* declared in the original papers, in order to compare them in a machine-independent way.

$C - sat$, presented by O. Dubois, P. Andre, Y. Boufkhad and J. Carlier [76], is a backtrack algorithm with a specialized branching rule and a local preprocessing at nodes of search tree. It is considered one of the fastest algorithms for SAT. Its times are in column labeled C-SAT. 2*cl*, presented by A. Van Gelder and Y. K. Tsuji [101], consists in a combination of branching and limited resolution. Its times are in column labeled 2cl. $TabuS$, presented by B. Jaumard, M. Stan and J. Desrosiers [145], is an exact algorithm which includes a tabu search heuristic and reduction tests other than those of the Davis-Putnam-Loveland scheme. Its times are in column labeled TabuS.

BRR, presented by D. Pretolani [211], makes use of directed hypergraph transformation of the problem, to which it applies a B-reduction, and of a pruning procedure. Its times are in column labeled BRR.

A noticeable performance superiority of *ACS* can be observed, expecially on unsatisfiable problems.

Problem	*n*	*m*	*lit*	*sol*	ACS	C-sat	2cl	TabuS	BRR
aim-100-1_6-no-1	100	160	480	UNSAT	0.20	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-no-2	100	160	480	UNSAT	0.93	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-no-3	100	160	480	UNSAT	1.35	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-no-4	100	160	480	UNSAT	0.96	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-1	100	160	479	SAT	0.09	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-2	100	160	479	SAT	0.03	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-3	100	160	480	SAT	0.26	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-4	100	160	480	SAT	0.01	n.a.	n.a.	n.a.	n.a.
aim-100-2_0-no-1	100	200	600	UNSAT	0.01	52.19	19.77	409.50	5.78
aim-100-2_0-no-2	100	200	600	UNSAT	0.38	14.63	11.00	258.58	0.57
aim-100-2_0-no-3	100	200	598	UNSAT	0.12	56.63	6.53	201.15	2.95
aim-100-2_0-no-4	100	200	600	UNSAT	0.11	0.05	11.66	392.23	4.80
aim-100-2_0-yes1-1	100	200	599	SAT	0.03	0.03	0.32	16.75	0.29
aim-100-2_0-yes1-2	100	200	598	SAT	0.09	0.03	0.21	0.24	0.43
aim-100-2_0-yes1-3	100	200	599	SAT	0.22	0.03	0.38	2.10	0.06
aim-100-2_0-yes1-4	100	200	600	SAT	0.04	0.12	0.11	0.03	0.03
aim-100-3_4-yes1-1	100	340	1019	SAT	0.44	n.a.	n.a.	n.a.	n.a.
aim-100-3_4-yes1-2	100	340	1017	SAT	0.53	n.a.	n.a.	n.a.	n.a.
aim-100-3_4-yes1-3	100	340	1020	SAT	0.01	n.a.	n.a.	n.a.	n.a.
aim-100-3_4-yes1-4	100	340	1019	SAT	0.12	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-1	100	600	1797	SAT	0.08	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-2	100	600	1799	SAT	0.07	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-3	100	600	1798	SAT	0.19	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-4	100	600	1796	SAT	0.04	n.a.	n.a.	n.a.	n.a.

Table 3: Results of ACS, C-SAT, 2cl(limited resolution), DPL with Tabu Search, B-reduction, on the *aim-100* series: 3-SAT artificially generated problems. From K. Iwama, E. Miyano and Y. Asahiro. Times are normalized according to *dfmax* results, as if they were obtained on the same machine.

3.6.4 The series *jnh*

The series *jnh* is constituted by random instances generated by J. Hooker. As stated in [130], the parameter were carefully chosen to result in hard problems [193], because otherwise random problem tend to be too easy. Each variable occurs in a given clause with probability p, and it occurs direct or negated with equal probability. The probability is chosen so that

the expected number of literals per clause is 5. Empty clauses and unit clauses are rejected. Such problems are hardest [139] when the number of variable is 100 and the number of clauses is between 800 and 900. Results on this set are reported in table 4 (part a and b).

Problem	n	m	lit	sol	ACS	DPL	JW	GU	$B\&C$	CS
jnh1	100	850	4392	SAT	0.03	10.5	18.6	53.1	20.8	107.9
jnh2	100	850	4192	UNSAT	0.05	1007.2	15.4	363.1	26.3	37.0
jnh3	100	850	4168	UNSAT	0.28	672.4	239.1	970.1	148.0	195.0
jnh4	100	850	4160	UNSAT	0.07	661.0	50.3	2746.9	108.9	36.0
jnh5	100	850	4164	UNSAT	0.06	670.8	42.8	120.2	88.3	39.1
jnh6	100	850	4155	UNSAT	0.32	1274.5	84.2	23738.2	149.9	217.0
jnh7	100	850	4160	SAT	0.03	5.9	7.2	160.3	51.4	69.2
jnh8	100	850	4147	UNSAT	0.05	165.6	62.8	624.4	58.7	95.8
jnh9	100	850	4156	UNSAT	0.09	345.2	78.9	1867.9	82.6	81.3
jnh10	100	850	4164	UNSAT	0.08	340.4	36.9	313.6	82.0	160.2
jnh11	100	850	4132	UNSAT	0.19	2280.6	135.1	4182.2	165.0	134.6
jnh12	100	850	4171	SAT	0.03	120.6	5.1	398.0	28.8	70.1
jnh13	100	850	4132	UNSAT	0.06	776.8	45.3	503.1	34.6	139.7
jnh14	100	850	4163	UNSAT	0.04	184.2	69.0	2610.4	76.7	39.9
jnh15	100	850	4126	UNSAT	0.08	1547.2	83.1	585.3	65.3	130.5
jnh16	100	850	4172	UNSAT	4.92	13238.7	542.4	20112.2	573.6	434.4
jnh17	100	850	4133	SAT	0.03	140.1	10.8	32.3	58.1	143.1
jnh18	100	850	4169	UNSAT	0.62	2261.0	158.2	2980.6	132.0	191.5
jnh19	100	850	4148	UNSAT	0.07	294.5	87.5	4184.4	153.8	132.3
jnh20	100	850	4154	UNSAT	0.07	648.6	124.5	203.7	126.3	187.3

Table 4a: Results of ACS, Davis-Putnam-Loveland, Jeroslow-Wang, Gallo-Urbani, Branch and Cut, Column Subtraction on the *jnh* series: randomly generated hard problems. From J.N. Hooker. In this table only, the last five columns show times on a different machine, hence times cannot be directly compared.

For most of them we have also results obtained by several other complete algorithms coded in Fortran and run on a Sun Sparc Station 330 in Unix environment, as shown in [130]. In this case only, we cannot calculate the exact computational performance relationship between their and our machine (probably our is at least an order of 10 faster), so we simply report *the original times* for Davis-Putnam-Loveland [176] (column labeled DPL), Jeroslow-Wang [146] (column labeled JW), Gallo-Urbani [96] (column labeled GU), Branch and Cut [139] (column labeled $B\&C$), Column Subtraction [131] (column labeled CS) methods.

Problem	n	m	*lit*	*sol*	ACS	*DPL*	*JW*	*GU*	*B&C*	*CS*
jnh201	100	800	4154	SAT	0.02	8.0	6.3	5.9	28.4	40.2
jnh202	100	800	3962	UNSAT	0.03	3515.2	47.4	710.5	42.7	34.6
jnh203	100	800	3906	UNSAT	0.18	939.8	66.6	294.6	186.3	241.6
jnh204	100	800	3914	SAT	0.41	1109.9	8.4	8905.8	78.1	220.0
jnh205	100	800	3911	SAT	0.05	309.1	12.7	1176.5	57.2	149.3
jnh206	100	800	3905	UNSAT	0.18	1556.6	126.6	3863.9	96.4	85.0
jnh207	100	800	3936	SAT	0.03	3.2	119.8	1037.0	65.1	48.0
jnh208	100	800	3908	UNSAT	0.17	388.3	51.6	958.0	63.6	33.8
jnh209	100	800	3902	SAT	0.11	4.2	50.8	1239.2	77.9	175.4
jnh210	100	800	3915	SAT	0.04	6.1	9.3	576.0	37.6	39.6
jnh211	100	800	3888	UNSAT	0.08	n.a.	n.a.	n.a.	n.a.	n.a.
jnh212	100	800	3932	SAT	0.26	n.a.	n.a.	n.a.	n.a.	n.a.
jnh213	100	800	3900	SAT	0.04	n.a.	n.a.	n.a.	n.a.	n.a.
jnh214	100	800	3896	UNSAT	0.12	n.a.	n.a.	n.a.	n.a.	n.a.
jnh215	100	800	3898	UNSAT	0.08	n.a.	n.a.	n.a.	n.a.	n.a.
jnh216	100	800	3888	UNSAT	0.19	n.a.	n.a.	n.a.	n.a.	n.a.
jnh217	100	800	3939	SAT	0.23	n.a.	n.a.	n.a.	n.a.	n.a.
jnh218	100	800	3905	SAT	0.01	n.a.	n.a.	n.a.	n.a.	n.a.
jnh219	100	800	3889	UNSAT	0.24	n.a.	n.a.	n.a.	n.a.	n.a.
jnh220	100	800	3923	SAT	0.06	n.a.	n.a.	n.a.	n.a.	n.a.
jnh301	100	900	4654	SAT	0.12	12528.6	65.8	271.3	116.0	77.5
jnh302	100	900	4441	UNSAT	0.03	161.6	13.0	380.4	17.0	84.3
jnh303	100	900	4380	UNSAT	0.16	388.6	111.4	307.1	98.2	40.0
jnh304	100	900	4417	UNSAT	0.15	132.0	27.3	409.2	43.4	43.0
jnh305	100	900	4406	UNSAT	0.06	652.7	68.0	138.9	101.7	196.2
jnh306	100	900	4425	UNSAT	1.25	4202.2	195.7	32270.0	221.9	205.1
jnh307	100	900	4365	UNSAT	0.04	6.7	32.1	19.7	25.6	180.3
jnh308	100	900	4410	UNSAT	0.20	1196.5	127.7	6188.3	159.7	164.6
jnh309	100	900	4415	UNSAT	0.03	131.0	14.7	298.2	48.1	42.7
jnh310	100	900	4369	UNSAT	0.03	262.8	25.9	406.7	9.8	43.9

Table 4b: Results of ACS, Davis-Putnam-Loveland, Jeroslow-Wang, Gallo-Urbani, Branch and Cut, Column Subtraction on the *jnh* series: randomly generated hard problems. From J.N. Hooker. In this table only, the last five columns show times on a different machine, hence times cannot be directly compared.

3.6.5 The series *ssa*

The series *ssa* is constituted by instances generated by A. Van Gelder and Y. Tsuji. They are encoding of application problems of circuit fault analysis, used in checking for circuit "single-stuck-at" fault. These instances are large in size but not particularly hard. Results on this set are reported in table 5.

The series were used in the test set of the Second DIMACS Implementation Challenge [149]. We also report the results of the four faster complete

algorithms of that challenge: $C - sat$, $2cl$, $TabuS$, and BRR, already described in 5.3. Times are normalized according to their result with *dfmax*, in order to compare them in a machine-independent way.

Problem	n	m	*literals*	*sol*	ACS	C-sat	2cl	TabuS	BRR
ssa7552-038	1501	3575	8248	SAT	0.19	0.49	0.86	0.01	0.25
ssa7552-158	1363	3034	6827	SAT	0.08	0.33	0.53	5.41	0.17
ssa7552-159	1363	3032	6822	SAT	0.15	0.36	0.53	0.75	0.20
ssa7552-160	1391	3126	7025	SAT	0.20	0.36	0.67	0.75	0.21

Table 5: Results of ACS on the *ssa* series: circuit fault analysis problems.

3.6.6 The series *des*

The series *des* is constituted by instances [5] arising from a practical application: verification and Cryptanalysis of Cryptographic Algorithms [232]. Such problems, which are nowadays showing their importance, can be encoded into instances which can be also very large. They are always satisfiable by construction, but we are interested in finding the satisfying truth assignment. Results on this set are reported in table 9.

Problem	n	m	*literals*	*sol*	ACS 1.0	SATO 3.2
des-1-1	316	1687	5186	SAT	0.11	1.94
des-1-4	1010	6446	20016	SAT	0.98	0.11
des-2-1	600	3531	10746	SAT	0.66	0.09
des-2-4	2062	13387	41224	SAT	2.45	0.19

Table 6: Results of ACS on the *des* series: cryptography problems.

3.7 MUS Selection

In the following tables, we report results of core selection on some series of unsatisfiable instances. columns labeled n and m shows, as before, number of variables and number of clauses. On the contrary, column labeled 'Core n' report the number of variables appearing in the unsatisfiable subformula selected. Column labeled 'Core m' are the number of clauses of the unsatisfiable subformula selected.

[5] Available upon request.

Problem	n	m	Core n	Core m
aim-50-1_6-no-1	50	80	20	22
aim-50-1_6-no-2	50	80	28	32
aim-50-1_6-no-3	50	80	28	31
aim-50-1_6-no-4	50	80	18	20
aim-50-2_0-no-1	50	100	21	22
aim-50-2_0-no-2	50	100	28	31
aim-50-2_0-no-3	50	100	22	28
aim-50-2_0-no-4	50	100	18	22

Table 7: Results of core selection on the *aim-50* series: 3-SAT artificially generated problems.

Problem	n	m	Core n	Core m
aim-100-1_6-no-1	100	160	43	48
aim-100-1_6-no-2	100	160	46	54
aim-100-1_6-no-3	100	160	51	57
aim-100-1_6-no-4	100	160	43	48
aim-100-2_0-no-1	100	200	18	19
aim-100-2_0-no-2	100	200	37	40
aim-100-2_0-no-3	100	200	25	27
aim-100-2_0-no-4	100	200	26	32

Table 8: Results of core selection on the *aim-100* series: 3-SAT artificially generated problems.

Problem	n	m	Core n	Core m
aim-200-1_6-no-1	200	320	52	55
aim-200-1_6-no-2	200	320	76	87
aim-200-1_6-no-3	200	320	73	86
aim-200-1_6-no-4	200	320	45	48
aim-200-2_0-no-1	200	400	49	55
aim-200-2_0-no-2	200	400	46	50
aim-200-2_0-no-3	200	400	35	37
aim-200-2_0-no-4	200	400	36	42

Table 9: Results of core selection on the *aim-200* series: 3-SAT artificially generated problems.

Problem	n	m	Core n	Core m
jnh2	100	850	100	223
jnh3	100	850	100	311
jnh4	100	850	100	295
jnh5	100	850	100	254
jnh6	100	850	100	291
jnh8	100	850	100	278
jnh9	100	850	100	250
jnh10	100	850	100	176
jnh11	100	850	100	202
jnh13	100	850	100	253
jnh14	100	850	100	175
jnh15	100	850	100	217
jnh16	100	850	100	428
jnh18	100	850	100	261
jnh19	100	850	100	198
jnh20	100	850	100	174

Table 10: Results of core selection on the *jnh* series: randomly generated hard problems.

Smaller unsatisfiable subformulae are detected in all the unsatisfiable instances solved. These can be remarkably smaller then the original formula, and give an approximation of a MUS. Such subformule represent the *core* problems of the solved instances.

3.8 Conclusions

We present a clause based tree search paradigm for Satisfiability testing, which makes use of a new adaptive branching rule, and the original techniques of core search, used to speed-up the procedure although maintaining the feature of complete method. We therefore obtain an enumeration technique altogether denominated Adaptive Core Search, which is able to sensibly reduce computational times.

By using the above technique, we observed a better performance improvement on instances which are not uniformly hard, in the sense they contain subsets of clauses having different difficulty degrees. This is mainly due to the ability of our adaptive device in pinpointing hard sub-formulae during the branching tree exploration earlier than other methods. We stress that techniques to perform a fast complete enumeration are widely proposed in literature. Adaptive Core Search, on the contrary, can reduce the set that enumeration works on.

Comparison of ACS with two simpler versions of it, one not using core search, and one not using neither core search nor the adaptive part of the branching rule, clearly reveals the great importance of this two strategies. Comparison with several published results shows the effectiveness of the proposed procedure. Comparison of ACS with the state-of-the-art solver Sato is particularly encouraging. In fact, ACS, in its first release 1.0, is sometimes faster that Sato 3.2, which has evolved for several years. In particular, Sato is faster mainly when the instances are big and $flat$, due to its very carefully implementation. We belive running times can further improve on big-sized instances by further polishing our implementation, and by using several techniques available in literature to perform a fast enumeration. Example of this could be to reduce clause revisits by saving and reusing global inferences revealed during search, as some other modern solvers do. This could be suitably introduced in our core search scheme, by evaluating our fitness function for the global inferences as well, and using this as a criterion to discard them. Future work will explore the introduction of similar tighter bounds in presented scheme, in order to reduce branching tree exploration.

In addition, a core search is a successful approach to the problem of MUS detection. This problem is of great practical relevance. In many practical application, in fact, it is useful to know, in addition to the unsolvability of an instance, which parts of the instance cause the unsolvability.

Chapter 4

Orthogonalization of a Logic Formula

4.1 Introduction

In addition to the normal forms presented in chapter 1, we can define in this chapter the orthogonal boolean form. Such form is of great relevance in many practical applications. We moreover propose a procedure for transforming an arbitrary CNF or DNF to an orthogonal one, and present the results of computational experiments carried out on randomly generated Boolean formulae.

Consider $B = \{0, 1\}$, or, equivalently, $\{True, False\}$. A Boolean function (see chapter 1) $f(x_1, x_2, ..., x_n)$ from the Boolean hypercube B^n to the Boolean set B can be represented by means of a Boolean formula $\mathcal{F}$ in Conjunctive (CNF) or Disjunctive (DNF) normal form.

A Boolean CNF formula is the logic conjunction ($\wedge$) of m clauses, which are logic disjunction ($\vee$) of literals, which can be either posited proposition (x_i) or negated ($\neg x_i$). The general structure is therefore the following.

$$(x_{i_1} \vee ... \vee x_{j_1} \vee \neg x_{k_1} \vee ... \vee \neg x_{n_1}) \wedge \ldots \wedge (x_{i_m} \vee ... \vee x_{j_m} \vee \neg x_{k_m} \vee ... \vee \neg x_{n_m})$$

Conversely, a Boolean DNF formula is the logic disjunction ($\vee$) of m terms, which are logic conjunction ($\wedge$) of literals. The general structure is:

$$(x_{i_1} \wedge ... \wedge x_{j_1} \wedge \neg x_{k_1} \wedge ... \wedge \neg x_{n_1}) \vee \ldots \vee (x_{i_m} \wedge ... \wedge x_{j_m} \wedge \neg x_{k_m} \wedge ... \wedge \neg x_{n_m})$$

In order to handle both CNF and DNF, in section 2 we introduce a general notation for normal forms. The orthogonal form results of great

relevance in solving several hard problems, e.g. in the reliability theory. A basic procedure to reach the orthogonal form is described in section 4. During the above process, the size of the formula tends to exponentially increase. We therefore present, in section 5, some improvements of the basic procedure, with the aim of minimizing the size of the formula both in the final result and during the computation. The proposed procedure is tested on a set of artificially generated Boolean formulae. Results are in section 6.

4.2 Notation

We will develop a procedure that applies both to CNF and DNF. We therefore need a notation which can represent both forms.

Clauses and terms can be viewed as sets of literals. We will call both of them monomials m_i. A CNF or DNF formula $\mathcal{F}$ is therefore a collection of sets of literals, hence a collection of monomials. We have an operator applied between monomial, that will be here indicated with the symbol $\perp$ (external operator), and an operator applied between literals of the same monomial, that will be here indicated with the symbol $\top$ (internal operator). Both CNF and DNF will therefore be represented as follows.

$$(x_{i_1} \top ... \top x_{j_1} \top \neg x_{k_1} \top ... \top \neg x_{n_1}) \perp \ldots \perp (x_{i_m} \top ... \top x_{j_m} \top \neg x_{k_m} \top ... \top \neg x_{n_m})$$

Where the following conventions hold:

$\perp$ means $\wedge$ if we are considering CNF, and $\vee$ if we are considering DNF
$\top$ means $\vee$ if we are considering CNF, and $\wedge$ if we are considering DNF

Given a monomial m_i, we have a set $T_i \subseteq B^n$ where m_i is 1 ($True$), and a set $F_i \subseteq B^n$ (the complement of T_i with respect to B^n) where m_i is 0 ($False$). When solving several NP problems on Boolean formulae, e.g. the Satisfiability problem, what we actually want to know is some information about the set of $True$ points $T = \{X \in B^n : f(X) = 1\}$ or, equivalently, about the set of $False$ points $F = \{X \in B^n : f(X) = 1\}$ for the whole function. Due to normal form, we already have some relations between the T_i and the global T, and between the F_i and F.

Proposition 2.1. *In the case of CNF (a conjunction), it results:*

$$T = \bigcap_{i=1}^{n} T_i$$

whereas, in the case of DNF (a disjunction), it results:

$$T = \bigcup_{i=1}^{n} T_i$$

Of course, specular results hold for the $False$ set F:

Proposition 2.2. *In the case of CNF (a conjunction) it results:*

$$F = \bigcup_{i=1}^{n} F_i$$

whereas, in the case of DNF (a disjunction), it results:

$$F = \bigcap_{i=1}^{n} F_i$$

In the general case, such sets are not disjoint, but can overlap each other: it can be $T_i \cap T_j \neq \phi$ or $F_i \cap F_j \neq \phi$ for some $i, j \in \{1 \dots n\}$. For the above reason, to find respectively the cardinality $|T|$ and $|F|$, we need to identify, at least in an implicit way, respectively all the T_i and all the F_i. The cardinality $|T|$ or $|F|$ gives us, for example, the solution of the feasibility version of the propositional Satisfiability problem, which is well known NP-complete. This theoretically means, moreover, that every problem in NP can be polynomially reduced to the problem of finding this cardinality.

Since the number of points in T_i and F_i is, in the worst case, exponential in the length of the monomials m_i, the approach of identifying all the T_i and all the F_i has exponential worst-case time complexity. This is not surprising. On the other hand, if all the T_i (resp. all the F_i) would be pairwise disjoint sets, in order to find the cardinality $|T|$ (resp. $|F|$) it would suffice to know the cardinality of the T_i (resp. F_i), and sum them. Such cardinalities are, in fact, trivially computable.

In order to complete notation unification, let us consider again the Satisfiability problem. In the case of CNF formulae, it consists in finding if, in the Boolean hypercube B^n, there is at least one true point for all the clauses. Conversely, for DNF formulae, it consists in finding if there is at least one false point for all the terms. Altogether, false points are *bad* for CNF, while true points are *bad* for DNF.

We will call the set of such *bad* points U, with the convention that $U = F$ for CNF, and $U = T$ for DNF. Moreover, every monomial m_i has his set

of *bad* points U_i of the Boolean hypercube B^n, with the convention that $U_i = F_i$ for CNF, and $U_i = T_i$ for DNF. (More intuitively, every m_i *forbids* a set of points: in the case of CNF, every m_i forbids its F_i, while, in the case of DNF, every term m_i forbids its T_i).

Conversely, we will call V the set of *good* points, with the convention that $V = T$ for CNF, and $V = F$ for DNF. Every monomial m_i has therefore his set of *good* points V_i, with the convention that $V_i = T_i$ for CNF, and $V_i = F_i$ for DNF.

Form	internal op.	external op.	bad pt.	good pt.
Unified	$\top$	$\perp$	U	V
CNF	$\vee$	$\wedge$	F	T
DNF	$\wedge$	$\vee$	T	F

Table 4.1: Convention used to unify notation for CNF and DNF.

Proposition 2.3. *The cardinality of the above U_i and V_i are easily computable. Let n be the number of variables, and $l(m_i)$ be the number of distinct literals appearing in monomial m_i, we have $|U_i| = 2^{n-l(m_i)}$, and $|V_i| = 2^n - |U_i| = 2^n - 2^{n-l(m_i)}$.*

We will indicate with (ϕ) the empty monomial, i.e. the monomial m_ϕ which is an empty set of literals. According to proposition 2.3, $U_\phi = B^n$, (m_ϕ has only *bad* points). We will instead indicate with ϕ the empty formula, i.e. the formula $\mathcal{F}_\phi$ which is an empty set of monomials. By definition, $\mathcal{F}_\phi$ will always evaluate to (V).

4.3 The Orthogonal form

We declare a formula to be in orthogonal form when, for every pair of monomials m_i and m_j, at least one boolean variable x_k appears direct in one (for instance m_i) and negated in the other (for instance m_j). Any two monomials have therefore the following structure:

$$m_i = (\ldots \top x_k \top \ldots), \;\; m_j = (\ldots \top \neg x_k \top \ldots) \quad \forall \; i, j \in \{1 \ldots n\}$$

We will say that the above terms are orthogonal, or *clash* [49] on x_k, or *resolve* [223] on x_k, or also *hit* [38] on x_k. This holds both for CNF and DNF.

Theorem 3.1. *For a Boolean formula in orthogonal form, the sets U_i are pairwise disjoint.*

This particularizes for CNF as:

$$F_i \cap F_j = \phi \qquad \forall\, i, j \in \{1 \dots m\}$$
$$(T_i \cap T_j \text{ can be } \neq \phi \qquad \text{for some } i, j \in \{1 \dots m\})$$

and for DNF as:

$$T_i \cap T_j = \phi \qquad \forall\, i, j \in \{1 \dots m\}$$
$$(F_i \cap F_j \text{ can be } \neq \phi \qquad \text{for some } i, j \in \{1 \dots m\})$$

Proof: Orthogonal form $\Rightarrow U_i \cap U_j = \phi \quad \forall\, i, j \in \{1 \dots m\}$.

U_i corresponding to monomial m_i is a set of points in B^n defined by a pattern obtainable from the m_i itself. For example, U_i corresponding to the CNF clause $(x_1 \vee \neg x_3)$ on B^4 is defined by the pattern (0, *, 1, *), representing the 4 points (0, 0, 1, 0), (0, 0, 1, 1), (0, 1, 1, 0), (0, 1, 1, 1). If two monomials m_i and m_j clash on at least one variable x_c, the corresponding U_i and U_j are defined by two patterns which respectively have 0 and 1 in at least position c, hence they define two sets U_i and U_j which cannot have any common point.

Proof: Orthogonal form $\Leftarrow U_i \cap U_j = \phi \quad \forall\, i, j \in \{1 \dots m\}$.

Let us consider two Boolean point $x' = (x'_1, x'_2, ..., x'_n) \in U_i$ and $x'' = (x''_1, x''_2, ..., x''_n) \in U_j$, with $U_i \cap U_j = \phi$. x' and x'' must be different (and binary), hence at least one component is respectively 0 and 1. Let us call that component x_c Monomials m_i and m_j corresponding to U_i and U_j must therefore both contain the variable x_c, and clash on it.

Example: Suppose we are interested in solving the Satisfiability problem for the following CNF. To solve our problem we need to check whether the global F covers the whole B^5. In the general case, we can only proceed by identifying F as the intersection of the F_i.

It is straightforward to find the corresponding F_i (and their cardinality).

$$\begin{array}{rcll}
(\ x_1 \vee \neg x_2 \vee \ x_3 \vee \ x_4 \vee \ x_5) & \rightarrow & F_1 = \{0,1,0,0,0\} & |F_1| = 1 \\
(\neg x_1 \vee \neg x_2 \vee \ x_3 \vee \ x_4 \vee \ x_5) & \rightarrow & F_2 = \{1,1,0,0,0\} & |F_2| = 1 \\
(\ x_2 \vee \ x_3 \vee \ x_4 \vee \ x_5) & \rightarrow & F_3 = \{*,0,0,0,0\} & |F_3| = 2 \\
(\ x_3 \vee \neg x_4 \vee \ x_5) & \rightarrow & F_4 = \{*,*,0,1,0\} & |F_4| = 4 \\
(\ x_3 \vee \ x_4 \vee \neg x_5) & \rightarrow & F_5 = \{*,*,0,0,1\} & |F_5| = 4 \\
(\ x_3 \vee \neg x_4 \vee \neg x_5) & \rightarrow & F_6 = \{*,*,0,1,1\} & |F_6| = 4 \\
(\neg x_3) & \rightarrow & F_7 = \{*,*,1,*,*\} & |F_7| = 16
\end{array}$$

No more in a straightforward way, by identifying all the points of the intersection of the F_i , we can observe that F actually covers B^5 (see picture 1 below). Hence, given CNF is unsatisfiable. The number of points in this intersection is, unfortunately, exponential (in the worst case) in the size of the formula. This gives worst case exponential time complexity to such procedure.

On the other hand, we could observe that the CNF is in orthogonal form, hence we have pairwise disjoint U_i. In the case of CNF, this means pairwise disjoint F_i.

On this basis, we easily have $|F| = |F_1| + |F_2| + |F_3| + |F_4| + |F_5| + |F_6| + |F_7| = 32$. Since F covers B^5 iff $|F| = 2^5 = 32$, this is the case, and given CNF is unsatisfiable. This is obtained just by counting, as shown in [144].

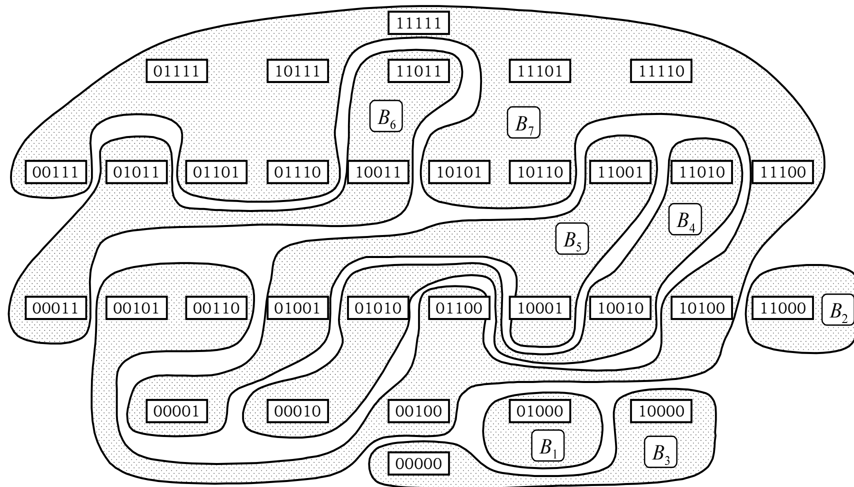

Figure 4.1: Individuation of the $False$ sets F_i on the Boolean hypercube B^5

4.4 Basic Orthogonalization operation

Given an arbitrary Boolean formula $\mathcal{F}$ in normal form, representing the Boolean function $f(x_1, x_2, ..., x_n)$, our aim is to put it in the orthogonal form $\mathcal{O}$ while still representing the same $f(x_1, x_2, ..., x_n)$. This can always be done as follows.

We define the multiplication ($\diamond$) of two monomials m_i and m_j as a new monomial containing all their literals (but without repeated ones) when the two monomials are not orthogonal, and ϕ when they are orthogonal. Note that ϕ means an empty formula, i.e. a formula for which there are only *good* points, and not a formula made by an empty monomial, i.e. a formula for which there are only *bad* points. Formally:

$$m_i \diamond m_j = (x_{ih} \top ... \top x_{ik}) \diamond (x_{jh} \top ... \top x_{jk}) = \begin{cases} \phi \quad \text{if } m_i \text{ and } m_j \text{ are orthogonal} \\ (x_{ih} \top x_{jh} \top ... \top x_{ik} \top x_{jk}) \quad\quad \text{else} \end{cases}$$

Proposition 4.1. *Consider any two monomials m_i and m_j, with their corresponding sets U_i, V_i, U_j and V_j. Let $m_k = m_i \diamond m_j$ be their product. The set of the bad points for m_k is $U_k = U_i \cap U_j$, while the set of good points is $V_k = V_i \cup V_j$*

We can use such multiplication operation to make any two monomials orthogonal each other. In fact:

Theorem 4.1. *Consider an arbitrary Boolean formula $\mathcal{F}$ in normal form representing the Boolean function $f(x_1, x_2, ..., x_n)$. If we multiply an arbitrary monomial $m_i \in \mathcal{F}$ by the negation $\neg m_j$ of another arbitrary monomial $m_j \in \mathcal{F}$, we obtain a new Boolean formula $\mathcal{F}'$ still representing the same $f(x_1, x_2, ..., x_n)$.*

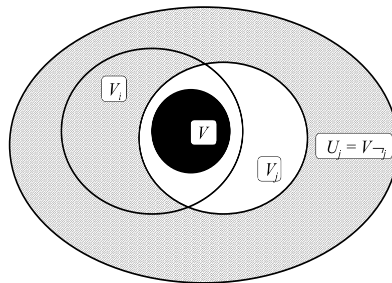

Figure 4.2: Individuation of the *False* sets F_i on the Boolean hypercube B^5

Proof: We need to prove that sets U and V are the same for $\mathcal{F}$ and $\mathcal{F}'$. As we can observe in the above figure 4.2, monomial m_j determines in B^n a partition in U_j and V_j. Its negation $\neg m_j$ determines a partition $U_{\neg j} = V_j$ and $V_{\neg j} = U_j$.

Now we multiply another monomial m_i by $\neg m_j$, obtaining the new monomial m_i', add m_i' and remove m_i form the formula. The set V_i' corresponding to this new monomial, by proposition 4.1, is $V_i \cup V_{\neg j}$, which is $\supseteq V_i$. So the set of *good* points for the formula V, which is the intersection of all the V_i, cannot decrease. It could increase in the area of $V_{\neg j}$, but such area is forbidden by the fact that $V \subseteq V_j$. Hence, V is the same for $\mathcal{F}$ and $\mathcal{F}'$, and therefore U also remains the same. The thesis follows.

Given an arbitrary monomial $m_i = (x_h \top x_{h+1} \top ... \top x_k)$, its negation (by De Morgan's laws) is easy computable as the following set of monomials connected by our external operator. $(\neg x_h) \bot (\neg x_{h+1}) \bot ... \bot (\neg x_k) = \neg(m_i)$. But the expression for $\neg m_i$ is not unique. We could, in fact, consider a negation which is in orthogonal form, namely the orthogonal negation $\neg^o(m_i)$ of m_i. $\neg^o(m_i)$ is made of k monomials $o_1^{m_i} \bot o_2^{m_i} \bot \ldots \bot o_k^{m_i}$ corresponding to the negation of the first literal, the first and the negation of the second, and so on, as follows.

$$(\neg x_h) \bot (x_h \top \neg x_{h+1}) \bot ... \bot (x_h \top x_{h+1} \top ... \top \neg x_k)$$

Example The orthogonal negation of

$$m = (x_1 \top x_2 \top \neg x_3)$$

is

$$\neg^o(m) = o_1^m \bot o_2^m \bot o_3^m = (\neg x_1) \bot (x_1 \top \neg x_2) \bot (x_i \top x_2 \top x_3)$$

Basing on the above results, we can develop the following procedure.

Basic Orthogonalization Operation: Without loss of generality, consider any two monomials m_1 and m_2 not already orthogonal.

$$m_1 = (x_i \top ... \top x_j \top x_{j+1} \top ... \top x_h) \quad m_2 = (x_{j+1} \top ... \top x_h \top x_{h+1} \top ... \top x_k)$$

m_1 and m_2 have, in general, a common set of literals $c_{1,2} = (x_{j+1} \top ... \top x_h)$ and two sets of literals d_1 and d_2 which are not in common: $d_1 = (x_i \top ... \top x_j)$

for m_1, and $d_2 = (x_{h+1} \top ... \top x_k)$ for m_2. Note that, since they are not orthogonal, they cannot contain complementary literals: if $x_i \in m_1 \Rightarrow \neg x_i \notin m_2$.

Choose one of them, say d_1, and consider its orthogonal negation $\neg^o(d_1) = o_1^{d_1} \bot o_2^{d_1} \bot \ldots \bot o_j^{d_1}$.

The (sub)formula $m_1 \bot m_2$, is equivalent to the following (sub)formula, in the sense that they both represent the same Boolean function.

$$m_1 \bot o_1^{d_1} \diamond m_2 \bot o_2^{d_1} \diamond m_2 \bot \ldots \bot o_j^{d_1} \diamond m_2$$

Note that the obtained (sub)formula is in orthogonal form. The number of monomials is 1 plus the cardinality of the set of non-common literals (d_1) we used. In order to obtain a smaller number of monomials, we always choose the set of non-common literals of minimum cardinality.

Example Given a formula made up of the two monomials m_1 and m_2.

$$m_1 = (x_1 \top \neg x_2 \top x_5) \bot (\neg x_2 \top x_3 \top x_4) = m_2$$

the sets of non-common literals are

$$d_1 = (x_1 \top x_5) \qquad \text{and} \qquad d_2 = (x_3 \top x_4)$$

Their cardinality is the same. We choose d_1, and its orthogonal negation is the following.

$$(\neg x_1) \bot (x_1 \top \neg x_5)$$

By performing the orthogonalization operation, the above formula is equivalent to

$$(x_1 \top \neg x_2 \top x_5) \bot ((\neg x_1) \diamond (\neg x_2 \top x_3 \top x_4)) \bot ((x_1 \top \neg x_5) \diamond (\neg x_2 \top x_3 \top x_4))$$

which is the following orthogonal formula.

$$(x_1 \top \neg x_2 \top x_5) \bot (\neg x_1 \neg x_2 \top x_3 \top x_4) \bot (x_1 \top \neg x_2 \top x_3 \top x_4 \top \neg x_5)$$

The above is a general procedure to orthogonalize any two monomials. Given any formula, by iterating this orthogonalization operation to exhaustion, until every pair of monomials are orthogonal, we can always reach the orthogonal form.

4.5 Improvements

Unfortunately, by repeatedly applying above operation to exhaustion, the size of the formula tends to exponentially increase. To reduce the size of the formula, we will make use of the following observation.

4.5.1 Recognition of Internally Orthogonal Sets of Terms

A set of monomials $\mathcal{S}$ is internally orthogonal when each monomial $m_i \in \mathcal{S}$ of them is orthogonal to every $m_j \in \mathcal{S}$, for all $m_i, m_j \in \mathcal{S}$. Given a generic formula, some sets of monomials may already be internally orthogonal.

We can partition the set of monomials in sets $\mathcal{S}_i$ which are already internally orthogonal $\mathcal{S}_1, ..., \mathcal{S}_p$. The original formula can therefore be represented as follows.

$$\mathcal{S}_1 \perp ... \perp \mathcal{S}_p$$

Given a set of monomials $\mathcal{S}_i$, we can consider its orthogonal negation $\neg^o(\mathcal{S}_i)$, which is a set of new monomials $o_j^{\mathcal{S}_i}$ corresponding to the negation of $\mathcal{S}_i$ in orthogonal form. This is obtainable in a straightforward way from the definition of orthogonal negation of a monomial.

$$\neg^o(\mathcal{S}_i) = o_1^{\mathcal{S}_i} \perp o_2^{\mathcal{S}_i} \perp \ldots \perp o_k^{\mathcal{S}_i}$$

This lead us to the extended orthogonalization operation: We define the multiplication ($\diamond$) of two sets of monomials $\mathcal{S}_1$ and $\mathcal{S}_2$ as a new set of all the monomials obtained by calculating $m_i \diamond m_j$ for all $m_i \in \mathcal{S}_1$ and all $m_j \in \mathcal{S}_2$. (The multiplication $m_i \diamond m_j$ is defined above.)

Without loss of generality, given any two internally orthogonal sets, the multiplication ($\diamond$) of

$$\mathcal{S}_1 = (m_i \perp m_{i+1} \perp \ldots \perp m_j) \quad \mathcal{S}_2 = (m_h \perp m_{h+1} \perp \ldots \perp m_k)$$

The (sub)formula $\mathcal{S}_1 \perp \mathcal{S}_2$ is equivalent, in the sense specified above, to the following (sub)formula.

$$\mathcal{S}_1 \perp \neg^o(\mathcal{S}_1) \diamond \mathcal{S}_2$$

The above is a general procedure to orthogonalize any two sets of internally orthogonal monomials. Given any formula, by iterating this orthogonalization operation to exhaustion, we can always reach the orthogonal form.

We can moreover take advantage of the two following simplifying operations.

4.5.2 Absorption

One monomial is implied by another one if it contains another one. Given a formula containing the following two monomials,

$$m_1 = (x_i \top \dots x_j \top x_{j+1} \top \dots \top x_h \top x_{h+1} \top \dots \top x_k) \quad m_2 = (x_{j+1} \top \dots \top x_h)$$

the first can be deleted obtaining a new formula which is equivalent, in the sense specified above, to the original one. This operation is particularly useful in reducing the number of monomials in the formula.

4.5.3 Synthesis Resolution

This operation is a special case of the general operation called resolution [223, 21] in the case of CNF, and consensus [220] in the case of DNF. Given a formula containing two monomials which are identical except for one literal x_i appearing positive in one monomial and negative in the other, hence with the following structure:

$$m_1 = (x_i \top x_h \top \dots \top x_k) \quad m_2 = (\neg x_i \top x_h \top \dots \top x_k)$$

we can add to the formula their resolvent [223] obtaining a new formula which is equivalent, in the sense specified above, to the original one.

$$t_3 = (x_h \top \dots \top x_k)$$

In particular, in this case, such resolvent absorbs both its parents. We can therefore remove from the formula both its parents, obtaining a new formula which is equivalent, in the sense specified above, to the original one. This operation helps in reducing the number of monomials in the formula.

4.6 Complete Orthogonalization Operation

So far, we have a set of operation that can be performed on the original formula in order to put it in orthogonal form. Being our aim not to increase too much the size of the formula, we define the quality Q of an orthogonalization step as the number of clauses orthogonalized divided by the number of new clauses created. A simple way to The algorithm is therefore

1. Find a partition in already orthogonal sets of clauses $\mathcal{S}_1, ..., \mathcal{S}_p$
2. Perform all extended orthogonalization steps of quality $Q \geq Q_{limit1}$

3. Perform all basic orthogonalization steps of quality $Q \geq Q_{limit2}$
4. Perform all possible synthesis resolution.
5. Perform all possible absorption
6. Repeat until all orthogonal

4.7 Testing

The algorithm was tested on a set of CNF formulae representing satisfiability instances. They are obtained from the SATLIB web site of the Darmstadt University of Technology. Such instances are 3-sat artificially generated problem.

Problem	n	m	literals	sol	time	$m_{orthogonal}$
uf20-01	20	91	273	Y	6.52	130
uf20-02	20	91	273	Y	5.00	100
uf20-03	20	91	273	Y	5.32	132
uf20-04	20	91	273	Y	3.42	134
uf20-05	20	91	273	Y	0.32	31
uf20-06	20	91	273	Y	1.80	40
uf20-07	20	91	273	Y	1.54	85
uf20-08	20	91	273	Y	5.55	136
uf20-09	20	91	273	Y	15.47	144
uf20-010	20	91	273	Y	5.84	128
uf20-011	20	91	273	Y	1.07	130
uf20-012	20	91	273	Y	8.34	190
uf20-013	20	91	273	Y	2.28	65
uf20-014	20	91	273	Y	7.07	167
uf20-015	20	91	273	Y	5.67	120
uf20-016	20	91	273	Y	4.77	102
uf20-017	20	91	273	Y	4.22	109
uf20-018	20	91	273	Y	8.06	105
uf20-019	20	91	273	Y	3.56	70
uf20-020	20	91	273	Y	9.17	98
uf20-021	20	91	273	Y	1.92	73
uf20-022	20	91	273	Y	2.48	79
uf20-023	20	91	273	Y	11.53	211
uf20-024	20	91	273	Y	5.28	171
uf20-025	20	91	273	Y	4.39	82
uf20-026	20	91	273	Y	3.50	59
uf20-027	20	91	273	Y	2.49	63
uf20-028	20	91	273	Y	3.38	93
uf20-029	20	91	273	Y	2.24	66
uf20-030	20	91	273	Y	1.96	90

From the above table we can observe that the number of monomials in the orthogonalized formula generally increase, although in some cases this does not hold. Moreover, intermediate formulae contains many more monomials. This turn out to be a general rule in performing such kind of operation. However, there are practical applications where the orthogonal form is of great relevance, and the advantages completely surmount the disadvantage of such size increase.

An example comes from the Reliability theory, in the case we need to compute the fault probability of a system which is made of a connection (serial and\or parallel) of elements whose fault probabilities are known.

4.8 Conclusions

The orthogonal form of a Boolean formula has remarkable properties. Several hard problems become easy when in orthogonal form. Every logic formula can be transformed in orthogonal form. A general procedure for orthogonalization is developed. The problem is indeed computationally demanding. As predictable, in the initial phase of the procedure, the size of the formula tends to exponentially increase. On the other hand, the size of the formula decreases again when approaching to the final phase. In spite of this size growth, orthogonalization appears to be the preferable way to solve some practical problems, for instance in the field of Reliability theory. Due to relatively novelty of the topic, the presented algorithm is still under construction.

Chapter 5

Logic Programming Techniques for an Error Free Data Collecting

When dealing with a large number of collected information, a relevant problem arises: perform all the requested elaboration considering only correct data. Examples of data collecting are for instance cases of statistical investigations, marketing analysis, experimental measures, etc. Corresponding examples of data elaboration could therefore be calculating statistical parameters, tracing consumers profiles, estimating unknown measured parameters, etc. Data correctness is a crucial aspect of data quality, and, in practical cases, it has always been a very computationally demanding problem.

Seldom data are single values. Generally, they are structured into sets of values, whose elements have a specific meaning, and are binded by specific relations. This set of p values v_i (the data) for a set of p fields f_i (their meaning) is usually called a record R (among other, in the field of databases). We will therefore consider records in the following form.

$$R = \{f_1 = v_1, f_2 = v_2, \ldots, f_p = v_p\}$$

The problem of error detection is generally approached by formulating a set of rules that the records must respect in order to be reliable, or *consistent*. In the absence of further information, consistent records are declared *correct*. Instead, inconsistent records are declared *erroneous*. The more accurate and careful the rules are, the more truthful individuation of correct and erroneous data can be achieved. Such rules can involve the value

of a single field (e.g. a value v_i must be within a set V) or a combination of values within a record (e.g. a value v_i must be equal to a value v_j plus a value v_k).

A first problem arising from this fact is the validation of such set of rules. In fact, the rules could contain some contradiction among themselves, or some rule could be implied by some other. This could result in erroneous records to be declared correct, and vice versa. This point is discussed in more detail is in section 2. The above problem of checking the set of rules against inconsistencies and redundancies is transformed in a Propositional Logic problem. This is done by encoding the rules in clauses, as explained in section 3. A sequence of propositional Satisfiability problems (SAT for short) is therefore solved, as illustrated in section 4. This procedure allows, moreover, to check if a new rule is in contradiction with the previous ones, or if they already imply it. This will reveal its importance in a phase of updating.

By choosing this clausal representation, the detection of erroneous records simply becomes the problem of testing if a truth assignment for its logical variables satisfies a propositional logic formula. See section 5 for details.

In the subsequent phase of error correction [186], the erroneous (or *inconsistent*) records must be changed in order to satisfy the above rules. This should be done by keeping as much as possible the correct information contained in such erroneous records. Two general principles should be followed: to apply the minimum changes to erroneous data, and to modify as less as possible the marginal and joint frequency distribution of the data [78]. This is described in more detail in section 6. The above problem is modeled by encoding the rules by linear inequalities, and solving a sequence of set covering problems, as explained in section 7.

The proposed procedure is tested by performing the entire process of error detection and correction in the case of a real world Census. The application and part of the data were kindly provided by the Italian National Statistic Institute (ISTAT). Results are in section 8.

5.1 Data Collecting through Questionnaires

Our attention will be focused on the problem of statistic projections carried out by processing answers obtained through a collection of questionnaires. We stress that such problem is used just as an example to apply our methodology, and of course does not exhaust field of application of the proposed procedure.

A record, in this case, is the set of the answers to one questionnaire Q. We therefore have

$$Q = \{f_1 = v_1, f_2 = v_2 \ldots, f_p = v_p\}$$

We will consider, in particular, the case of a census of population. Examples of fields f_i are `age` or `marital status`, corresponding examples of values v_i are `18` or `single`. Fields can be distinguished in quantitative and qualitative ones. Roughly speaking, a quantitative field require its value to be either a real number a in some interval $[a_1, a_2]$, or an integer number n in some set N, or at least a value belonging to an ordered set. In a quantitative field, in fact, the order operators '$<$', '$\leq$', '$>$', '$\geq$' must be defined. On the other hand, a qualitative field require its value d to be a member of some discrete set $D = \{d_1, d_2, \ldots, d_n\}$.

Errors, or, more precisely, inconsistencies between answers or out of range answers, can be due to the original compilation of the questionnaire, or can be introduced during any later phase of information processing, such as data input or conversion. Inconsistent questionnaires could contain information that deeply modifies the aspects of interest (just think of maximum or minimum of some value), and thus, without their detection, our statistical investigation would produce erroneous results. We can distinguish between stochastic errors and systematic errors. Stochastic errors are randomly introduced, and can therefore be unpredictable and have in general low or no correlation. On the other hand, systematic errors consists in a repetition of the same error. This can be, for instance, due to some defect in the questionnaire. Of course, both of these kinds of errors must be identified.

Generally, real world statistics involve such a high number of questionnaires that an automatic procedure to carry out error detection is needed. Usually, National Statistic Offices perform the task of detecting inconsistencies by formulating rules that must be respected by every correct set of answers. More precisely, rules are written in form of *edits*. An edit expresses the error condition. The following example will clarify this point.

Example 2.1. An inconsistent answer can be to declare

`marital status` as `married` and `age` as `10` years old.

The rule to detect this kind of errors could be the following

if `marital status` is `married`, `age` must be not less than, say, `14`.

The rule must be put in form of an edit, which expresses the error condition. Since we have the error if `marital status` = `married` and `age` < `14`, the edit for this example would therefore be

$$(\texttt{marital status} = \texttt{married}) \wedge (\texttt{age} < \texttt{14})$$

The set of all the edits is sometimes called the set of edit rules, or Check plan, or Compatibility plan, of a statistical investigation. Such set of edits is used to split the set of all questionnaires in the two subsets of correct ones and erroneous ones. Questionnaires which verify the condition defined in at least one edit are declared erroneous. We will say that such questionnaires activate the edits.

Obviously, the individuation of the set of edits itself plays a crucial role. In fact, the set of edits must be free from inconsistency (i.e. edits must not be in contradiction each other), and, preferably, from redundancy (i.e. do not contain edit which are logically implied from other edits). In the case of real questionnaires, edits can became very numerous. The cardinality of the set of edits, in fact, increases with the number of questions in the questionnaire. Moreover, a high number of edits allows a more accurate error detection. Test for contradictions and redundancies must be automatically performed as well. Therefore, a form of edits representation that can be treated by automatic elaboration is needed.

Many commercial software systems deal with the problem of questionnaires correction, and they make use of a variety of different (and sometimes *naive*) edits encoding and solution algorithm [10, 269, 209]. In practical case, however, they suffer from severe limitations, due to the inherent computational complexity of the problem. Some methods ignore edit testing, and just separate erroneous questionnaires from correct ones. Their limitations are that results are incorrect if edits are incorrect, and edits updating turns out to be very difficult. This cause that number of edits must be small enough to be validated by inspection by a human operator.

On the other hand, other methods try to check for contradiction and redundancy by generating all implied edits, such as the 'Fellegi Holt' procedure [78]. Their limitation is that, as the number of edits slightly increases, they produce very poor performance. This happens, of course, because of the huge demand of computational resources required for generating all implied edits, whose number exponentially grows with the number of original edits. The above limitations prevented to now the use of a set of edits whose cardinality is above a certain value. Another serious drawback is

that simultaneous processing of quantitative and qualitative fields is seldom allowed.

5.2 A Logical Representation of the Set of Edits

The usefulness of logic or Boolean techniques is proved by many approaches to similar problems of information representation ([25, 221] among others). This should not be surprising, when considering the role of the science of Logic. A representation of the set of edit by means of first-order logic is not new. This methodology turns out to be equivalent to the 'Fellegi-Holt' one [13], with consequent computational limitations. In this paper we propose an edit encoding by means of the simpler propositional logic. Treatment of numerical data is performed by a process called binarization [25].

A propositional logic formula is composed of propositions, i.e. logical variables (also called binary, or Boolean, variables), which assume values in the set $\{True, False\}$, or, equivalently, $\{1, 0\}$, and of the logical connectives $\{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, with their usual meaning of 'and', 'or', 'implies', 'is equivalent'. Propositions can be positive (a logical variable α) or negative (a negated logical variable $\neg\alpha$). Every Propositional Logic formula can be put in conjunctive normal form (CNF), namely a conjunction ($\wedge$) of disjunctions ($\vee$). A CNF formula $\mathcal{F}$, with n logical variables and m clauses, has the following general structure:

$$(\alpha_{i_1} \vee ... \vee \alpha_{j_1} \vee \neg\alpha_{k_1} \vee ... \vee \neg\alpha_{n_1}) \wedge \ldots \wedge (\alpha_{i_m} \vee ... \vee \alpha_{j_m} \vee \neg\alpha_{k_m} \vee ... \vee \neg\alpha_{n_m})$$

Given truth values ($True$ or $False$) to the logical variables, we have a truth value for the whole formula. A formula $\mathcal{F}$ is satisfiable if and only if there exists a truth assignment that makes the formula $True$ [65, **?**, 146, 101]. If this does not exist, the formula $\mathcal{F}$ is unsatisfiable.

The problem of testing satisfiability of propositional formulae in conjunctive normal form, named SAT, plays a protagonist role in mathematical logic and computing theory. Actually, it is fundamental in Artificial Intelligence, Expert Systems, Deductive Database theory, due to its ability of formalizing deductive reasoning, and thus solving logic problems by means of automatic computation. It is known to be NP-complete [98]. Satisfiability problems indeed are used for encoding and solving a wide variety of problems arisen from different fields, e.g. VLSI logic circuit design and testing, programming language project, computer aided design.

A SAT formulation can be used to solve the problem of logical implication, i.e. to detect if a given proposition is logically implied by a set of

propositions [163, 102, 176, 64].

In the case of questionnaires, every edit can be encoded in a propositional logic clause. Moreover, since the edit have a very precise syntax, this encoding can be done by an automatic procedure [31]. The set of edits E, written by the person entrusted of this task at the ISTAT, and according to the grammar used by them, is therefore transformed in a CNF propositional formula $\mathcal{E}$, following the sequence of steps listed below and described in further subsections:

Edit propositional encoding procedure

1. *Identification of the domains* D_f *for each one of the* p *field* f*, considering that we are dealing with errors.*
2. *Identification of subsets* $S_f^1, S_f^2, \ldots$ *in every domain* D_f*, defined by breakpoints, or cut points,* $b_f^1, b_f^2, \ldots$ *obtained from the edits.*
3. *Identification of equivalent subsets* $S_f^j, S_k^q, \ldots$*, and definition of equivalence classes* $C_f^h = [S_f^j]$.
4. *Definition of* n *logical variables* $\alpha_{[S_f^j]}$ *to represent the equivalence classes* $[S_f^j]$.
5. *Expression of all the edits by means of clauses defined over the introduced logical variables* $\alpha_{[S_f^j]}$.
6. *Identification of congruency clauses to supply the information not contained in edits.*

Note that the above procedure is merely formal, i.e. not depending by the meaning of the involved propositions. Therefore, it can be entirely performed by means of automatic elaboration.

5.2.1 Identification of the domains for the fields

In this step, the totality of possible answers need to be identified, including incorrect ones and blank answer. Such possibilities depend, of course, on the nature of the field (qualitative or quantitative), but also on the typographical aspect of the question in the questionnaire (for instance, single choice question, text box question, etc.). Such sets of possible answer, for the generic field f, will be indicated as D_f. D_f can include intervals of real numbers, sets of elements, etc. A generic value belonging to domain D_f will be indicated as $v_f \in D_f$.

Example 3.1. Consider a `field marital` status represented as follows:

Marital status:
☐ single ☐ married ☐ separate ☐ divorced ☐ widow

Answer can vary only on a discrete set of possibilities in mutual exclusion, or, due to errors, be missing or not meaningful (for instance when we have more than one choice). Both latter cases are expressed with the value `blank`.

$$D_{\texttt{marital status}} = \{\texttt{single}, \texttt{married}, \texttt{separate}, \texttt{divorced}, \texttt{widow}, \texttt{blank}\}$$

Example 3.2. Consider a field `age` represented as follows:

Age: ________ years

Due to errors, the totality of possible answers can be any number or be `blank`. Although this may seem too pessimistic, note that similar choices improve robustness of the procedure. We have

$$D_{\texttt{age}} = (-\infty, +\infty) \cup \{\texttt{blank}\}$$

From the above example we can see that a quantitative field can have a domain whose elements are not only numbers. To perform such identification of the domains D_f, a characterization of the qualitative or quantitative

nature of every field and of its typographical aspect must be given in input to the procedure.

5.2.2 Identification of the breakpoints for the fields

Edits are propositions involving one or more values $v_{f_1} \in D_{f_1}, v_{f_2} \in D_{f_2}, \ldots$ for one or more fields $f_1, f_2, \ldots$. As told, they state the error condition. In our application, they may have one of the two following logical structure:

$$f_1 < relation > v_{f_1}$$

$$(f_1 < relation > v_{f_1}) < logic\ connective > (f_2 < relation > v_{f_2})$$

where $< relation >$ is one of '=', '<', '>', '≤', '≥', '∈', etc., and $<$ *logic connective* $>$ is one of ⇒, ⇔, ∧ (cfr. Example 2.1.). Of course, order relations are used only in the case of ordered domains (quantitative fields).

Values v_f appearing in the edits are called *breakpoints*, or *cut points*, for the domains. They represent the logical *watershed* between values of the domain. Such particular values will be indicated with b_f^j. All the breakpoints b_f^j can be automatically detected by reading the edits.

We can observe that the expression $(f < relation > v_f)$ represents a set of values of the domain D_f. This can be a single values $d_f \in D_f$ (when the relation is '='), or a sets of values with cardinality > 1, or an interval of numbers. In all the above cases, anyway, $(f < relation > v_f)$ denotes a subset of domain D_f, and, in order to avoid too many case distinctions, they will all be called subsets $S_f^j \subseteq D_f$. We congruently have

$$D_f = \bigcup_j S_f^j$$

In order to identify all subsets S_f^j, the breakpoints b_f^j are used to partition domain D_f according to the edits. Some domains can be also very fragmented.

Example 3.3. For the field `marital status`, by reading an imaginary set of edits, we have the following breakpoints

$$b^1_{\texttt{marital status}} = \texttt{single}$$
$$b^2_{\texttt{marital status}} = \texttt{married}$$
$$b^3_{\texttt{marital status}} = \texttt{separate}$$
$$b^4_{\texttt{marital status}} = \texttt{divorced}$$
$$b^5_{\texttt{marital status}} = \texttt{widow}$$
$$b^6_{\texttt{marital status}} = \texttt{blank}$$

and, by using the breakpoints and the edits to cut the domain $D_{\texttt{marital status}}$, we have the subsets

$$
\begin{array}{l}
S^1_{\texttt{marital status}} = \{\texttt{single}\} \\
S^2_{\texttt{marital status}} = \{\texttt{married}\} \\
S^3_{\texttt{marital status}} = \{\texttt{separate}\} \\
S^4_{\texttt{marital status}} = \{\texttt{divorced}\} \\
S^5_{\texttt{marital status}} = \{\texttt{widow}\} \\
S^6_{\texttt{marital status}} = \{\texttt{blank}\}
\end{array}
$$

Example 3.4. For the field age, by reading an imaginary set of edits, we have the following breakpoints

$$
\begin{array}{l}
b^1_{\mathsf{age}} = \mathsf{0} \\
b^2_{\mathsf{age}} = \mathsf{14} \\
b^3_{\mathsf{age}} = \mathsf{18} \\
b^4_{\mathsf{age}} = \mathsf{26} \\
b^5_{\mathsf{age}} = \mathsf{120} \\
b^6_{\mathsf{age}} = \texttt{blank}
\end{array}
$$

and, by using the breakpoints and the edits to cut the domain D_{age}, we have the subsets

$$
\begin{array}{l}
S^1_{\mathsf{age}} = (-\infty, \mathsf{0}) \\
S^2_{\mathsf{age}} = [\mathsf{0}, \mathsf{14}) \\
S^3_{\mathsf{age}} = [\mathsf{14}, \mathsf{18}) \\
S^4_{\mathsf{age}} = \{\mathsf{18}\} \\
S^5_{\mathsf{age}} = (\mathsf{18}, \mathsf{26}) \\
S^6_{\mathsf{age}} = [\mathsf{26}, \mathsf{120}] \\
S^7_{\mathsf{age}} = (\mathsf{120}, +\infty) \\
S^8_{\mathsf{age}} = \{\texttt{blank}\}
\end{array}
$$

Note that, for real valued numerical fields, depending on the relations in the edit ('$<$', '$>$' '$\leq$', '$\geq$'), subsets are intervals close, open, left close, left open, etc.

5.2.3 Individuation of equivalent classes of subsets

Depending on the edits, some subsets S^a_f, S^b_f, ... can be equivalent. This happens when, given a value d_f for a field f, the following alternative cases

$$d_f \in S^a_f,\ d_f \in S^b_f,\ \ldots$$

activate exactly the same edits for every other combination of values for other fields. This means that, if d_f is in $S_f^a \cup S_f^b \cup \ldots$, changing its value still in $S_f^a \cup S_f^b \cup \ldots$ can never change correctness result for any questionnaire. By considering such equivalence relationship, we introduce equivalence classes for that. The above equivalence class will be indicated as $[S_f^a]$. Such equivalent subsets, can be identified from the edits as follows. Given a group of subsets $d_f \in S_f^a$, $d_f \in S_f^b$, $\ldots$, if all the edits where they appear are identical except for $< relation >$ and value d_i (the two elements which define the subset of the field f - see edit structure in subsect. 3.2) $d_f \in S_f^a$, $d_f \in S_f^b$, $\ldots$ are equivalent. Equivalence classes can therefore be automatically identified.

Example 3.5. If all the edits where `married` and `separate` appear would be

`marital status` $=$ `married` $\wedge$ `age` $<$ `14`
meaning: if `marital status` is `married`, `age` must be not less than `14`.

`marital status` $=$ `separate` $\wedge$ `age` $<$ `14`
meaning: if `marital status` is `separate`, `age` must be not less than `14` (must be married to be separate).

the two subsets `married` and `separate` would be equivalent. Since, instead, they also appear in other edits which do not satisfy the above condition, `married` and `separate` are not equivalent. In this case the equivalence condition never holds for the field `marital status`, hence we have no equivalent subset in it. Therefore, there is a different class for every subset.

Example 3.6. On the contrary, for the field `age`, some subsets are equivalent. In particular, edits representing the concept *out of normal* are

(`age` $<$ `0`)
meaning: `age` cannot be less than `0`.

(`age` $>$ `120`)
meaning: `age` cannot be more than `120`.

(`age` $=$ `blank`)
meaning: `age` cannot be `blank`.

The above subsets does not appear in any other edit, and the above edits are identical except for $< relation >$ and value d_i, hence the subsets $\{(-\infty, 0)\}, \{(120, +\infty)\}, \{\texttt{blank}\}$ are equivalent and collapse in to the same

class. Moreover, also the subsets $\{\texttt{18}\}$ and $\{(\texttt{18}, \texttt{26})\}$ results to be equivalent. There are no further equivalent subsets. Altogether, for the field `age`, we have the classes

$$
\begin{aligned}
C^1_{\texttt{age}} &= [\{\texttt{blank}\}] = \{(-\infty, \texttt{0}), (\texttt{120}, +\infty), \texttt{blank}\} \\
C^2_{\texttt{age}} &= [[\texttt{0}, \texttt{14})] = \{[\texttt{0}, \texttt{14})\} \\
C^3_{\texttt{age}} &= [[\texttt{14}, \texttt{18})] = \{[\texttt{14}, \texttt{18})\} \\
C^4_{\texttt{age}} &= [(\texttt{18}, \texttt{26})] = \{\texttt{18}, (\texttt{18}, \texttt{26})\} \\
C^5_{\texttt{age}} &= [[\texttt{26}, \texttt{120}]] = \{[\texttt{26}, \texttt{120}]\}
\end{aligned}
$$

5.2.4 Definition of logic variables

The logical variables are used to encode the information about which class contains the value of every field. This can be done in several ways, and pursuing different aims. We choose the following, with the aim to produce an easier CNF formula. The definition of logical variables is slightly different in the case of qualitative or quantitative fields.

For every qualitative field with n classes, we use $n-1$ logic variables α_i corresponding to $n-1$ classes. If the qualitative field f has a value belonging to the class C^j_f we put $\alpha_{C^j_f} = True$. The same holds for the other classes of the field, except for one, for instance the last one C^n_f. When the field f has a value in this last class C^n_f, we put all variables at $False$.

Example 3.7. The field `marital status` is divided in 6 equivalence classes (see example 3.5). We therefore have 6-1 = 5 logical variables $\alpha_{[\texttt{single}]}$, $\alpha_{[\texttt{married}]}$, $\alpha_{[\texttt{separate}]}$, $\alpha_{[\texttt{divorced}]}$, $\alpha_{[\texttt{widow}]}$, as shown below.

$$v_{\texttt{marital status}} \in [\texttt{single}] \quad \Rightarrow \quad \begin{cases} \alpha_{[\texttt{single}]} = True \\ \alpha_{[\texttt{married}]} = \alpha_{[\texttt{separate}]} = \\ = \alpha_{[\texttt{divorced}]} = \alpha_{[\texttt{widow}]} = False \end{cases}$$

$$v_{\texttt{marital status}} \in [\texttt{married}] \quad \Rightarrow \quad \begin{cases} \alpha_{[\texttt{single}]} = False \\ \alpha_{[\texttt{married}]} = True \\ \alpha_{[\texttt{separate}]} = \alpha_{[\texttt{divorced}]} = \alpha_{[\texttt{widow}]} = False \end{cases}$$

$$v_{\texttt{marital status}} \in [\texttt{separate}] \quad \Rightarrow \quad \begin{cases} \alpha_{[\texttt{single}]} = \alpha_{[\texttt{married}]} = False \\ \alpha_{[\texttt{separate}]} = True \\ \alpha_{[\texttt{divorced}]} = \alpha_{[\texttt{widow}]} = False \end{cases}$$

$$v_{\texttt{marital status}} \in [\texttt{divorced}] \quad \Rightarrow \quad \begin{cases} \alpha_{[\texttt{single}]} = \alpha_{[\texttt{married}]} = \alpha_{[\texttt{separate}]} = False \\ \alpha_{[\texttt{divorced}]} = True \\ \alpha_{[\texttt{widow}]} = False \end{cases}$$

$$v_{\texttt{marital status}} \in [\texttt{widow}] \quad \Rightarrow \quad \begin{cases} \alpha_{[\texttt{single}]} = \alpha_{[\texttt{married}]} = \\ = \alpha_{[\texttt{separate}]} = \alpha_{[\texttt{divorced}]} = False \\ \alpha_{[\texttt{widow}]} = True \end{cases}$$

$$v_{\texttt{marital status}} \in [\texttt{blank}] \quad \Rightarrow \quad \begin{cases} \alpha_{[\texttt{single}]} = \alpha_{[\texttt{married}]} = \alpha_{[\texttt{separate}]} = \\ = \alpha_{[\texttt{divorced}]} = \alpha_{[\texttt{widow}]} = False \end{cases}$$

For every quantitative field with n classes, we use $n-1$ variables. The difference with former case is that these variables do not correspond to classes. They correspond instead to nested intervals, all of them having as initial point the smaller feasible value for that field, and as final point the identified breakpoints. Externally from the bigger nested interval it remains the 'out of range' class. This choice for variable association results, in our case, in shorter clauses. In fact, in most of edits appear similar intervals. They can therefore be expressed with one variable instead of more ones.

If the quantitative field f has a value in one of the nested intervals `[a,b]` we put $\alpha_{[\texttt{a,b}]} = True$. The other variable must be set accordingly, as illustrated below. The same holds for the other classes of the field, except for one, for instance the last one C_f^n. When the field has a value in the 'out of range' class, we put all variables at $False$.

Example 3.8. The field `age` is divided in 5 equivalence classes (see example 3.6). We therefore have 5-1 = 4 logical variables $\alpha_{[0,14)}$, $\alpha_{[0,18)}$, $\alpha_{[0,26)}$, $\alpha_{[0,120]}$, as illustrated below.

$$
\begin{aligned}
v_{\texttt{age}} \in [[0,14)] &\Rightarrow \begin{cases} \alpha_{[0,14)} = \alpha_{[0,18)} = \\ = \alpha_{[0,26)} = \alpha_{[0,120]} = True \end{cases} \\
v_{\texttt{age}} \in [[14,18)] &\Rightarrow \begin{cases} \alpha_{[0,14)} = False \\ \alpha_{[0,18)} = \alpha_{[0,26)} = \alpha_{[0,120]} = True \end{cases} \\
v_{\texttt{age}} \in [[18,26)] &\Rightarrow \begin{cases} \alpha_{[0,14)} = \alpha_{[0,18)} = False \\ \alpha_{[0,26)} = \alpha_{[0,120]} = True \end{cases} \\
v_{\texttt{age}} \in [[26,120]] &\Rightarrow \begin{cases} \alpha_{[0,14)} = \alpha_{[0,18)} = \alpha_{[0,26]} = False \\ \alpha_{[0,120]} = True \end{cases} \\
v_{\texttt{age}} \in [\texttt{blank}] &\Rightarrow \begin{cases} \alpha_{[0,14)} = \alpha_{[0,18)} = \\ = \alpha_{[0,26]} = \alpha_{[0,120]} = False \end{cases}
\end{aligned}
$$

We therefore have the following graphical representation of the nested intervals.

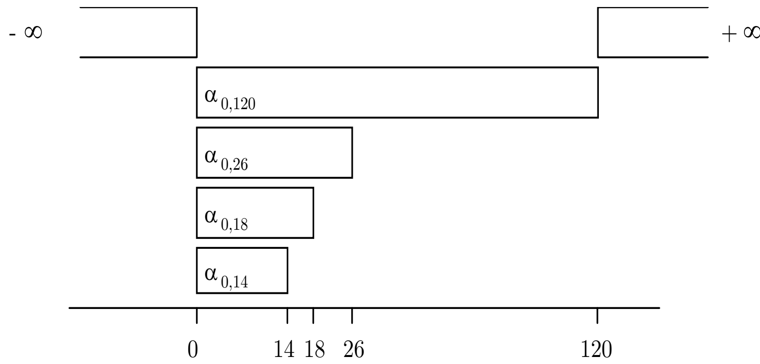

Figure 5.1: Nested subsets for the field `age`.

5.2.5 Encoding of edits as clauses

So far, every edit can be encoded in clauses by using the defined variables. Every expressions ($f_i < relation > v_{f_i}$) can in fact be substituted by the corresponding logical variable, obtaining therefore a sequence of logic variables connected by logic operators, hence a logic formula.

We are interested in producing clauses which are satisfied by consistent answers. Being edits the description of the error condition, we now need to negate the obtained logic formula. This negated logic formula can be then transformed in CNF format, by following simple syntactical logic rules. The so generated CNF formula is satisfied by every set of correct questionnaires answers, and are not satisfied by every set of inconsistent or out of range answer. With the particular edit structure considered, every edit produces one and only one clause. However, in a general case, this could not hold, but the procedure works fine as well.

Example 3.9. Following the above example 3.1, the given edit is

$$\texttt{marital status} = \texttt{married} \wedge \texttt{age} < \texttt{14}$$

By substituting the logical variables, we have the following logic formula

$$\alpha_{[\texttt{married}]} \wedge \alpha_{[\texttt{0,14})}$$

By negating it, and applying De Morgan's law, we obtain the following clause

$$\neg\alpha_{[\texttt{married}]} \vee \neg\alpha_{[\texttt{0,14})}$$

5.2.6 Identification of congruency clauses

In addition to information given by edits, there is information which is not contained in edits, and that a human operator would consider obvious, but which must be provided to an automatic elaboration. In our case, we need to express that fields must have one and only one value, and therefore other clauses, named congruency clauses, need to be added.

By using $n-1$ variables for n equivalence classes, there is no need to add clauses expressing that we must have a value being in at least one class, because it does not exist a truth assignment for variables that doesn't verify this.

Instead, it must be expressed that the value for a field must be in only one class. Considering the case of qualitative fields (such as the example of `marital status`), we have the n variables corresponding to the $n+1$ disjoint classes. The above condition becomes that only one variable can be true. This is imposed by adding clauses constituted by all the possible couples of negated variables. Their number is therefore $\binom{n}{2}$ (number of combination of class 2 of n objects).

Example 3.10. In the case of the qualitative field `marital status`, we have 5 variables, hence the congruency clauses are $\binom{5}{2} = 10$, as follows:

$$
\begin{array}{ccc}
\neg\alpha_{[single]} & \vee & \neg\alpha_{[married]} \\
\neg\alpha_{[single]} & \vee & \neg\alpha_{[separate]} \\
\neg\alpha_{[single]} & \vee & \neg\alpha_{[divorced]} \\
\neg\alpha_{[single]} & \vee & \neg\alpha_{[widow]} \\
\neg\alpha_{[married]} & \vee & \neg\alpha_{[separate]} \\
\neg\alpha_{[married]} & \vee & \neg\alpha_{[divorced]} \\
\neg\alpha_{[married]} & \vee & \neg\alpha_{[widow]} \\
\neg\alpha_{[separate]} & \vee & \neg\alpha_{[divorced]} \\
\neg\alpha_{[separate]} & \vee & \neg\alpha_{[widow]} \\
\neg\alpha_{[divorced]} & \vee & \neg\alpha_{[widow]}
\end{array}
$$

Considering now the case of quantitative fields (such as the example of `age`), we have n variables corresponding to n nested subsets. As observed above, we can have either the case of a quantitative field divided into nested intervals of reals, or the case of a quantitative field divided into nested sets of integer numbers. We will discuss the procedure speaking about nested intervals, but, of course, the case of nested sets of integers is completely analogous.

The congruency condition becomes that, if a variable $\alpha_{[a_1,b_1]}$, corresponding to an interval $[a_1, b_1]$, is $True$, also all the variables $\alpha_{[a_2,b_2]}, \alpha_{[a_3,b_3]}, \ldots,$ corresponding to all the intervals $[a_2, b_2], [a_3, b_3], \ldots,$ containing $[a_1, b_1]$, must be $True$. The above condition is imposed by adding $(n-1)+(n-2)+\ldots+1 = \binom{n}{2}$ clauses expressing

$$\alpha_{[a_1,b_1]} \Rightarrow \alpha_{[a_2,b_2]}, \alpha_{[a_1,b_1]} \Rightarrow \alpha_{[a_3,b_3]}, \ldots, \alpha_{[a_2,b_2]} \Rightarrow \alpha_{[a_3,b_3]}, \ldots,$$

Converting in CNF by applying elementary logic rules ($\alpha_a \Rightarrow \alpha_b$ is equivalent to $\neg\alpha_a \vee \alpha_b$), we obtain

$$\neg\alpha_{[a_1,b_1]} \vee \alpha_{[a_2,b_2]}, \neg\alpha_{[a_1,b_1]} \vee \alpha_{[a_3,b_3]}, \ldots, \neg\alpha_{[a_2,b_2]} \vee \alpha_{[a_3,b_3]}, \ldots,$$

Example 3.11. In the case of the quantitative field `age`, we have 4 variables corresponding to 4 nested subsets, hence the congruency clauses

are $\binom{4}{2} = 6$, as follows:

$$
\begin{array}{lcl}
\neg\alpha_{[0,14)} & \vee & \alpha_{[0,18)} \\
\neg\alpha_{[0,14)} & \vee & \alpha_{[0,26)} \\
\neg\alpha_{[0,14)} & \vee & \alpha_{[0,120]} \\
\neg\alpha_{[0,18)} & \vee & \alpha_{[0,26)} \\
\neg\alpha_{[0,18)} & \vee & \alpha_{[0,120]} \\
\neg\alpha_{[0,26)} & \vee & \alpha_{[0,120]}
\end{array}
$$

So far, given the set of edits, we have a set of m clauses, and, given the set of answers to a questionnaire, we have a truth assignment for the n logical variables. By construction, a record which does not activate any edit, will satisfy all the clauses, and a record which activates some edits will not satisfy the corresponding clauses. We therefore have that the truth assignment given by a record must satisfy all the clauses to be declared correct. We will hence consider the conjunction of all the clauses, that is a CNF formula $\mathcal{E}$, and say, briefly, that the questionnaire Q must satisfy $\mathcal{E}$ to be declared correct.

5.3 Edits Validation

As told, edits must be free from inconsistency (i.e. edits must not be in contradiction each other), and, preferably, from redundancy (i.e. do not contain edit which are logically implied from other edits). The test for contradictions and redundancies is very hard for a human operator, and impossible above a certain number of edits. On the other hand, when such test is automatically performed, it historically turns out to be very computationally demanding.

A diffuse approach follows the so-called 'Fellegi-Holt' methodology [78]. This consists in checking for contradiction and redundancy by generating all the (new) edits which are implied by the given edits. The generation of all implied edits allows to check if the given edits imply some *hidden* rule that should not hold. Unfortunately, but predictably, the number of implied edits is exponential in the number of given edits. This is the main reason for which this methodology, although theoretically irreprehensible, is recently recognized to be not applicable for many real-world problems.

Edit representation by means of first-order logic was already proposed, for instance in [13]. This allows a formal and convenient description of the edit and the operations. However, something analogous to the generation

of all implied edits must be performed in this case also. The computational complexity of the problem remains exponential.

Being our proposal to perform edit validation in the case of real world problems, we must get rid of the generation of all implied edits. By means of a propositional logic representation, the problem of checking for inconsistency and redundancy can be formalized as follows.

5.3.1 Complete Inconsistency in the Set of Edits

When every possible set of answers to the questionnaire is declared incorrect, we have the situation called *complete inconsistency* of the set of edits. In fact, reminding that edits describe the inconsistent situation, let's consider the following example.

Example 4.1. The following is a situation of Complete Inconsistency. This is, of course, a very simple and evident one. More complex ones, involving dozens of edits, are not so easily visible. Below every edit its meaning is explained.

`seaside house` = `no`
meaning: everybody must have a seaside house.

`mountain house` = `no`
meaning: everybody must have a mountain house.

(`seaside house` = `yes`) $\wedge$ (`mountain house` = `yes`)
meaning: it is not allowed to have both seaside and mountain house.

Using these edits, every questionnaire fails the check, because it cannot exist any set of answers which appear consistent. Edits are always activated.

In a large set of edits, or in a phase of edits updating performed by people different from the original edit writers, the situation of complete inconsistency may occur.

Claim 4.1. *By encoding edits in clauses, complete inconsistency correspond to a CNF formula that cannot be satisfied, namely an unsatisfiable formula.*

Complete inconsistency can therefore be detected by checking the satisfiability of the whole CNF formula.

Moreover, when having a complete inconsistency and in order to restore consistency, it would be very useful to know which are the inconsistent edits. This corresponds to selecting which part of the unsatisfiable CNF cause the unsolvability, i.e. a minimal unsatisfiable subformula (MU) [164, 79, 33].

This could be done by means of the used SAT solver, which, in the case of unsatisfiable instances, is able to select a subset of clauses which are still unsatisfiable, and thus causes the unsatisfiability of the whole instance [33]. Therefore, the inconsistent edits are the ones corresponding to such clauses, and should be changed (by the human adviser who writes the edits, of course). The usefulness of the individuation of such inconsistent subset of edits is easily understandable when thinking of the prospect of checking, for instance, a dozen of edits instead of one thousand.

5.3.2 Partial Inconsistency in the Set of Edits

More insidious because less easy to detect without an automatic procedure in a large set of edits is the situation of *partial inconsistency* of the set of edits. This happens when some questionnaires, which are correct, are declared erroneous, only due to a particular value $v_f^\dagger$ of a single field f. In the intentions of the edit writer, $v_f^\dagger$ was a feasible value for the field f. Due to an unfortunate edit combination, however, there is an hidden and unwanted implied edit which forbids the value $v_f^\dagger$ for the field f.

$v_f^\dagger$ will be called a *sentence value*. When a correct questionnaire contains a sentence value, it is (erroneously) declared incorrect, due to edit's fault. For questionnaires not containing sentence values, partial inconsistency does not cause any problem. Let us consider indeed the following example.

Example 4.2. The following is a situation of partial inconsistency. Below every edit its meaning is explained.

$(\texttt{annual income} \geq \texttt{1000}) \Rightarrow (\texttt{seaside house} = \texttt{no})$
meaning: if annual income is greater then or equal to 1000, then the subject must have a seaside house.

$(\texttt{annual income} \geq \texttt{2000}) \Rightarrow (\texttt{mountain house} = \texttt{no})$
meaning: if annual income is greater then or equal to 2000, then the subject must have a mountain house.

$(\texttt{seaside house} = \texttt{yes}) \wedge (\texttt{mountain house} = \texttt{yes})$
meaning: it is not allowed to have both seaside and mountain house.

Using the above edits, every questionnaires where the subject has an `annual income` $\geq$ `2000` is declared erroneous, even if it should not. In fact, that answer necessarily activates the edits. Note that, for `annual income` $<$ `2000`, this partial inconsistence does not show any effect, and error detection proceeds in the correct way. The value `2000`, (and any other greater value) is a sentence value for the field `annual income`.

We can have more than one sentence value $v_f^\dagger$ forbidden by the same unwanted implied edit. The set of all values forbidden by an unwanted implied edit is in fact constituted by the subset S_f^j containing $v_f^\dagger$, together with all other subsets belonging to the same equivalence class (see sec. 3). The logical variable corresponding to the equivalence class $[S_f^j]$ will be called *sentence variable* $\alpha_f^\dagger$ for the field f. The set of all the values forbidden by one unwanted implied edit will be called *sentence set* $S_f^\dagger$.

$$S_f^\dagger \quad = \bigcup_{k:\, S_f^k \in [S_f^j],\, v_f^\dagger \in S_f^j} S_f^k$$

We will therefore say that we have partial inconsistency with regard to the set $S_f^\dagger$ for the field f, or, equivalently, with regard to the variable $\alpha_f^\dagger$ for the field f. Note that we can have more unwanted implied edits forbidding more variables of the same field.

In case of partial inconsistency, the CNF formula obtained from the edits is satisfiable. However, if for field f we have a sentence value, this corresponds to fixing the sentence variable $\alpha_f^\dagger$ to $True$.

Claim 4.2. *If we fix $\alpha_f^\dagger$ to $True$, and remove, as standard, satisfied clauses and all negated occurrence of that variable, the resulting formula becomes unsatisfiable.*

Basing on the above result, partial inconsistency with regard to any single variable α_k of the CNF formula can be detected by checking the satisfiability of all the CNF formulae obtained by independently fixing $\alpha_k = True$, for $k = 1, \ldots, n$.

Partial inconsistency with respect to a single variable α_k, will be called first-level partial inconsistency. Partial inconsistency with respect to a couple of variables α_k and α_h, will be called second-level partial inconsistency, and so on. We don't check all levels, otherwise we could not get rid from the exponential complexity that affects other procedures. What we do is

to check all first-level partial inconsistencies, and provide a tool that allows to check, if desired, higher-level partial inconsistencies. Note that neither 'Fellegi-Holt' methods check such partial inconsistencies, since they do dot lead to a complete contradiction when deriving all implied edits, and therefore are not automatically signaled. With 'Fellegi-Holt' methods, in fact, they could only be found by a (hypothetic) human inspector which examines every implied edit generated.

Moreover, in order to restore consistency, the used SAT solver is still able to select a subset of clauses which are unsatisfiable, and thus causes the situation of partial inconsistency. Individuation of inconsistent edits can therefore be carried out also in the case of partial inconsistency.

5.3.3 Redundancy in the set of edits

Some edits could be logically implied by others, being therefore redundant. It would be obviously preferable if we could remove them, because decreasing the number of edits while maintaining the same power of inconsistency detection can simplify the whole process and make it less error prone.

Example 4.3. The following is a (very simplified) situation of edit redundancy. Below every edit its meaning is explained.

$(\texttt{role} = \texttt{head of the house}) \wedge (\texttt{annual income} < 100)$
meaning: head of the house must have an annual income greater then or equal to 100.

$\texttt{annual income} < 100$
meaning: everybody must have an annual income greater then or equal to 100.

The first edit is clearly redundant.

A SAT formulation is generally used to solve the problem of logical implication, i.e., given a set of statements S called axioms, to decide whether another statement s logically derives from them, in symbols $S \Rightarrow s$. Representing statements S and s with clauses, we have $S \Rightarrow s$ if and only if $S \cup \neg s$ is an unsatisfiable formula [163, 176]. In our case this means:

Claim 4.3. *The clausal representation of an edit e_j is implied by the clausal representation of a set of edits E if and only if the CNF formula obtained by $E \cup \neg e_j$ is unsatisfiable.*

It can be consequently checked if an edit is redundant by removing its clausal representation e_j from the CNF formula, by adding its negation $\neg e_j$ to the formula, and by testing if the resulting formula is unsatisfiable. The redundancy of every edit can be checked by independently applying to each one of them the above operation

5.4 Individuation of Erroneous Records

Once we have a valid set of edit rules, they are used to detect erroneous records, in our case questionnaires. A correct questionnaire will be indicated with Q^r (right) and an erroneous one with Q^w (wrong). By using the propositional logic representation, this trivially becomes the problem of checking if the truth assignment corresponding to each questionnaire Q satisfies the CNF formula $\mathcal{E}$ obtained from the set of edits plus the congruency clauses. This operation can be performed with an extremely small computational effort. It results therefore suitable to check even a very large number of questionnaires.

5.5 The Problem of Imputation

After detection of erroneous records, if information collecting has no cost, we could just cancel erroneous records and collect new information until we have enough correct records. Since usually information collecting has a cost, we would like to use also the information contained in the erroneous records. This means changing the erroneous records in order to try to restore the unknown correct values. Such operation is called imputation.

We will call *original data* the unknown data that we would have if we had no errors, and *original frequency distributions* the unknown frequency distributions that we would have if we had no errors. We will correspondingly call *original questionnaire* Q^o the questionnaire that we would have if we had no errors.

In our case, therefore, given an erroneous questionnaire Q^w, the imputation process consists in changing some of his values in order to obtain a *corrected questionnaire* Q^c which satisfies the formula $\mathcal{E}$.

Imputation should be done by keeping as much as possible the correct information contained in erroneous data. Two general principles should be followed [78]:

- To apply the minimum changes to erroneous data.

- To modify as less as possible the marginal and joint original frequency distribution of the data.

The above principles frequently clash. We give an example to clarify this point.

Example 6.1. Consider this case of error: someone whose `age` is `70` and `marital status` is `married`. The original questionnaire would be

$Q^o = \{ \ldots \texttt{ age} = \texttt{70}, \ldots \texttt{ marital status} = \texttt{married}, \ldots \}$

However, the subject forgets to write the zero of `70` and writes `7`. The questionnaire we actually have is

$Q^w = \{ \ldots \texttt{ age} = \texttt{7}, \ldots \texttt{ marital status} = \texttt{married}, \ldots \}$

Such record is immediately detected as erroneous, since we have an edit which says 'if `marital status` is `married`, `age` must be $\geq$ `14`'. Suppose that, in virtue of some procedure (presented below), we know that the errors is in the field `age`. We therefore want to correct the value of `age`. The point is how to correct that. Imagine in fact that we have many records which are in the above situation, in the sense that their original values for `age` were any value above `14`, but the erroneous values we have on the questionnaire are all below `14`. If we just try to correct `age` by changing it as less as possible, we put all of such values at `14`, as follows.

$Q^c = \{ \ldots \texttt{ age} = \texttt{14}, \ldots \texttt{ marital status} = \texttt{married}, \ldots \}$

The statistical distribution of the original `age` would therefore be remarkably altered, by having a wrong peak in `14`. Note that, even if we try to correct `age` by choosing a random value above `14` which has the same frequency distribution of the `age` of married people (distribution obtained from the correct questionnaires) we could alterate the joint frequency distribution of the field `age` with respect to other fields. To continue the example, imagine that our subject (who is 70 years old), has, for the field `work`, the value `retired`. Imagine that the random value used to correct `age` is `50` (quite plausible). We have:

$Q^c = \{ \ldots \texttt{ age} = \texttt{50}, \ldots \texttt{ marital status} = \texttt{married}, \ldots \texttt{ work} = \texttt{retired} \}$

We can observe that the original (unknown) joint frequency distribution of the two fields `age` and `work` is altered. The same could happen for the other original joint frequency distributions.

There are two main approaches to the above problem of imputation: the deterministic and the probabilistic one. Such approaches can sometimes be combined.

Given an erroneous record, a deterministic imputation reconstruct the record with a deterministic procedure, renouncing to keep the original unknown joint frequency distributions. Note that errors can be stochastic or systematic. Stochastic errors are randomly introduced, and can therefore be unpredictable and have in general low or no correlation. Systematic errors consists in a repetition of the same error. This can be, for instance, due to some structural defect in the questionnaire. Deterministic methods are valid in the case of systematic errors, being in fact a systematic correction. Deterministic imputation just require a priori decisions, and therefore do not present specific computational difficulties.

A probabilistic imputation, on the contrary, for every erroneous record, tries to correct it by choosing new values which are not predetermined. The same erroneous records can therefore be corrected in more than one way. Such methods are generally preferred in the case of stochastic errors, because they assure a more uniform data correction, and can salvage the original frequency distributions. The drawback of probabilistic procedures usually is their computational burden.

5.5.1 Error Localization

A first problem arising when a questionnaire is declared erroneous, is to locate the error, namely to understand which are the erroneous fields. We could assume that the erroneous fields are the smallest set of fields that, if changed, permit to restore consistency, in the sense of not activating the edits anymore. This assumption is based on the fact that, when error is something unintentional, the more likely event is that errors are the smaller number. This is coherent with the principle of minimum change. In addiction to the above, one can argue that some fields can be more reliable than others. This means that the probability they are erroneous is lower. What is generally done is to give, for every field f_i, a measure $r_i \in \mathbb{R}_+$ of the reliability, which corresponds to a "preference" in keeping unchanged the value of field f_i. We define the total cost of a correction as the sum of the reliabilities r_i of the fields to modify f_i^w. By calling W the set of the f_i^w, the cost of such correction is

$$c\left(W\right) = \sum_{i:f_i^w \in W} r_i$$

Therefore, the problem of error localization is the following. Given the

erroneous questionnaire Q^w and the CNF formula $\mathcal{E}$ to be satisfied, we want to find a set W of fields f_i^w such that:

- The corrected questionnaire Q^c satisfying $\mathcal{E}$ can be obtained from the erroneous one Q^w by changing (only and all) the values of the $f_i^w \in W$.
- The total cost of required changes $c(W)$ is minimum.

We can in fact have more than one set of fields whose imputation can restore consistency, and we are interested in the one which changes as few as possible Q^w.

As for the values to put in such fields, two approaches are generally considered. One is to generate the imputed values by means of some (stochastic) function, although this could change the original unknown frequency distribution of the data (see example 6.1). The second is to use a donor questionnaire Q^d.

5.5.2 Imputation through a Donor

A donor questionnaire Q^d is a correct questionnaire which, according to some distance function $d(Q^w, Q^d) \in \mathbb{R}_+$, is the nearest one to the erroneous questionnaire Q^w we want to correct. Therefore, it represents a subject which has very similar characteristics. Given Q^w and Q^d, we simply copy the values of the fields f_i^w that we need to change from the donor Q^d to the erroneous Q^w. This procedure is generally recognized to cause a low alteration of the original frequency distributions.

Example 6.2. Consider this case of the error in example 6.1: someone whose `age` is `70` and `marital status` is `married`. The erroneous questionnaire we have is

$$Q^w = \{\ \ldots\ \texttt{age} = \texttt{7},\ \ldots\ \texttt{marital status} = \texttt{married},\ \ldots\ \}$$

Assume that the field `marital status` has a cost $c_{\texttt{marital status}} = 8$, and the field `age` has a cost $c_{\texttt{age}} = 3$. This means that the answer to the field `marital status` is considered more reliable than the answer to the field `age`. We find that the set W of minimum cost is just the field $\{\texttt{age}\}$. We therefore want to perform the imputation of the field `age`.

Assume that, if we consider the values of all the fields of Q^w, the subject appears to be an elderly man. Let therefore assume that, by searching for a correct questionnaire at minimum distance (considering all fields) from Q^w we find another elderly man:

$Q^d = \{$ ... `age` = `72`, ... $\}$

We can now proceed with the imputation of the field `age` from the donor. This restores consistency. We obtain

$Q^c =\{$ ... `age` = `72`, ... `marital status` = `married`, ... $\}$

Note that, in this case, the corrected questionnaire Q^c is very similar to the original one Q^o. This does not happen by chance, but is due to the mechanism of imputation trough a donor.

However, two problems arise from the use of a donor. The first is that, if we have not enough correct questionnaires to find a donor which is close to Q^w, the imputed values can be not so similar to the original ones. The second is that, by using a donor, we are not guaranteed that the set of erroneous fields of minimum cost W is enough to restore consistency of Q^w. In fact, W is that set of fields at minimum cost that can be changed to restore consistency, but the use of a donor does not let to change such fields as much as we like. We could need to take more fields from the donor, or we could need to take the fields of another set $W' \neq W$, before restoring consistency. However, we have the guarantee that, since the donor is a correct questionnaire, there exists in Q^d at least one set of fields whose values are able to restore consistency of Q^w. The second problem is related to the first, since, having a very numerous set of correct questionnaires, the case when we need to copy from the donor a set of fields different from W before restoring consistency is rare. Moreover, the number of fields we could need to add in this case is low. Consider the following (very rough) example.

Example 6.3. We have an erroneous questionnaire Q^w

$Q^w = \{$ ... `age` = `17`, ... `car` = `no`, ... `city of residence` = `aaa`, ...
..., `city of work` = `bbb`, ... `time to go to work` = `20`, ... $\}$

where the set of erroneous fields of minimum cost is { `age`, `car` }, with a total cost of 5.5. Imagine we could restore consistency only if, for these two fields, we have `age`$\geq$ `18` and `car` = `yes`. Searching for a donor, however, we find Q^d such that

$Q^w = \{$ ... `age` = `16`, ... `car` = `yes`, ... `city of residence` = `aaa`, ...
... `city of work` = `aaa`, ... `time to go to work` = `20`, ... $\}$

If we proceed with imputation of the two selected fields { `age`, `car` }, we do not restore consistency at all. We need to choose a different set of fields. In this case, imagine that the imputation of the set of fields { `age`, `city of work` }, with a total cost of $c = 6$, restores consistency. By taking them from the donor, we obtain:

$Q^w =$ { ... `age` = `17`, `car` = `yes`, `city of residence` = `aaa`, ...
... `city of work` = `aaa`, ... `time to go to work` = `20`, ... }

Therefore, the problem of imputation trough a donor is the following. Given the erroneous questionnaire Q^w, the donor questionnaire Q^d, and the CNF formula $\mathcal{E}$ to be satisfied, we want to find a set D of fields f_i^d such that:

- The corrected questionnaire Q^c satisfying $\mathcal{E}$ can be obtained from the erroneous one Q^w by copying from the donor Q^d (only and all) the values of the $f_i^d \in D$.
- The total cost of the correction $c(D)$ is minimum.

In this case also we can have more than one set of fields whose imputation can restore consistency, and we are interested in the one which changes as few as possible Q^w. Due to the motivations noted above, we have that $c(W) \leq c(D)$. Variants to the above procedure are possible (the number of donors, how to choose a donor), but the spirit of the imputation trough a donor remains the same.

5.6 A Set Covering Formulation

We can observe that, in both the cases of the use of an imputation function or of the use of a donor, we actually want to find a set of changes of minimum cost such as we can restore consistency, i.e. satisfy the CNF formula $\mathcal{E}$. The above problem can be modeled as a weighted set covering problem [196].

Given a ground set S of n elements s_i, and a collection $\mathcal{A}$ of m subsets A_j of S (for example, the collection may consists of all subsets of size $k \leq n$), the set covering problem is the problem of taking a set of elements s_i of minimum cardinality such as we have at least one element for every A_j. A little more general problem is the following. Given a ground set S of n elements s_i, each one with a cost $c_i \in \mathbb{R}_+$, and a collection $\mathcal{A}$ of m sets A_j of elements of S, the weighted set covering problem is the problem of taking

the set of elements s_i of minimum total weight such as we have at least one element for every A_j. Note that the first problem is a special case of the second when all the costs are 1.

Let a^j be the incidence vector of A_j, i.e. a vector in $\{0,1\}^n$ whose i-th component a_i^j is 1 if $s_i \in A_j$, and 0 if $s_i \notin A_j$. Consider a vector of variables $x \in \{0,1\}^n$ which is the incidence vector of the set of the elements s_i we take. We can give the following mathematical model for the above problems.

$$\begin{array}{ll} \min & \sum_{i=1}^{n} c_i x_i \\ \text{s. t.} & \sum_{i=1}^{n} a_i^j x_i \geq 1 \qquad j = 1 \dots m \end{array} \tag{5.1}$$

The set covering problem is a classical combinatorial optimization problem, with binary variables which assume values in {0,1}. It is of great relevance for modeling and solving a variety of problems arising form many practical fields. It is known to be NP-complete [98]. Set covering formulations are used, for instance, in the fields of telecommunications, transportation, facility location, crew scheduling, and, in general, when the problem has the structure of a set of *something* that must be *covered*.

In order to work in the field of binary optimization, we now transform the logic variables α_i taking values in $\{True, False\}$ into binary variables x_i taking values in {0,1}. The difference is only formal, and the conversion is straightforward. While the operations defined on the logical variables where $\{\wedge, \vee, \dots\}$, the operations defined on the binary variables are $\{+, -, \dots\}$ In particular, a positive literal α_i corresponds to a binary variable x_i, and a negative literal $\neg\alpha_i$ corresponds to a negated binary variable $\bar{x}_i$. Note that $\bar{x}_i$ and x_i must have opposite values, just like $\neg\alpha_i$ and α_i.

A set of answers to a questionnaire, which corresponded to a truth assignment in $\{True, False\}^n$, will now correspond to a binary vector in $\{0,1\}^n$. The following propositional clause c_j

$$(\alpha_i \vee \dots \vee \alpha_j \vee \neg\alpha_k \vee \dots \vee \neg\alpha_n)$$

will now correspond to the following linear inequality, by defining the set A_π of the logical variables which appear positive in c_j, and the set A_ν of the logical variables which appear negative in c_j, and the corresponding incidence vectors a^π and a^ν

$$\sum_{i=1}^{n} a_i^\pi x_i + \sum_{i=1}^{n} a_i^\nu \bar{x}_i \geq 1$$

For each clause we have an inequality of the above type. If we write all of them, we obtain that the logic formula $\mathcal{E}$ that the truth assignment must satisfy

$$(\alpha_{i_1} \vee ... \vee \alpha_{j_1} \vee \neg\alpha_{k_1} \vee ... \vee \neg\alpha_{n_1}) \wedge \ldots \wedge (\alpha_{i_m} \vee ... \vee \alpha_{j_m} \vee \neg\alpha_{k_m} \vee ... \vee \neg\alpha_{n_m})$$

corresponds now to the following system of linear inequalities that the binary vector must satisfy.

$$\begin{pmatrix} a^{\pi}_{1,1} & ... & a^{\pi}_{1,n} \\ & ... & \\ a^{\pi}_{m,1} & ... & a^{\pi}_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} a^{\nu}_{1,1} & ... & a^{\nu}_{1,n} \\ & ... & \\ a^{\nu}_{m,1} & ... & a^{\nu}_{m,n} \end{pmatrix} \begin{pmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_n \end{pmatrix} \geq \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Example 7.1. Suppose that the truth assignment corresponding to the (correct) questionnaire Q is

$$\{\alpha_1 = False, \alpha_2 = False, \alpha_3 = True\}$$

and that the logic formula $\mathcal{E}$ used to detect erroneous questionnaires is

$$(\neg\alpha_1 \vee \alpha_2 \vee \neg\alpha_3) \wedge (\neg\alpha_1 \vee \neg\alpha_2) \wedge (\alpha_2 \vee \alpha_3)$$

The binary vector corresponding to the questionnaire Q is

$$\{x_1 = 0, x_2 = 0, x_3 = 1\}$$

and the system of linear inequalities corresponding to $\mathcal{E}$ is

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \bar{x}_3 \end{pmatrix} \geq \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Moreover, in order to take into account the reliability of every field, we estimate a vector of n costs $\{c_1, \ldots, c_n\}$ corresponding to the variables $\{\alpha_1, \ldots, \alpha_n\}$. We pay c_i when we change α_i. Note that, since every field corresponds to more variables, such costs are not directly the reliability of a field. On the other hand, an estimation of an individual cost for every variable allows more subtile evaluations.

We can now model the imputation problems as follows.

5.6.1 Error Localization

In the case of error localization, we have

- The binary vector $e = \{e_1, \ldots, e_n\} \in \{0,1\}^n$ corresponding to the erroneous questionnaire Q^w.
- The binary variables $x = \{x_1, \ldots, x_n\} \in \{0,1\}^n$ and their complements $\bar{x} = \{\bar{x}_1, \ldots, \bar{x}_n\} \in \{0,1\}^n$, with the coupling constraints $x_i + \bar{x}_i = 1$. The variables x represent the truth assignment corresponding to the corrected questionnaire Q^c that we want to find.
- The system of linear inequalities $A^\pi x + A^\nu \bar{x} \geq 1$, with $A^\pi, A^\nu \in \{0,1\}^{m \times n}$, that e does not satisfy. We know that such system has binary solutions, since $\mathcal{E}$ is satisfiable and has more than one solution.
- The vector $c' = \{c_1, \ldots, c_n\} \in \mathbb{R}^n_+$ of costs that we pay for changing e. We pay c_i for changing e_i.

We introduce furthermore a vector of binary variables $y = \{y_1, \ldots, y_n\} \in \{0,1\}^n$ representing the changes we introduce in e. We have

$$y_i = \begin{cases} 1 & \text{if we change } e_i \\ 0 & \text{if we keep } e_i \end{cases}$$

According to the principle of the minimum change, we want to change the erroneous questionnaire Q^w minimizing the total cost of the changes. This can be expressed as

$$\min_{y_i \in \{0,1\}} \sum_{i=1}^{n} c_i y_i = \min_{y \in \{0,1\}^n} c' y \tag{5.2}$$

We want a corrected questionnaire Q^c which satisfies the system of inequalities

$$A^\pi x + A^\nu \bar{x} \geq 1 \tag{5.3}$$

A key issue is that there is a relation between variables y and x (and consequently $\bar{x}$). This depends on the values of e, as follows:

$$y_i = \begin{cases} x_i \quad (= 1 - \bar{x}_i) & \text{if } e_i = 0 \\ 1 - x_i \quad (= \bar{x}_i) & \text{if } e_i = 1 \end{cases} \tag{5.4}$$

In fact, when $e_i = 0$, to keep it unchanged means to put $x_i = 0$. Since we do not change, $y_i = 0$. On the contrary, to change it means to put $x_i = 1$. Since we change, $y_i = 1$. Altogether, $y_i = x_i$.

When, instead, $e_i = 1$, to keep it unchanged means to put $x_i = 1$. Since we do not change, $y_i = 0$. On the contrary, to change it means to put $x_i = 0$. Since we change, $y_i = 1$. Altogether, $y_i = 1 - x_i$.

By using the above result, we can express the problem of error localization with the following formulation. Our objective function (2) becomes

$$\min_{x_i, \bar{x}_i \in \{0,1\}} \quad \sum_{i=1}^{n} (1 - e_i) c_i x_i + \sum_{i=1}^{n} e_i c_i \bar{x}_i \tag{5.5}$$

Subject to the following constraints

$$\begin{aligned} & A^{\pi} x + A^{\nu} \bar{x} \geq 1 \\ & x_i + \bar{x}_i = 1 \\ & x, \bar{x} \in \{0,1\}^n \end{aligned}$$

We will call *satisfiability* constraints the first kind of constraints, deriving in fact from the propositional satisfiability problem. We will call *coupling* constraints the second kind of constraint. The third kind are the *binary* constraints over the variables. The above formulation is a set covering problem, as defined in (1).

5.6.2 Imputation through a Donor

The case of imputation through a donor is very similar. We have

- The binary vector $e = \{e_1, \ldots, e_n\} \in \{0,1\}^n$ corresponding to the erroneous questionnaire Q^w.
- The binary vector $d = \{d_1, \ldots, d_n\} \in \{0,1\}^n$ corresponding to the donor questionnaire Q^d.
- The binary variables $x = \{x_1, \ldots, x_n\} \in \{0,1\}^n$ and their complements $\bar{x} = \{\bar{x}_1, \ldots, \bar{x}_n\} \in \{0,1\}^n$, with the coupling constraints $x_i + \bar{x}_i = 0$. x corresponds to the corrected questionnaire Q^c that we want to find.
- The system of linear inequalities $A^{\pi} x + A^{\nu} \bar{x} \geq 1$, with $A^{\pi}, A^{\nu} \in \{0,1\}^{m \times n}$, that e does not satisfy. We know that such system has binary solutions, since $\mathcal{E}$ is satisfiable and has more than one solution.
- The vector $c' = \{c_1, \ldots, c_n\} \in \mathbb{R}_+^n$ of costs that we pay for changing e. We pay c_i for changing e_i.

We use, as before, a vector of binary variables $y = \{y_1, \ldots, y_n\} \in \{0,1\}^n$ representing the elements that we copy from d to e. We have

$$y_i = \begin{cases} 1 & \text{if we copy } d_i \text{ in } e_i \\ 0 & \text{if we keep } e_i \end{cases}$$

According to the principle of the minimum change, we want to change the erroneous questionnaire Q^w minimizing the total cost of the changes. This can be expressed as

$$\min_{y_i \in \{0,1\}} \sum_{i=1}^{n} c_i y_i = \min_{y \in \{0,1\}^n} c' y \tag{5.6}$$

We want a corrected questionnaire Q^c which satisfies the system of inequalities

$$A^{\pi} x + A^{\nu} \bar{x} \geq 1 \tag{5.7}$$

In this case also, there is a relation between variables y and x (and consequently $\bar{x}$). This depends on the values of e and d, as follows:

$$y_i = \begin{cases} x_i \quad (= 1 - \bar{x}_i) & \text{if } e_i = 0 \text{ and } d_i = 1 \\ 1 - x_i \quad (= \bar{x}_i) & \text{if } e_i = 1 \text{ and } d_i = 0 \\ 0 & \text{if } e_i = d_i \end{cases} \tag{5.8}$$

In fact, when $e_i = 0$ and $d_i = 1$, not to copy the element means to put $x_i = 0$. Since we do not change, $y_i = 0$. On the contrary, to copy the element means to put $x_i = 1$. Since we change, $y_i = 1$. Altogether, $y_i = x_i$.

When, instead, $e_i = 1$ and $d_i = 0$, not to copy the element means to put $x_i = 1$. Since we do not change, $y_i = 0$. On the contrary, to copy the element means to put $x_i = 0$. Since we change, $y_i = 1$. Altogether, $y_i = 1 - x_i$.

Finally, when $e_i = d_i$, we have no gain from copying the element, and therefore we do not change. Altogether, $y_i = 0$.

By using the above result, we can express the problem of imputation through a donor with the following formulation. Our objective function (2) becomes

$$\min_{x_i, \bar{x}_i \in \{0,1\}} \sum_{i=1}^{n} (1 - e_i) d_i c_i x_i + \sum_{i=1}^{n} e_i (1 - d_i) c_i \bar{x}_i \tag{5.9}$$

Subject to the following constraints

$$\begin{aligned} & A^{\pi} x + A^{\nu} \bar{x} \geq 1 \\ & x_i + \bar{x}_i = 1 \\ & x, \bar{x} \in \{0,1\}^n \end{aligned}$$

Similarly with the former case, we will call *satisfiability* constraints the first kind of constraints, deriving in fact from the propositional satisfiability problem. We will call *coupling* constraints the second kind of constraint. The third kind are the *binary* constraints over the variables. Also the above formulation is a set covering problem, as defined in (1).

The above problems are therefore solved with the volume algorithm [11], which is a recently proposed and very effective combinatorial optimization algorithm.

5.7 Implementation

We first implemented a series of procedures in order perform the automatic conversion of the set of edits into a CNF formula. After that, in virtue of modeling our problems as classical optimization problems, we could take advantage of the huge amount of research done in the fields of satisfiability and set covering solvers.

The satisfiability problems are solved by means of a procedure of Adaptive Core Search [33], in particular with the ACS solver. Such solver, in fact, turns out to be a very effective one, as proved in [32]. Moreover, in the case of unsatisfiable instances, ACS is able to select a subset of clauses which are still unsatisfiable, and thus cause the unsatisfiability. This is a key feature in our case, since, in the cases of inconsistencies, we need to remove them, and this is possible only when the 'culprit' edits can be located.

All the edits validation process was implemented to run sequentially, and interactively asking, for every inconsistence of redundancy found, whether stop the process and let the edit-writer mend the set of edit, or ignore it and continue.

The set covering problems are then solved by implementing the Volume Algorithm [11, 12]. This very effective procedure recently developed by Barahona is an extension of the subgradient algorithm, with the key feature is that is able to produce primal as well dual solutions. This gives a fast method for producing approximations for large scale linear programs.

We added a simple heuristic in order to obtain an integer solution to our set covering problem, together with a bound allowing to verify the quality of such solution. Such heuristic consists in a rounding of the fractionary solution. Rounding is deterministic (and just cuts at 1/2) for some fractionary values, while being probabilistic for particular combinations of fractionary values.

The choice of the Volume Algorithm made possible to solve problems

whose size is not solvable by a commercial Branch-and-Bound solver (Xpress).

5.8 Results

We performed the process of edits validation and data imputation in the case of a Census of Population. The set of edits are kindly provided by the Italian National Statistic Institute (ISTAT). In the course of this work, we actually did several tests, by considering different sets of edits. They ranged from 100 to 400 edits. Such procedure were considered to be quite representative of how a real census proceeds.

All the described procedure were implemented in C++ and tested on a Pentium II 450MHz PC under MS Windows Operating System.

In this census, the data of every family were collected by using a single questionnaire. Individuals were identified by number, and edits were originally written with this structure. Edits have been rewritten identifying individuals by roles, which are: `head of the family`, `consort`, `father of head`, `mother of head`, `brother/sister of head`, `son`, `additional son`, `father in law`, `mother in law`, `daughter in law`, `son in law`, `grandchild`. This was done by taking obviously care of maintaining the exact meaning of the original edits. This choice produces a more compact set of edits, while its congruency and redundancy properties remain the same.

Being a questionnaire for an entire family, we can have similar edits repeated for every member of the family. Edit replication, called explosion, is the first automatically performed step.

It is then applied the procedure of automatic conversion of edits into clauses. The implemented software takes in input a file containing the set of edits, a file containing a description of the fields, a file containing lists of all feasible values of qualitative fields, and a file containing the list of family roles. The procedure identifies logical variables, according to outlined procedure, and gives in output a file listing the used logical variables, with their meaning, and the CNF formula which encodes the set of edits. The generation of the logic formula is not a costing operation.

For the problem we deal with, resulting formulas range from 315 variables and 650 clauses to 450 variables and 1100 clauses. Since this kind of problems are solved quite easily, in order to test the limits of the procedure, we moreover considered artificially generated satisfiability instances of bigger size. They ranged from 1000 variables and 5000 clauses to 15000 variables and 75000 clauses.

5.8.1 Edit Validation

Our algorithm begins with solving the satisfiability problem for the CNF formula, in order to detect complete inconsistency for the set of edits. After this, in order to detect all 1-level partial inconsistency, the procedure goes ahead fixing in turn every variable to $True$ and then all the variables of every field to $False$, and solving at every step the resulting SAT instance.

The test for redundancies is then made by negating in turn every clauses, and solving at every step the resulting SAT instance.

Hence, for every instance with n variables and m clauses, the implemented algorithm solves in cascade about $1 + n + n/10 + m$ satisfiability problems of non trivial size. Although such problems do not reveal to be structurally hard, the computational burden is substantial. Therefore, there is the need of an efficient SAT solver. We used ACS, a recently proposed and very efficient SAT solver, which uses an enumeration scheme with a new adaptive technique [33]. In the following tables we report number of variables (n) and number of clauses (m) of the propositional CNF formula, the number of problem we had to solve (# of problems), and total time for solving all of them (time).

In table 1 we report results on formulas which are the encoding of the real census edits. In table 2 we report results on artificially generated formulas.

n	m	# of problems	time
315	650	975	0.99
350	710	1090	1.35
380	806	1219	1.72
402	884	1321	2.21
425	960	1428	2.52
450	1103	1599	3.24

Table 1: Results of the edit validation procedure on real sets of edits .

Our solver is able, in the case of an unsatisfiable instance, of providing a set of clauses which cause the unsolvability, in order to understand were is the inconsistency. In the analyzed set of real edits, which were supposed to be error free, a partial inconsistency was found, that was due to one of the edit concerning divorced people, as explained below. Married people must be at least `14` years old, and a divorce procedure takes at least four years. It results that divorced people must be at least `18` years old. There was an edit representing this, and it was erroneously written

$$(\texttt{marital status} \in \{\texttt{single}, \texttt{married}, \texttt{separate}, \texttt{widow}\}) \wedge (\texttt{age} < 18)$$

The correct edit should have been the following.

$$(\texttt{marital status} = \texttt{divorced}) \wedge (\texttt{age} < \texttt{18})$$

Such problem could have caused errors in the phase of individuation of erroneous records.

By checking for redundancy, several clauses resulted redundant. Further tests, performed after deliberately introduction of inconsistency or redundancy in the set of edits, lead in the totality of the cases to their detection.

n	m	# of problems	time
1000	5000	6101	15
3000	15000	18301	415
5000	25000	30501	1908
8000	40000	48801	7843
10000	50000	61001	16889
15000	75000	91501	>36000

Table 2: Results of the edit validation procedure on artificially generated sets of edits .

In the case of artificially generated instances, partial inconsistencies and redundancies were found. Their description is not significative, being artificially generated problems. They were used in order to understand which size of problems the proposed procedure is able to treat.

Afterwards, detection of erroneous questionnaires answers have been performed, as a trivial task. It proceeds converting the set of answers in values for the logical variables, and simply testing if such truth assignment satisfies the CNF formula obtained from the edits.

5.8.2 Data Imputation

The procedure for data imputation was tested on simulated questionnaires, and by using the two sets of real edits and the artificially generated ones. We considered both the problems of error localization and imputation through a donor. For each set of edits we considered various simulated erroneous answers. In particular, we considered the percentage of activated edits for the erroneous answer. Since errors are usually a small part of the answers, we realistically considered small edit activation percentages (1%, 2%, 3%, etc.). Note that a set of edit corresponding to n logic variables and m clauses corresponds here to a set covering formulation with $2n$ variables and m satisfiability constraints plus n coupling constraints, altogether $m + n$.

We solved the set covering formulation by means of a set covering solver based on the Volume Algorithm. Moreover, since this is an approximate procedure, we compared its results with the commercial Branch-and-Bound solver Xpress. In some cases of high error percentage, the solver based on the Volume Algorithm could not find a feasible integer solution. In those cases we reported "-". We report a different table for each instance, with time and solution value in all the above cases. Problems of error localization follow.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	0.01	3.18	43.0	43.0
0.9%	0.10	3.18	221.6	221.6
1.3%	0.10	3.45	328.4	328.4
2.4%	0.14	2.86	729.5	729.5
5.0%	0.22	3.50	300.8	300.8

Table 3: Error localization procedure on a real set of edits. The set covering instance has 400 var. and 1400 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	0.04	1.91	58.6	58.6
0.7%	0.10	1.91	108.6	108.6
1.0%	0.11	2.54	140.1	140.1
1.6%	0.16	1.90	506.7	506.7
2.5%	0.20	2.50	1490.1	1490.1
3.5%	0.23	2.98	2330.8	2330.8

Table 4: Error localization procedure on a real set of edits. The set covering instance has 480 var. and 1880 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.7%	0.51	2.44	463.2	463.2
1.5%	0.68	3.02	394.5	394.5
2.1%	0.55	3.32	612.5	612.5
3.6%	0.98	3.73	1254.8	1254.8
6.7%	1.00	4.40	2341.2	2341.2

Table 5: Error localization procedure on a real set of edits. The set covering instance has 800 var. and 3200 const.

From the above tables, we can observe that real problems are solved efficiently both by VA and B&B. Surprisingly, VA reaches in all cases the optimal integer solution, given by the B&B. Times are very small in both cases, and they increase with error percentage (and, of course, with the size of the problem).

In addition, we tested the procedure on artificially generated instances, corresponding to the artificially generated satisfiability instance considered in the case of edit validation.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	0.66	87.90	329.4	329.4
1.0%	1.30	81.88	1010.4	1010.4
1.1%	0.73	97.71	952.9	952.9
1.8%	1.92	109.05	1982.0	1982.0
4.3%	3.65	87.16	5549.8	5549.8

Table 6: Error localization procedure on an artificially generated set of edits. The set covering instance has 2000 var. and 6000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	11.60	3813.4	1273.7	1273.6
0.8%	4.75	3477.7	2266.1	2266.1
1.0%	5.10	2796.9	3604.3	3604.3
2.1%	4.76	4117.2	6064.2	6064.2
4.0%	11.63	3595.0	1544.7	1544.6

Table 7: Error localization procedure on an artificially generated set of edits. The set covering instance has 6000 var. and 18000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	8.27	18837.8	2018.1	2018.1
0.7%	10.87	19210.5	3817.1	3817.1
1.3%	11.80	19493.6	6409.2	6409.2
1.7%	28.58	18506.6	9673.7	9673.7
3.0%	41.35	17000.3	21742.5	21742.4

Table 8: Error localization procedure on an artificially generated set of edits. The set covering instance has 10000 var. and 30000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	9.16	>21600	3058.4	-
0.9%	20.36	>21600	6897.7	-
1.3%	40.64	>21600	11682.9	-
2.0%	66.60	>21600	20113.0	-
4.0%	73.54	>21600	37069.1	-

Table 9: Error localization procedure on an artificially generated set of edits. The set covering instance has 16000 var. and 48000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	44.44	>21600	5135.4	-
0.8%	30.61	>21600	8640.5	-
1.3%	31.42	>21600	14466.7	-
2.1%	83.41	>21600	23093.7	-
3.6%	74.64	>21600	39448.2	-

Table 10: Error localization procedure on an artificially generated set of edits. The set covering instance has 20000 var. and 60000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	35.74	>21600	6754.7	-
0.9%	47.33	>21600	12751.3	-
1.3%	107.97	>21600	20135.4	-
2.0%	94.42	>21600	31063.6	-
4.0%	186.92	>21600	66847.4	-

Table 11: Error localization procedure on an artificially generated set of edits. The set covering instance has 30000 var. and 90000 const.

From the above tables, we can observe that artificially generated problems of very big size are all solved by VA, while B&B cannot solve the bigger instance within the time limit of 6 hours. However, we begin to notice that the solution found by VA is not optimal, although the difference is numerically negligible.

Further occasional tests with higher error percentage shows that VA does not increase running time, but the heuristic is not able to find a feasible integer solution, hence we have no solution, while B&B would reach such solution but in an a prohibitive amount of time.

In the case of a donor, we considered a simulated donor, which is a correct solution very similar to the erroneous one. Results both for real and artificially generated sets of edits follow.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	0.04	0.01	43.4	43.4
0.9%	0.04	0.01	227.5	227.5
1.3%	0.04	0.01	348.7	348.7
2.4%	0.04	0.04	726.6	726.6
5.0%	0.03	0.03	365.8	365.8

Table 12: Imputation through a donor on a real set of edits. The set covering instance has 400 var. and 1400 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	0.04	0.02	63.6	63.6
0.7%	0.06	0.02	144.7	144.7
1.0%	0.06	0.02	264.5	264.5
1.6%	0.06	0.02	643.5	643.1
2.5%	0.07	0.02	1774.3	1774.1
3.5%	0.06	0.03	2369.9	2369.5

Table 13: Imputation through a donor on a real set of edits. The set covering instance has 480 var. and 1880 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.7%	0.08	0.04	280.1	280.1
1.5%	0.08	0.06	453.5	453.1
2.1%	0.08	0.06	655.4	655.0
3.6%	0.08	0.07	1378.0	1378.0
6.7%	0.10	0.07	2455.1	2455.0

Table 14: Imputation through a donor on a real set of edits. The set covering instance has 800 var. and 3200 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	0.26	0.01	331.2	331.2
1.0%	0.10	0.31	1015.2	1015.1
1.1%	0.04	0.01	952.9	952.9
1.8%	0.08	0.31	1997.1	1997.1
4.3%	-	0.20	-	5592.2

Table 15: Imputation through a donor on an artificially generated set of edits. The set covering instance has 2000 var. and 6000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	0.17	0.30	1291.6	1291.4
0.8%	0.17	0.10	2305.4	2305.4
1.0%	0.70	0.30	3660.1	3660.1
2.1%	0.38	0.30	6124.0	6123.1
4.0%	0.65	0.30	1585.7	1585.0

Table 16: Imputation through a donor on an artificially generated set of edits. The set covering instance has 6000 var. and 18000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	1.01	0.6	2120.1	2118.9
0.7%	1.30	0.6	3877.7	3877.2
1.3%	1.40	0.3	6411.7	6411.7
1.7%	1.60	0.6	9690.1	9690.1
3.0%	1.05	0.3	21823.1	21802.4

Table 17: Imputation through a donor on an artificially generated set of edits. The set covering instance has 10000 var. and 30000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	0.44	0.6	3064.2	3064.2
0.9%	4.59	0.6	6899.7	6899.6
1.3%	0.86	0.8	11702.9	11698.3
2.0%	-	0.6	-	19814.5
4.0%	-	0.9	-	36496.7

Table 18: Imputation through a donor on an artificially generated set of edits. The set covering instance has 16000 var. and 48000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.5%	2.85	0.9	5139.1	5139.0
0.8%	3.28	0.9	8688.0	8687.0
1.3%	3.41	0.9	14500.4	14478.5
2.1%	3.82	0.9	23411.7	23072.0
3.6%	-	0.9	-	44026.8

Table 19: Imputation through a donor on an artificially generated set of edits. The set covering instance has 20000 var. and 60000 const.

	Time		Value	
error	VA	B&B	VA	B&B
0.4%	3.58	1.27	6788.1	6788.1
0.9%	3.22	1.27	12760.0	12759.5
1.3%	0.97	0.9	20140.1	20140.0
2.0%	2.89	1.27	31258.1	31082.2
4.0%	-	1.59	-	67764.94

Table 20: Imputation through a donor on an artificially generated set of edits. The set covering instance has 30000 var. and 90000 const.

From the above tables, we can observe that all problems are solved is very short times. B&B is always able to find a solution within seconds. VA, on the other hand, is not able to find a solution when error increases too much. Moreover, the solution found by VA is not optimal in several cases, although the difference is very small.

This can be explained by noting that, in the problem of imputation through a donor, some variables are fixed to the value thy have in the donor. This results in a problem with much less variables but with a high error percentage. In this conditions the preferable solution method is a complete one, such like B&B.

To summarize the results, VA outperform B&B in the case of error localization in very big instances with a moderate error. On the contrary, in the case of smaller problems with higher error (not realistic), or in some cases of imputation through a donor, B&B is more reliable.

5.9 Conclusions

A binary encoding is the more direct and effective representation both for records and for the set of edit rules in a process of data collecting. It allows to automatically detect inconsistencies and redundancies in the set of

edits rules. Erroneous records detection is carried out with an inexpensive procedure. The proposed encoding allows, moreover, to automatically perform error localization and data imputation. The related computational problems are overcome by using state-of-the-art solvers. Approached real problems have been solved in extremely short times. Artificially generated problems are effectively solved until sizes which are orders-of-magnitude larger than the above real-world problems. Hence, noteworthily qualitative improvements in a general process of data collecting are made possible. The implemented software is tested in the case of a real Population Census. Edits are kindly provided by the Italian National Statistic Institute (ISTAT). Results are extremely encouraging.

References

[1] L. Abel. On the order of connections for automatic wire routing. IEEE Trans. on Computers, 1227–1233, Nov. 1972.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. Data Structures and Algorithms. Addison Wesley, Reading, 1985.

[3] A.V. Aho, D.S. Johnson, R.M. Karp, S.R. Kosaraju, C.C. McGeoch, C.H. Papadimitriou, and P. Pevzner. Theory of computing: Goals and directions, March 15, 1996.

[4] S.B. Akers. Binary decision diagrams. IEEE Transactions on Computers, C27(6):509–516, Jun. 1978.

[5] S.B. Akers. On the use of the linear assignment algorithm in module placement. In Proc. of the 18th ACM/IEEE Design Automation Conference, pages 137–144, 1981.

[6] Y. Asahiro, K. Iwama and E. Miyano. Random Generation of Test Instances with Controlled Attributes. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:377–393, 1996.

[7] P. Ashar, A. Ghosh, and S. Devadas. Boolean satisfiability and equivalence checking using general Binary Decision Diagrams. Integration, the VLSI journal, 13:1–16, 1992.

[8] B. Aspvall, M.F. Plass, and R.E. Tarjan. A lineartime algorithm for testing the truth of certain quantified Boolean formulas. Information Processing Letters, 8(3):121–132, Mar. 1979.

[9] B. Aspvall. Recognizing disguised NR(1) instances of the satisfiability problem. Journal of Algorithms 1, pages 97–103, 1980.

[10] M. Bankier. Experience with the New Imputation Methodology used in the 1996 Canadian Census with Extensions for future Census. UN/ECE Work Session on Statistical Data Editing, Working Paper n.24, Rome, Italy, 2-4 June 1999.

[11] F. Barahona and R. Anbil. The Volume Algorithm: producing primal solutions with a subgradient method. IBM Research Report RC21103, 1998.

[12] F. Barahona and F. Chudak. Near-optimal solutions to large scale facility location problems. IBM Research Report RC21606, 1999.

[13] G. Barcaroli. Un approccio logico formale al problema del controllo e della correzione dei dati statistici. Quaderni di Ricerca ISTAT n.9/1993.

[14] R.J. Bayardo and D.P. Miranker A Complexity Analysis of Space Bounded Learning Algorithms for the Constraint Satisfaction Problem. In Proc. 13th Nat. Conf. on Artificial Intelligence 558-562 1996.

[15] A. Ben-Tal, G. Eiger, and V. Gershovitz. Global minimization by reducing the duality gap. Mathematical Programming, 63:193–212, 1994.

[16] K. Berman, J. Franco, and J. Schlipf. Unique satisfiability for Horn sets can be solved in nearly linear time. Proc. Seventh Advanced Research Institute in Discrete Applied Mathematics (ARIDAM), New Brunswick, New Jersey, May, 1992. In Discrete Applied Mathematics 60:77–91, 1995.

[17] W. Bibel. Automated theorem proving. Vieweg, 1982.

[18] A. Billionnet and A. Sutter. An efficient algorithm for the 3-Satisfiability Problem. Operations Research Letters, 12:29–36, 1992.

[19] J.R. Bitner and E.M. Reingold. Backtrack programming techniques. Comm. of ACM, 18(11):651–656, Nov. 1975.

[20] C. Blair, R.G. Jeroslow, and J.K. Lowe. Some results and experiments in programming techniques for propositional logic. Computers and Operations Research 13:633-645, 1988.

[21] A. Blake. Canonical Expressions in Boolean Algebra. Ph.D. thesis, University of Chicago, 1937.

[22] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver efficient workload balancing. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994. To appear in Annals of Mathematics and Artificial Intelligence.

[23] E. Boros, Y. Crama, and P.L. Hammer. Polynomialtime inference of all valid implications for Horn and related formulae. Annals of Mathematics and Artificial Intelligence 1, pages 21–32, 1990.

[24] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A complexity index for Satisfiability Problems. SIAM Journal on Computing, 23:45–49, 1994.

[25] E. Boros, P.L. Hammer, T. Ibaraki and A. Kogan. Logical analysis of numerical data. Mathematical Programming, 79:163–190, 1997.

[26] E. Boros, P.L. Hammer, and A. Kogan. Computational experiments with an exact SAT solver. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994.

[27] E. Boros, P. L. Hammer, and X. Sun. Recognition of q-Horn formulae in linear time. Discrete Applied Mathematics 55, pages 1–13, 1994.

[28] C.A. Brown, L.A. Finklestein, and P.W. Purdom. Backtrack Searching in the Presence of Symmetry. 6th International Conference on Algebraic Algorithms and Error Correcting Codes (AAECC), Lecture Notes in Computer Science, Vol. 357, pp. 99110.

[29] C.A. Brown and P.W. Purdom. An average time analysis of backtracking. SIAM J. on Computing, 10(3):583–593, Aug. 1981.

[30] C.A. Brown and P.W. Purdom. An empirical comparison of backtracking algorithms. IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI4(3):309–316, May 1982.

[31] R. Bruni. Satisfiability Techniques for Questionnaires Error Detection. Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza" , Technical Report 37-99, 1999.

[32] R. Bruni and A. Sassano. CLAS: a Complete Learning Algorithm for Satisfiability. Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza" , Technical Report 6-99, 1999.

[33] R. Bruni and A. Sassano. Finding Minimal Unsatisfiable Subformulae in Satisfiability Instances. in proc. of 6th Internat. Conf. on Principles and Practice of Constraint Programming, Lecture notes in Computer Science 1894, Springer-Verlag, 500–505, 2000.

[34] M. Bruynooghe and L.M. Pereira. Deduction Revision by Intelligent Backtracking, pages 194–215. Ellis Horwood Limited, 1984.

[35] R.E. Bryant. Graphbased algorithms for Boolean function manipulation. IEEE Trans. on Computers, C35(8):677–691, Aug. 1986.

[36] R.E. Bryant. Symbolic Boolean manipulation with ordered binarydecision diagrams. ACM Computing Surveys, 24(3):293–318, Sept. 1992.

[37] H. Kleine Büning. On generalized Horn formulas and k resolution. Theoretical Computer Science 116, pages 405–413, 1993.

[38] H. Kleine Büning and T. Lettmann. Aussagenlogik: Deduktion und Algorithmen. B.G. Teubner, Stuttgart, 1993. Reprinted in English.

[39] K.M. Bugrara and P.W. Purdom. Clause order backtracking. Technical Report 311, Indiana University, 1990.

[40] M. Buro and H.K. Büning. Report on a SAT competition. Bulletin of the European Association for Theoretical Computer Science, 49:143–151, Feb. 1993.

[41] G. Butler. Computing in Permutation and Matrix Groups II: Backtrack Algorithm. Math. Comp. 39: 671–680, 1982.

[42] V. Chandru and J.N. Hooker. Extended Horn sets in propositional logic. J. ACM 38, pages 205–221, 1991.

[43] V. Chandru and J.N. Hooker. Optimization Methods for Logical Inference. Wiley, New York, 1999.

[44] Y.J. Chang and B. W. Wah. Lagrangian techniques for solving a class of zero-one integer linear programs. In Proc. Computer Software and Applications Conference, pages 156–161, Dallas, TX, August 1995. IEEE.

[45] M. T. Chao and J. Franco. Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k satisfiable problem. Information Sciences, 51:289–314, 1990.

[46] R. Chandrasekaran. Integer programming problems for which a simple rounding type of algorithm works. In W. Pulleyblank, ed. Progress in Combinatorial Optimization. Academic Press Canada, Toronto, Ontario, Canada, pages 101106, 1984.

[47] W.T. Chen and L.L. Liu. Parallel approach for theorem proving in propositional logic. Inform. Sci., 41(1):61–76, 1987.

[48] A. Church. A note on the Entscheidungsproblem. J. of Symbolic Logic 1:40-41, 1936.

[49] V. Chvatal. Resolution Search. Discrete Applied Mathematics, 73:81–99, 1997.

[50] V. Chvátal and E. Szemerédi. Many hard examples for resolution. J. of ACM, 35:759–770, 1988.

[51] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In Proceedings on the Foundations of Computer Science, 1992.

[52] W. F. Clocksin and C. S. Mellish. Programming in Prolog (2nd Edition). Springer-Verlag, Berlin, 1984.

[53] J. Cohen, editor. Special Section on Logic Programming. Comm. of the ACM, volume 35, number 3. 1992.

[54] M. Conforti, G. Cornuéjols, A. Kapoor, K. Vusković, and M. R. Rao. Balanced Matrices. Mathematical Programming: State of the Art. J. R. Birge and K. G. Murty, eds. Braun-Brumfield, United States. Produced in association with the 15th Int'l Symposium on Mathematical Programming, University of Michigan, 1994.

[55] S.A. Cook. The complexity of theorem-proving procedures. In Proceedings of the Third ACM Symposium on Theory of Computing, pages 151–158, 1971.

[56] S.A. Cook. Find hard instances of the satisfiability problem. Presented at the 1996 DIMACS Workshop on Satisfiability Problem: Theory and Applications. March 11, 1996.

[57] Y. Crama, P. L. Hammer, B. Jaumard, and B. Simeone. Product form parametric representation of the solutions to a quadratic boolean equation. R.A.I.R.O. Recherche op'erationnelle/Operations Research 21:287–306, 1987.

[58] Y. Crama, P. Hansen, and B. Jaumard. The basic algorithm for pseudoboolean programming revisited. Discrete Applied Mathematics 29:171–185, 1990.

[59] J. Crawford and L. Auton. Experimantal results on the crossover point in Satisfiability problems. In Proc. AAAI-93, pages 22–28, 1993.

[60] J.M. Crawford. Solving satisfiability problems using a combination of systematic and local search. Submitted to the DIMACS Challenge II Workshop, 1994.

[61] M. Dalal, and D. W. Etherington. A hierarchy of tractable satisfiability problems. Information Processing letters 44, pages 173–180, 1992.

[62] G. Dantzig. Programming in a Linear Structure. Comptroller, USAF, Washington, D.C., February, 1948.

[63] G. Dantzig. Linear Programming and Extensions. Princeton University Press, Princeton, New Jersey, 1963.

[64] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. Comm. Assoc. for Comput. Mach., 5:394–397, 1962.

[65] M. Davis and H. Putnam. A computing procedure for quantification theory. Jour. Assoc. for Comput. Mach., 7:201–215, 1960.

[66] R. Dechter. Learning while searching in constraint satisfaction problems. In Proceedings of AAAI'86, 1986.

[67] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence, 41(3), 1990.

[68] R. Dechter. Directional resolution: The Davis-Putnam procedure revisited. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994.

[69] R. Dechter and J. Pearl. Networkbased heuristics for constraintsatisfaction problems. Artificial Intelligence, 34:1–38, 1988.

[70] J. Doyle. A truth maintenance system. Artificial Intelligence, 12:231–272, 1979.

[71] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming 1, pages 267–284, 1984.

[72] D.Z. Du, J. Gu, and P.M. Pardalos, editors. The Satisfiability (SAT) Problem. DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996.

[73] O. Dubois. Counting the number of solutions for instances of satisfiability. Theoretical Computer Science, 81:49–64, 1991.

[74] O. Dubois and J. Carlier. Probabilistic approach to the satisfiability problem. Theoretical Computer Science, 81:65–75, 1991.

[75] O. Dubois and Y. Boufkhad. Analysis of the space of solutions for random instances of the satisfiability problem. Proceedings of the fourth International Symposium on Artificial Intelligence and Mathematics, Ft. Lauderdale, Florida, p. 175, 1996.

[76] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Editors: D.S. Johnson and M.A. Trick, 26:415–436, 1996.

[77] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multi-commodity flow problems. SIAM J. on Computing, 5(4):691–703, 1976.

[78] P. Fellegi and D. Holt. A Systematic Approach to Automatic edit and Imputation. Journal of the American Statistical Association, 17:35–71(353), 1976.

[79] H. Fleischner and S. Szeider. Polynomial-time Recognition of Mimimal Unsatisfiable Formulas with Fixed Clause-Variable Difference. submitted to Elsevier Preprint. 1999.

[80] C.A. Floudas and P.M. Pardalos. A Collection of Test Problems for Constrained Global Optimization Algorithms. Springer Verlag, New York, 1990.

[81] C.A. Floudas and P.M. Pardalos, editors. Recent Advances in Global Optimization. Princeton University Press, New York, 1992.

[82] S. Foldes and P.L. Hammer. Normal Forms of Pseudo-Boolean Functions. Rutgers University, RUTCOR Research Report RRR 1-2000.

[83] J. Franco. On the probabilistic performance of algorithms for the satisfiability problem. Information Processing Letters, 23:103–106, 1986.

[84] J. Franco. Elimination of infrequent variables improves average case performance of satisfiability algorithms. SIAM J. on Computing, 20:1119–1127, 1991.

[85] J. Franco and Y.C. Ho. Probabilistic performance of heuristic for the satisfiability problem. Discrete Applied Mathematics, 22:35–51, 1988/89.

[86] J. Franco and M. Paull. Probabilistic analysis of the Davis Putnam procedure for solving the Satisfiability Problem. Discrete Applied Mathematics, 5:77–87, 1983.

[87] J. Franco. 1993. On the occurrence of null clauses in random instances of satisfiability. Discrete Applied Mathematics 41, pp. 203–209.

[88] J. Franco and R. Swaminathan. Toward a good algorithm for determining unsatisfiability of propositional formulas. 1996.

[89] J. Franco, J. Goldsmith, J. Schlipf, E. Speckenmeyer, R. P. Swaminathan. An algorithm for the class of pure implicational formulas. Proceedings of the Workshop on Satisfiability, Siena, Italy, 1996.

[90] J.W. Freeman. Improvements to the davis-putnam procedure for satisfiability. Ph.D. Thesis, University of Pennsylvania, 1995.

[91] E.C. Freuder. A sufficient condition for backtrack-bounded search. J. ACM, 32(4):755–761, Oct. 1985.

[92] E.C. Freuder and M.J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In Proceedings of 9th IJCAI, pages 1076–1078, 1985.

[93] A. M. Frieze, and S. Suen. Analysis of simple heuristics for random instances of 3-SAT. 1993.

[94] Z. Galil. On the complexity of regular resolution and the DavisPutnam procedure. Theoretical Computer Science, pages 23–46, 1977.

[95] G. Gallo, and M. G. Scutella. Polynomially solvable satisfiability problems. Information Processing Letters 29, pages 221–227, 1988.

[96] G. Gallo and G. Urbani. Algorithms for testing the Satisfiability of Propositional Formulae. Journal of Logic Programming, 7:45–61, 1989.

[97] G. Gallo and D. Pretolani. Hierarchies of polynomially solvable SAT problems. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994.

[98] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, San Francisco, 1979.

[99] J. Gaschnig. Performance Measurements and Analysis of Certain Search Algorithms. Ph.D. thesis, Carnegie-Mellon University, Dept. of Computer Science, May 1979.

[100] A.V. Gelder. A satisfiability tester for nonclausal propositional calculus. Information and Computation, 79(1):1–21, Oct. 1988.

[101] A. Van Gelder and Y.K. Tsuji. Satisfiability testing with more reasoning and less guessing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:559–586, 1996.

[102] M.R. Genesereth and N.J. Nilsson. Logical Foundations of Artificial Intelligence. Morgan Kaufmann Publishers, Los Altos, California, 1987.

[103] A. Ginzberg. Algebraic Theory of Automata. Academic Press, New York, 1968.

[104] F. Glover. Tabu search — Part I. ORSA Journal on Computing, 1(3):190–206, Summer 1989.

[105] K. Gödel. Die Vollständigkeit der Axiome des logischen Functionkalküs. Monatshefte für Mathematik und Physik 37:349–360, 1930

[106] K. Gödel. Über Formal Unentscheidbare Sätze Der Principia Mathematica und Verwandter Systeme I. Monatshefte für Mathematik und Physik 38: 173-198, 1931.

[107] M.X. Goemans and D.P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. J. of ACM, 42(6):1115-1145, 1995.

[108] A. Goldberg. Average case complexity of the satisfiability problem. In proc. Fourth Workshop on Automated Deduction, 1-6, 1979.

[109] J. Gu. Parallel algorithms and architectures for very fast search. Ph.D thesis. Technical Report UUCSTR88005, July 1988.

[110] J. Gu. How to solve Very Large-Scale Satisfiability problems. Technical Report UUCSTR88032, 1988, and Technical Report UCECETR90002, 1990.

[111] J. Gu. Efficient local search for very large-scale satisfiability problem. SIGART Bulletin, 3(1):8–12, Jan. 1992, ACM Press.

[112] J. Gu. Local search for satisfiability (SAT) problem. IEEE Trans. on Systems, Man, and Cybernetics, 23(4):1108–1129, Jul. 1993, and 24(4):709, Apr. 1994.

[113] J. Gu. Optimization Algorithms for the Satisfiability (SAT) Problem. In Advances in Optimization and Approximation. Ding-Zhu Du (ed), pages 72–154. Kluwer Academic Publishers, 1994.

[114] J. Gu. Optimization by multispace search. Kluwer Academic Publishers, 1997.

[115] J. Gu. Constraint-Based Search. Cambridge University Press, New York, 1997.

[116] J. Gu and Q.P. Gu. Average time complexities of several local search algorithms for the satisfiability (SAT) problem. Technical Report UCECETR91004, 1991. In Lecture Notes in Computer Science, Vol. 834, pp. 146154, 1994 and to appear in IEEE Trans. on Knowledge and Data Engineering.

[117] J. Gu and Lizhoudu. An efficient implementation of the SAT1.5 algorithm. Technical Report, USTC, Sept. 1995.

[118] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. DIMACS Series in Discrete Mathematics, 1999.

[119] J. Gu and R. Puri. Asynchronous circuit synthesis by Boolean satisfiability. IEEE Transactions on CAD of Integrated Circuits and Systems, 14(8):961–973, Aug. 1995.

[120] J. Gu and W. Wang. A novel discrete relaxation architecture. IEEE Trans. on Pattern Analysis and Machine Intelligence, 14(8):857–865, Aug. 1992.

[121] J. Gu, W. Wang, and T.C. Henderson. A parallel architecture for discrete relaxation algorithm. IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI9(6):816–831, Nov. 1987.

[122] Y. Gurevich. Average case completeness. J. of Computer and Systems Sciences, 42(3):346–398, 1991.

[123] A. Haken. The intractability of resolution. Theoretical Computer Science, 39:297–308, 1985.

[124] P.L. Hammer and S. Rudeanu. Boolean Methods in Operations Research and Related Areas. Springer-Verlag, New York, 1968.

[125] E.R. Hansen. Global optimization using interval analysis. M. Dekker, New York, 1992.

[126] P. Hansen and B. Jaumard. Uniquely solvable quadratic Boolean equations. Discrete Applied Mathematics 12:147–154, 1985.

[127] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. Computing, 44:279–303, 1990.

[128] P. Hansen, B. Jaumard, and G. Plateau. An extension of nested satisfiability. Les Cahiers du GERAD, G9327, 1993.

[129] R. M. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, 14:263–313, 1980.

[130] F. Harche, J.N. Hooker, and G.L. Thompson. A computational study of Satisfiability Algorithms for Propositional Logic. ORSA Journal on Computing, 6:423–435, 1994.

[131] F. Harche and G.L. Thompson. The Column Subtraction algorithm, an exact method for solving weighted Set Covering, Packing and Partitioning Problems. Computers and Operations Research, 21:689–705, 1990.

[132] W. Harvey, and M. Ginsberg. Limited Discrepancy Search, Proceedings International Joint Conference Artificial Intelligence (IJCAI) 1995.

[133] P.V. Hentenryck. Constraint Satisfaction in Logic Programming. The MIT Press, Cambridge, 1989.

[134] P. Heusch. The Complexity of the Falsifiability Problem for Pure Implicational Formulas. Proc. 20th Int'l Symposium on Mathematical Foundations of Computer Science (MFCS'95), J. Wiedermann, P. Hajek (Eds.), Prague, Czech Republic. Lecture Notes in Computer Science (LNCS 969), Springer-Verlag, Berlin, pages 221–226, 1995.

[135] J.N. Hooker and V. Vinay. Branching Rules for Satisfiability. Journal of Automated Reasoning, 15:359–383, 1995.

[136] J.N. Hooker. Generalized resolution and cutting planes. Annals of Operations Research, 12:217–239, 1988.

[137] J.N. Hooker. A quantitative approach to logical inference. Decision Support Systems, 4:45–69, 1988.

[138] J.N. Hooker. Resolution vs. cutting plane solution of inference problems: Some computational experience. Operations Research Letter, 7(1):1–7, 1988.

[139] J.N. Hooker and C. Fedjki. Branch and Cut solution of Inference Problems in Propositional Logic. Annals of Mathematics and AI, 1:123–139, 1990.

[140] B.K.P. Horn and M.J. Brooks, editors. Shape from Shading. The MIT Press, Cambridge, 1989.

[141] R. Horst and H. Tuy. Global Optimization: Deterministic Approaches. Springer Verlag, Berlin, 1990.

[142] J. Hsiang. Refutational theorem proving using termrewriting systems. Artificial Intelligence, pages 255–300, 1985.

[143] A. Itai and J. Makowsky. On the complexity of Herbrand's theorem. Working paper 243, Department of Computer Science, Israel Institute of Technology, 1982.

[144] K. Iwama. CNF satisfiability test by counting and polynomial average time. SIAM J. on Computing, pages 385–391, 1989.

[145] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the Satisfiability Problem. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:457–477, 1996.

[146] R.E. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. Annals of Mathematics and AI, 1:167–187, 1990.

[147] R.G. Jeroslow. Computation-oriented reductions of predicate to propositional logic. Decision Support Systems, 4:183–197, 1988.

[148] D.S. Johnson. Approximation Algorithms for Combinatorial Problems. J. of Computer and Systems Sciences, 9:256–278, 1974.

[149] D.S. Johnson and M.A. Trick, editors. Clique, Graph Coloring, and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series Vol. 26. American Mathematical Society, 1996.

[150] J.L. Johnson. A neural network approach to the 3-satisfiability problem. J. of Parallel and Distributed Computing, 6:435–449, 1989.

[151] A. E. W. Jones and G. W. Forbes. An adaptive simulated annealing algorithm for global optimization over continuous variables. Journal of Optimization Theory and Applications, 6:1–37, 1995.

[152] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende. Computational experience with an interior point algorithm on the satisfiability problem. Annals of Operations Research, 25:43–58, 1990.

[153] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende. A continuous approach to inductive inference. Mathematical Programming, 57:215–238, 1992.

[154] N. Karmarkar. A new polynomialtime algorithm for linear programming. Combinatorica, (4):373–395, 1984.

[155] R.M. Karp. Reducibility among combinatorial problems. In Complexity of Computer Computations, R.E. Miller and J.W. Thatcher eds., Plenum Press, New York, 85–103, 1972.

[156] S. Kirkpatrick, C.D. Gelat, and M.P. Vecchi. Optimization by simulated annealing. Science, 220:671–680, 1983.

[157] L. M. Kirousis, E. Kranakis, and D. Krizanc. A better upper bound for the unsatisfiability threshold. In DIMACS series in Discrete Mathematics and Theoretical Computer Science, 60, 1996.

[158] E. de Klerk, H. van Maaren, and J.P. Warners. Relaxations of the Satisfiability Problem using Semidefinite Programming. Technical Report SEN-R9903, CWI, Amsterdam, 1999. In I.P. Gent and T. Walsh eds. SAT 2000, Journal of Automated Reasoning, 2000.

[159] D. Knuth. Fundamental Algorithms, The Computer Programming, Volume 2nd ed., Addison-Wesley, Reading, Massachusetts 1968.

[160] D. E. Knuth. Estimating the efficiency of backtracking programs. Mathematics of Computation, 29(129):121–136, Jan. 1975.

[161] D. Knuth. Nested satisfiability. Acta Informatica 28:16, 1990.

[162] R. Kowalski. A proof procedure using connection graphs. J. ACM, 22(4):572–595, Oct. 1975.

[163] R. Kowalski. Logic for Problem solving. North Holland, 1978.

[164] O. Kullmann. A systematic approach to 3-SAT-decision, yielding 3-SAT-decision in less than 1:5045 n steps. Theoretical Computer Science, to appear.

[165] O. Kullmann. An application of matroid theory to the SAT problem, ECCC TR00-018, February 2000.

[166] V. Kumar. Algorithms for constraint satisfaction problems: A survey. The AI Magazine, 13(1):32–44, 1992.

[167] T. Larrabee. Test pattern generation using Boolean satisfiability. IEEE Trans. on Computer Aided Design, 11(1):4–15, Jan. 1992.

[168] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for Synthesis of Hazard free Asynchronous Circuits. In Proc. of 28th DAC, pages 302–308, 1991.

[169] C.Y. Lee. Representation of Switching Circuits by Binary Decision Programs. Bell Systems Technical Journal, 38:985–999, July 1959.

[170] H.R. Lewis. Renaming a set of clauses as a Horn set. Journal of the Association for Computing Machinery 25, pages 134–135, 1978.

[171] G.J. Li and B. W. Wah. Parallel iterative refining A*: An efficient search scheme for solving combinatorial optimization problems. In Proc. Int'l Conf. on Parallel Processing, pages 608–615, University Park, PA, August 1991. Pennsylvania State Univ. Press.

[172] D. Lichtenstein. Planar formulae and their uses. SIAM Journal on Computing 11:329–343, 1982.

[173] K.J. Lieberherr and E. Specker. Complexity of partial satisfaction. J. of ACM, 28:411–421, 1981.

[174] S. Lin. Computer solutions of the traveling salesman problem. Bell Sys. Tech. Journal, 44(10):2245–2269, Dec. 1965.

[175] J.W. Lloyd. Foundations of Logic Programming. 2nd ed. Springer-Verlag, New York, 1995.

[176] D.W. Loveland. Automated Theorem Proving: a Logical Basis. North Holland, 1978.

[177] D.G. Luenberger. Linear and Nonlinear Programming. Addison-Wesley, Reading, 1984.

[178] H. Van Maaren. Elliptic approximations of propositional formulae. Technical Report 96-65, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1996. To appear in Discrete Applied Mathematics.

[179] H. Van Maaren. On the use of second order derivatives for the satisfiability problem. In D. Du, J. Gu, and P.M. Pardalos eds. Satisfiability problem: Theory and applications, vol. 35 of DIMACS series in Discrete Mathematics and Computer Science. Americam Mathematical Society, 677-687, 1997.

[180] A.K. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8:99–119, 1977.

[181] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In Proc. of ACM/IEEE In ternational Conference on CAD, 1988.

[182] J.A. Makowsky and A. Sharell. On average case complexity of SAT symmetric distribution. J. of Logic and Computation, 5(1):71–92, Feb. 1995.

[183] C. Mannino and A. Sassano. Solving Hard Set Covering problems. Operations Research Letters, 18:1-5, 1995.

[184] C. Mannino and A. Sassano. Augmentation, Local Search and Learning. AI IA Notizie, XIII:34–36, Mar. 2000.

[185] T.A. Marsland and J. Schaeffer. Computers, Chess, and Cognition. Springer-Verlag, New York, 1990.

[186] M. Masselli, M. Signore, F. Panizon. Il sistema di controllo della qualitá dei dati. Manuale di tecniche di indagine ISTAT, vol. 6, 1992.

[187] J. Mayer, I. Mitterreiter and F.J. Radermacher. Running time experiments on some algorithms for solving propositional satisfiability problems. Annals of Operations Research 55:139-178, 1995.

[188] B. Mazure, L. Sais, and E. Gregoire. Tabu search for SAT. In Proceedings of CP'95 Workshop on Solving Really Hard Problems, pages 127–130, 1995.

[189] C.R. McLean and C.R. Dyer. An analog relaxation processor. In Proceedings of the 5th International Conference on Pattern Recognition, pages 58–60, 1980.

[190] K. Mehlhorn. Data Structures and Algorithms: Graph Algorithms and NP-Completeness. Springer-Verlag, Berlin, 1984.

[191] Z. Michalewicz. Genetic Algorithms + Data Structure = Evolution Programs. Springer Verlag, 1994.

[192] M. Minoux. The unique Horn-satisfiability problem and quadratic Boolean equations. Annals of Mathematics and Artificial Intelligence special issue on connections between combinatorics and logic, J. Franco, M. Dunn, W. Wheeler, (eds.) 1992.

[193] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT Problems. In Proceedings of AAAI'92, pages 459–465, Jul. 1992.

[194] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. Discrete Applied Mathematics, 10:117–133, 1983.

[195] B.A. Nadel. Constraint satisfaction algorithms. Computational Intelligence, 5:188–224, 1989.

[196] G.L. Nemhauser and L.A. Wolsey. Integer and Combinatorial Optimization. J. Wiley, New York, 1988.

[197] N.J. Nilsson. Principles of Artificial Intelligence. Tioga Publishing Company, Palo Alto, California, 1980.

[198] P. Nobili and A. Sassano. Strengthening Lagrangian Bounds for the MAX-SAT Problem. In Workshop on the Satisfiability Problem, Abstracts, Siena 53-62, 1996.

[199] C.H. Papadimitriou and K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice Hall, Englewood Cliffs, 1982.

[200] C.H. Papadimitriou. On selecting a satisfying truth assignment. In Proceedings of the 32nd Annual Symposium of the Foundations of Computer Science, pages 163–169, 1991.

[201] P. M. Pardalos. Complexity in numerical optimization. World Scientific, Singapore-and River Edge, N.J., 1993.

[202] G. Patrizi. The equivalence of an LCP to a parametric linear program with a scalar parameter. European Journal of Operational Research, 51:367-386, 1991.

[203] J. Pearl. Heuristics. Addison-Wesley, Reading, 1984.

[204] L.M. Pereira and A. Porto. Selective Backtracking, pages 107–114. Academic Press, 1982.

[205] C. Petrie. Revised dependency-directed backtracking for default reasoning. In Proceedings of AAAI'87, pages 167–172, 1987.

[206] J. Peysakh. A fast algorithm to convert Boolean expressions into CNF. IBM Computer Science RC 12913, No. 57971, T.J. Watson, New York, 1987.

[207] T. Pitassi and A. Urquhart. The complexity of the hajós calculus. SIAM J. on Disc. Math., 8(3):464–483, 1995.

[208] J. Plotkin, J. Rosenthal, and J. Franco. Correction to probabilistic analysis of the Davis Putnam Procedure for solving the Satisfiability problem. Discrete Applied Mathematics, 17:295–299, 1987.

[209] C. Poirier. A Functional Evaluation of Edit and Imputation Tools. UN/ECE Work Session on Statistical Data Editing, Working Paper n.12, Rome, Italy, 2-4 June 1999.

[210] D. Pretolani. A linear time algorithm for unique Horn satisfiability. Information Processing Letters 48:61–66, 1993.

[211] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. DIMACS Series Volume: Clique, Graph Coloring, and Satisfiability — Second DIMACS Implementation Challenge. Editors: D.S. Johnson and M.A. Trick, American Mathematical Society, 1996.

[212] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem Computational Intelligence 9 (3) 268-299 1993.

[213] P. Purdom. Survey of average time SAT performance. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994.

[214] P.W. Purdom. Search rearrangement backtracking and polynomial average time. Artificial Intelligence, 21:117–133, 1983.

[215] P.W. Purdom. Solving satisfiability with less searching. IEEE Trans. on Pattern Analysis and Machine Intelligence, 6:510–513, 1984.

[216] P.W. Purdom. Average time for the full pure literal rule. Information Sciences, 1994 78:269–291.

[217] P.W. Purdom and G.N. Haven. Probe Order Backtracking. SIAM J. on Computing, 1997.

[218] R. Puri and J. Gu. An efficient algorithm for computer microword length minimization. IEEE Transactions on CAD of Integrated Circuits and Systems, 12(10):1449–1457, Oct. 1993.

[219] R. Puri and J. Gu. A BDD SAT solver for satisfiability testing. Annals of Mathematics and Artificial Intelligence, Vol. 17, pp. 315–337, 1996.

[220] W.V. Quine. A way to simplify truth fubctions. American Mathematical Monthly, 62:627–631, 1955.

[221] J.R. Quinlan. Induction of decision trees. Machine Learning, 1:81–106, 1986.

[222] M. Resende and T. Feo. A GRASP for MAXSAT. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994.

[223] J.A. Robinson. A machine–oriented logic based on the resolution principle. Journal of the ACM, pages 23–41, 1965.

[224] A. Rosenfeld. Computer vision: Basic principles. Proceedings of the IEEE, 76(8):863–868, Aug. 1988.

[225] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs, 1995.

[226] Saab and V. B. Rao. Combinatorial optimization by stochastic evolution. IEEE Transactions on CAD, CAD10(4):525–535, Apr. 1991.

[227] A. Saldanha, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. A Framework for Satisfying Input and Output Encoding Constraints. In Proceedings of ACM/IEEE Design Automation Conference, pages 170–175, 1991.

[228] A. Samal and T.C. Henderson. Parallel consistent labeling algorithms. International Journal of Parallel Programming, 1988.

[229] I. Schiermeyer. Solving 3Satisfiability in less than O(1:579 n) steps. Lecture Notes in Computer Science 702, pages 379–394, 1993.

[230] I. Schiermeyer. Pure literal lookahead: an O(1:497 n) 3-Satisfiability algorithm. In Proc. of the Workshop on Satisfiability, Universitá delgi Studi, Siena, Italy, pages 63–72, 1996.

[231] J. S. Schlipf, F. Annexstein, J. Franco, and R. Swaminathan. On finding solutions for extended Horn formulas. Information Processing Letters 54, pages 133–137, 1995.

[232] B. Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, 1994.

[233] M. G. Scutella. A note on Dowling and Gallier's topdown algorithm for propositional Horn satisfiability. Journal of Logic Programming 8, pages 265–273, 1990.

[234] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Proceedings of AAAI'92, pages 440–446, Jul. 1992.

[235] B. Selman, H.A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In DIMACS Series Volume: Clique, Graph Coloring, and Satisfiability — Second DIMACS Implementation Challenge. American Mathematical Society, pp. 290–295, 1996.

[236] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In Proc. of the 12th National Conf. on Artificial Intelligence, pages 337–343, Seattle, July 1994.

[237] R.C.J. Shi, A. Vannelli, and J. Vlach. An improvementon Karmarkar'salgorithm for integer programming. COAL Bulletin of the Mathematical Programming Society, 21:23–28, 1992.

[238] R. Sosic and J. Gu. Fast search algorithms for the n-queens problem. IEEE Trans. on Systems, Man, and Cybernetics, SMC21(6):1572–1576, Nov./Dec. 1991.

[239] R. Sosic and J. Gu. Efficient local search with conflict minimization. IEEE Trans. on Knowledge and Data Engineering, 6(5):661–668, Oct. 1994.

[240] R. Sosic, J. Gu, and R. Johnson. The Unison algorithm: Fast evaluation of Boolean expressions. ACM Transactions on Design Automation of Electronic Systems, (1)4:456–477, Oct. 1996.

[241] R. Stallman and G.J. Sussman. Forward reasoning and dependency directed backtracking. Artificial Intelligence, 9(2):135–196, 1977.

[242] L. Sterling and E. Shapiro. The Art of Prolog, Advanced Programming Techniques. The MIT Press, Cambridge, Massachusetts, 1986.

[243] H.S. Stone and J.M. Stone. Efficient search techniques – an empirical study of the n-queens problem. IBM J. Res. Develop., 31(4):464–474, July 1987.

[244] R. P. Swaminathan and D. K. Wagner. The arborescence–realization problem. Discrete Applied Mathematics 59, pages 267–283, 1995.

[245] A. Törn and A. Źilinskas. Global Optimization. Springer-Verlag, 1989.

[246] P. P. Trabado, A. Lloris-Ruiz, and J. Ortega-Lopera. Solution of Switching Equations Based on a Tabular Algebra. IEEE Trans. on Computers, C42:591–596, May 1993.

[247] K. Truemper. Alpha-balanced graphs and matrices and GF(3)representability of matroids. Journal of Combinatorial Theory B 32, pages 112–139, 1982.

[248] K. Truemper. Monotone Decomposition of Matrices. Technical Report UTDCS194, University of Texas at Dallas. 1994.

[249] K. Truemper. Polynomial algorithms for problems over d-systems. Presented in the 3rd International Symposium on AI and Mathematics, Jan. 1994.

[250] K. Truemper. Effective Logic Computation. Wiley, New York, 1998

[251] G.S. Tseitin. On the Complexity of Derivations in Propositional Calculus. In Structures in Constructive Mathematics and Mathematical Logic, Part II, A.O. Slisenko, ed., pages 115–125. 1968.

[252] J.D. Ullman. Principles of Database Systems. Computer Science Press, Rockville, 1982.

[253] A. Urquhart. Hard examples for resolution. J. of ACM, 34:209–219, 1987.

[254] A. Urquhart. The relative complexity of resolution and cutfree gentzen systems. Annals of Mathematics and Artificial Intelligence, 6:157–168, 1992.

[255] A. Urquhart. The complexity of propositional proofs. The Bulletin of Symbolic Logic, 1(4):425–467, 1995.

[256] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. J. of VLSI Signal processing, 1993.

[257] B. W. Wah, G. J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. IEEE Computer, 18(6):93–108, June 1985. Also in Tutorial: Computers for Artificial Intelligence Applications, ed. B. W. Wah, IEEE Computer Society, 1986, pp. 173188.

[258] B. Wah and G.J. Li. Computers for Artificial Intelligence Applications. IEEE Computer Society Press, Washington D. C., 1986.

[259] B.W. Wah, editor. New Computers for Artificial Intelligence Processing. IEEE Computer, volume 20, number 1. IEEE Computer Society Press, 1987.

[260] B. W. Wah, G. J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. In L. Kanal, V. Kumar, and P. S. Gopalakrishnan, editors, Parallel Algorithms for Machine Intelligence and Pattern Recognition, pages 102–145. SpringerVerlag, New York, NY, 1990.

[261] B. W. Wah and L.C. Chu. Combinatorial search algorithms with metacontrol: Modeling and implementations. Int'l J. of Artificial Intelligence Tools, 1(3):369–397, September 1992.

[262] B. W. Wah and Y. Shang. A comparison of a class of IDA* search algorithms. Int'l J. of Artificial Intelligence Tools, 3(4):493–523, October 1995.

[263] B. W. Wah and Y. Shang. A discrete lagrangian-based global search method for solving satisfiability problems. In DingZhu Du, Jun Gu, and Panos Pardalos, editors, Proc. of the DIMACS Workshop on Satisfiability Problem: Theory and Applications. American Mathematical Society, March 1996.

[264] B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best first search. Trans. on Software Engineering, SE11(9):922–934, September 1985.

[265] W. Wang and C.K. Rushforth. An adaptive local search algorithm for channel assignment problem. IEEE Trans. on Vehicular Technology, Vol. 45, No. 3, pp. 459–466, Aug. 1996.

[266] M. B. Wells. Elements of Combinatorial Computing. Pergamon Press, Oxford, 1971.

[267] H.P. Williams. Linear and integer programming applied to the propositional calculus. Systems Research and Information Sciences, 2:81–100, 1987.

[268] M.J. Wilson. Compact normal forms in propositional logic and integer programming formulations. Computers and Operations Research, 90:309-314, 1990.

[269] W.E. Winkler. State of Statistical Data Editing and current Research Problems. UN/ECE Work Session on Statistical Data Editing, Working Paper n.29, Rome, Italy, 2-4 June 1999.

[270] P.H. Winston. Artificial Intelligence. Addison-Wesley, Reading, 1984.

[271] L. Wos, G. A. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy. Journal of the ACM, 12:536–541, 1965.

[272] L.C. Wu and C.Y. Tang. Solving the satisfiability problem by using randomized approach. Information Processing Letters, 41:187–190, 1992.

[273] T.Y. Young and K.S. Fu, editors. Handbook of Pattern Recognition and Image Processing. Academic Press, Orlando, 1986.

[274] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In Proceedings of AAAI'88, pages 155–160, 1988.

[275] V.N. Zemlyachenko, N.M. Korneeko, and R.I. Tyshkevich. Graph isomorphism problem. J. of Soviet Mathematics, 29:1426–1481, 1985.

[276] S. Zhang and A. G. Constantinides. Lagrange programming neural networks. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 39(7):441–452, 1992.

[277] H. Zhang. SATO: An Efficient Propositional Prover. in Proc. of International Conference on Automated Deduction (CADE-97), Lecture notes in Artificial Intelligence 1104, Springer-Verlag, 308–312, 1997.

[278] H. Zhang and M.E. Stickel. Implementing the Davis-Putnam Method. Technical Report, The University of Iowa, 1994.