Efficient Decision Procedures for the Integration of Planning and Formal Verification in Advanced Robotic Systems

Marco Maratea

A dissertation in
Robotics

Thesis Supervisors: Chiar.$^{\underline{\text{mo}}}$ Prof. Enrico Giunchiglia

Dipartimento di Informatica, Sistemistica e Telematica
Università degli Studi di Genova.

Dalla mia tesi di Laurea, ancora

"Credi che sia facile nascondere le lacrime?
Devo ancora fingere, io che vorrei soltanto piangere"

Francesco Renga – Ancora di lei

Dedicato a tutti i bambini del sud-est
asiatico vittime dello Tsunami

"Angelo, prenditi cura di lei,
Lei non sa, vedere al di la di quello che ha"

Francesco Renga – Angelo
Vincitore del Festival di Sanremo 2005

L'ultima dedica è per Karol Wojtila, il nostro papa,
la cui scomparsa ha lasciato un grande vuoto.
Non ti dimenticheremo.

Ringraziamenti

Ed eccoci ancora arrivati all'ultimo passo nella scrittura di una tesi anche se compare solitamente nelle prime pagine.

Prima di tutti vorrei ringraziare i miei genitori, che mi sono stati sempre vicini in questi anni nonostante forse non fossero completamenti convinti di questa scelta. Questo lavoro è ancora dedicato a voi.

Poi Enrico, il mio supervisore, che anche prima del dottorato mi è stato sempre vicino e mi ha spronato ad andare avanti, a volte a modo suo ... non so cosa succedera in futuro, ma ti ringrazio per il supporto che mi hai dato e avermi seguito in questo lungo cammino.

Per quanto riguarda l'ambiente del laboratorio, vi sono molte persone da citare. Partiamo da Alessandro, che se ricordo bene nella tesi di laurea avevo definito "l'altra meta" dell'MRG. Ora "l'altra metà" è anche un fatto "istituzionale" ... che peccato almeno da un punto di vista sentimentale ... Ora, oltre che per i caffè e le chiaccherate posso ringraziarti anche per aver lavorato insieme ed avermi aiutato ... spero succeda ancora.

Per quanto riguarda il resto del lab (ora STAR+AI) in questi anni ho trovato diversi amici oltre che colleghi. Parto dagli "storici" LucaC e Massimo: il lavoro, gli impegni ed il fatto di essere "fisicamente" in luoghi diversi ci permette di parlare e stare meno insieme, ma per me siete sempre due grandi amici.

Passiamo poi a Claudio, Jacopo e LucaP, le persone con cui ho avuto più a che fare ultimamente, e non solo lavorativamente. Ho scoperto amici in voi prima che colleghi, spero che continueremo in questa direzione.

Poi Tac che, tra alti e bassi, mi è comunque stato vicino sin dalla mia tesi di laurea e mi ha insegnato diverse cose ... addirittura ora riprenderemo a lavorare insieme dopo un paio d'anni, è un piacere ... ed ancora auguri a te e tua moglie per il piccolo che tra poco battezzerete.

E poi voglio citare persone che ho conosciuto in ambito lavorativo nel laboratorio e che hanno lasciato il loro segno in me ... Felice, compagno di tesi per molti mesi, Silvio ora in Francia, utili tutti i suoi consigli, Corrado, Sabrina, Giuliano, Adriano, Ilenia ... grazie a tutti.

Spostandosi di pochi metri voglio ringraziare alcune persone di altri laboratori che mi hanno spesso tenuto compagnia. Lorenzo, amico anche lui, si ci diverte a Boston? Non sto ricevendo da te e-mail simili alle mie da Stanford...:-) MarcoFato, un vero mito ... Carlos, Giorgio, Ivan, Silvia, Elisa, Barbara, Federica ... grazie a tutti ... ed anche Mauri, ex-biolab. Ho avuto "l'onore" di essere citato come amico nella tua tesi di laurea ... anche per me lo sei.

In questo lungo periodo, ho affrontato anche un paio di lunghi periodi all'estero. Sono stato in posti bellissimi, ringrazio i Professori Vladimir Lifschitz e David Dill per avermi dato la possibilità di lavorare con loro. Ma a volte, quando si è così lontani da casa non è sempre facile, quindi ringrazio Marco e Laura "di Austin" (ora pisani ... auguri, tra qualche mese ...) per il

sostegno e la compagnia che ci siamo fatti; ed AlessandroC e LucaB per tutto il tempo passato insieme in California ... e per le risate, che continuano anche ora via e-mail e quando ci vediamo ...

C'e' anche un mondo fuori da Villa Bonino, ma in certi momenti si corre il rischio di dimenticarselo ...

Grazie ai miei più cari amici "genovesi", Andrea, Davide, Paolo, MarcoJ, MarcoL, MarcoP, Danilo, Simone, LucaM, Clara ...sicuramente, per vari motivi, negli ultimi tempi (un po lunghi) ci vediamo molto meno rispetto a prima. Mi preme dirvi, che io vi voglio bene come quando ci vedevamo quasi tutti i giorni ... voi per me resterete sempre "i miei amici".

Vorrei ora ringraziare i miei parenti, zii e cugini, per il loro sostegno in questo percorso. Grazie ai miei vicini Anna, Carlo e Laura, per questo e per tutto quello che hanno fatto per me da quando solo piccolo.

Grazie a Stefano, Barbara, Massimo, Silvia ed Angela. Vi avrei già citato come "cugini", ma meritate una citazione particolare per il rapporto che abbiamo sempre avuto. E da Stefano un ringraziamento particolare, insieme a Sara ... poi si capirà ...

Grazie alle persone che mi sono lontane, anche se così vicine. In particolare penso a tutte le persone, mia nonna, i miei zii, cugini amici che vivono al sud, e che hanno rappresentato e rappresentano una parte molto importante della mia vita. Tra tutti cito solo alcuni, con i quali per motivi diversi ho avuto un rapporto particolare. Ciccio, ci siamo un po allontanati negli ultimi anni, a volte non è facile ... Ma per me tu rimani sempre lo stesso, spero che il tempo ci riavvicinerà... quanto bene che voglio anche ai tuoi genitori. Pasqualino e Luigi, che piacere che siete venuti a trovare per il vs. viaggio di nozze un amico lontano: Vi voglio bene. E poi GiuseppeV ... una "nuova entrata" dalla mia tesi di laurea .. forse per le differenza d'età ma solo ultimamente ho scoperto che sei un grande amico ... spero di passare ancora molto tempo con te a parlare, e ... ci vengo a Napoli prima o poi :-)

Ed Enzo ... sembra incredibile che, nonostante ci sentiamo e vediamo così poco siamo talmente amici, invece è così ... ci bastano pochi secondi e sembra che non ci siamo visti tutti i giorni da anni. E' veramente una cosa bellissima, ma io comunque non perdo la speranza di vederti più spesso in futuro.

E, come nella mia tesi di laurea, l'ultimo pensiero è per chi non c'e' più ... purtroppo ora le persone da ricordare sono diventate tre ... a volte la vita non è facile, e la morte di persone così care ti lascia sgomento senza possibilità di fare nulla per rimediare ... Nonni, mi mancate tanto ... Un'altro caro ricordo è per mio zio Carlo, che è mancato molto tempo fa. Con questo ricordo spero anche di colmare una grave mancanza che vi era nella mia tesi di laurea riguardo il suo ricordo ... zio, è passato molto tempo ed ero molto piccolo, ma ricordo la persona che eri ... e non ti ho mai dimenticato ne ti dimenticherò...

... Ma stavolta no, voglio terminare i miei ringraziamenti con un messaggio positivo e di speranza ... che è naturalemente rivolto a te, Angelica. Chi l'avrebbe detto quel giorno in cui Stefano e Sara ci hanno fatto conoscere ... anche se forse ci speravamo entrambi. Da li è inziato un percorso che ci ha portato per ora fino a qui. Hai portato serenità e felicità nella mia vita, che sono cose importantissime e non semplici da ottenere, e per questo ti ringrazio infinitamente. E spero con tutto il cuore di proseguire per sempre questo percorso insieme a te amore mio ...

Contents

1 Introduction
1.1 Research area, motivations and goals . . . . . .
1.2 Contribution of the thesis iii
1.3 Structure of the thesis iv

I Propositional Satisfiability vi

2 Solving the satisfiability (SAT) problem vii
2.1 Propositional logic vii
2.2 Algorithms for SAT viii
2.3 The Davis-Logemann-Loveland (DLL) algorithm x
2.3.1 Backjumping and Learning xiii
2.4 The SAT solver(s) SIM(O) xiv
2.4.1 The SAT solver SIM xiv
2.4.2 The SAT solver SIMO xv
2.5 Extending the DLL algorithm to work as an enumerator xvi

II Safe Planning via Propositional Logic xviii

3 Introduction xix
3.1 Possible and Valid Plans xix
3.1.1 Safe Plans xxi
3.1.2 Safe Planning: Plan Validation xxii
3.1.3 Safe Planning: Plan Generation xxiii
3.2 Summing up xxiv

4 Dependent and Independent Variables xxvi
4.1 Variable Dependency xxvi
4.2 DLL and Variable Dependency xxvii
4.3 Experimental results xxviii
4.3.1 Test set xxviii
4.3.2 DLL implementation xxviii
4.3.3 Effects of the IVS Heuristic xxix
4.4 Summing up xxxiii

5 Testing the efficiency of improvements **xxxiv**
5.1 Experimenting with look-ahead techniques . . . xxxv
5.1.1 Iterative version of the (modern) SAT solver SIMO . . . xxxv
5.1.2 Experimental Setting . . . xxxvii
5.1.3 Simo-Up vs. Simo-Fp . . . xxxviii
5.2 (In)Effectiveness of failed literal detection . . . xxxix
5.2.1 Introducing oracles in Simo-Fp . . . xl
5.2.2 Simo-Up vs. Simo-Fp(*) . . . xli
5.2.3 Ineffectiveness of failed literal in a modern SAT solver . . xliii
5.3 Summing up . . . xlv

III Safe Planning via Temporal Reasoning **xlvi**

6 Introduction **xlvii**
6.1 Working example . . . xlviii

7 From DLL to a decision procedure for TR **xlix**
7.1 Separation logic . . . xlix
7.2 Syntax . . . xlix
7.2.1 Semantics . . . xlix
7.2.2 Valuations . . . l

8 The TSAT++ solver **lv**
8.1 Optimizations . . . lv
8.1.1 IS_n Pre-processing . . . lv
8.1.2 Early Pruning . . . lvii
8.1.3 Model Reduction . . . lviii
8.1.4 Best Reason Detection . . . lix
8.2 Related work . . . lxi
8.3 Implementation and Experimental Analysis . . . lxiii
8.3.1 Disjunctive Temporal Problems . . . lxiv
8.3.2 Real-world problems . . . lxv
8.3.3 Hand-made problems . . . lxvi
8.3.4 Hardware verification problems . . . lxvii
8.4 Summing up . . . lxix

IV Safe Planning via Answer Set Programming **lxx**

9 Introduction **lxxi**
9.1 Answer Set Programming . . . lxxi
9.2 From SAT formulas to logic programs . . . lxxiii
9.3 From logic programs to propositional formulas . . . lxxiv

10 From DLL to a decision procedure for ASP **lxxv**
10.1 SAT-Based Answer Set Solvers . . . lxxvi

11 Answer Set solver Cmodels2 **lxxx**
11.1 Integration in CMODELS lxxx
11.2 Experimental Results lxxxii
11.2.1 Finding one answer set lxxxiii
11.2.2 Finding all answer sets lxxxv
11.3 Summing Up lxxxvii

12 On the relation between SAT and ASP **lxxxviii**
12.1 On the relation between ASP and SAT xc
12.1.1 Answer Set Programming xc
12.1.2 Propositional reasoning xci
12.1.3 Translation from ASP to SAT xci
12.1.4 Translation from SAT to ASP xcii
12.2 On the relation between ASP and DLL xcii
12.2.1 ASP xcii
12.2.2 DLL xciv
12.2.3 ASP, DLL and their relation xcv
12.3 On the relation between AS and SAT solvers xcvii

V Safe Planning via SAT-related optimization problems **ci**

13 Introduction **cii**

14 From optimization functions to CNF formulas **civ**
14.1 Naive encoding civ
14.2 Warners's encoding cv
14.2.1 Extension to general coefficients cviii
14.3 Bailleux and Boufkhad encoding cviii
14.3.1 Optimizing the encoding cxi
14.3.2 Extension to general coefficients cxi

15 The OPTSAT solver **cxiii**
15.1 Propositional solvers tested cxvi

16 Experimental analysis **cxvii**
16.1 Benchmarks cxvii
16.1.1 Randomly generated problems cxvii
16.1.2 Real-world problems cxviii
16.2 Rival systems cxviii
16.3 Results cxix
16.3.1 Analysis on Max-SAT problem cxix
16.3.2 Analysis on Min(Max)-ONE problem cxxi

17 Conclusions **cxxv**
17.1 Future work cxxvi

Bibliography **cxxvii**

Chapter 1

Introduction

1.1 Research area, motivations and goals

The increasing complexity of the services requested to robotic devices in space applications results in a need for more and more sophisticated and autonomous systems. A compelling requirement for space autonomy led to the development of systems that perform complex, time consuming and critical tasks, possibly without the need of human intervention [MNPW98]. Planning is a research area in Artificial Intelligence (AI) aiming at the construction of systems – called planners – that enable a robot to autonomously synthesize a series of actions that will achieve its goals. Planning has been studied since the early years of AI; recently the interest has been renewed, with systems able to automatically generate plans with hundreds (or even thousands) of actions.

On the other hand, the same increasing complexity of the requested services causes an analogous increase in the complexity of the specifications and of the programs controlling the robotic devices. Concerns arise about the correctness of the specifications of the program and/or the environment in which the robots are supposed to operate [SFM00]. Model checking (MC) (see, e.g., [CGL94]), is a research area in Computer Science devoted to the definition of procedures for the automatic verification of programs and specifications. Current model checkers are able to efficiently verify systems with more than 10^{120} states [BCL+94]. The characteristic that makes MC particularly attractive is that it is a "push-button" technology: It is completely automatic.

There are two important similarities between planning and MC: Both perform state-space exploration, and both are fully automatic. However, the planning and Formal Verification (FV) community fields have been evolved separately until recently, each adopting and developing its own techniques and tools. In the last few years, we have seen a cross-fertilization between these two areas in which ideas originally thought in FV have been successfully employed in planning [CGGT97, GT99], and vice-versa [BCCZ99].

Throughout this thesis, we address the problem of "safe planning". By safe planning, we mean the task of generating/validating plans that not only achieve the goal, but verify also a set of other user-defined properties, e.g., safety properties. This is particularly important in the context of space applications, where autonomy and safety are indeed two crucial properties any system has

to guarantee. In particular, safe planning boils down to further exploit the similarities between planning and MC by defining procedures:

- for the validation of user-defined plans with respect to given (safety) properties, and
- for the generation of plans satisfying the desired properties.

The problem of safe planning can be addressed via reduction to:

- a propositional formula, that can be solved by a satisfiability solver (introduced in [ACGT01]);
- a logic program, to be solved under the Answer Set Semantic (ASP) [GL88a] (proposed in this thesis);
- a propositional formula enhanced with temporal constraints (called Separation Logic, SL for short [Pra77]), that can be solved by a dedicated solver (proposed in this thesis);
- an optimization problem related to the solutions returned by a satisfiability solver (proposed in this thesis).

The choice of the above logics was dictated by several considerations, the main one being the fact that most problems of interest in the fields of planning and formal verification can be naturally encoded in them.

The "Safe Planning via Propositional Logic" approach was introduced in [ACGT01]. Given a planning problem (expressed, e.g., in PDDL language) and the safety properties the plan has to comply (expressed in Linear Temporal Logic, LTL), a procedure, that rely on the "planning as satisfiability approach", for generating plans that not only achieve the goals but verify also a set of other safety properties was introduced. This procedure is based on a reduction to a propositional formula. The planning as satisfiability approach was introduced in [KS92] in the classical setting (in which all actions have a deterministic and "a-priori" known effect) in 1992, and then extended to the nondeterministic (in which an action can have several effects when executed) domains in [TCG00]. Each component of a planning problem (action/fluent) is mapped into a (series of) propositional variables, and the planning problem is translated into a propositional formula. Then a satisfiability solver is called, and the solution (if any) is mapped-back into a plan. The approach can leverage on the fact that modern satisfiability solvers can deal with problems having millions of variables in few seconds.

The "Safe Planning via Answer Set Programming" approach is introduced in this thesis. Answer set programming [GL88a] is a new declarative paradigm for solving search problems appearing in knowledge representation and reasoning. To solve a problem, a programmer designs a logic program such that models of the program are solutions to the problems. The procedure and the approach presented in [ACGT01] are encoded as logic programs (to be solved under the Answer Set semantic). The approach presented rely on the strong link between ASP and SAT: A SAT formula can be translated into a logic problem in a modular way. Despite the strong link, ASP and SAT are different is many ways: Among them, ASP is a non monotonic logic while SAT is monotonic;

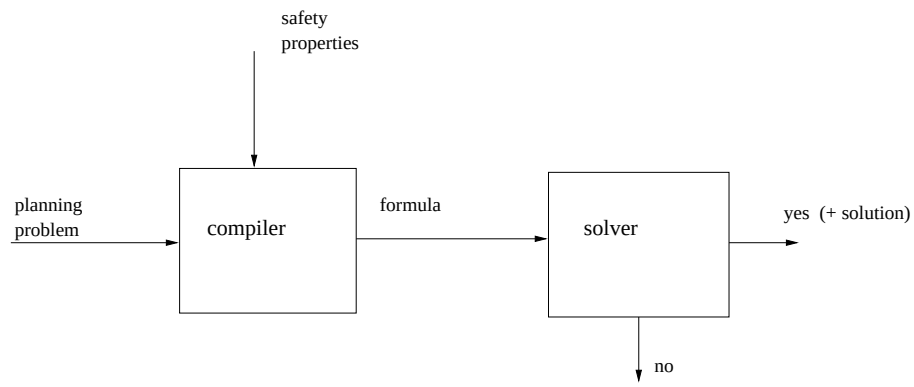


Figure 1.1: Implementing Safe Planning.

ASP allows in general for a more "compact" representation of the problems, while SAT solvers are more optimized than ASP solvers; and the two formalism are best suited for different problem domains.

The "Safe Planning via Temporal Reasoning" approach uses propositional logic enhanced with constraints. In this thesis we will restrict to constraints of the type $x - y \leq c$, where x, y are arithmetic variables, and c is a numeric constant. The resulting theory is known as Separation (or Difference) Logic [Pra77], in the area of FV. This theory is enough for our approach due to the way the encoding of actions (and fluents) is implemented (see Chapter 6). Despite its simplicity, SL can be often used to encode interesting problems from the planning, scheduling and FV domains. A solver for this theory must have a specialized reasoner for checking the arithmetic consistency of the problem: Fortunately, this can be done efficiently in polynomial time.

The last approach is related to some optimization problems that can be applied to a propositional formula. Among them, there is the "Min-One" problem: "Given a propositional formula, find the satisfying assignment with fewer variables assigned to true". The "Safe Planning via SAT-related Optimization Problems" approach rely on an encoding of a planning problem (+ safety properties) to a propositional formula and a minimization of a functional cost (that defines the Min-ONE problem and the fact that the propositional satisfying assignment must be "minimal" w.r.t. this function). The functional cost is defined on the actions of the problem. This functional cost can be easily encoded in a propositional formula as well: Then, running a slightly modified satisfiability solver on the overall formula guarantees (under some conditions to be imposed on the variables of the resulting propositional formula) that the first satisfying assignment is "optimal", i.e., the solution corresponds to a plan with minimal "cost".

Summing up, all the encodings we have described can be part of a general schema for solving the problem of Safe Planning reported in Fig. 1.1.

Given in input the description of the planning problem (e.g. described in the STRIPS/PDDL language) and the safety properties that the plan has to enjoy (expressed in LTL), a compiler is called. A compiler is one of the approaches we have seen before, and a formula in the correspondent logic (due to the compiler

used) is given in output. Then the formula is fed to a solver that is able to decide the satisfiability of a formula in the underlined logic.

1.2 Contribution of the thesis

The contributions of this thesis are both from a "theoretical" side and from a "practical", implementation oriented side. On the theoretical side, we have introduced (as anticipated before) three new approaches for implementing Safe Planning

Safe Planning via Temporal Reasoning where the problem of Safe Planning is compiled into a Temporal Reasoning problem via a Separation formula;

Safe Planning via Answer Set Programming where the problem of Safe Planning is compiled into a logic program to be solved under the Answer Set semantic;

Safe Planning via SAT Optimization where the problem of Safe Planning is compiled into an optimization problem composed by a propositional formula and a functional cost to be minimized defined on a subset of the variables of the propositional formula (the actions of the planning problem).

On the "practical", implementation oriented side the contribution of the thesis is on the design, the implementation and the experimental testing of four systems:

SIMO (Satisfiability Internal Modulo Object-oriented), an efficient new generation decision procedure for propositional satisfiability. SIMO is the evolution of the SIM solver;

Cmodels2 (Computing models), a decision procedure, based on propositional satisfiability, for finding AS of logic programs;

TSAT++ (Temporal SATisfiability), a decision procedure , based on propositional satisfiability, for solving formula expressed in SL;

OPTSAT (OPTimal SATisifability), a decision procedure for optimization problems related to propositional satisfiability.

Each system embeds both state-of-the-art techniques, and new ideas and algorithms that were developed in the course of the thesis to improve the current available decision procedures in all the logics that we study. As a result, SIMO, Cmodels2, TSAT++ and OPTSAT compare favorably with analogous systems from other researchers, and in particular:

- SIMO compares favorably with other systems. It ranked in the first 5 position (out of 28 competitors) in the SAT 2002 competition. It is the object-oriented version of the solver SIM (Satisfiability Internal Modulo) previously developed in the STAR-Lab group Then it has been used in particular as satisfiability search engine in solvers for more expressive formalism (namely Cmodels2, TSAT++ and OPTSAT);

- Cmodels2 is continuously under development, and it is a joint project with the University of Texas at Austin. Among other SAT solver, it embeds SIMO as internal propositional solver, and it compares very well w.r.t. rival systems. It uses a new algorithm for solving these problems w.r.t. state-of-the-art systems;
- TSAT++ is faster than other state-of-the-art systems and it features newly introduced techniques;
- OPTSAT is competitive with other comparable state-of-the-art systems: It features a newly introduced algorithm to solve the problem, and other improvements are coming.

The systems that we developed are interesting not only from the research perspective, but also from an application point of view. SIM(O) has been at the heart of two technology-transfer projects. The first one is Thunder/Forecast, an open architecture for verification of hardware designs developed by the Logic Validation Technologies Group at the Israel Development Center of Intel Corp. An early version of SIMO was integrated into Thunder and used to debug the design of some Pentium IV components. The results obtained with Thunder/SIMO outperformed the results obtained with other traditional validation technologies implemented in Forecast. The second technology transfer project centered around SIM has been in cooperation with Istituto per la Ricerca Scientifica e Tecnologica (IRST-ITC) in Trento and involved the integration of SIM into the NuSMV model checker, a tool for debugging and verifying hardware designs and software protocols.

The other systems presented in this thesis aim to go in the same direction.

1.3 Structure of the thesis

The thesis is composed by five parts each one focusing on the details about each of the approach as well as the decision procedure used.

Propositional satisfiability focuses on propositional logic and the problem of determine the consistency of a propositional formula.
Chapter 2 introduces basic definitions and notations to be used in the remainder of the part; the basic Davis-Logemann-Loveland (DLL) algorithm for propositional satisfiability is described. The SAT solver SIM and SIMO on which all the other work is based on are presented, as well as how to extend DLL algorithm to work as an enumerator of propositional satisfiable assignments.

Safe Planning via Propositional Logic focuses on the approach of implementing Safe planning via a reduction to a propositional formula and some optimization on the decision of propositional formulas.

- Chapter 3 defines the approach.
- Chapter 4 introduces and experiments an optimization in the DLL algorithm that takes into account the "meaning" of the propositional variables, by means of "independent" variables, those variables such that once they are assigned all the others are as well.

- Chapter 5 introduces and studies a method for testing (possible) improvements that a new technique could introduce "a-priori" i.e. before optimally implement it. A case study of failed-literal is also introduced.

Safe Planning via Temporal Reasoning focuses on the approach of implementing Safe planning via a reduction to a Separation formula. Then the extension of the DLL algorithm is presented as well as an efficient solver for this logic.

- Chapter 6 defines the approach.
- Chapter 7 extends the DLL algorithm in order to solve Temporal (or Separation) formulas.
- Chapter 8 presents and evaluate a decision procedure, TSAT++, for this theory.

Safe Planning via Answer Set Programming introduces answer set programming; focuses on Safe Planning via Answer Set Programming reduction; presents and tests a solver for logic programs under the answer set semantic.

- Chapter 9 defines the approach and answer set programming.
- Chapter 10 extends the DLL algorithm in order to solve logic programs under answer set semantic.
- Chapter 11 introduces and evaluates an ASP solver, Cmodels2, for logic programs under the answer set semantic.
- Chapter 12 studies both from a theoretical and practical point of view the relation between SAT and ASP procedures.

Safe Planning via SAT-related Optimization problems focuses on the approach of implementing Safe planning via reduction to some SAT-related optimization problems.

- Chapter 13 introduces the approach and the SAT-related optimization problems Max(Min)-SAT and Min(Max)-One
- Chapter 14 describes the encodings from linear constraints to propositional formulas needed by the encoding.
- Chapter 15 introduces the (new) algorithm, completely based on propositional satisfiability search, for solving the optimization problems, and the related solver OPTSAT.
- Chapter 16 introduces the benchmarks used and evaluate the OPTSAT decision procedure.

Part I

Propositional Satisfiability

Chapter 2

Solving the satisfiability (SAT) problem

The purpose of this chapter is to set out basic terminology about SAT, and the concepts and issues that subsequent chapters will discuss in more detail when needed. What is presented here is intended mainly for reference and it is not meant to be detailed and complete. Extensive references will be provided for all the results herewith presented.

The chapter consists of five sections. the first one presents some basic definitions and notation. The second lists the most widely used strategies for solving SAT. The third describes the Davis-Logemann-Loveland (DLL) algorithm, the basic building block on top of which we develop all the SAT techniques presented in this thesis. The fourth presents the SAT solvers SIM and SIMO. The last section extends the DLL algorithm to work as an enumerator of satisfying assignments.

2.1 Propositional logic

A *proposition* (or *Boolean variable*) is a symbol from a set $P = \{p_1 \ldots p_n\}$, our *propositional alphabet*. A *literal* is either a proposition (say, p) or the complement of one (denoted by $\neg p$ or $\overline{p}$); in the first case, we say that l is a *positive literal*, and in the second, we say that l is a *negative literal*. We say that a proposition p and a literal l are *associated with* each other if either $l \equiv p$ or $l \equiv \neg p$. A *clause* is a finite disjunction of literals. A propositional formula is in *conjunctive normal form* (*CNF*) if it is a finite conjunction of clauses. For example,

$$\begin{array}{lcl} cl_1 & : & p_1 \vee p_2 \\ cl_2 & : & p_2 \vee p_3 \\ cl_3 & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ cl_4 & : & \neg p_1 \vee \neg p_3 \end{array} \tag{2.1}$$

is in CNF, where each cl_i $(1 \leq i \leq 4)$ is a clause. We will assume that no clause contains two literals of the form p and $\neg p$. We will also assume that the literals in a given clause are unique. This latter assumption allows us to treat clauses

as sets of literals, and formulas as sets of clauses. So, for example:

$$\begin{array}{l}\{\{p_1, p_2\} \\ \ \{p_2, p_3\} \\ \ \{\neg p_1, \neg p_2, p_3\} \\ \ \{\neg p_1, \neg p_3\}\}\end{array} \tag{2.2}$$

is an equivalent way of looking at the CNF of 2.1. Given a CNF formula φ, the *length* of each clause $c \equiv \bigvee_{i=1}^{w} l_i$ is simply w the number of literals in it, also denoted as $|c|$; the length of φ is $|\varphi| = \sum_{i=1}^{m} |c_i|$ the sum of the length of each clause; $n(\varphi)$ is the number of propositions in φ and $m(\varphi)$ is the number of clauses in φ. When φ is understood from the context, we will refer to these parameters as n and m respectively.

Any propositional formula can be converted to an equi-satisfiable CNF: By allowing the introduction of auxiliary variables (see, for example, [PG86a, Tse70]) the worst case computation time remains polynomial in the size of the original formula.

A truth assignment v is a partial function from the set of propositions to $1, 0$ (denoting respectively the values TRUE and FALSE for logical truth and falsehood respectively). We can extend the definition of v in a natural way so that it assigns truth values to literals, clauses and formulas: For a literal l, if $l \equiv p$ for some proposition p, then $v(l) = v(p)$, otherwise $v(l) = \neg^v v(p)$. For a clause $c \equiv \bigvee_{i=1}^{w} l_i$, $v(c) = (\bigvee_{i=1}^{w})^v v(l_i)$. For a formula $\varphi =\equiv \bigwedge_{i=1}^{m} c_i$, $v(\varphi) = (\bigwedge_{i=1}^{m})^v v(c_i)$. The interpretation of the connectives $\neg^v$, $(\bigvee_{i=1}^{w})^v$ and $(\bigwedge_{i=1}^{m})^v$ are the usual ones. If $*$ is either a proposition, literal, clause, or formula, we say that v *satisfies* $*$ if $v(*) = 1$, and v *falsifies* $*$ if $v(*) = 0$. When we discuss a truth assignment v in the context of a particular formula φ, we will assume that v's domain is restricted to the set of propositions in φ. Since v is a partial function, we will say that propositions, literals, clauses and formulas can have one of three possible truth values with respect to v: true, false and *undefined* (also *open*). A proposition, etc., that is not open with respect to v is said to be *valued* by v. If v values every proposition in a formula φ, we say that v is complete w.r.t. φ, otherwise it is incomplete. v need not be complete in order to value φ.

The CNF-satisfiability problem (SAT) is simply this [GJ79]: Given a propositional formula φ in CNF, is there a truth assignment v that satisfies φ?

Finally, we define some special kinds of clauses. Given a CNF formula φ, a *unary clause* or *unit clause* in φ is a clause with length one, and a *binary clause* in φ is a clause of length two. An *open unary clause* is an open clause containing exactly one open literal, and an *open binary clause* is a binary clause containing exactly two open literals. When it is clear from the context we shall blur the distinction and talk about unary (resp. binary) clauses even when we to refer to open unary (resp. binary) clauses.

2.2 Algorithms for SAT

Needless to say, SAT is such a fundamental problem in logic and computer science that it received a lot of attention and many different algorithms and techniques have been devised to try and solve it efficiently (see, for a good

survey [GPFW97]. In particular, [GPFW97] classifies SAT algorithms according to three dimensions:

1. sequential vs. parallel
2. discrete vs. continuous
3. constrained vs. unconstrained

In the remainder of this section (and of this thesis) we shall be concerned with *sequential* and *discrete* algorithms to solve the SAT problem. This leaves the third dimension to be sorted out.

Constrained formulations. The goal is to satisfy all the constraints, i.e., all the clauses in φ.

Unconstrained formulations. The goal is to minimize the number of unsatisfied clauses.

We will now briefly review some relevant contributions for both formulations. The list presented is an excerpt from the one in [GPFW97] and is by no means exhaustive.

- The Davis-Putnam (DP) algorithm [DP60] and the improved Davis-Logemann-Loveland (DLL) algorithm [DLL62] perform a backtracking search through the space of possible truth assignments for φ.
- Ordered Binary Decision Diagrams (OBDDs) [Bry92] provide a canonical representation for propositional (non-CNF) formulas.
- The resolution algorithm [Rob65] derives implicates of the formula until a contradiction is found or no more new implicates can be generated.
- The Stålmarck's method [SS98], a saturation based theorem proving algorithm with special treatment for equalities.
- Greedy local search [SK93] algorithms and their variants (see [HS00] for a recent survey) randomly select an assignment for φ and then try to minimize the unsatisfied clauses using various strategies.
- other approaches, such as term rewriting, production systems, enumerating assignments, some of which are covered in [GPFW97]

Many of the discrete constrained algorithms eliminate one variable at a time. This can be done either by making repeated use of resolution, as was done in the original version of the DP algorithm [DP60], or by assigning some variable each possible value and generating a sub-formula for each value, as was done in David, Logemann and Loveland's DLL algorithm [DLL62]. Resolution generates only one new formula, but in the worst case the number of clauses in the new formula will be proportional to the square of the number of the clauses in the original one. Except some peculiar case, the DLL procedure based on *splitting* is faster than its ancestor DP. As reported by [Fre95] the DP variable elimination rule has four key disadvantages:

1. it is more difficult to implement than the splitting rule;

2. it tends to rapidly increase the number and the length of the clauses;
3. it tends to generate a lot of duplicate or subsumed clauses;
4. and it very rarely generates new unit clauses.

The DLL splitting rule also makes it easier to construct a certificate of satisfiability, i.e., a satisfying truth assignment, although DP's variable elimination rule makes it easier to construct a certificate of unsatisfiability, i.e., a resolution refutation.

Ordered Binary Decision Diagrams take a different approach, since they directly encode all the solutions of the propositional formula [Bry92]. Although this may lead to an exponential blow up in the size of the diagram, many problems have been successfully tackled with OBDDs (that, we recall, handle the more expressive language of Quantified Boolean logic). The comparison between OBDDs and DLL presented in [US94] shows no clear winner, but recent results in the field of formal verification (e.g., [BCC+99], [BCRZ99], [WBCG00] and [CFG+01]) show that DLL based solvers might have an edge over OBDDs also in their traditional field of application. These results are also confirmed by recent SAT competitions where the most efficient solvers are based on the DLL algorithm.

Stålmarck's method is at the heart of the commercial tool PROVER, and it has been developed since the beginning to tackle formal verification problems in large system designs. As outlined in [SS98], the strength of PROVER seems to rely on the particular inference method, but most of PROVER strength probably comes from a very effective implementation. The algorithm presented in [GW00] is a close sibling of the Stålmarck's method and can be characterized as a breadth-first back-tracking algorithm. In the preface of [vM00] the editors conclude that, because of its features, HeerHugo (a SAT solver based on this method) provides an approach that is complementary (not a substitute) to DLL algorithms.

The resolution method lies at the root of DLL ancestor, the Davis-Putnam algorithm [DP60], but resolution based theorem provers (like SPASS [WGR96]) are generally targeted to first-order rather than propositional logics. Although they can deal with propositional problems and the DLL algorithm itself is a sibling of the resolution method, it is widely accepted that DLL-based implementations outrank resolution-based theorem provers on pure propositional satisfiability problems [Zha97].

2.3 The Davis-Logemann-Loveland (DLL) algorithm

The basic version of DLL, shown in Figure 2.1, performs a depth-first search through the space of all possible truth assignments until it either finds a satisfying truth assignment or explores the entire search space without finding any. This function takes a propositional formula φ and a stack of assigned literals μ (also seen as a set of literals) and returns TRUE if the formula is satisfiable and FALSE otherwise. Initially, we call it with $\mu = \emptyset$. The literal l in DLL is called the *premise* and the proposition associated with l is called the *branching propo-*

sition or *branching variable*. Although SAT is technically a decision problem, most implementations of DLL return the satisfying truth assignment they find.

Before describing in detail the pseudo-code of Figure 2.1, we introduce the concepts of *look-ahead algorithms* and *heuristic choice* of premises.

A look-ahead algorithm or *constraint propagator*, is a function that takes a literal l and a formula φ and returns a simplified formula φ' such that the function runs in low-order polynomial time, φ' is satisfiable iff φ is satisfiable, and φ' is in some sense easier to satisfy than φ. For example, φ' may have more valued propositions than φ, or φ' may have fewer open propositions or clauses than φ, or φ' may stipulate some relationship between the truth values of two or more open propositions in φ. SAT search algorithms typically run one or more look-ahead algorithms at every node of the search tree to simplify the remaining problem as much as possible without making any explicit guesses. Also, heuristics (discussed below) can run one or more look-ahead algorithms to compute an estimate of how effective a particular choice will be. Note that if this is the case, the heuristics themselves can detect some simplifications to the current problem.

A search algorithm that selects premises in a predetermined order is called *static*, and one that selects them at run time is called *dynamic*. As reported by [Fre95], static algorithms are generally much slower in practice than dynamic algorithms, although some recent results in the field of formal verification, such as [Sht00] and [ABC$^+$02], revamped static heuristics, especially for very large but otherwise very structured problems. Dynamic search algorithms require a heuristic for selecting premises at each search node. For our purposes a heuristic consists of three components:

1. A *branching strategy* or *branching policy* for selecting the next branching proposition.

2. A *priority function* that takes open proposition p in φ and uses information about the occurrences of p in φ to compute a numerical priority for p in accordance with the given strategy.

3. A *branching order* for determining whether the next branching propositions should be satisfied first or falsified first.

The heuristic is a fundamental part of the DLL algorithm and it is the only font of non-determinism of the algorithm. Finding a "good" literal on which branch on is either very important (and moreover different heuristics work well on different problem domains) and "difficult". We do not report detailed proofs here, but it has been showed in [Lib00] that deciding whether a literal is optimal is both NP-hard and co-NP hard, and in PSPACE.

In the algorithm:

1. Simplify(l,φ) simplifies the formula φ under the assumption that the literal l is true. This is done by removing from φ all clauses in which l appears and by removing $\overline{l}$ from all clauses in which $\overline{l}$ appears;

2. ChooseLiteral(φ) picks a literal l in φ according to some heuristic function. We have seen before the complexity of choosing the "best" literal.

Some heuristics operate by computing Simplify(l, φ) and Simplify($\neg l, \varphi$) for each open proposition p. Consequently they can detect some simplifications

to the current problem, thereby blurring the distinction between look-ahead algorithms and heuristics. Given the definition of the Simplify(l, φ) function we can make another important definition: We say that an open literal l is a *failed literal* if Simplify(l, φ) is false.

In the rest of the thesis, we could remove φ from the calls of the two procedures above described when it is clear from the context the formula we are referring about.

We conclude this section by describing the algorithm in Figure 2.1 in detail.

- As a first step the algorithm checks whether we satisfied φ or we falsified it: In both cases the algorithm returns from the current call with a result.
- If none of the above two cases holds, if there are any unit clauses l, then φ is correspondingly simplified assigning l with the right value.
- If there are no more unit clause to simplify, then we must select a new premise with the function ChooseLiteral().
- Finally, the algorithm branches on the truth values of the premise l: The current state φ is simplified with Simplify(l) and then handed over to the recursive call of DLL.

The search space is explored with a depth-first search. In this way, the second call is done iff the first one failed to discover a satisfying truth assignment.

function DLL(φ,μ)
1 **if** $\{\} \in \varphi$ **then return** FALSE
2 **if** $\varphi = \emptyset$ **then return** TRUE
3 **if** $\{l\} \in \varphi$ **then return** DLL(Simplify(l,φ),$\mu \wedge l$)
4 $l :=$ ChooseLiteral(φ)
5 **return** DLL(Simplify(l,φ),$\mu \wedge l$) or
DLL(Simplify($\bar{l}$,φ),$\mu \wedge \bar{l}$)

Figure 2.1: The DLL algorithm.

DLL(φ,$\emptyset$) terminates and returns TRUE if φ is satisfiable and FALSE otherwise; in the former case, when line 2 is reached, μ propositionally entails φ.

There is empirical evidence in the literature (see, e.g. [LS03, LS05]) that DLL is the best among the complete algorithms for solving the SAT problem. A number of improvements to DLL have been proposed, especially on the heuristic function used in ChooseLiteral(φ), on the data structures employed, on the way unit propagation and backtracking are performed, but the basic algorithm still stands unchanged.

Lemma 1 *(Soundness and completeness of* DLL*) Let φ be a formula. Then* DLL(φ, μ) *returns* TRUE *if φ is satisfiable, and* FALSE *otherwise.*

Proof. The statement is proved in detail in the original paper [DLL62]

2.3.1 Backjumping and Learning

As we mentioned before, along the years a number of optimizations have been made to the basic DLL algorithm in order to improve its efficiency and the scope of the formulas it can deal with. Nowadays, modern SAT solver can (easily) solve problems up to a million of variables and millions of clauses.

One of the most important achievements in the SAT community was the introduction in 1997 [BS97, SS96] of optimized backtracking (i.e., look-back) techniques in order to recover from "error" (or back-end) in the propositional search tree. DLL algorithm is basically a depth-first backtrack search algorithm. This means that, once a failure is reached in the search tree, the algorithm backtrack until the closest choice point (ChooseLiteral()) and change the value assigned to the correspondent variables. But it is not infrequent (indeed, most of the time this is the case) that recently made choice are not directly responsible for the conflict, and flipping the correspondent value does not change the state of the search, and leads to explore an useless part of the search tree.

In order to possibly avoid this problem, a technique called *backjumping* was introduced. This technique needs some more informations to be stored in order to set up the machinery to implement it. Each time a literal is assigned as a unit clause, it is stored also the clause that forced the literal to be assigned as *reason* of the literal. When a dead-end is reached, there is a clause that has become empty (*empty clause*). We create a *working reason* resolving (on the "conflicting" literal, the literal that caused the contradiction) the empty clause and the reason of the "conflicting" literal. Starting from this working reason we (backward) unassigned all the units we encounter, resolving the working reason with the reason of the literal we are de-assigning, thus creating a new working reason. When we reach a choice point, we check if the associated variable is in the working reason: If it is the case, we flip the value of the correspondent literal and we go on (forward) on the search. Otherwise we continue backtracking because this choice point was not directly responsible for the conflict (and flipping its value would lead to explore a completely useless part of the search tree) until a "responsible" choice point is reached or we have backtracked until the root (and in this last case the formula is to be considered FALSE , or unsatisfiable).

Even quite powerful, backjumping does not ensure that the same mistake can not be done over and over again in the propositional search. This is because backjumping is a "local" technique. In order to avoid the possibility of making an already made error again, a new technique (highly related to backjumping) was introduced (in [BS97, SS96]): *Learning*. Learning is the ability of the SAT solver to add a clause (some of the working clause seen before) to the formula in order to avoid the repetition of the same mistakes. Obviously, learning has to be treated carefully: An exponential number of clauses can be learned and add to the initial formula, causing an exponential blow up in space. For this reason are of fundamental importance methods for bound the dimension of the clauses to be learned and policy to delete (no anymore useful) clauses, in order to maintain "polynomial" the number of added clauses at each moment (we will be discussing some of this methods in the next Section). It is interesting to note [Pro93b] that it is not obvious a-priori which is the "best" clause to be learned in order to maximize the advantage of the two techniques. A clause that can be more beneficial for the backjumping effects can be less effective for learning

and vice-versa: The point is that (as anticipated before) that backjumping is a *local* technique, while learning is a *global* technique.

Moreover, it is not even clear that backjumping and learning help in reducing the CPU time for solving a problem (they have some overheads): In general, and as a result of recent SAT Competitions [LS03, LS05], these techniques help a lot when dealing with problems arising from real-world applications (such as bounded model checking, planning, formal verification), but are useless for randomly generated problems. As long as our focus (in the systems we will describe) will be on resolving real-world problems, unless explicitly specified, our (propositional) solver(s) will be using these two techniques.

Finally, we will not explicitly introduce the two techniques in the algorithms we will see (and we saw) in this thesis: This is because it is hard to formally introduce them in the (recursive version of the) DLL algorithm (unless introducing the propositional stack explicitly with the result of greatly complicate the schema). Nevertheless, all the consideration and proofs we will see also hold using backjumping and learning; and how these techniques can be integrated into a DLL algorithm schema will be more clear when we will introduce the iterative version of the DLL algorithm in Chapter 5.

2.4 The SAT solver(s) SIM(O)

In this section we will describe the two SAT solvers (based on DLL algorithm) on which most of the research in this thesis is based on. For each solver, we will be describing its main features and applications.

2.4.1 The SAT solver SIM

SIM [1] [GMTZ01], is a library designed and implemented at DIST, University of Genova. It comprises of a series of techniques and heuristics from other state-of-the-art solvers. It compares favorably with these solvers and moreover with SIM it is possible to "create" new SAT solvers newly combining different strategies and heuristics in the library not previously done in other solvers.

The data structure is basically composed by proposition (or variables) and clauses. Each proposition p has two list of occurrences (in clauses): $posList$, where it occurs positively, and $negList$, where it occurs negated. If a variable p is assigned to TRUE , all the clauses in which it occurs positively (resp. negated) is deleted (resp. simplified), on the contrary if it is assigned to FALSE .

SIM contains a number of heuristics (described in [GMTZ01]) in order to choose the next literal to branch on. Mainly, all the heuristics rely on "static" information on the formula (the number of occurrences in (some) clauses), or [LA97, BS97] on information extracted from the formula by performing some additional reasoning (failed literal detection for examples, see Chapter 5) at each branching node.

Regarding learning, SIM implements two kinds of learning:

- *size learning*, a clause corresponding to a conflict is stored if it has no more than *Order* literals. Size learning is implemented in SATO, and is used by default with *Order*=20.

[1] `http://www.star.dist.unige.it/~sim/sim/`

- *relevance learning*, a clause corresponding to a conflict is always stored. A "forget" mechanism eliminates the learned clauses in which more than *Order* literals are open or have been assigned differently since the time they have been added to the set of input clauses. Relevance learning is implemented by RELSAT, and is used by default with *Order*=3.

SIM was the heart of a technology transfer project in cooperation with Istituto per la Ricerca Scientifica e Tecnologica (IRST-ITC) in Trento and involves the integration of SIM into the NuSMV model checker[2][CCG+02], a tool for debugging and verifying hardware designs and software protocols.

Moreover, SIM is the back-end solver for dealing with the propositional part of the problem of the academic planner CPLAN[3][CGT03], designed and implemented at DIST, and of the solver for quantifier-free first order theories MathSAT [ABC+02]

2.4.2 The SAT solver SIMO

SIMO [4] [GMT03] is the C++ and Object-oriented evolution of SIM. But it is not just a re-implementation of it. SIMO incorporates the most recent advancements in the propositional solvers. In particular:

Simplify(l) is implemented as described in [MMZ+01], Sec. 2, "Optimized BCP". The function is based on a data structure that, for each clause, keeps reference to two literals in the clause ("watched literals" in the terminology of [MMZ+01]) with the proviso that they should not be both assigned to FALSE . This data structure has several features: (i) it identifies all unit and empty clauses, (ii) it minimizes the number of accesses to clauses, and (iii) it does not need expensive updates while backtracking.

ChooseLiteral() shares many similarities with the heuristic described in [MMZ+01]. Literal scores are updated only once each N calls to HEURISTIC (N is set by default in SIMO to 255); once the scores have been updated, the literals are stored in a priority queue; at each branch, the highest open literal in the priority queue is selected. The rules to update the literal score are:

- each literal has a counter, initialized to the number of occurrences in the formula;
- when a clause is learned during the look-back process the counter associated with each literal in the clause is incremented;
- the score of a literal is updated by considering its old score (divided by a constant) plus the increment in the value of its counter since the last scoring occurred.

backjumping/learning follows the learning method described in Sec. 2 of [ZMMM01] as "1-UIP learning". The implementation of this look-back method in SIMO involves some hairy details which are beyond the scope of this thesis. Please see [ZMMM01] for a detailed explanation of 1-UIP (and others) learning methods.

[2] `http://nusmv.irst.itc.it/`
[3] `http://www.ai.dist.unige.it/drwho/Cplan/`
[4] `http://www.star.dist.unige.it/~sim/simo/`

SIMO participated to the SAT 2002 Competition [SBH02], ranking (globally) in the first five positions (out of 28 solvers).

Moreover, SIMO was integrated into Thunder/Forecast, an open architecture for verification of hardware designs developed by the Logic Validation Technologies Group at the Israel Development Center of Intel Corp. and used to debug the design of some Pentium IV components. The results obtained with Thunder/SIMO outperformed the results obtained with other traditional validation technologies implemented in Forecast.

Finally, SIMO is the heart of all the tool for more expressive formalisms that we developed during the thesis and that are presented in the subsequent Sections.

2.5 Extending the DLL algorithm to work as an enumerator

In order to set up the machinery required by the SAT-based approach, we need a modification of a DLL algorithm. Figure 2.2 shows the resulting procedure.

function DLL_ENUM(φ,μ)
1 **if** $\{\} \in \varphi$ **then return** FALSE
2 **if** $\varphi = \emptyset$ **then** Print(μ); **return** FALSE
3 **if** $\{l\} \in \varphi$ **then return** DLL_ENUM(Simplify(l,φ),$\mu \wedge l$)
4 l := ChooseLiteral(φ)
5 **return** DLL_ENUM(Simplify(l,φ),$\mu \wedge l$) or
DLL_ENUM(Simplify($\bar{l}$,φ),$\mu \wedge \bar{l}$)

Figure 2.2: DLL algorithm as enumerator.

The only difference with respect to the procedure of Figure 2.1 is when $\varphi = \emptyset$. In this case the current valuation, μ, is printed and FALSE is returned in order to force backtracking.

Lemma 2 *(DLL as an enumerator) Let φ be a propositional formula.* DLL_ENUM*(φ,$\emptyset$) prints a set of valuations which is propositionally complete for φ.*

Proof. The statement is proved in detail in [GGT00]. We do not report here the entire proof, but just a sketch. Consider that returning FALSE at line 2 after μ has been printed forces backtracking, undoing all Simplify operations done since the last time ChooseLiteral was called upon, and flipping the value of the last literal chosen by ChooseLiteral. Line 2 will be hit again next time a model is found. On the other hand, if φ is unsatisfiable, line 2 will never be reached and nothing will be printed.

The disjunction of all models printed by DLL_ENUM(φ,$\emptyset$) after its execution terminates is a DNF (Disjunctive Normal Form) of φ. Notice, that it is not required that all models are stored anywhere.

We have slightly modified the SIM and SIMO solvers in order to act as enumerators (as in the Figure 2.2). In this way the two SAT solvers (in particular SIMO)

are at the basis of all the academic tools presented in this thesis, taking into account the "propositional" part of the problem presented. The functionality of being able to enumerate all the propositional satisfying assignments of a propositional formula is the most important feature for a SAT solver used as back-end solver in applications that rely on abstraction/compilation into a SAT theory.

Part II

Safe Planning via Propositional Logic

Chapter 3

Introduction

This Chapter reviews the "Safe Planning via Propositional Logic" approach introduced in [ACGT01]. The concepts of possible, valid and safe plans, (safe) plan generation and validation are introduced.

The approach described in this Chapter is the starting point for the other approaches reported in the next Parts.

3.1 Possible and Valid Plans

We start with a set of atoms partitioned into a set of *fluent symbols* and a set of *action symbols*. Intuitively, a fluent represents a property of the world that changes over time, because of the execution of actions. A *formula* is a propositional combination of atoms. An *action* is an interpretation of the action symbols. A *state* is an interpretation of the fluent signature. A *transition* is a triple $\langle s, a, s' \rangle$ where s, s' are states and a is an action: Intuitively s is the initial state of the transition, and s' is its resulting state. Executing an action a means to execute concurrently the "elementary actions" represented by the action symbols satisfied by a.

An *action description* D is a finite set of expressions describing how actions change the state of the world, i.e., the set of possible transitions. We do not make any assumption on D, except the following:

1. the effects of actions depend on the state of the world in which they are executed.

This restriction does not allow for "non-markovian" action descriptions like the one definable in $\mathcal{ARD}$ [GL95]. Under the assumption that D is Markovian, it is possible to associate a transition diagram with each action description D. The *transition diagram* represented by D is the directed graph which has the states of D as vertexes, and which includes an edge from s to s' labeled a for every transition $\langle s, a, s' \rangle$ that is possible according to D. Notice that D may be deterministic or non-deterministic.

Besides the above assumption, we also assume that

2. it is possible to compute a propositional formula tr_i^D whose satisfying assignments correspond to the possible transitions caused by the execution of an action in D.

More in detail, we assume that in tr_i^D there is a propositional variable A_i for each ground action symbol A, and two propositional variables F_i and F_{i+1} for each ground fluent F in D: Intuitively, F_i represents the value of F in the initial state of the transition, and F_{i+1} represents the value of F in the resulting state, after having performed the action.

As in the standard planning literature, a *planning problem* for D is characterized by two formulas I and G in the fluent signature, i.e., is a triple $\langle I, D, G\rangle$, where I and G encode the initial and goal state(s) respectively. A *plan* (of *length* $n \geq 0$) is a finite sequence $a^1; \ldots; a^n$ of actions.

Consider a planning problem $\pi = \langle I, D, G\rangle$. Intuitively, to ensure that a plan $a^1; \ldots; a^n$ is valid, we have to check that

- the plan is "always executable" in any initial state, i.e., executable for any initial state and any possible outcome of the actions in the plan, and
- any "possible result" of executing the plan in any initial state is a goal state.

In order to make the above definition precise we have to give the following definitions.

A *history* for an action description D is a path in the corresponding transition diagram, that is a finite sequence

$$s^0, a^1, s^1, \ldots, a^n, s^n \tag{3.1}$$

($n \geq 0$) such that $s^0, s^1, \ldots, s^n$ are states, $a^1, \ldots, a^n$ are actions, and

$$\langle s^{i-1}, a^i, s^i\rangle \qquad (1 \leq i \leq n)$$

are transitions which are possible according to D. n is the *length* of the history (3.1).

A plan $\vec{a} = a^1; \ldots; a^n$ is *possible* for π if there exists a history (3.1) for D such that

- s^0 is an initial state, and
- s^n is a goal state.

An action a is *executable* in a state s if for some state s', $\langle s, a, s'\rangle$ is a possible transition according to D. Let s^0 be a state. The plan $a^1; \ldots; a^n$ is *always executable in* s^0 if for any history

$$s^0, a^1, s^1, \ldots, a^k, s^k$$

with $k < n$, a^{k+1} is executable in s^k. Assume that $\vec{a}$ is a plan which is always executable in a state s^0. A state s^n is a *possible result of executing* $\vec{a}$ *in* s^0 if there exists a history (3.1) for D.

A plan $\vec{a} = a^1; \ldots; a^n$ is *valid* for π if for any initial state s^0,

- $\vec{a}$ is always executable in s^0, and
- any possible result of executing $\vec{a}$ in s^0 is a goal state.

Indeed, if D is deterministic, any possible plan is also valid. However, this is not the case if D is nondeterministic. A possible plan *may* lead to the goal; while valid plans are ensured to reach a goal state despite the potential multiple initial states and nondeterminism in the action description. In the planning literature, valid plans are also called "conformant" [SW98].

3.1.1 Safe Plans

Consider a planning problem $\pi = \langle I, D, G \rangle$. In the previous section we have formally defined the notion of valid plan as a sequence of actions which is guaranteed to reach a goal state. Depending on the characteristics of π (e.g., depending on the language used to describe D) different planning procedures can be used. However, as we have already argued in the introduction, we do not only want that a plan is "valid", but also "safe" with respect to a set of user-defined properties. For example, we do not want a plan which can go through a state which is potentially dangerous for humans.

In the FV literature, these (safety) properties are nicely formalized by means of temporal logics. Here we consider a simple temporal logic, whose syntax and semantics resembles Pnueli's Linear Temporal Logic (LTL) [Pnu77]. The definition of temporal formula (used for the specification of the properties) is the following:

- A fluent symbol or the negation of a fluent symbol is a *temporal formula*, and
- If α and β are temporal formulas, also

$$\begin{array}{cccc} (\alpha \wedge \beta), & (\alpha \vee \beta), & \mathcal{X}_s \alpha, & \mathcal{X}_w \alpha, \\ \mathcal{F}\alpha, & \mathcal{G}\alpha, & (\alpha \mathcal{U} \beta), & (\alpha \mathcal{R} \beta), \end{array}$$

  are temporal formulas.

Notice that we have assumed temporal formulas to be in negation normal form, i.e., that negations occur only in front of fluent symbols. This is not a limitation given that —according to the semantics defined below— (assuming we relax for a moment the assumption to be in negation normal form)

$$\begin{array}{c} \neg \mathcal{X}_s \alpha \equiv \mathcal{X}_w \neg \alpha, \\ \neg \mathcal{F} \alpha \equiv \mathcal{G} \neg \alpha, \\ \neg (\alpha \mathcal{U} \beta) \equiv (\neg \alpha \mathcal{R} \neg \beta). \end{array} \tag{3.2}$$

Some of the above equivalences hold in standard LTL. However, the meaning of the temporal connectives $\mathcal{X}_w, \mathcal{F}, \mathcal{G}, \mathcal{R}$ have to be adjusted in order to accommodate the fact that we are dealing with a finite time line (see [Eme90], pag. 1006). Intuitively, if n represent our horizon,

- $\mathcal{X}_s \alpha$ reads "there exists a successor moment and α holds there",
- $\mathcal{X}_w \alpha$ reads "if there exists a successor moment then α holds there",
- $\mathcal{F}\alpha$ reads "for some subsequent time $\leq n$ α holds",
- $\mathcal{G}\alpha$ reads "for all subsequent times $\leq n$ α holds",
- $\alpha \mathcal{U} \beta$ reads "for some subsequent time $\leq n$ β holds, and α holds until then",
- $\alpha \mathcal{R} \beta$ reads "if for some subsequent time $\leq n$ β does not hold, then α holds in a state before then".

According to the above intuitive meanings, the following equivalences hold:

$$\mathcal{F}\alpha \equiv \top\mathcal{U}\alpha, \qquad \mathcal{G}\alpha \equiv \bot\mathcal{R}\alpha, \tag{3.3}$$

where $\top, \bot$ are the symbols for truth and falsity respectively.

Let h be a history (3.1), and α a temporal formula. We say that h *satisfies* α *($h \models^n \alpha$)* iff $h \models_0^n \alpha$, where, for any $i \leq n$, the definition of $h \models_i^n \alpha$ is given inductively on the structure of the formulas:

1. $h \models_i^n \alpha$ if $s^i \models \alpha$ and α is a fluent literal,
2. $h \models_i^n (\alpha \wedge \beta)$ if $h \models_i^n \alpha$ and $h \models_i^n \beta$,
3. $h \models_i^n (\alpha \vee \beta)$ if $h \models_i^n \alpha$ or $h \models_i^n \beta$,
4. $h \models_i^n \mathcal{X}_s\alpha$ if $i < n$ and $h \models_{i+1}^n \alpha$,
5. $h \models_i^n \mathcal{X}_w\alpha$ if $i = n$ or $h \models_{i+1}^n \alpha$,
6. $h \models_i^n \mathcal{F}\alpha$ if $\exists j$: $i \leq j \leq n$, $h \models_j^n \alpha$,
7. $h \models_i^n \mathcal{G}\alpha$ if $\forall j$: $i \leq j \leq n$, $h \models_j^n \alpha$,
8. $h \models_i^n \alpha\mathcal{U}\beta$ if $\exists j$: $i \leq j \leq n$, $h \models_j^n \beta$ and $\forall k$: $i \leq k < j$, $h \models_k^n \alpha$,
9. $h \models_i^n \alpha\mathcal{R}\beta$ if $\forall j$: $i \leq j \leq n$, $h \models_j^n \beta$ or $\exists k$: $i \leq k < j$, $h \models_k^n \alpha$.

Given the above semantics, it is easy to see that both (3.2) and (3.3) hold.

Temporal logic allows us to specify different interesting requirements on plans. The user can specify a safety property with the formula $\mathcal{G}\alpha$, intuitively standing for "the property α should be maintained", or $\mathcal{G}\neg\beta$, meaning for instance that "a dangerous or undesired situation β should be avoided". The formula $\mathcal{G}\mathcal{F}\alpha$ can be used to specify that "always a state where α holds should be eventually reached", while $\mathcal{F}\mathcal{G}\beta$ states that "the system should get to the point where property β can be maintained". As further examples, formulas can be composed in order to specify a sequential ordering in which properties should be verified: $\mathcal{F}(\alpha \wedge \mathcal{F}\beta)$ states that "α should be eventually true before β", while $\mathcal{F}(\alpha \wedge \mathcal{F}\mathcal{G}\beta)$ can be used to specify that "α should be reached first, and then β should be reached and maintained".

3.1.2 Safe Planning: Plan Validation

We now consider the problem of plan validation with respect to a finite set of properties. Let α be the conjunction of the properties, let $\pi = \langle I, D, G\rangle$ a planning problem, and let $\vec{a} = a^1; \ldots; a^n$ be a valid plan.

We say that $\vec{a}$ is *safe* with respect to α if each history (3.1) satisfies α. The problem of validating $\vec{a}$ with respect to α can be re-cast as a satisfiability problem. The basic idea is to check whether the execution of the actions in $\vec{a}$ "entails" the property α, reformulated as a propositional formula. In the following, for any number i and formula H, H_i is the expression obtained from H by substituting each atom B with B_i. Intuitively, the subscript i represents time:

- If F is a fluent symbol, the atom F_i expresses that F holds at time i.

- If A is an action symbol, the atom A_i expresses that A is among the elementary actions executed at time i.

The reformulation $[\alpha]^n$ of α as a propositional formula is defined as $[\alpha]^n_0$, and $[\alpha]^n_i$ is:

1. $[\alpha]^n_i$ is α_i, if α is a fluent literal,
2. $[\alpha \wedge \beta]^n_i$ is $[\alpha]^n_i \wedge [\beta]^n_i$,
3. $[\alpha \vee \beta]^n_i$ is $[\alpha]^n_i \vee [\beta]^n_i$,
4. $[\mathcal{X}_s\alpha]^n_i$ is $\perp$ if $i = n$, and is $[\alpha]^n_{i+1}$ otherwise,
5. $[\mathcal{X}_w\alpha]^n_i$ is $\top$ if $i = n$, and is $[\alpha]^n_{i+1}$ otherwise,
6. $[\mathcal{F}\alpha]^n_i$ is $\vee^n_{j=i}[\alpha]^n_j$,
7. $[\mathcal{G}\alpha]^n_i$ is $\wedge^n_{j=i}[\alpha]^n_j$,
8. $[\alpha\mathcal{U}\beta]^n_i$ is $\vee^n_{j=i}([\beta]^n_j \wedge^{j-1}_{k=i} [\alpha]^n_k)$,
9. $[\alpha\mathcal{R}\beta]^n_i$ is $\wedge^n_{j=i}([\beta]^n_j \vee^{j-1}_{k=i} [\alpha]^n_k)$.

For any formula, the correspondence between its semantics and its translation as a propositional formula is immediate. Given the above mapping, and under the assumption that $\vec{a}$ is valid for π, the check that $\vec{a}$ is safe with respect to α amounts to check that

$$I_0 \wedge \wedge^{n-1}_{i=0} A_i \wedge \wedge^{n-1}_{i=0} tr^D_i \models [\alpha]^n. \tag{3.4}$$

For example, if α is a safety property $\mathcal{G}\beta$, (3.4) holds if for any history (3.1) in which s^0 is an initial state, s^i satisfies β for any $i : 0 \leq i \leq n$.

3.1.3 Safe Planning: Plan Generation

Consider a finite set of properties, Let α be their conjunction, and let $\pi = \langle I, D, G\rangle$ a planning problem. Safe planning is the task of finding a valid plan which is also safe with respect to α. As before, we do not make any assumption about D: it can be deterministic or nondeterministic, it can allow for the concurrent execution of actions or not. However, for simplicity we assume that I is satisfied by at most one state.

The simplest approach for the generation of safe plans, is to generate (possible) plans, and test whether each generated plan is valid and safe. In this way, we obtain

1. a correct but possibly incomplete safe procedure in general, and
2. a correct and complete safe procedure if we generate and test all the possible plans.

Given a planning problem π and a natural number n, we say that a procedure for safe planning is

- *correct* (for π, n) if any returned plan $a_1; \ldots; a_n$ is valid and safe for π, and

- *complete* (for π, n) if it returns *False* when there is no valid and safe plan $a_1; \ldots; a_n$ for π.

In the planning as satisfiability framework, all the possible plans can be generated by enumerating the assignments satisfying

$$I_0 \wedge \wedge_{i=0}^{n-1} tr_i^D \wedge G_n. \tag{3.5}$$

The check whether a plan $\vec{a} = a^1, \ldots, a^n$ is safe —assuming we already know is valid— can be done as described in Section 3.1.2. Indeed, if I is satisfied by only one state (as we assumed at the beginning of the Section) and D is deterministic, any possible plan is also valid, and thus we are done: we can easily devise a procedure that generates valid plans till one which is also safe is found.

However, as described in [Giu00], the check whether $\vec{a}$ is valid is complicated in the case that D is non deterministic. In fact, the check that a possible plan is also valid amounts to perform an entailment check similar to the one in (3.4). More in detail, a plan $a^1; \ldots; a^n$ is valid iff

$$I_0 \wedge \neg Z_0 \wedge \wedge_{i=0}^{n-1} A_i \wedge \wedge_{i=0}^{n-1} trt_i^D \models G_n \wedge \neg Z_n, \tag{3.6}$$

where trt_i^D is defined on the basis of tr_i^D, and Z is a newly introduced fluent symbol, see [Giu00]. Thus, given what we said so far, in order to have a correct and complete procedure for safe planning we should:

- generate (all) the possible plans by satisfying (3.5),
- test if each generated plan $a^1; \ldots; a^n$ is valid by checking whether (3.6) holds, and
- if $a^1; \ldots; a^n$ is valid, check whether it is also safe with respect to a property α, by checking whether (3.4) holds.

This is not necessary. Indeed, the last two checks can be combined into one: A (possible) plan $a^1; \ldots; a^n$ is both valid for π and safe with respect to α iff

$$I_0 \wedge \neg Z_0 \wedge \wedge_{i=0}^{n-1} A_i \wedge \wedge_{i=0}^{n-1} trt_i^D \models G_n \wedge \neg Z_n \wedge [\alpha]^n.$$

Furthermore, as an optimization, if in the generation phase we consider only the plans corresponding to the assignments satisfying

$$I_0 \wedge \wedge_{i=0}^{n-1} tr_i^D \wedge G_n \wedge [\alpha]^n \tag{3.7}$$

we still get a correct and complete procedure for safe planning: The assignment satisfying (3.5) and not (3.7) for sure do not correspond to safe plans.

3.2 Summing up

In this Chapter we have presented the Safe Planning approach via a translation to a propositional formula. The formula are not restricted to be in CNF, the form used by most of the propositional solvers. Nonetheless, it is well known that there exist efficient and linear-time (in the size of the formula) conversion methods to a CNF formula [Tse70, PG86b].

To be noticed that in the rest of this Part, all the techniques and optimizations we will see are based on (but not limited to) the CNF representation of the formulas.

As long as the approach has been previously invented by others, we focus our attention on optimizing the DLL computation (*i*) using "independent" variables to guide the search, and (*ii*) introducing a new method based on oracles to predict if a technique (when implemented efficiently) can help in reducing the time needed to solve a propositional formula.

Chapter 4

Dependent and Independent Variables

In this Chapter, we investigate the "independent variable selection (IVS)" heuristic, i.e., given an input formula on the set of variables N, the selection is restricted to a – possibly small – subset S which is sufficient to determine a truth value for all the variables in N. During the search phase, scoring and selection of the literal to assign next in ChooseLiteral() are restricted to S, and the truth values for the remaining variables are determined by the pruning techniques of the solver. The completeness of the solver is maintained because all the variables are assigned once those in S are. Moreover, the worst-case size of the search space goes down from $2^{|N|}$ to $2^{|S|}$. IVS has been proposed and used several times in combination with different scoring and selection techniques. The results have been mixed, but most of the times big benefits have been reported. In particular, in [CB94] the IVS heuristic has been first proposed and applied to scheduling problems: no benefits are reported on these instances; in [GMS98] IVS has been applied to planning problems generated with MEDIC [EMW97]: improvements reach 4 orders of magnitude; in [Sht00] and [CFG+01] IVS is experimented in the context of formal verification: the authors report significant improvements on most instances. On the implementation side, the SAT solvers SIM and SIMO both incorporate the IVS heuristic along with a variety of underlying scoring and selection techniques.

We discuss the possible advantages and disadvantages of using the IVS heuristic.

4.1 Variable Dependency

The idea behind variable dependency in DLL is simple. A variable x is *dependent* on a set of independent variables $x_1, \ldots, x_n$ if for any assignment to $x_1, \ldots, x_n$ the value of x becomes determined by LOOK-AHEAD.[1] Thus, the notion of variable dependency is defined with respect to a given set of variables

[1] With LOOK-AHEAD we refer, at least, to the repeated application of Simplify() on the unit clauses encountered, also called BCP (Boolean Constraint Propagation). Nonetheless, LOOK-AHEAD can perform additional reasoning as we will see in the reminder of this Chapter.

and a specific LOOK-AHEAD procedure. For example, in the formula:

$$\{\{\neg l \vee l_1 \vee \ldots l_n\}, \{\neg l_1 \vee l\}, \ldots, \{\neg l_n \vee l\}\}$$

where $l, l_1, \ldots, l_n$ are distinct literals, we can say that $|l|$ depends on $|l_1|, \ldots, |l_n|$ if LOOK-AHEAD enforces BCP. Our working hypotheses for the remainder of the paper are that LOOK-AHEAD is limited to BCP as in Figure 2.1, and that the set of independent variables is known a priori in all the problems that we consider.

4.2 DLL and Variable Dependency

If S is a set of independent variables, it is rather easy to modify CHOOSE-LITERAL to enforce IVS, i.e., to restrict scoring and selection to S. Now the question is: what can we expect by using IVS? For one thing, the unrestricted CHOOSE-LITERAL can simulate the restricted one, but the converse is not true. If the input formula φ is satisfiable, this is not necessarily a problem: they are both allowed to pick a set of literals satisfying φ. On the other hand, if the input formula is unsatisfiable, we may witness a substantial degradation of DLL performances when using the IVS heuristic. For example, let φ be an unsatisfiable formula on the set of variables S. Assume that refuting φ with DLL requires a search tree of exponential size (the existence of such formulas is discussed, e.g., in [Urq95]). Now let ψ be the following formula on the set of variables $N = S \cup \{a, b, c\}$:

$$(\varphi \leftrightarrow a) \wedge (a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (b \leftrightarrow \neg c).$$

Clearly, ψ itself is unsatisfiable and the variables a, b, c are dependent on the set of variables S. If we try to have DLL refute ψ by enabling the IVS heuristic, then we are bound to explore a search tree of exponential size. At the end of each branch we determine a truth value for a, and then backtrack occurs because of an inconsistency found by running LOOK-AHEAD. On the other hand, it is sufficient for an unrestricted CHOOSE-LITERAL to pick any of the variables in the set $\{a, b, c\}$ in order to quickly refute ψ.

Statistic	**DES**	**Parity**	**BMC**	**Pretolani**	**Planning**
Min	0.01	0.01	0.05	0.35	0.67
$\mathbf{Q_1}$	0.01	0.02	0.10	0.34	0.68
$\mathbf{Q_2}$	0.03	0.03	0.18	0.37	0.70
$\mathbf{Q_3}$	0.06	0.05	0.24	0.37	0.71
Max	0.18	0.13	0.40	0.46	0.74

Table 4.1: Statistics on the ratio $|S|/|N|$.

Notwithstanding the above considerations that may suggest to abandon the idea of restricting CHOOSE-LITERAL, in practice it is easy to see how exponential improvements can be obtained with the IVS heuristic. For example, given a formula φ on a set S of variables, we can construct a super-formula ψ of φ that introduces denumerately many new variables, all depending on S. If we

introduce exponentially many new such variables and try to solve ψ, the first call to an unrestricted CHOOSE-LITERAL is enough to cause a substantial worsening with respect to a restricted one.

4.3 Experimental results

4.3.1 Test set

The test set used to evaluate the IVS heuristic consists of 157 CNF formulas, 102 satisfiable and 55 unsatisfiable, yielding an overall 64% chance of a satisfiable instance. Some of these instances are also known to be challenging for currently available SAT solvers, but selecting hard instances was not our focus. In assembling the test set, we privileged problems that have already been used to compare SAT solvers; we tried to maximize the number of different problem classes, and the number of problems in each class. Our choice was constrained by the availability of instances for which we are able to determine a set of independent variables. In particular, we chose:

- 32 Data Encryption Standard (DES) problems, see [MM00, Li00];
- the "famous" 30 parity problems, see, e.g., [SKM97];
- 34 instances of bounded model checking (BMC), see [BCCZ99, Li00];
- 24 Pretolani problems, see, e.g., [Li00].
- 37 planning problems generated by BlackBox [KS98];

For the sake of our analysis, the most representative parameter is the $|S|/|N|$ ratio, i.e., the fraction of the set of total variables N which is also in the set S of independent variables.

Table 4.1 gives an idea of how the ratio $|S|/|N|$ is distributed among the instances in each of the above problem classes. We remind that the p%-percentile is the value x such that p% of the observed data is smaller than x. In Table 4.1: Q_1 denotes the 25%-percentile, Q_2 the 50%-percentile, and Q_3 the 75%-percentile of the observed ratios; "Min" and "Max" denote, respectively, the minimum and the maximum of the observed ratios. This set of statistics is known as the "five-number summary" and is very useful for comparing distributions [MM93]. From Table 4.1 we can see that the parameter $|S|/|N|$ is distributed quite differently across the problem classes considered. For instance, while $|S|/|N|$ is almost normally distributed around 0.18 from a minimum of 0.05 to a maximum of 0.40 in BMC problems, in DES and Parity instances the distribution is biased toward small values of $|S|/|N|$ since the 75% of the data is smaller than 0.06 for DES and 0.05 for Parity problems, but the maximum ratio is 0.18 for DES and 0.13 for Parity problems.

4.3.2 DLL implementation

The SAT solvers SIM and SIMO support the IVS heuristic. Moreover, SIM comes with different implementations of CHOOSE-LITERAL that can be selected from the command line. Using compile-time options, it is also possible to augment SIM with backjumping and learning. Therefore, SIM provided the ideal platform for

our purposes. In our experiments, we run SIM with 5 different implementations of CHOOSE-LITERAL, with and without backjumping and learning. For each version of CHOOSE-LITERAL we run the default implementation as well as the one with the IVS heuristic, for a total of 20 different configurations tested. Unless explicitly mentioned, the version of SIM used to produce the data is the one with chronological backtracking. The implementations of CHOOSE-LITERAL that we evaluated are:

- MOMS (M), introduced in [Pre93], which prefers variables that occur frequently in the shortest clauses;
- Jeroslow-Wang (JW), see [JW90], where the occurrences of variables in short clauses are exponentially better than those in long clauses;
- Böhm (B), discussed in [BB92], which considers occurrences in clauses of any length and, in case of ties, prefers variables occurring frequently in short clauses;
- SATZ (S), as explained in [LA97], which features a complex scoring mechanism based on BCP and a modified version of JW;
- Unitie0 (U0), introduced by [CFG+01] under the name "Unit", which prefers variables producing the highest simplification with BCP.

In all the cases above, using the IVS heuristic amounts to restricting the scoring and selection process to the variables in S. To distinguish between the unrestricted CHOOSE-LITERAL and the one with the IVS heuristic, we use "*" as a suffix for the latter. For instance, "M*" means MOMS scoring with IVS enabled.

4.3.3 Effects of the IVS Heuristic

Tables 4.2 to 4.6 report the results of our experimental analysis using the instances and the configurations of SIM described in the previous sections. The ordering of the tables is in accordance with the value of Q_2 for each problem class, from the lowest (DES instances) to the highest (Planning instances). Indeed, considering the discussion in Section 4.2, we expected that using the IVS heuristic would produce greater benefits for problems with small values of $|S|/|N|$.

Each line in the tables reports the following data:

- the configuration of SIM, e.g., "M" and "M*" for MOMS;
- the cumulative sum $C(t)$ of the problems solved by the configuration within time t, for $t = \{1, 2, 4, \ldots, T_f\}$; T_f is 1200 seconds;
- the root mean square (RMS) of $C(t)$ calculated as:

$$\mathrm{RMS}[C(t)] = \sqrt{\frac{\sum_{s=1}^{T_f/T} C(sT)^2}{T_f/T}}$$

  where T is a sampling constant that divides the range $[0; T_f]$ into intervals of equal length; in our calculations $T = 12$.

The value RMS[$C(t)$] summarizes the performances of a given configuration, and enables the comparison among different configurations. Intuitively, it represents the weighted distance of $C(t)$ from the "worst possible configuration" $C_0(t) = 0$, i.e., the configuration which cannot solve any problem within T_f. The use of RMS[$C(t)$] privileges the configurations that converge quickly and solve many problems. RMS[$C(t)$] is going to be small for configurations that either converge slowly or saturate at a small number of problems. This captures precisely our intuition of a "bad" behavior of a configuration on a problem class.[2]

Heur	**RMS**	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	$\mathbf{T_f}$
M	15.81	3	3	5	6	10	13	14	16	16	16	16
M*	15.90	3	4	5	9	12	14	15	16	16	16	16
B	15.83	3	4	5	6	9	13	15	16	16	16	16
B*	15.91	4	5	5	8	12	15	16	16	16	16	16
JW	15.66	2	2	3	4	4	8	13	15	16	16	16
JW*	15.83	2	2	4	4	7	13	14	16	16	16	16
S	16.23	16	16	16	16	16	16	16	16	16	16	17
S*	16.22	16	16	16	16	16	16	16	16	16	16	17
U0	23.66	16	18	21	22	22	22	22	22	23	24	24
U0*	16.00	16	16	16	16	16	16	16	16	16	16	16

Table 4.2: Data Encryption Standard, 32 problems, all satisfiable.

In Table 4.2 the data about DES problems is reported. Notice that none of the configurations is able to solve all the problems in the class. Looking at the statistics for $|S|/|N|$ in Table 4.1, we see small ratios for these instances, always less than 0.18. Nevertheless, introducing the IVS heuristic does not help in speeding up the convergence of any configuration, modulo some minor improvements on MOMS, Böhm and Jeroslow-Wang. The null impact on SATZ, is contrasted by a sharp worsening of the Unitie0 heuristic: "U0*" solves 1/3 less problems than the unrestricted "U0" configuration.

In Table 4.3 we report the data about Parity problems. As for the DES class, none of the configurations is able to solve all the problems in the class. This is to be expected, since these problems have been shown to be very challenging for the current state-of-the-art SAT solver. On the other hand, the statistics for the $|S|/|N|$ ratio in Table 4.1 tell us that the percentage of independent variables is quite small (13% at most). Indeed, introducing the IVS heuristic helps a little in speeding up the convergence of some configurations, namely MOMS, Böhm, Unitie0, but it has a negative impact on Jeroslow-Wang and a null impact on SATZ configurations. Moreover, the IVS heuristic does not help in solving more problems (see the T_f column). Parity problems are thus a clear example of how the IVS heuristic can be successful for some configurations and unsuccessful for others.

In Table 4.4 the data about BMC problems is reported. In this problem class (see Table 4.1) the median $|S|/|N|$ ratio is 0.18, and it is higher than the maximum ratio observed on DES and Parity problems. Nevertheless, with the

[2] All the experiments run on a network of identical workstations equipped with PentiumIII 600MHz, 128MB of RAM, and `Linux Suse ver. 6.2`. SIM is compiled with `gcc 2.95.2`; time is measured in CPU seconds.

Heur	RMS	1	2	4	8	16	32	64	128	256	512	T_f
M	19.99	10	10	12	17	19	20	20	20	20	20	20
M*	20.00	12	13	15	19	20	20	20	20	20	20	20
B	19.99	11	13	13	16	20	20	20	20	20	20	20
B*	20.00	12	15	20	20	20	20	20	20	20	20	20
JW	19.99	13	16	19	20	20	20	20	20	20	20	20
JW*	20.00	10	13	15	17	18	20	20	20	20	20	20
S	19.99	12	15	16	18	20	20	20	20	20	20	20
S*	19.99	12	15	16	18	20	20	20	20	20	20	20
U0	19.63	11	11	12	12	14	16	18	19	19	20	20
U0*	20.00	14	17	19	20	20	20	20	20	20	20	20

Table 4.3: Parity, 30 problems, all satisfiable.

Heur	RMS	1	2	4	8	16	32	64	128	256	512	T_f
M	26.11	9	11	12	13	16	16	18	19	21	26	30
M*	29.71	11	12	15	16	18	19	20	24	26	32	32
B	25.70	8	10	12	13	16	16	18	19	21	26	30
B*	29.67	11	12	14	16	18	19	20	24	26	32	32
JW	17.10	9	9	10	11	13	13	14	15	17	17	18
JW*	28.58	11	14	16	17	18	20	21	23	25	28	32
S	21.22	10	12	12	14	14	16	17	17	20	21	24
S*	21.20	10	12	12	14	14	16	17	17	20	21	24
U0	17.95	9	9	10	12	12	13	14	15	17	18	20
U0*	21.96	11	12	13	13	14	16	18	18	20	22	25

Table 4.4: Bounded Model Checking, 34 problems, all unsatisfiable.

only exception of SATZ scoring, introducing the IVS heuristic helps in solving more problems (see the T_f column), and also in speeding up convergence (see the RMS column). The data is in accordance with the good results reported on similar problems in the literature (see, e.g., [Sht00] and [CFG+01]).

Table 4.5 reports about Pretolani instances. On this class we see again some improvement brought by the IVS heuristic. The distribution of $|S|/|N|$ is concentrated in the interval $[0.35; 0.46]$ with a median of 0.37. Notwithstanding the fairly big number of variables left to examine, in at least one configuration, namely Böhm, introducing the IVS heuristic enables to solve more problems within T_f. In some configurations, namely MOMS, Jeroslow-Wang and Unitie0, the convergence of the configurations featuring IVS is quicker than their unrestricted counterparts. On the other hand, SATZ performances worsen with the introduction of IVS. Indeed, SATZ with IVS does not examine all the variables and this amounts to a loss of accuracy in the selection of the next variable. Such loss is not compensated by the speed gain obtained by examining an approximate 40% of the variables only.

We conclude our report with the data about Planning problems in Table 4.6. From the table we can immediately grasp the inadequacy of non-BCP based heuristics on this problem class. MOMS, Böhm and Jeroslow-Wang fail to be

Heur	RMS	1	2	4	8	16	32	64	128	256	512	T_f
M	8.68	4	5	5	5	7	7	8	8	8	9	9
M*	8.76	6	8	8	8	8	8	8	8	8	9	9
B	8.65	3	3	3	3	4	5	6	6	6	6	6
B*	14.75	11	12	12	12	12	12	12	12	12	12	12
JW	14.45	11	13	14	14	14	14	14	14	14	14	15
JW*	14.55	13	14	14	14	14	14	14	14	14	14	15
S	12.55	8	10	10	11	11	11	11	11	12	13	13
S*	9.15	7	7	7	8	8	8	8	8	9	9	10
U0	14.22	10	11	13	14	14	14	14	14	14	14	15
U0*	14.66	12	13	14	14	14	14	14	14	14	15	15

Table 4.5: Pretolani, 24 problems, 12 satisfiable, 12 unsatisfiable.

Heur	RMS	1	2	4	8	16	32	64	128	256	512	T_f
M	5.63	0	2	3	4	4	5	5	5	5	5	7
M*	7.80	0	2	5	5	5	6	6	6	6	7	9
B	2.04	0	0	0	0	0	0	2	2	2	2	3
B*	2.33	0	0	0	0	0	0	2	2	2	2	3
JW	0.97	0	0	0	0	0	0	0	1	1	1	1
JW*	0.97	0	0	0	0	0	0	0	1	1	1	1
S	27.99	10	11	13	17	19	20	21	24	27	28	31
S*	27.99	10	11	13	17	19	20	21	24	27	28	31
U0	27.79	7	10	10	19	22	23	24	25	26	28	30
U0*	26.92	5	11	16	20	21	21	22	22	24	27	30

Table 4.6: Planning (BlackBox), 37 problems, 28 satisfiable, 9 unsatisfiable.

really competitive, and the improvements introduced by the IVS heuristic are quite marginal, with the only exception of MOMS. SATZ and Unitie0 configurations are much more performant, but again the IVS heuristic is not able to really improve them in any sense.

In Table 4.7 we summarize the results of our experiments in a compact way, with the following notation. Each line corresponds to a configuration, and the comparison is between the unrestricted version and the one featuring the IVS heuristic. There are two blocks of results in the table. The upper block summarizes the results obtained using SIM with chronological backtracking, while the data in the lower block is obtained using SIM augmented with backjumping and learning. Each column corresponds to a problem class. To compute the data in the Table, we use the RMS values for each configuration. Given $C(t)$ and $R = \text{RMS}[C(t)]$ for an unrestricted configuration, we denote with $C^*(t)$ and $R^* = \text{RMS}[C^*(t)]$ the corresponding parameters for the same configuration with the IVS heuristic. With these definitions in mind, a "+"(resp., "−") in Table 4.7 means that $R^* > R$ (resp. $R^* < R$), and a "=" means that $R^* = R$. To quantify how much the RMS changes going from $C(t)$ to $C^*(t)$, let P be the

Heur	DES	Parity	BMC	Pretolani	Planning
M	+	+	+ +	+	+
B	+	+	+ +	+ + +	+
JW	+	+	+ + +	+	=
S	−	=	−	−−	=
U0	− − −	+	+ +	+	−
M	+	+	+	−−	+
B	+	+	+	−	+
JW	+	−	++	−−	+
S	−	=	−	−	=
U0	−	+	++	−	+

Table 4.7: Evaluation of the IVS heuristic.

total number of problems in the class and r be the ratio

$$r = \frac{|R^* - R|}{P}$$

In Table 4.7, given $\odot \in \{+, -\}$, a single "$\odot$" means that $r \leq 0.1$, a "$\odot\odot$" means that $0.1 < r \leq 0.2$, and a "$\odot\odot\odot$" means that $r > 0.2$. For instance, the "+ +" corresponding to BMC and U0 in Table 4.7 means that U0* is 10% to 20% better than U0. The upper block of Table 4.7 evidences that both the problem class and the underlying scoring technique determine the behavior of IVS heuristic, and the lower block of Table 4.7 evidences that also the backtracking scheme should be included among the factors that make up for IVS effectiveness.

4.4 Summing up

Overall, on the basis of the results in Table 4.7, we can conclude that whether IVS improves performances or not strictly depends on the type of problems considered, on the underlying scoring and selection technique, and also on the backtracking scheme. Still, according to the data in Table 4.7, IVS introduces some benefits on average: in the Table there are 40 "+", and 19 "−".

Determining whether the heuristic will improve the performances or not is far from being obvious. The experimental analysis that we presented confirms this intuition. Nevertheless, the potential of the heuristic weighted against its simplicity is fairly big, and this suggests that, whenever possible, IVS should be included in the experimental assessment of SAT instances.

Chapter 5

Testing the efficiency of improvements

As already mentioned, in the last years, we have seen a tremendous boost in the performances of SAT solvers, such boost mostly due to Chaff [MMZ+01]. Chaff owes its efficiency to four components: (*i*) efficient data structures, fitted to the look-ahead and look-back methods implemented, (*ii*) an innovative look-back method, (*iii*) an innovative heuristic, based on the information extracted from the look-back phase, and (*iv*) low-level optimizations of the code. zChaff (Chaff latest incarnation) was the best among the complete SAT solvers on industrial benchmarks in the last SAT competitions [SBH02, LS05]. Further, all the other top-performers in these two categories (with the exception of 2CLS+EQ [Bac02]) were "Chaff-like" SAT solvers, as they incorporated some/most of the ideas behind Chaff. In this paper, when we speak of modern SAT solvers, we have in mind a "Chaff-like" engine. It is worth pointing out that the only look-ahead method implemented by most modern SAT solvers is still the simple, yet effective, unit clause propagation method, but they all feature far more sophisticated heuristics and look-back methods.

In this Chapter we investigate the effect of adding a failed literal detection method to the traditional unit clause propagation method in the look-ahead component of a modern SAT solver. Failed literal detection was first introduced in POSIT [Fre95], used extensively, e.g., in SATZ [LA97] and RelSAT [BS97], and similar techniques led to positive results on real-world instances in [Bac02]. Our analysis is mostly experimental and has been performed using our solver SIMO modified to incorporate ideas (*i*) - (*iii*) above. With these modifications SIMO is within a factor of 3 slower than zChaff, the remaining gap due to the absence of low-level code optimizations. We have run our experiments using several challenging real-world benchmarks. Such instances are either well-known in the literature or submitted to the SAT 2002 competition. On the basis of the collected data, we anticipate that:

- enhanced look-ahead based on failed literal detection does not pay off in modern SAT solvers;
- even assuming we had an oracle answering whether a literal will fail (thus saving the time necessary to try the literals which will not fail), enhanced

```
DLL-SOLVE()
 1 do
 2    r ← LOOK-AHEAD()
 3    if r = TRUE then
 4       r ← CHOOSELITERAL()
 5    else
 6       r ← LOOK-BACK()
 7 while r = UNDEF
 8 return r
```

Figure 5.1: Iterative version of DLL algorithm.

look-ahead is not effective;

- even assuming we had an oracle giving us the list of literals which will fail (thus saving the time necessary to scan the list of literals), enhanced look-ahead is not effective.

We reached such conclusions by analyzing CPU times, but also, and perhaps most convincingly, on the basis of CPU independent data such as the total number of assignments performed by SIMO. Therefore, enhancing the look-ahead method with powerful deduction techniques seems to be unsuccessful, at least in the case of failed literal detection and real-world instances.

5.1 Experimenting with look-ahead techniques

In this section we introduce and compare two versions of SIMO: Simo-Up, i.e., SIMO in its default configuration, and Simo-Fp, i.e., SIMO enhanced with failed literal detection. In Subsec. 5.1.1 we briefly describe the implementation of SIMO, the differences between Simo-Up and Simo-Fp, and the failed literal detection algorithm. In Subsec. 5.1.2 we describe the benchmarks selected for our analysis and the computing platform on which the experiments have been carried out. Finally, in Subsec. 5.1.3 we show experimental data indicating that the search space explored by Simo-Fp appears to be smaller than the one explored by Simo-Up, but Simo-Fp performance is much worse than Simo-Up.

5.1.1 Iterative version of the (modern) SAT solver simo

In order to better understand the modification (in the algorithm) introduced in this Chapter, we introduce here the "iterative" version of the DLL algorithm (DLL-SOLVE()). This description introduces the concept of "state of the search" r (that can be undefined if the formula is not yet satisfiable nor unsatisfiable), but more important there is the explicit indication of the LOOK-BACK() phase.

Particular emphasis is given to the look-ahead engine, since this is the only part in which Simo-Fp differs with respect to Simo-Up. The conventions we use to describe algorithms are those of [CLR98] (pages 4,5).

The main search loop of SIMO is DLL-SOLVE, Fig. 5.1, an iterative implementation of the DLL method [DLL62]. Inside DLL-SOLVE:

```
UNIT-PROPAGATE()
r = TRUE
while there is {l} and r = TRUE
    r = Simplify(l)
return r
```

Figure 5.2: The UNIT-PROPAGATE() phase of the algorithm.

```
LOOK-AHEAD()
r ← UNIT-PROPAGATE()
if r = FALSE then
    return FALSE
else
    return FAILED-PROPAGATE()
```

Figure 5.3: The LOOK-AHEAD() phase of the algorithm.

```
FAILED-PROPAGATE()
for each open atom a
    r ← UNIT-PROPAGATE(a)
    LOOK-BACK()
    if r = FALSE then
        r ← UNIT-PROPAGATE(¬a)
        if r = FALSE then return FALSE
    else
        r ← UNIT-PROPAGATE(¬a)
        LOOK-BACK()
        if r = FALSE
            UNIT-PROPAGATE(a)
return TRUE
```

Figure 5.4: The FAILED-LITERAL() phase of the algorithm.

- LOOK-AHEAD simplifies the input formula and returns FALSE if an empty clause is deduced, TRUE otherwise;
- CHOOSELITERAL assigns an open literal according to some criterion and returns TRUE in case no open literal can be found, UNDEF otherwise;
- LOOK-BACK backtracks to a previous open branch in the search tree, and returns FALSE if such branch cannot be found, UNDEF otherwise.

LOOK-AHEAD, Fig. 5.3, is where the distinction between Simo-Up and Simo-Fp comes into play: in Simo-Up, lines 2-5 of LOOK-AHEAD are replaced with the instruction "**return** r", i.e., Simo-Up simplifies the formula using unit clause propagation only; in Simo-Fp, the implementation of LOOK-AHEAD is exactly as detailed in Fig. 5.3. FAILED-PROPAGATE, in Fig. 5.4, scans all the open atoms of the formula (line 1). For each such atom a, it tentatively assigns a to TRUE and performs unit clause propagation (line 2), and then it retracts the tentative assignment (line 3). If UNIT-PROPAGATE deduces an empty clause, i.e., the literal a is failed, then FAILED-PROPAGATE tries to assign a to FALSE (line 5): if even this fails, then it returns FALSE , i.e., it forces backtracking (line 6). If the propagation of a succeeds, then FAILED-PROPAGATE repeats the same process for $\neg a$ (lines 8-11). Notice that, in case $\neg a$ is failed (line 11), FAILED-PROPAGATE already knows that a is not, and so there is no need to check the result of UNIT-PROPAGATE. If there was no atom a such that both a and $\neg a$ failed, FAILED-PROPAGATE returns TRUE .

The functions UNIT-PROPAGATE, CHOOSELITERAL and LOOK-BACK called by the algorithms in Fig. 5.1 are implemented as described in [MMZ+01] and [ZMMM01], and detailed in Subsec 2.4.2.

5.1.2 Experimental Setting

The test set that we used for our experimental analysis consists of 483 real world instances. The benchmarks have been selected considering classical SAT problems as well as instances submitted to the last SAT 2002 competition [SBH02]. The benchmarks are divided into 10 families:

Beijing'96 16 instances from the Beijing'96 SAT competition, a mixture of scheduling [CB94], planning and circuit verification problems;

bmc 78 instances including

- 30 instances from CMU described in [BCCZ99],
- 13 instances from IBM, described in [Sht00], and
- 22 "dining philosophers" and 6 "counter" model checking instances described in [SBH02];

des 28 instances of logic cryptanalysis for security protocols described in [MM00];

fev 8 instances, encoding the problem of comparing the output of different adder circuit implementations (see [SBH02]);

fpga 102 instances of FPGA routing problems, including "homer" and "bart" instances described in [MAAS02];

fvp-unsat 2.0 22 instances of formal property verification problems;

mediator 4 instances of query optimization problems in complex mediator systems, described in [Pre02];

miters 25 instances encoding problems of Combinational Equivalence Checking described, e.g., in [SG99];

sss-sat 1.0 100 instances related to formal property verification of super-scalar microprocessors;

vliw-sat 1.0 100 instances related to formal property verification of VLIW microprocessor.

The three families fvp-unsat 2.0, sss-sat 1.0 and vliw-sat 1.0 are collectively described in [VB01]. The size of the problems varies considerably, from a minimum of 25 atoms and 60 clauses (an instance in the fev family), to a maximum of 315,312 atoms and 1,725,384 clauses (an instance in the fpga family), with an overall average of 11,520 atoms and 108,229 clauses. To confirm that the test set is indeed challenging for modern SAT solvers, we used zChaff [MMZ+01], the winner of the latest SAT competitions on industrial categories (see [SBH02]). By running zChaff on the test set we obtained the following results:

- 438 problems can be solved within a time limit of 20 minutes;
- 303 instances are found satisfiable, 135 unsatisfiable, and the remaining timed out;
- the total run time (excluding timed out instances) is about 5 and a half hours, with an average run time of about 50 seconds.

All the experiments presented in this paper, including the above exploratory analysis, have been run on two identical PCs equipped with a Pentium IV 1.8 GHz processor, and 512MB of RAM (DDR 266Mhz), running Linux RedHat 8.0 (kernel version 2.4.18-14). Since this platform cannot normally measure CPU times below the threshold of $10^4 \mu$sec, to overcome this limitation we patched the kernel with the tool perfctr contained in the distribution of the package papi 2.3.1 [DLM+01]. We used a binary distribution of zChaff 2001.2.17 for the exploratory analysis, and our solver SIMO (version 3.0) to run all the experiments presented. SIMO has been compiled with gcc 3.2. For practical reasons, a timeout of 1200 seconds has been placed on the run time of SIMO on each instance.

5.1.3 Simo-Up vs. Simo-Fp

Using the algorithms presented in Subsec. 5.1.1 and the test set presented in Subsec. 5.1.2 we run the comparison between Simo-Up and Simo-Fp. For both versions of SIMO, the distributions of the run time and the number of calls to CHOOSELITERAL (branches), are summarized in Fig. 5.5. The x-axis in both plots is an ordinal in the range (0-180), and the y-axis is, respectively, CPU seconds in Fig. 5.5 (left) and number of branches in Fig. 5.5 (right). The total number of problems visualized is 181 out of 483 since we discarded

- instances in which either Simo-Up or Simo-Fp exceeded the time out, and

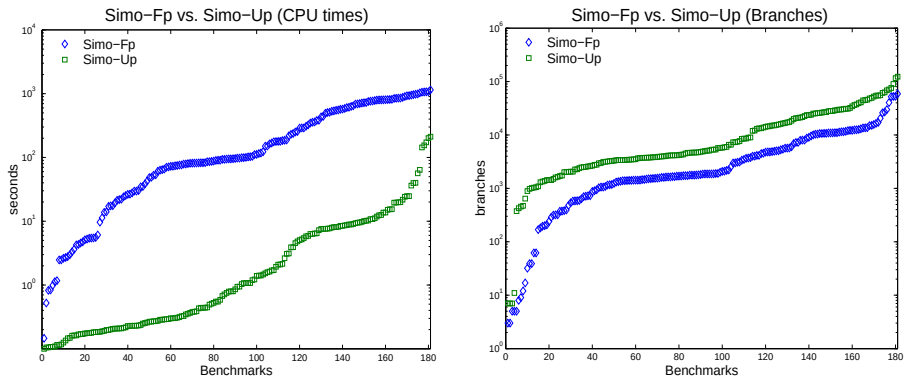


Figure 5.5: Simo-Up vs. Simo-Fp considering CPU time (left) and Branches (right).

- instances in which the run time of Simo-Up was less than 0.1 seconds.

Both plots in Fig. 5.5 are obtained by ordering the results of Simo-Up and Simo-Fp independently and in ascending order.

As you can see from Fig. 5.5, the performance of Simo-Fp in terms of CPU time is clearly worse than the performance of Simo-Up, also considering that Simo-Up solves another 161 instances (not shown in Fig. 5.5) on which Simo-Fp times out. On the other hand, if we consider the number of branches, then the search space explored by Simo-Fp appears to be always, although not considerably, smaller than the one explored by Simo-Up. At first glance, this seems to point out a specific inefficiency in the implementation of FAILED-PROPAGATE in Simo-Fp. As we will show in the next section, this is indeed not the case.

5.2 (In)Effectiveness of failed literal detection

In this section we will motivate the experimental results shown in the previous one, and we will give strong empirical reasons against the effectiveness of failed literal detection in modern SAT solvers like SIMO. To accomplish this, we introduce a family of oracle-based algorithms, where an oracle is an ideal procedure that performs some computation for free (see, e.g., [Pap94]). In particular, we will show how the CPU time of algorithms incorporating oracles can be calculated starting from the CPU time of Simo-Fp, and then we compare the performances of such oracle-based algorithms with Simo-Up.

In the following, we use *tries* as a CPU independent performance measure, instead of branches. The number of tries is the number of times that a literal is assigned a value, for whatever reason, be it a choice of the heuristic, a unit literal, a failed literal, or a tentative assignment performed during FAILED-PROPAGATE. Clearly, the number of branches is always less than (or equal to) the number of tries. As we will see, tries are more informative than branches as a CPU independent performance measure, because most of the overall run time of the solver is spent on assigning literals (as also reported in [MMZ+01]).

This point is confirmed also by our experiments (not reported in details

here), where the correlation between CPU time and tries is higher than the correlation between CPU time and branches.

5.2.1 Introducing oracles in Simo-Fp

As we saw in Subsec. 5.1.1, Simo-Fp scans the list of open literals and propagates each open literal to check whether it fails. We can think of three oracle-based versions of Simo-Fp, that we call Simo-Fp(TO), Simo-Fp(FO), and Simo-Fp(FRO). In particular, we assume to have

- in Simo-Fp(TO), an oracle testing whether a literal will fail, thus saving the time necessary to try the literals which will not be failed;
- in Simo-Fp(FO), an oracle returning the sequence of literals which will fail in Simo-Fp, thus saving also the time necessary to scan the list of open literals;
- in Simo-Fp(FRO), an oracle returning the sequence of literals which will fail in Simo-Fp and their reasons, thus saving also the time necessary to calculate the reasons of the failed literals.

Since the oracles cannot be implemented in practice, we need to calculate the performance of the oracle-based versions using the experimental data of Simo-Fp. In order to accomplish this, we introduce four CPU time counters inside FAILED-PROPAGATE (line numbers refer to FAILED-PROPAGATE in Fig. 5.1):

- Total time (T_f) is the sum of the run times of each call to FAILED-PROPAGATE.
- Time spent on failed (T_s) is the sum of the run times spent to perform literal propagations when the literals are failed (lines 2-5 and lines 8-11, when the tests on lines 4 and 10 are successful, respectively).
- Time wasted on failed (T_w) is the same as above, but when the literals are not failed (lines 2-3 and lines 8-9, when the tests on lines 4 and 10 are not successful, respectively).
- Time spent on reason (T_r) is the sum of the run times spent to calculate the reason of each failed literal when the literal is failed (lines 2-3 and lines 8-9 when the tests on lines 4 and 10 are successful, respectively).

Let T be the CPU time of Simo-Fp, and $T(*)$ be the CPU time of Simo-Fp(*). The performance of the oracle-based algorithms can be calculated as follows:

$$\begin{aligned} T(\mathsf{TO}) &= T - T_w \\ T(\mathsf{FO}) &= T - T_f + T_s \\ T(\mathsf{FRO}) &= T - T_f + T_s - T_r \end{aligned}$$

Notice that $T(\mathsf{FRO}) = T(\mathsf{FO}) - T_r$.

To calculate the tries of the oracle-based versions of Simo-Fp we introduce four more counters (still with reference to FAILED-PROPAGATE in Fig. 5.1):

- Total tries (N_f) is the sum of the tries performed in each call to FAILED-PROPAGATE.

- Tries spent on failed (N_s) is the sum of the tries spent to perform literal propagations when the literals are failed (the tries performed by UNIT-PROPAGATE in lines 2,5 or 8,11 when the literal is failed).
- Tries wasted on failed (N_w) is the same as above, but when the literals are not failed (the tries performed by UNIT-PROPAGATE in lines 2 or 8 when the literal is not failed).
- Tries spent on reason (N_r) is the sum of the tries spent to calculate the reason of each failed literal when the literal is failed (the tries performed by UNIT-PROPAGATE in lines 2 or 8 when the literal is failed).

Let N be the number of tries performed by Simo-Fp, and $N(*)$ the number of tries performed by Simo-Fp(*). The number of tries performed by the oracle-based algorithms can be calculated as follows:

$$\begin{aligned} N(\mathsf{TO/FO}) &= N - N_w \\ N(\mathsf{FRO}) &= N - N_f + N_s - N_r \end{aligned}$$

Notice that $N(\mathsf{TO}) = N(\mathsf{FO})$, so we do not need two distinct measures for the tries of Simo-Fp(TO) and Simo-Fp(FO).

5.2.2 Simo-Up vs. Simo-Fp(*)

Considering the oracles presented in the previous Subsection, we can enrich the comparison between Simo-Up and Simo-Fp with the data of Simo-Fp(TO), Simo-Fp(FO), and Simo-Fp(FRO). For all such real and oracle-based versions of SIMO, the distributions of the run time and the number of tries are summarized in Fig. 5.6. The layout of the plots, as well as the selection of the data to show, follows the same guidelines of Fig. 5.5. By looking at Fig. 5.6 we can immediately conclude that aggressive failed literal detection is bound to be ineffective, both in terms of run time and, more interestingly, also in terms of search space explored. The only version of Simo-Fp that can barely compete with Simo-Up is Simo-Fp(FRO), the version of Simo-Fp embodying the most powerful oracle presented in Subsec. 5.2.1. In spite of its power, the number of tries performed by Simo-Fp(FRO) is, on average, only about 80% of the number of tries performed by Simo-Up. As we can deduce from the plots, Simo-Fp performances are influenced by two major factors:

(i) the time (and the tries) spent to check whether a given literal is failed or not, and

(ii) the time (and the tries) spent to calculate the reasons of failed literals.

By looking at Fig. 5.6 (right) we can see two order-of-magnitude gaps in the number of tries: one between Simo-Fp and Simo-Fp(TO/FO), which confirms point (i), and one between Simo-Fp(TO/FO) and Simo-Fp(FRO), which confirms point (ii).

In the following we concentrate on tries as a performance measure. The explanation for the performance gap between Simo-Fp and Simo-Fp(TO/FO) lies in the fact that FAILED-PROPAGATE, on a formula with N atoms, performs $O(N^2)$ tests along each path of the search tree. Moreover, each test involves

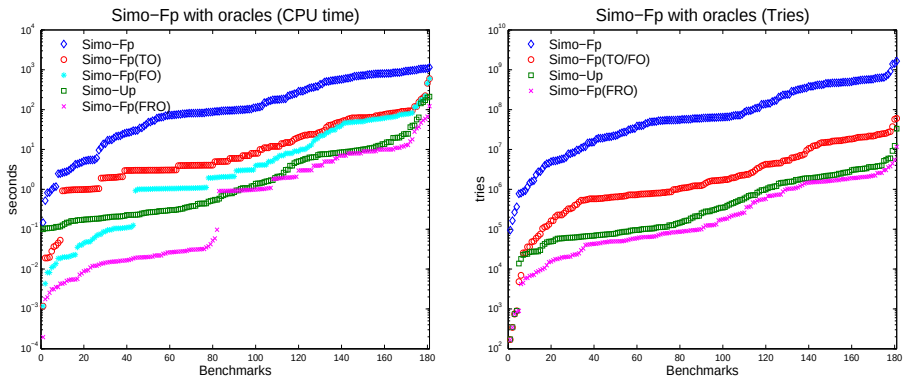


Figure 5.6: Simo-Up, Simo-Fp and oracle-based versions of Simo-Fp, considering CPU time (left) and Tries (right).

Benchmarks				
Family	Sat	Tot	At#	Cl#
Beijing-1996	8	8	8,226	53,390
bmc	14	30	10,466	52,995
des	7	7	3,285	20,539
fev	0	3	1,324	3,819
fpga	10	30	32,612	194,786
fvp-unsat.2.0	0	5	1,468	15,206
mediator	2	2	561.50	12,086
miters	3	12	2,261	6,119
sss-sat.1.0	79	79	5,022	51,043
vliw-sat.1.1	5	5	20,780	284,509

Table 5.1: Data for each benchmark family.

a round of unit propagation which performs $O(N)$ tries. The total complexity of failed propagation along a single path is therefore $O(N^3)$ tries. Considering that the percentage of tests that yield a failed literal propagation is always less than 5% and about 0.2% on average, the gap between Simo-Fp and Simo-Fp(TO/FO) grows as $O(N^3)$. The reason of the performance gap between Simo-Fp(TO/FO) and Simo-Fp(FRO) is explained by the fact that in order to calculate the reason of a failed literal, Simo-Fp(TO/FO) has to pursue the "wrong" branch till it generates a contradiction. This involves one assignment plus several unit propagations: in our test set, such propagations cost about 160 tries on average.

To complete our experimental analysis, we need to confirm that the cumulative results of Fig. 5.6 are true also of each single family, i.e., there are no compensation effects among different families of benchmarks. In Tables 5.1 and 5.2 we present the data regarding Simo-Up, Simo-Fp and the oracle-based versions of Simo-Fp. Each row of the Tables contains data about a single family among those detailed in Subsec. 5.1.2. For each family we report: in Table 5.1 the number of benchmarks left in the family after instances have been filtered out as described in Subsec. 5.1.3 (column Tot); the number of satisfiable instances (column Sat); the average number of atoms and clauses (columns At# and Cl#, respectively); in Table 5.2 the total number of tries performed by

SIMO Tries (x1000)				
Family	Up	Fp	Fp(FRO)	Fp(TO/FO)
Beijing-1996	1,151	4,475,504	113	5,141
bmc	95,253	5,006,227	33,669	290,623
des	3,073	85,967	786	13,554
fev	1,636	168,740	786	2,814
fpga	10,326	960,441	9,232	48,375
fvp-unsat.2.0	8370	1,047,241	5,438	52,900
mediator	3,689	22,472	1,289	12,833
miters	27,398	2,505,136	23,478	72,405
sss-sat.1.0	75,227	17,553,074	56,333	601,287
vliw-sat.1.1	242	2,657,969	127	8,374

Table 5.2: Simo-Up, Simo-Fp and oracle-based versions of Simo-Fp, considering Tries arranged by benchmark family.

Simo-Up, Simo-Fp, Simo-Fp(TO/FO) and Simo-Fp(FRO) divided by 1,000. Notice that clauses and atoms statistics have been rounded to 1, and tries statistics have been rounded to 1,000. Although we cannot show here the complete data, compensation effects are absent also when looking at single instances in each family. In other words, there is no single instance in our test set on which Simo-Fp(TO/FO) performs less tries than Simo-Up or Simo-Fp(FRO), while on all the instances Simo-Fp(FRO) performs less tries than Simo-Up.

5.2.3 Ineffectiveness of failed literal in a modern SAT solver

All the points we made so far might explain some of the deficiencies of failed literal propagation, but they still do not explain why Simo-Fp(TO/FO) loses the comparison with Simo-Up. Considering the implementation of SIMO, as shown in Fig. 5.1, it is easy to see that the solver is parametric on UNIT-PROPAGATE, CHOOSELITERAL and LOOK-BACK. In particular, while the efficient implementation of UNIT-PROPAGATE is not likely to interfere negatively with the performances of Simo-Fp, the same is not true for CHOOSELITERAL and LOOK-BACK. Therefore, in order to understand the impact of CHOOSELITERAL and LOOK-BACK we introduce four more versions of SIMO:

Simo(R)-Up and Simo(R)-Fp are obtained, respectively, from Simo-Up and Simo-Fp by substituting the Chaff-like heuristic with a random one; in Simo(R)-Up, Simo(R)-Fp and its variants Simo(R)-Fp(*) CHOOSELITERAL chooses a literal uniformly at random from the set of open literals at each branch.

Simo(CR)-Up and Simo(CR)-Fp are obtained from Simo(R)-Up by substituting the advanced learning technique with a straightforward chronological backtracking routine; in Simo(CR)-Up, Simo(CR)-Fp and its variants Simo(CR)-Fp(*) LOOK-BACK always backtracks to the latest open branch node, irrespective of whether the branch caused the conflict or not, and learning is disabled, i.e., no clauses are ever added to the initial set.

Further, we selected about one hundred instances from the initial test set of 483 described in Subsec. 5.1.2 by considering all the benchmarks that Simo-Fp solved in less than 10 seconds. The resulting test set contains about 50% of

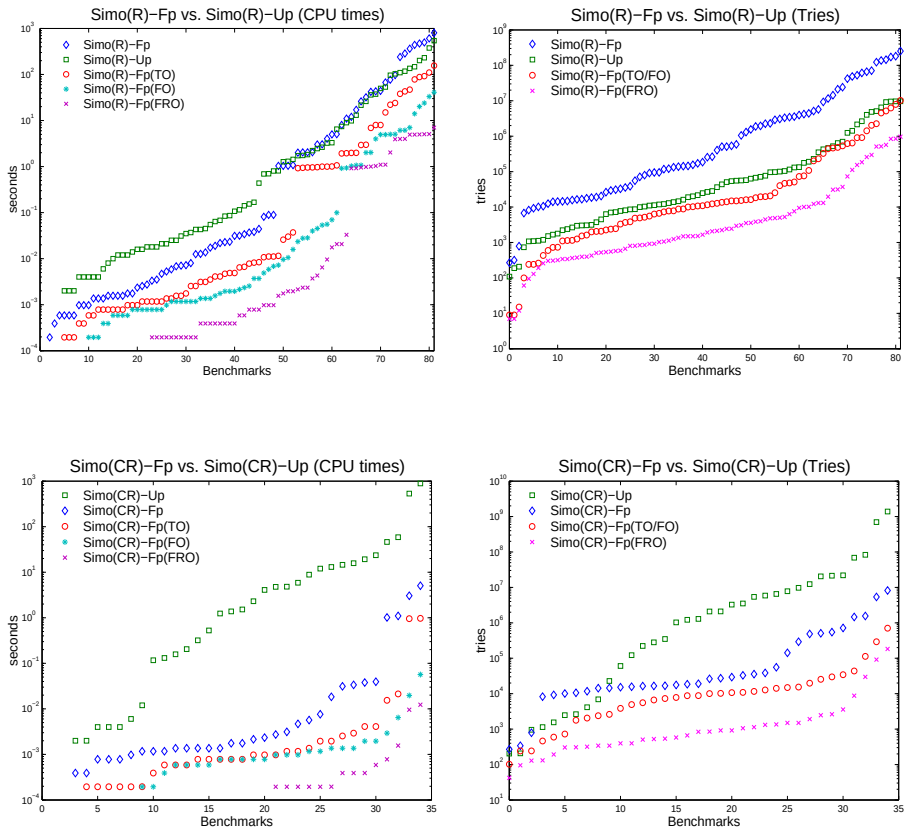


Figure 5.7: Simo(R)-Up, Simo(R)-Fp and oracle-based versions of Simo(R)-Fp (top); Simo(CR)-Up, Simo(CR)-Fp and oracle-based versions of Simo(CR)-Fp (bottom); CPU times (left) and Tries (right).

satisfiable formulas, and about 10% of formulas that could not be solved by the fastest of the four versions of SIMO described above. The results presented hereafter are preliminary in the sense that we did not run multiple instances of the randomized algorithms.

In Fig. 5.7 we present the results obtained by running Simo(R)-Up, Simo(R)-Fp, Simo(CR)-Up and Simo(CR)-Fp on the reduced test set. The results are arranged into four plots, where each set of dots represents the runtime (resp., tries) distribution of a particular version of SIMO. Considering Fig. 5.7 (top row) we can observe that Simo(R)-Up is still superior to Simo(R)-Fp: on average, Simo(R)-Fp performs one order of magnitude tries more than Simo(R)-Up, and it is barely competitive in terms of CPU. To this, we must add that Simo(R)-Fp times out on 15 instances, while Simo(R)-Up can solve an additional 8. On the other hand, Simo(R)-Fp(TO/FO) is very close to the performances of Simo(R)-Up: Simo(R)-Fp(TO/FO) performs, on average, about 50% of the number of tries of Simo(R)-Up, and Simo(R)-Fp(TO) is about a factor of 6 faster than Simo(R)-Up. Therefore, although the real version of Simo(R)-Fp does not show any real advantage over Simo(R)-Up, the least powerful of the oracles introduced

in Subsec. 5.2.1 is sufficient to yield a substantial improvement. Considering Fig. 5.7 (bottom row) we can observe that Simo(CR)-Fp dominates Simo(CR)-Up: on average, Simo(CR)-Fp performs about 50 times less tries than Simo(CR)-Up, and it is about 500 times faster. Moreover, Simo(CR)-Up times out on 64 instances, while Simo(CR)-Fp can solve an additional 30.

The preliminary results presented in Fig. 5.7 point out that (*i*) the look-back based heuristic and (*ii*) the advanced look-back techniques implemented in Simo-Up are the major causes of the ineffectiveness of failed literal detection. The fact that failed literal detection *per se* appears to be an effective technique is witnessed by the comparison between Simo(CR)-Up and Simo(CR)-Fp. Nevertheless, the addition of a clever look-back technique can change this result dramatically, as witnessed by the comparison between Simo(R)-Up and Simo(R)-Fp. Our current explanation of this, is that the look-back schema implemented in SIMO *shadows* the effects of failed literal propagation. Intuitively, even though Simo(R)-Up may assign failed literals late in the search tree, its look-back engine will assign the negation of the failed literals and place them at the proper level in the search stack. Although we do not have yet a precise intuition about the underlying phenomena in the case of the heuristic, by contrasting the results of Simo(R)-Up and Simo(R)-Fp with those presented in Subsec. 5.2.2, it is clear that also the implementation of CHOOSELITERAL is playing a major role in the ineffectiveness of failed literal detection.

5.3 Summing up

Concluding, the Chapter presented strong empirical evidence that enhanced look-ahead based on failed literal detection does not pay off in modern SAT solvers, at least in the case of real-world problems. In particular we showed that

- the number of tries performed by Simo-Fp is, on average, about three orders of magnitude bigger than the number of tries performed by Simo-Up;
- the number of tries performed by Simo-Fp(TO/FO) is, on average, about one order of magnitude bigger than the number of tries performed by Simo-Up;
- the number of tries performed by Simo-Fp(FRO) is, on average, only about 80% of the number of tries performed by Simo-Up.

The CPU times of Simo-Fp(*) and Simo-Up share a similar behavior.

On the basis of the preliminary results shown in Subsec. 5.2.3 we conjecture that look-back based heuristics and advanced look-back techniques account for the ineffectiveness of Simo-Fp.

Apart from the results on the case study related to failed literal that we have presented, in this Chapter we have introduced a new methodology for testing if a technique (before optimally implement it) have the potential to improve the performances of an algorithm if enhanced with the technique.

Part III

Safe Planning via Temporal Reasoning

Chapter 6

Introduction

In the previous part we have seen how to encode the problem of Safe Planning into a propositional formula, i.e., to decide if a planning is also "safe" w.r.t. some given safety properties via decision about the satisfiability of a propositional formula. However, this is not the only possible approach.

We can reduce the Safe Planning problem to a decision of a formula that involves both propositional logic and (a limited form of) linear constraints. We focus here on the case where the action description D is deterministic.

The idea is quite simple. Remember that if A is an action symbol, the atom A_i expresses that A is among the elementary actions executed at time i. Then, instead of assuming that in tr_i^D there is a propositional variable A_i for each ground action symbol A, we assume that in tr_i^D there is a constraint of the type $A = i$ for each ground action symbol A. This means that now tr_i^D can not be longer assumed to be a propositional formula, but now is a "Temporal Reasoning" formula or, a formula in Separation Logic (SL)[1] [Pra77]. The approach has a disadvantages w.r.t. the one that uses only propositional logic: An action can be executed only once, because only one of the constraints $A = i$ can be true in the different tr_i^D. Nevertheless, the problem can be overcome by introducing, instead of a unique constraint, a bunch of constraints (for the same A), $A_0 = 0, A_1 = 1, \ldots, A_i = i, \ldots, A_n = n$. Are also possible constraints that induce some "orders" among the actions, e.g., $A - B <= k$, that encodes the fact that the action A must be executed at most k time units after the execution of B.

The final formula to check is

$$I_0 \wedge \wedge_{i=0}^{n-1}(A_i = i) \wedge \wedge_{i=0}^{n-1} tr_i^D \models [\alpha]^n. \tag{6.1}$$

where, now,

- I_0 and $[\alpha]^n$ are propositional formulas
- $\wedge_{i=0}^{n-1} tr_i^D$ is a formula in SL

[1] SL has also other connectives other than "=" as we will see later. Moreover, constraints in the form $A = i$ are not exactly SL constraints: We need to add a slack variable Z_0 in order to create the SL constraint $A - Z_0 = i$.

This approach boils down to checking the satisfiability of a formula in Temporal Reasoning (or in Separation Logic) in order to check if a planning is also safe under given conditions.

With respect to the approach of checking a formula expressed in pure propositional logic, obviously there are both advantages and disadvantages.

Clearly, an SL formula is strictly more expressive w.r.t. a propositional formula, and the "encoding" most likely result in a more compact formula. On the other hand, very specialized and efficient solver for propositional reasoning exist, and they have had a tremendous boost in the last years; while decision procedures for propositional logic enhanced with linear arithmetic is not so a mature area.

In order to possibly overcame the last point, we have chosen an approach to solve SL formulas mainly based on reasoning on propositional formulas (SAT-based reasoning).

6.1 Working example

In order to better understand and appreciate all the definitions and optimizations that we will present in the next Chapter, we introduce here a working example, borrowed from the FV community, and we will use it in the next Chapter.

Example 1 *In [BLS02], the case-study is introduced of a bounded model checking problem for the memory unit of the Motorola Elf microprocessor. The unit is initially modeled as* $20K$ *lines of VERILOG, with* 80 *integer-valued variables and* 70 *propositional atoms. After some translation stages, the problem is reduced to checking satisfiability of a formula in SL, a fragment of which, call it* ϕ_{Elf}*, looks like this:*

$$
\begin{array}{ll}
(p_1 \vee \neg(VPred = I_{RR})) & \wedge \\
(\neg p_1 \vee VPred = I_{RR}) & \wedge \\
(\neg p_2 \vee VPred < I_{RR} + 1) & \wedge \\
(p_2 \vee \neg(VPred < I_{RR} + 1)) & \wedge \\
(p_3 \vee p_4) & \wedge \\
(p_3 \vee \neg p_4 \vee \neg VenI' = VenI) & \wedge \\
(p_5 \vee \neg(VenI' + 2 = VenI)) & \wedge \\
(\neg p_5 \vee VenI' + 2 = VenI) &
\end{array}
$$

In the above formula, VPred, I_{RR}*, VenI, VenI' are variables and* p_1*,* p_2*,* p_3*,* p_4*,* p_5 *are propositional letters.* $VPred < I_{RR} + 1$ *is a constraint, and* p_5 *and* $\neg(VenI' + 2 = VenI)$ *are literals.* $\star$

Chapter 7

From DLL to a decision procedure for TR

In this Chapter we will see how to turn a decision procedure for DLL into a decision procedure able to decide temporal reasoning formulas, or Separation formulas (formulas in Separation Logic, SL).

In the rest of this part, without loss of generality we will speak only of SL and SL formulas.

We will introduce same necessary theoretical background about SL, and then we will present a decision procedure for this logic.

7.1 Separation logic

7.2 Syntax

Let $\mathcal{V}$ and $\mathcal{P}$ be two disjoint sets of symbols, called *variables* and *propositional atoms* respectively. A *constraint* is an expression of the form $x - y \bowtie c$ where $x, y \in \mathcal{V}$, $\bowtie \in \{\leq, <, >, \geq, =, \neq\}$ and c is a numeric constant. The notations $x \bowtie y + c$ and $x - c \bowtie y$ will also be freely used in place of $x - y \bowtie c$. An *atom* is either a constraint or a propositional atom. A *formula* is a combination of atoms via the unary connective "$\neg$" for negation and the n-ary connectives "$\wedge$" and "$\vee$" ($n \geq 0$) for conjunction and disjunction respectively. We will write $\top$ and $\bot$ for the empty conjunction and the empty disjunction respectively. A *literal* is either an atom or its negation. If a is an atom, then $\overline{a}$ abbreviates $\neg a$ and $\overline{\neg a}$ stands for a.

7.2.1 Semantics

Let the set D (*domain of interpretation*) be either the real or the integer numbers. An *assignment* is a total function mapping variables to D and propositional atoms to the truth values FALSE and TRUE.

Let σ be an assignment and ϕ be a formula. Then $\sigma \models \phi$ (σ satisfies *a formula* ϕ) is defined as follows

$$\sigma \models x - y \bowtie c \text{ if and only if } \sigma(x) - \sigma(y) \bowtie c,$$

$\sigma \models p$ with $p \in \mathcal{P}$ if and only if $\sigma(p) = true$,

$\sigma \models \neg\phi$ if and only if it is not the case that $\sigma \models \phi$,

$\sigma \models (\wedge_{i=1}^{n}\phi_i)$ if and only if for each $i \in [1, n]$, $\sigma \models \phi_i$, and

$\sigma \models (\vee_{i=1}^{n}\phi_i)$ if and only if for some $i \in [1, n]$, $\sigma \models \phi_i$.

If $\sigma \models \phi$ then σ will also be called a *model* of ϕ. We also say that

- a formula ϕ is *satisfiable* if and only if there exists an assignment satisfying it;
- a formula ϕ is *valid* if and only if every assignment satisfies it;
- two formulae ϕ and ψ are *logically equivalent* if and only if the formula $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ is valid.

Here we deal with deciding whether a formula is satisfiable or not in the given domain of interpretation D. Notice that the satisfiability of a formula depends on D, e.g. $x - y > 0 \wedge x - y < 1$ is clearly satisfiable if D is $\mathbb{R}$ but unsatisfiable if D is $\mathbb{Z}$. However, the problems of checking satisfiability in $\mathbb{Z}$ and $\mathbb{R}$ are closely related and will be treated uniformly almost always. Therefore, from now on, we will drop the distinction, and we will introduce it back only when needed.

Example 2 *Consider Example 1.* ϕ_{Elf} *is satisfiable, and a model is* $\sigma = \{p_1 \mapsto true, VPred \mapsto 12, I_{RR} \mapsto 12, p_2 \mapsto true, p_3 \mapsto true, p_4 \mapsto true, p_5 \mapsto true, VenI \mapsto 10, VenI' \mapsto 8\}$. $\star$

7.2.2 Valuations

A *valuation* is a finite set μ of literals such that for each atom a, if $a \in \mu$ then $\neg a \notin \mu$. In the following if μ is a valuation, then by μ we also denote the formula $\wedge_{l \in \mu} l$. Context will make clear what is intended. Moreover, we say that

1. a valuation μ *propositionally entails* a formula φ if $(\neg\mu \vee \varphi)$ can be proved in propositional logic;
2. two formulae are *propositionally logically equivalent* if one formula propositionally entails the other, and vice versa.

The following result shows the importance of valuations.

Theorem 1 *A formula* ϕ *is satisfiable if and only if there exists a valuation* μ *such that*

1. μ *is satisfiable,*
2. *all atoms occurring in* μ *also occur in* ϕ*, and*
3. μ *propositionally entails* ϕ*.*

Proof. The right to left direction is trivial. For the left to right direction, first notice that it is always possible to convert ϕ to an equivalent formula in Disjunctive Normal Form (DNF). Let S be the set of disjuncts in the DNF. Then by the semantics of $\vee$ it follows that ϕ is satisfiable if and only if there is $\mu \in S$ such that μ is satisfiable. Further, for such μ, also the second and third properties hold.

Given the above result, in order to check satisfiability of a formula ϕ, the issue becomes that of efficiently building a set S of valuations which is *propositionally complete for* ϕ, i.e. such that the disjunction of the valuations in S is propositionally logically equivalent to ϕ. Given such a set, we can then separately check the satisfiability of its elements.

function TSAT(ϕ,μ)
1 **if** $\{\} \in \phi$ **then return** FALSE
2 **if** $\phi = \emptyset$ **then return** SatCheck(μ)
3 **if** $\{l\} \in \phi$ **then return** TSAT(Simplify(l,ϕ),$\mu \wedge l$)
4 $l :=$ ChooseLiteral(ϕ)
5 **return** TSAT(Simplify(l,ϕ),$\mu \wedge l$) or
TSAT(Simplify($\bar{l}$,ϕ),$\mu \wedge \bar{l}$)

Figure 7.1: Basic SAT-based decision procedure based on DLL.

The DLL_ENUM algorithm that we have seen in Section 2.5 can be readily turned into a decision procedure for SL as show in Figure 7.1. Again, the modifications are limited to the case in which $\phi = \emptyset$. Instead of printing μ and unconditionally returning FALSE, we now return the result of invoking SatCheck(μ), where SatCheck(μ) is a satisfiability procedure for valuations, i.e. it returns TRUE if μ is satisfiable and FALSE otherwise. This procedure clearly depends on the decidable theory under consideration. As we will see shortly, a satisfiability procedure for SL valuations can be readily built by using the well-known Bellman-Ford procedure, which runs in polynomial time [CLRS01].

Theorem 2 *(Soundness and completeness of TSAT) Let ϕ be a formula. Then* TSAT*(ϕ,$\top$)*[1] *returns* TRUE *if ϕ is satisfiable, and* FALSE *otherwise.*

Proof. It readily follows from Theorem 1, from the soundness and completeness of the DLL algorithm and from Lemma 2.

It is a well known fact that the Bellman-Ford (BF) algorithm can be used to check the satisfiability of a finite set Q of constraints of the form $x - y \leq c$, see, e.g., [CLRS01]. This is done by first building a *constraint graph for* Q, i.e. a weighted directed graph whose nodes are the variables occurring in Q and having an edge from y to x of weight c for each constraint $x - y \leq c$ in Q. An extra node, the *source*, is also included and is linked to all the other nodes with edges of weight 0. BF is then used to solve the "single source shortest-paths" problem. The set of constraints Q is satisfiable if and only if the constraint

[1] Here we initialize μ with $\top$ because it is defined as a formula, and not as a set like in Chapter 2.

graph for Q contains no *negative cycles*, i.e. cycles with cumulative negative weight.

Here we show that satisfiability checking of a generic valuation μ can be done efficiently with BF. As a preliminary step, we turn μ into an equi-satisfiable set $\mu^{\leq,<}$ whose literals are of the form $x - y \leq c$ or $x - y < c$. This can be done by deleting all the literals of the form p and $\neg p$ where p is a propositional atom and by replacing constraint literals

- $y - x \geq -c, \neg(y - x < -c), \neg(x - y > c)$ with the logically equivalent constraint $x - y \leq c$, and
- $y - x > -c, \neg(y - x \leq -c), \neg(x - y \geq c)$ with the logically equivalent constraint $x - y < c$.

A further step is needed to transform the valuation $\mu^{\leq,<}$ into an equi-satisfiable set of constraints of the form $x - y \leq c$ whose satisfiability can be checked with BF. If the domain of interpretation is $\mathbb{Z}$, this can be done by replacing in $\mu^{\leq,<}$ every constraint of the form $x - y < c$ with $x - y \leq c'$ where c' is the maximum integer strictly smaller than c. It is easy to see that the resulting set of constraint is satisfiable if and only if $\mu^{\leq,<}$ is. If the domain of interpretation is $\mathbb{R}$, then we rely on the following result.

Lemma 3 *Let Q and Q' be two finite sets of constraints of the form $x - y \leq c$ and $x - y < c$ respectively. Let n be the number of variables in Q'. Let p be the maximum number of digits appearing to the right of the decimal point in any numeric constant in $Q \cup Q'$. If C is $x - y < c$, let $C_{\leq}$ be $x - y \leq c - \frac{1}{10^p(n+1)}$. Finally, let $Q'_{\leq} = \{C_{\leq} : C \in Q'\}$.*

$Q \cup Q'$ is satisfiable in $\mathbb{R}$ if and only if $Q \cup Q'_{\leq}$ is satisfiable in $\mathbb{R}$.

Proof. The right to left direction is trivial and therefore here we focus on the left to right direction. In the following, if $Q'' \subseteq Q \cup Q'$ is a set of constraints, by $Q''_{\leq}$ we mean the set obtained from Q'' by replacing each constraint C of the form $x - y < c$ with $C_{\leq}$. Further, ϵ is $\frac{1}{10^p(n+1)}$.

We proceed by contradiction and assume that $Q \cup Q'$ is satisfiable while $Q \cup Q'_{\leq}$ is not. In this case, there exists a subset Q'' of $Q \cup Q'$ such that

- Q'' is satisfiable and $Q''_{\leq}$ is not,
- $Q''_{\leq}$ has the form $\{x_1 - x_2 \leq c_1 - e_1, x_2 - x_3 \leq c_2 - e_2, \ldots, x_m - x_1 \leq c_m - e_m\}$, where each e_i is either 0 or ϵ, and
- in $Q''_{\leq}$, there are at least one and at most n constraints for which $e_i = \epsilon$, i.e. $1 \leq |Q'' \cap Q'| \leq n$.

Q'' is satisfiable and $Q''_{\leq}$ unsatisfiable imply $\sum_{i=1}^{m} c_i > 0$ and $\sum_{i=1}^{m}(c_i - e_i) < 0$ respectively (notice that it cannot be the case that $\sum_{i=1}^{m} c_i = 0$ because $Q'' \cap Q' \neq \emptyset$ and Q'' has to be satisfiable by hypothesis). Since $\sum_{i=1}^{m} c_i > 0$, then $\sum_{i=1}^{m} c_i \geq \frac{1}{10^p}$. But then we have a contradiction, because

$$\begin{array}{ll}
\sum_{i=1}^{m}(c_i - e_i) & = \\
\sum_{i=1}^{m} c_i - \sum_{i=1}^{m} e_i & \geq \\
\sum_{i=1}^{m} c_i - n\epsilon & = \\
\sum_{i=1}^{m} c_i - \frac{1}{10^p}\frac{n}{n+1} & \geq \\
\frac{1}{10^p} - \frac{1}{10^p}\frac{n}{n+1} & > 0
\end{array}$$

Notice that the application of the above result requires, if the domain of interpretation is $\mathbb{R}$, to determine the values of n and p which in turn depend on μ. The next result shows that the values for n and p can be computed beforehand and once and for-all, on the basis of the input formula ϕ.

Theorem 3 *Let ϕ be a formula with n variables. Let p be the maximum number of digits appearing to the right of the decimal point in any numeric constant in ϕ. Let μ be a valuation whose atoms occur in ϕ. The valuation μ is satisfiable in $\mathbb{R}$ if and only if the valuation obtained from $\mu^{\leq,<}$ by replacing each constraint $x - y < c$ with $x - y \leq c - \frac{1}{10^p(n+1)}$ is satisfiable in $\mathbb{R}$.*

Proof. Clearly, μ is satisfiable in $\mathbb{R}$ if and only if $\mu^{\leq,<}$ is satisfiable in $\mathbb{R}$. The thesis trivially follows from Lemma 3 once we observe that, given that the atoms in μ occur in ϕ,

- the number of variables in $\mu^{\leq,<}$ is less than or equal to n and
- the maximum number of digits appearing to the right of the decimal point in any of the numeric constants in $\mu^{\leq,<}$ is less than or equal to the maximum number of digits appearing to the right of the decimal point in any of the numeric constants in ϕ.

The above results allow us to use BF in order to check the satisfiability of any valuation. Given a valuation μ with n variables, BF runs in time $O(n \times |\mu|)$, and is the best known method for this task (see [CLRS01]). Further BF has the following advantages, in the case the valuation μ is unsatisfiable:

- each negative cycle in the constraint graph G corresponds to a minimal (with respect to set inclusion) unsatisfiable subset of μ, and
- assuming there are more than one negative cycles in G, and that R is the corresponding set of reasons (where a reason if a set of constraints involved in a negative cycle), it is easy to modify BF (i) in order to return the "best" reason under given conditions *among those in* R, and (ii) without modifying its overall complexity $O(n \times |\mu|)$.

Example 3 *Once again, let us consider Example 1. Assume, moreover, that ChooseLiteral simply returns the first atom in lexicographical order. Then here is how* TSAT*(ϕ_{Elf},$\top$) works:*

1. *since there are no unit clauses, p_1 is chosen and $\mu = \{p_1\}$;*
2. *after Simplify(p_1,ϕ_{Elf}) is executed, the second clause has become unit since $\neg p_1$ has been removed from it; therefore $VPred = I_{RR}$ is detected as appearing in a unit clause and added to μ;*
3. *same as Items 1 and 2, but with p_2 and $VPred < I_{RR} + 1$; now $\mu = \{p_1, VPred = I_{RR}, p_2, VPred < I_{RR} + 1\}$;*
4. *again, there are no unit clauses, therefore p_3 is chosen and added to μ;*
5. *after Simplify(p_3,ϕ_{Elf}) is executed, no unit clauses are left, so p_5 is chosen and added to μ;*

6. *lastly, VenI'+ 2 = VenI is detected in a unit clause and added to* μ *which is now* $\{p_1, VPred = I_{RR}, p_2, VPred < I_{RR} + 1, p_3, p_5, VenI' + 2 = VenI\}$*;*

7. ϕ_{Elf} *has now become empty; SatCheck is called and a model of* μ*, which also is a model of* ϕ_{Elf}*, is found; for instance, the model in Example 2.*

$\star$

Chapter 8

The TSAT++ solver

In this Chapter we present TSAT++ [ACGM05, ACGM04, ACG+04], our solver for SL. TSAT++ is the new version of our solver Tsat, both designed and implemented at MRG-Lab (now STAR-Lab and AI-Lab) at DIST. In the rest of the Chapter we present some key optimizations we have implemented of top of the basic version of the solver, prove their soundness and completeness properties, and show experimentally their effectiveness.

8.1 Optimizations

The clear separation between the enumeration of valuations propositionally entailing ϕ and the check of their satisfiability is the key feature of the SAT-based approach to building decision procedures. However, the naive application of this idea may suffer from the generation of exponentially many unsatisfiable valuations. The reason of this inefficiency is that the SAT solver is not aware of the properties of the background theory, in our case SL. To illustrate this point, let us again consider the problem of Example 1. If $V_Pred_1 = I_{RR}$ is assigned to TRUE, then it is pointless to assign FALSE to $V_Pred_1 < I_{RR} + 1$ as this valuation (or any extension thereof) will be later found to be unsatisfiable and hence rejected by SatCheck.

As a matter of fact most optimizations to the basic procedure that have been proposed in the literature aim at preventing the generation of unsatisfiable (and hence useless) valuations. In this section we describe four optimizations that—as shown in Section 8.3—make TSAT++ the fastest decision procedure for SL on a wide range of benchmark problems.

8.1.1 IS_n Pre-processing

To reduce the enumeration of unfruitful valuations at a reasonable price, [ACG99] introduced the so called IS_n pre-processing. The name stands for *Inconsistent Subsets* and the subscript number represents the size of the subsets sought for. Naively put, if P is the set of constraint literals occurring positively in the input formula, IS_n checks the satisfiability of all the valuations P' subset of P such that $|P'| \leq n$: For each unsatisfiable subset P', the clause $\vee_{l \in P'} \bar{l}$ is added to the input formula before calling the TSAT procedure.

Although IS_n can be exponential in general, for each fixed n polynomially many subsets of cardinality n exists, and if satisfiability checking is done in polynomial time the resulting procedure runs in polynomial time.

For a given value of n, it also makes sense to generalize the idea in order to check the satisfiability of sets P, with $|P| \leq n$, of literals whose atom occurs in the input formula. To ease the presentation, we restrict to the case in which $n = 2$. The generalization of IS_2 works as follows: for each unordered pair $\{c_i, c_j\}$ of distinct constraints appearing in ϕ such that they involve the same variables, all possible pairs of literals built out of them are checked for satisfiability.

The resulting optimized version of TSAT is given in Figure 8.1.

function TSAT_$IS_2(\phi)$
1 **let** $\phi_0 := \top$
2 **foreach** unordered pair of constraints $\{c_i, c_j\}$ in ϕ
3 involving the same variables,
4 **if** SatCheck$(c_i \wedge c_j)$=FALSE **then** $\phi_0 := \phi_0 \wedge (\neg c_i \vee \neg c_j)$
5 **else if** SatCheck$(\neg c_i \wedge c_j)$=FALSE **then** $\phi_0 := \phi_0 \wedge (c_i \vee \neg c_j)$
6 **else if** SatCheck$(c_i \wedge \neg c_j)$=FALSE **then** $\phi_0 := \phi_0 \wedge (\neg c_i \vee c_j)$
7 **else if** SatCheck$(\neg c_i \wedge \neg c_j)$=FALSE **then** $\phi_0 := \phi_0 \wedge (c_i \vee c_j)$
8 **return** TSAT$((\phi_0 \wedge \phi), \top)$

Figure 8.1: IS_2 pre-processing.

Theorem 4 *(Soundness and completeness of* TSAT_IS_2*) Let ϕ be a formula. Then* TSAT_$IS_2(\phi)$ *returns* TRUE *if ϕ is satisfiable, and* FALSE *otherwise.*

Proof. By Theorem 2, since ϕ_0 is logically valid and therefore ϕ and $\phi_0 \wedge \phi$ are equivalent.

Example 4 *Consider Example 1 once more. After the pre-processing step of* TSAT_$IS_2(\phi_{Elf})$*, the clauses*

$$\neg(VPred = I_{RR}) \vee VPred < I_{RR} + 1$$

and

$$\neg(VenI' = VenI) \vee \neg(VenI' + 2 = VenI)$$

are added to ϕ_{Elf}. These added clauses allow for more pruning while descending the search tree.

Consider Example 3. In TSAT_$IS_2(\phi_{Elf})$*, choosing p_1 forces $VPred = I_{RR}$ by unit propagation; but now, thanks to the clause added by IS_2, this also forces $VPred < I_{RR} + 1$, which in turn forces p_2.* TSAT$(\phi_{Elf}, \top)$ *on the other hand, had to branch on p_2.* $\star$

IS_2 is a simple way of guiding the generation phase by taking into account the structure of the constraints in the input formula. IS_2 has been proved to speed-up the search, especially on randomly generated problems such as the Disjunctive Temporal Problems (DTPs), which are made of binary clauses

containing constraints only (cf. Section 8.3.1). In that case, the effectiveness of the technique is dramatic, since adding more binary clauses, which is what IS_2 does, paves the way to detect and propagate more unit clauses once a literal has been selected by ChooseLiteral.

8.1.2 Early Pruning

An alternative approach that aims to limiting the generation of unsatisfiable valuations is based on the idea of checking the valuations while they are generated by TSAT. This technique is called *early pruning* (EP) and relies on the fact that no unsatisfiable valuation can be extended into a satisfiable one by adding more constraints. EP can be readily incorporated in TSAT as shown in Figure 8.2.

function TSAT_EP(ϕ,μ)
1 **if** $\{\} \in \phi$ **then return** FALSE
2 **if** $\phi = \emptyset$ **then return** SatCheck(μ)
3 **if** $\{l\} \in \phi$ **then return** TSAT_EP(Simplify(l,ϕ),$\mu \wedge l$)
4 **if** SatCheck(μ)=FALSE **then return** FALSE
5 l := ChooseLiteral(ϕ)
6 **return** TSAT_EP(Simplify(l,ϕ),$\mu \wedge l$) or
TSAT_EP(Simplify($\bar{l}$,ϕ),$\mu \wedge \bar{l}$)

Figure 8.2: TSAT with early pruning.

Theorem 5 *(Soundness and completeness of* TSAT_EP*) Let ϕ be a formula.* TSAT_EP*(ϕ,$\top$) returns* TRUE *if ϕ is satisfiable, and* FALSE *otherwise.*

Proof. By Theorem 2 we know that TSAT is sound and complete. Now, first notice that TSAT_EP only differs from TSAT in that one more recursion base case, possibly returning FALSE, has been introduced at line 4. This fact ensures soundness of the function: if TSAT finds no model of ϕ, neither will TSAT_EP.

As far as completeness is concerned, assume by contradiction that a satisfiable valuation μ is found by TSAT, which is not found by TSAT_EP. By the above consideration, this means that a subset of μ, call it μ', must have been reached by TSAT_EP and rejected. This means that μ' is unsatisfiable and μ, a superset of it, is satisfiable, which is contradictory.

Example 5 *Consider Example 1,* TSAT_EP *as in the Figure, and assume ChooseLiteral return the first literal that appears in the formula. Then,* TSAT_EP*(ϕ_{Elf},$\top$) picks and adds to μ, in turn, p_1, $VPred = I_{RR}$ and $\neg p_2$. The last choice forces $\neg(VPred < I_{RR} + 1)$ into μ by unit propagation, but clearly the valuation is now unsatisfiable. Therefore backtracking happens and both $\neg(VPred < I_{RR} + 1)$ and $\neg p_2$ are removed from μ. ChooseLiteral then switches to p_2 and the algorithm goes on as in Example 3.*

Notice that in this case TSAT*, with the same ChooseLiteral, would have explored a totally useless portion of the search space, namely checking all models prefixed with the unsatisfiable μ detected above by EP.* $\star$

8.1.3 Model Reduction

A further optimization, called *model reduction*, is based on the observation that a valuation μ generated by TSAT can be *redundant*, i.e. there might exist a valuation $\mu' \subset \mu$ that propositionally entails the input formula. When this is the case, we can check the satisfiability of μ' instead of μ. This has the following advantages:

1. If μ and μ' are either both satisfiable or both unsatisfiable, then the value returned by SatCheck is the same. However, checking the satisfiability of μ' can be easier if we use, e.g., the Bellman-Ford algorithm for this task.

2. If μ is unsatisfiable, it may nevertheless be the case that μ' is satisfiable: in this case SatCheck(μ') returns TRUE, thereby pruning any further search.

Model reduction can be easily incorporated in TSAT as shown in Figure 8.3. The main difference with respect to TSAT is that the reduced model μ',

function TSAT_MR(ϕ,μ)
1 **if** $\{\} \in \phi$ **then return** FALSE
2 **if** $\phi = \emptyset$ **then return** SatCheck(ReduceModel(μ))
3 **if** $\{l\} \in \phi$ **then return** TSAT_MR(Simplify(l,ϕ),$\mu \wedge l$)
4 l := ChooseLiteral(ϕ)
5 **return** TSAT_MR(Simplify(l,ϕ),$\mu \wedge l$) or
TSAT_MR(Simplify($\bar{l}$,ϕ),$\mu \wedge \bar{l}$)

Figure 8.3: TSAT with model reduction.

rather than μ, is checked for satisfiability. It is assumed that ReduceModel(μ) returns a valuation $\mu' \subseteq \mu$ propositionally entailing the initial input formula.

Theorem 6 *(Soundness and completeness of* TSAT_MR*) Let ϕ be a formula.* TSAT_MR*(ϕ,$\top$) returns* TRUE *if ϕ is satisfiable, and* FALSE *otherwise.*

Proof. It suffices to note that, since $\mu' \subseteq \mu$, there are three possible cases: both μ' and μ are satisfiable, both are unsatisfiable, or μ' is satisfiable, but μ is not. In the first two cases, SatCheck(ReduceModel(μ)) coincides with SatCheck(μ); in the third case, a satisfiable valuation propositionally entailing the input formula has been found, and the algorithm terminates.

Here again is important to check that, on average, the time spent in reducing the model does not overwhelm the advantage gained by reducing it. So far, we have been experimenting with two techniques for reducing models:

Triggering: If μ contains a literal l that does not belong to any clause in the input formula ϕ, then μ propositionally entails ϕ if and only if $\mu \setminus \{l\}$ does; therefore l can be safely removed from μ. This technique, introduced in [WW99], is called *triggering*. Triggering has a linear cost in $|\mu|$ if realized, e.g. via a simple table of the occurrences of literals in ϕ.

Minimization: A better idea is to remove as many redundant constraint literals as possible. This can be done by recursively eliminating from μ one

constraint literal l at a time such that for each clause C containing l, there exists another literal l' in $\mu \cup C$. Minimization can be done in linear time in the size of the input formula ϕ provided that a data structure associating to each literal l the clauses of ϕ whom l belongs to is available.[1]

Example 6 *Consider again ϕ_{Elf}; in this case, a possible valuation found by* TSAT_MR *is* $\mu = \{p_1, VPred = I_{RR}, p_2, VPred < I_{RR}+1, VenI' = VenI, p_4, p_3, p_5, VenI' + 2 = VenI\}$. *A reduced version of it, according to minimization, is* $\mu' = \{p_1, VPred = I_{RR}, p_2, VPred < I_{RR} + 1, p_3, p_4, p_5, VenI' + 2 = VenI\}$, *obtained from μ by removing the constraint literal $VenI' = VenI$. Further, while μ is unsatisfiable, μ' is not.* $\star$

8.1.4 Best Reason Detection

So far, we have discussed how to extend a SAT solver in order to obtain a decider for SL, focusing in particular on SAT solvers based on DLL. Our motivation for this has been that most of the state-of-the-art complete SAT solvers are based on DLL. However, such solvers extend the basic DLL procedure in different ways in order to be more effective on different classes of problems. Broadly speaking, we can divide such solvers in two categories, following the distinction that is usually made in the SAT competition [LS03]:

- those designed for real-world problems, e.g. zchaff [MMZ+01], the winner of the last SAT competition in this category. The features of these solvers are that they have a fast-to-compute heuristics, a simple but efficient pruning mechanism based on unit-propagation, and a sophisticated backtracking mechanism based on back-jumping and learning (see [MMZ+01]).
- those designed for solving difficult either randomly generated or handmade problems, e.g. kcnfs [DD04] and March_eq [HM04] the winners of the last SAT competition in these categories. These solvers have a complex-to-compute heuristics, sophisticated pruning mechanisms significantly extending unit-propagation, and a simple but efficient backtracking mechanism without learning.

The modifications needed in order to obtain a SAT-based solver for SL can be done along the lines so far outlined if we start from a solver without back-jumping and/or learning. Still, in case we want to use a backtracking schema based on learning, whenever FALSE is returned, a "reason" for the failure has to be computed. Intuitively, whenever we are backtracking from a valuation μ, a reason is a subset μ' of μ such that any valuation extending μ' will fail. While backtracking, these reasons μ' are used in order to back-jump over the literals which are not in μ'. Further, if the solver uses learning, the clause $\vee_{l \in \mu'} \overline{l}$ is (temporarily) added to the input set of clauses in order to avoid the future exploration of valuations extending μ'.

Thus, in order to use SAT solvers with learning, it is not enough for SatCheck(μ) to return FALSE when μ is not satisfiable. Indeed, SatCheck(μ) must also compute a reason for such a failure, i.e. an unsatisfiable subset μ' of μ. One such set is obviously μ itself. However in order to try to maximize the advantages of

[1] Notice that the clauses eventually added by IS_n need not be taken into account since valid.

learning, it is important that μ' be as "small" as possible with respect to some ordering relation on valuations. We have found useful to focus on some specific notions of minimality. In particular we will consider valuations

1. minimal under set inclusion,
2. of minimal cardinality, and
3. minimal with respect to the ordering with which μ has been built during the search. A valuation minimal in this sense will be said to be "shallowest" in the sequel.

Intuitively, there is no point in returning a reason which is not minimal under set inclusion: If we unnecessarily include a literal l in the reason, this may lead to branch on $\bar{l}$, and such branch is bound to fail. Among the reasons which are minimal under set inclusion, those with minimal cardinality have the further advantage that, once added to the input formula because of learning, they prune a larger portion of the search space. Finally, while backtracking from a valuation μ, and even returning a reason μ' with minimal cardinality, it may still be the case that the next branch being explored is deemed to fail. In fact, μ may still contain a shallowest reason.

Example 7 *Consider Example 1 once again and assume that the heuristics is such to first set p_1, forcing also $VPred = I_{RR}$ by unit propagation, then $\neg p_2$, forcing $\neg(VPred < I_{RR} + 1)$, and then $VenI' = VenI$, p_3 and p_5, this last one forcing also $VenI' + 2 = VenI$. The corresponding valuation $\{p_1, VPred = I_{RR}, \neg p_2, \neg(VPred < I_{RR} + 1), VenI' = VenI, p_3, p_5, VenI' + 2 = VenI\}$ propositionally entails ϕ_{Elf} but is unsatisfiable. The standard procedure detects that μ is unsatisfiable, but it backtracks only up to the choice of p_5, which is not involved in the unsatisfiability of μ; then a whole search branch is explored, which is totally useless, since the assignment still contains both $VPred = I_{RR}$ and $\neg(VPred < I_{RR} + 1)$, which are responsible of the contradiction. The same, even worse, goes for the choice of p_3.*

On the other hand, if reason detection is enabled, upon detection of the unsatisfiability of μ, a reason is found, backtracking starts up to a point where the contradiction corresponding to the reason is solved. In our example, there are two possible minimal reasons, namely $\xi = \{VPred = I_{RR}, \neg(VPred < I_{RR} + 1)\}$ and $\xi' = \{VenI' = VenI, VenI' + 2 = VenI\}$. Both ξ and ξ' are minimal under set inclusion of minimal size. However, ξ is the shallowest. Indeed, if the reason is set to ξ, backtracking will stop at the choice point where $\neg p_2$ was chosen. Also notice that, assuming the reason being returned is ξ', backtracking will stop at the choice $VenI' = VenI$: However, the following search is bound to fail given that the valuation will still contain ξ. $\star$

The above example and discussion seems to point out that a reason of minimal size is better than a reason minimal under set inclusion, and that the shallowest reason is better than a reason of minimal size. Indeed, the shallowest reason tries to remove as soon as possible the unsatisfiability from the valuation built so far. However, despite the "smartness" of the reason being returned, there is no guarantee whatsoever that the tree being explored with a "smart" reason mechanism will be smaller than the tree explored with another reason mechanism. As [Pro93a] pointed out, it may be the case that the a-priori known

fruitless exploration of a branch will lead to a failure and the discovery of a reason causing a long jump to the top of the search stack. To this extent, a simple implementation of SatCheck(μ) returning μ as reason whenever μ is not satisfiable, can turn out to be more effective than other implementations, at least in some cases. However, trivially, a solver with back-jumping and/or learning can never explore more nodes than a solver with backtracking, assuming, e.g. a lexicographic branching heuristics.

The first SAT-based solver for SL using a backtracking schema with learning has been proposed in [ABC+02]. However, in that paper, there is no indication about how the reason is computed when SatCheck(μ) fails.

8.2 Related work

Several systems tailored for SL, employing different approaches and techniques, have been built and tested along the years. We now give an overview of them, highlighting the pros and cons of each one and chronologically reviewing the techniques introduced by each one.

SK [SK98, SK00]. The procedure SK has been the first dealing with a significant fragment of SL. Its main features are the combined usage of *forward-checking*, back-jumping and Minimum Remaining Value (MRV) heuristics. Forward-checking works by checking whether the valuation built so far entails either a literal or its negation, for each literal not yet in the valuation. This actually reduces the search space, at the price of performing a potentially large number of useless satisfiability checks, namely those for which the check detects satisfiability. SK is also able to detect conflict sets and to improve on backtracking via a technique similar to back-jumping. MRV is used to choose literals which appear in disjunctions with the smallest number of unassigned disjuncts—in the case of the DTP, to which SK is limited, this actually boils down to unit propagation.

The main difference between SK and SAT-based procedures lies in the way valuations propositionally entailing the input formula are searched. In fact, SK is based on *syntactic branching*: given a disjunction $l \vee l'$, first l is added to the current valuation, and, upon failure, l' is considered. As explained below, this type of search may lead to the exploration of search space already explored.

Tsat [ACG99]. Tsat was the first application of the SAT-based approach to SL. The system employs a branching schema now known as *semantic branching*. Unlike syntactic branching, semantic branching selects a not yet assigned literal l, and considers in turn the case in which l is true and the case in which l is false. Notice that in the second case, the conjunction of $\bar{l}$ with $(l \vee l')$ forces the assignment of l' by unit propagation: As already observed in [D'A92], syntactic branching may lead to redundant exploration of parts of the search space, which semantic branching avoids. The following example, adapted from [ACG99], clearly illustrates this issue.

Example 8 *Let ϕ be a formula including the following clauses:*

$$\begin{array}{lcl} x_1 - x_2 \leq 3 & \vee & x_7 - x_8 \leq 20 \\ x_1 - x_3 \leq 4 & \vee & x_4 - x_3 \leq -2 \\ x_2 - x_4 \leq 2 & \vee & x_3 - x_2 \leq 1 \\ & \vdots & \end{array}$$

Let $\phi(i,j)$ denote the j-th disjunct of the i-th disjunction displayed in ϕ; for example, $\phi(1,2)$ is $x_7 - x_8 \leq 20$. Assume that the dots stand for further (possibly many) unspecified clauses such that no satisfiable extension of the valuation $\{\phi(1,1), \phi(2,1)\}$ exists.

Consider the behavior of syntactic versus semantic branching when $\{\phi(1,1), \phi(2,1)\}$ is the valuation built so far. Since no satisfiable extension of it exists, after some search, failure is necessarily detected; both procedures backtrack and remove $\phi(2,1)$ from the current valuation.

Now syntactic branching goes on with the valuation $\{\phi(1,1), \phi(2,2)\}$, whereas semantic branching proceeds with $\{\phi(1,1), \neg\phi(2,1)\}$ which leads immediately, via unit propagation, to $\{\phi(1,1), \neg\phi(2,1), \phi(2,2)\}$.

Working with the latter valuation rather than with the former may lead to considerable savings: assume that both procedures extend the valuation with $\phi(3,1)$; since $\{\phi(1,1), \neg\phi(2,1), \phi(2,2), \phi(3,1)\}$ is unsatisfiable, semantic branching immediately backtracks and considers $\phi(3,2)$, whereas syntactic branching may waste a big amount of resources in the vain attempt of finding a satisfiable extension of $\{\phi(1,1), \phi(2,2), \phi(3,1)\}$.

$\star$

Semantic branching was shown in [ACG99] to dramatically improve the performance with respect to SK, up to one order of magnitude on randomly generated DTPs.

In Tsat, also IS_2 was introduced, gaining to the system another order of magnitude in performance. This despite the fact that, to enumerate valuations, Tsat adapted a rather simple SAT solver, due to Böhm [BB92], which did not employ any modern optimization such as back-jumping and learning. Satisfiability checking used *lp_solve* v2.2 [Ber97], which provided a free implementation of the Simplex method.

CSPi [OC00]. CSPi features an essentially CSP-based solution schema, implementing an efficient incremental procedure for forward-checking. Semantic branching is used, showing results that are better than Tsat on small instances, and comparable on bigger ones. Notice that performance, up to [OC00], was measured in terms of how many calls to the satisfiability check function were done, rather than CPU time.

MathSAT [ABC+02]. MathSAT uses SIM [GMTZ01] as enumerator and a hierarchical satisfiability checker employing—in this order—equality reasoning, the Bellman-Ford method for constraints, the Simplex method for full linear arithmetic, and inequalities reasoning. The simplest solver is chosen on-the-fly, thereby obtaining both expressivity and efficiency at the same time. MathSAT also introduces a number of optimizations, among which preprocessing based

upon syntactic equivalence, enhanced early pruning, that is, early pruning conditioned upon a heuristic function, and back-jumping/learning based upon reason detection. Also, a form of model reduction is used, based upon triggering. On randomly generated DTPs, MathSAT improves the performance over Tsat in terms of CPU time. However, the gap between the two solvers decreases as the number of variables increases.

Epilitis [TP03]. Epilitis is, so far, the last CSP-based system. Epilitis is restricted to DTPs. It uses semantic branching, incremental forward checking, a MRV heuristics and *size-bounded learning* of size n [BM96]. This means that conflict clauses are retrieved and stored only if they contain less than n literals (in practice, $n = 10$ is used). Once stored, a clause is never forgotten. On randomly generated DTPs, Epilitis shows significantly better performance than Tsat in terms of CPU time, of up to one order of magnitude.

SEP [SSB02]. SEP is a back-end to the UCLID verification tool [LSB02], employing the so-called *eager* variant of the SAT-based approach. Given a formula ϕ, rather than enumerating valuations and checking them for satisfiability, SEP builds a propositional formula ϕ' whose satisfying valuations are ensured to correspond to satisfiable valuations of ϕ. The current version of SEP uses Chaff to find valuations satisfying ϕ'.

To the best of our knowledge, SEP is so far the only solver using the eager SAT-based approach to SL. SEP suffers from the fact that the size of ϕ' can be exponential in the size of ϕ. On the other hand, as reported in [SSB02], if SEP can get past the encoding phase, the problem is easy to solve for Chaff.

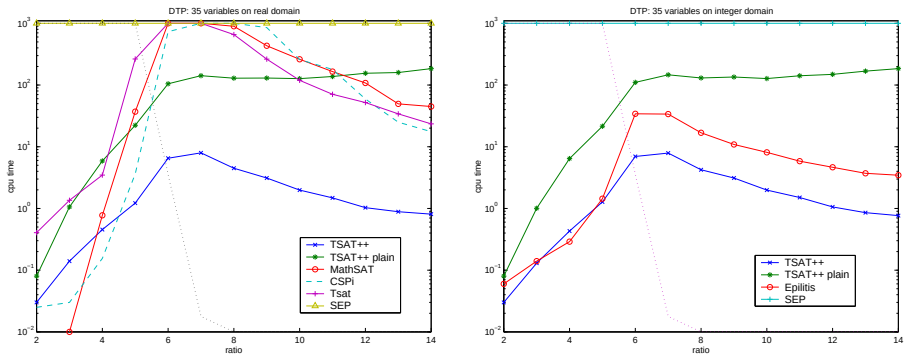
8.3 Implementation and Experimental Analysis

We have implemented the techniques described in Section 8.1 in a system called TSAT++. The system is based on a C++ implementation of an iterative version of the algorithm of Figure 7.1 featuring all optimizations presented in Section 8.1. TSAT++ uses two distinct modules for the enumeration of valuations μ propositionally entailing the input formula ϕ and for checking the satisfiability of μ.

In the current version, enumeration is done by a modified version of SIMO [GMT03]. SIMO features a number of SAT optimization techniques inspired by Chaff, among which 1-UIP learning, VSIDS heuristic, and two-literal watching [MMZ+01].

In order to assess the effectiveness of the optimizations described in Section 8.1 we have carried out a thorough experimental analysis using TSAT++ and TSAT++plain, on a wide variety of publicly available random, handmade, and real-world SL-formulae.[2] TSAT++plain is the same as TSAT++ except that IS_2, early pruning, and model reduction are disabled while best reason detection is set so to return a reason minimal with respect to set inclusion. Further, in order to evaluate the effectiveness of our system, we have compared TSAT++ with a number of rival, publicly available, and state-of-the-art systems

[2] The classification of the benchmarks in "random", "handmade", and "real-world" problems is borrowed from the SAT competitions [LS03, LS05].



(a) (b)

Figure 8.4: Performance on (a) randomly generated DTPs with 35 real valued variables and on (b) randomly generated DTPs with 35 integer valued variables. The dotted plot indicates satisfiability percentage both in (a) and in (b).

specifically designed for (a significant fragment of) SL or with a specialized satisfiability procedure for SL valuations.[3] We have thus considered the system presented in [SK98], that we will call SK; Tsat [ACG99], the predecessor of TSAT++; CSPi [OC00]; and Epilitis [TP03]. All these systems are restricted to DTPs (see Section 8.3.1). Moreover, we have considered SEP [SSB02] and MathSAT [ABC+02]. TSAT++ is as expressive as SEP and not comparable to MathSAT: while MathSAT allows for arbitrary linear constraints as atoms, it does not allow to consider the integers as domain of interpretation. After a first run, we have discarded SK, because clearly non competitive with respect to the others.

Each solver has been run on all the benchmarks it can deal with, not only on the benchmarks the solver was analyzed on by the authors. In particular, Epilitis can only handle DTPs with integer valued variables; CSPi and Tsat can only handle DTPs with real valued variables; MathSAT can handle arbitrary SL-formulae with real valued variables; SEP and TSAT++ can handle arbitrary SL-formulae with real or integer valued variables. Each solver has been run using the settings or the version of the solver suggested by the authors for the *specific* class of problems. All the experiments have been run on a Linux box equipped with a Pentium IV 2.4GHz processor and 1GB of RAM. CPU time is measured in seconds; timeout has been set to 1,000 seconds.

8.3.1 Disjunctive Temporal Problems

We start our analysis considering randomly generated DTPs as introduced in [SK98] and since then used as a benchmark in [ACG99, OC00, ABC+02, TP03]. DTPs are randomly generated by fixing the number k of disjuncts per clause, the number n of arithmetic variables, a positive integer L such that all the

[3]Notice that there exists other systems capable of handling SL, e.g., ICS [dMRSR04], CVC [SBD02], CVC-Lite [BB04], Verifun [FJOS03]. We did not include these solvers in our analysis since they are not tailored for SL. MathSAT has been included since it has a specialized satisfiability checker for SL based on Bellman-Ford.

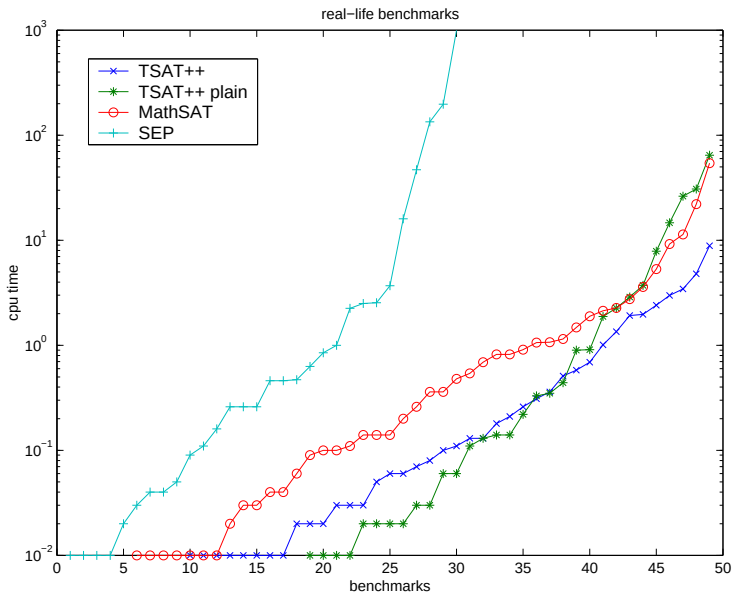


Figure 8.5: Performance on real-world problems.

constants are taken in $[-L, L]$. Then, (i) the number of clauses m is increased in order to range from satisfiable to unsatisfiable instances, (ii) for each tuple of values of the parameters, 100 instances are generated and then fed to the solvers, and (iii) the median of the CPU time is plotted against the m/n ratio. The results for $k = 2$, $L = 100$, and $n = 35$ are given in Figure 8.4: Plots (a) and (b) show the performance when the variables are real and integer valued respectively.

When $m/n \geq 6$, TSAT++ clearly outperforms the other systems, including TSAT++plain: In the peak region, the solver that is closer to TSAT++ in this domain, namely Epilitis, is a factor of 6 slower on 35 variables (cf. plot (b)). This is a very positive result, taking into account that Epilitis only works on DTP with $k = 2$, and it has been thoroughly tested and optimized on this type of problems (see [TP03]). All the other systems are about 2 orders of magnitude slower than TSAT++ in the peak region. Even more important is the fact that according to some experiments we have done, as the number of variables increases (we have experimented with problems up to 50 variables), the gap in performance between TSAT++ and the other systems also increases. For this class of problems TSAT++ has been run with early pruning and pre-processing enabled, with the best reason detection optimization set to return shortest reason, and with model reduction disabled. The role of the optimizations is fundamental for the performance on this test set: TSAT++ is more than one order of magnitude faster than TSAT++plain in the peak region.

8.3.2 Real-world problems

We have also carried out experiments on

1. the 40 post-office benchmarks introduced in [ABC+02], coming in 4 series (consisting of 7, 9, 11, and 13 instances respectively) of increasing difficulty. In these problems the domain of the interpretation is the set of real numbers.

2. the 16 hardware verification problems from [SSB02], 9 (resp. 7) of which are with real (resp. integer) valued variables.

The post-office benchmarks are bounded model checking problems for timed automata; the hardware verification suite includes scheduling, cache coherence protocol, load-store unit and out-of-order execution problems. Considering the results of MathSAT, SEP, and TSAT++ on the post-office problems, our first observation is that SEP is not competitive on these problems: on 13 of the hardest instances, SEP had a segmentation fault in 11 cases, and on the other 2 hardest instances SEP is outperformed by different orders of magnitude by TSAT++ and MathSAT. Our second observation is that TSAT++ (with IS_2 pre-processing, model reduction, and shortest reason detection) performs better than MathSAT up to a factor of 6 on *each single instance*: This is particularly remarkable given that the authors have customized a version of MathSAT explicitly for this kind of problems.[4] Considering the hardware verification problems, all of them are easy to solve (i.e. in less than 3s each) for all the three solvers, except for SEP that timeouts on one instance. Of the 9 (resp. 16) runs of MathSAT (resp. SEP and TSAT++), only 3 take more than 0.1s. These observations are confirmed by Figure 8.5, which gives the overall picture of the results for MathSAT, SEP, and TSAT++ on the 49 instances with real valued variables: The x-axis is the number of instances solved by each solver within the CPU time specified on the y-axis. The plot also shows that TSAT++plain can be faster than TSAT++ on the easy instances, i.e. those requiring less than 1s to be solved. For such problems, the overhead of the optimizations (and in particular of the pre-processing) outweighs the benefits.

8.3.3 Hand-made problems

Finally, we have considered the "hand-made" diamond problems from [SSB02]. A diamond problem is a formula ϕ that depends on a parameter $K > 0$ and such that there exists a number of unsatisfiable valuations propositionally entailing ϕ which is exponential in K. Moreover, hard instances having a single satisfiable valuation propositionally entailing them can be generated. A second parameter T is also used and it affects the number of variables and the size of the problem. Variables range over the reals.

Table 8.1 shows comparative results on the diamond problems for various settings of K and T. In particular, we considered all the settings corresponding to non trivially solvable instances reported in [SSB02]. The third column denotes whether the problem has a unique valuation propositionally entailing it; the remaining columns show CPU times for TSAT++, TSAT++plain, MathSAT, and SEP. For this class of problems TSAT++ has been run with best reason detection set to shortest reason, and with model reduction. The experimental results clearly show that TSAT++ performs best, often by orders of magnitude.

[4] As indicated by the authors, we have used this customized version of MathSAT on this class of problems.

Table 8.1: Diamond problems: "TIME" indicates that the solver does not solve the instance withing the time limit. "–" indicates that the solver exits abnormally.

K	T	unique	TSAT++	TSAT++plain	MathSAT	SEP
50	4	NO	0.00	0.02	0.05	0.12
50	4	YES	0.01	0.14	TIME	0.07
100	5	NO	0.01	0.11	0.61	1.18
100	5	YES	0.04	7.57	TIME	0.17
250	5	NO	0.08	0.76	5.40	52.20
250	5	YES	0.21	194.99	TIME	0.77
500	5	NO	0.29	4.46	21.22	742.99
500	5	YES	1.05	TIME	TIME	4.85
1000	5	NO	1.07	22.3	–	TIME
1000	5	YES	6.45	TIME	–	22.53
2000	5	NO	3.76	94.23	–	–
2000	5	YES	29.90	TIME	–	–

Instances with a unique solution are more difficult than non-unique ones, as expected, except for SEP.[5]

For this test set, it is of fundamental importance the model reduction optimization: without it, TSAT++ performance is significantly worse, up to the point that problems which are solved in 1 second by TSAT++, are not solved without model reduction within the time limit.

8.3.4 Hardware verification problems

Table 8.2 reports on the performance obtained on hardware verification problems. These unsatisfiable formulae, converted to SL from CLU, represent bounded model checking of safety properties for hardware models such as an out-of-order microprocessor design, a cache coherence protocol, a DLX pipeline and the memory unit of the Elf processor (see [BLS02] and citations therein for a more detailed explanation).

The formulae are represented as DAGs, and are converted to CNF by letting each node of the DAG correspond to a new Boolean variable defined as in the DAG; each definition, using the if-and-only-if operator, generates then at most three clauses. The DAGs contain up to 110K nodes and the resulting CNFs up to 330K clauses. Variables range over the integers, so MathSAT has been excluded from the comparison, since it does not support reasoning on this domain. Also CVC has a weak support for the integers[6] and it has anyway shown a poor performance on an initial round of tests, so it has been excluded too. Performance of SEP with conjunction matrix is not significantly differently from the plain configuration, so it is not reported.

[5]Following a suggestion by Ofer Strichman we have also tried SEP with an option that disables the use of a specialized data structure called "conjunction matrix" [SSB02]. This can have a dramatic impact on SEP: Some problems that are solved with conjunction matrix within the time limit are not solved without, and vice versa.

[6]Personal communication by Sergey Berezin.

Table 8.2: Hardware verification problems.

Instance	Lazy	Eager
	TSAT++	SEP
22s.smv	0.05	–
25s.smv	0.02	–
cache.inv8	0.02	–
cache.inv10	0.11	–
cache.inv12	75.08	–
dlx1c	TIME	–
elf.rf7	0.02	3.60
elf.rf8	0.74	MEM
elf.rf9	13.92	TIME
ooo.rf7	7.42	MEM
ooo.rf8	231.80	TIME
ooo.rf9	TIME	TIME
q2.10	6.40	–
q2.12	44.28	–
q2.14	230.69	–

Consider the Table. SEP looks quite prone to implementation problems, although on one instance (elf.rf7) it could carry the computation to the end; on two more instances (22s.smv and 25s.smv) it could finish the encoding phase (then the CNF converter crashed), showing timings around 19 seconds. Looking at columns 2 of the Table, one sees that TSAT++ does not incur in any crash, and performs indeed much better than SEP.

Discussion

Let us remark a few points about these last benchmarks:

1. although it is not tuned on this class of problems, TSAT++ clearly shows the best performance. Also, it is the most robust system.

2. a further closer look at TSAT++'s output reveals that the branching heuristics employed in TSAT++ is definitely not optimal for the hardware verification domain, and that getting a better one could make the solver much faster.

In order to test this latter conjecture, we have re-done the hardware verification tests with TSAT++ , but trying to give to the systems some domain-dependent information. We have experimented some static orderings of the input variables with TSAT++, again based upon the original structure of the DAGs. This has resulted in TSAT++ actually *managed to solve* the dlx1c instance in about 577 seconds.

8.4 Summing up

In this part, we have presented the basic procedure from [ACG99] along with some key optimizations, and we have presented a new solver, TSAT++. We have also shown how it is possible to check the satisfiability of valuations involving constraints of the form $x - y < c$ using BF. An extensive comparative experimental analysis shows that our solver TSAT++, built along the lines described in this Chapter, is currently the state-of-the-art on various classes of problems, including randomly generated, hand-made, and real-world instances.

Part IV

Safe Planning via Answer Set Programming

Chapter 9

Introduction

In this Part we introduce a new approach for implementing Safe Planning that rely on a compiler that create a logic program to be solved under the answer set semantic; we then present a solver for Answer Set programming that is a joint work between DIST and the University of Texas at Austin; and finally we show some recent results on the relation between SAT and Answer Set programming.

In the Chapter, we briefly review the basic concepts and definitions about Answer Set Programming and then we show how to modify the compiler in order to create a logic program (instead of a propositional formula and/or a SL formula)

9.1 Answer Set Programming

Answer Set Programming (ASP) is a new programming paradigm [GL88b, GL91, MT99, SNT02] for knowledge representation and reasoning that has emerged in the last years. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions.

A rule element is an expression of the type p or *not* p, where p is a literal and the symbol not is the *negation as failure* operator that is different different from the classical negation $\neg$, and is the "reason" of the non-monotonicity of the logic.

A rule is an expression of the form:

$$Head \leftarrow Boby \tag{9.1}$$

where $Head$ and $Body$ is a finite set of rule elements. A rule is called *constraint* if $Head = \emptyset$; it is called *disjunctive* if the $|Head| > 1$.

If

$$Head = \{p_0, \ldots, p_k, not\ p_{k+1}, \ldots, not\ p_l\}$$

and

$$Body = \{p_{l+1}, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$$

for $n \geq m \geq l \geq k \geq -1$ then we can write the rule as

$$p_0; \ldots; p_k; not\ p_{k+1}; \ldots; not\ p_l \leftarrow p_{l+1}, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n \quad (9.2)$$

We can also drop the $\leftarrow$ in (9.2) if the body is empty ($n = m = l$).
A *program* is a (finite) set of rules.

The notion of an answer set (AS) is defined first for programs that do not contain negation as failure ($l = k$ and $n = m$ in every rule of the type (9.2) of the program). Let Π be such a program, an let X be a consistent set of literals. We say that X is closed under Π if, for every rule (9.1) in Π, $Head \cap X \neq \emptyset$ whenever $Body \subseteq X$. We say that X is an *answer set* for Π if X is minimal among the sets closed under Π (relative to set inclusion).

For instance, the program that contains the two rules

$$p; q \leftarrow,\ \neg r \leftarrow p \quad (9.3)$$

has two answer sets: $\{p, \neg r\}$ and $\{q\}$. If we add the constraint:

$$\leftarrow q$$

to (9.3), we will get the program whose only answer set is $\{p, \neg r\}$. The last example illustrates a general property of a constraint: Adding a constraint to a program affects its collection of answer sets by eliminating the answer sets that "violate" the constraint. This is also true for program with negation as failure.

To extend the definition of an answer set to programs whit negation as failure, take any program Π, and let X be a consistent set of literals. The reduct Π^X of Π related to X is the set of rules

$$p_0; \ldots; p_k \leftarrow p_{l+1}, \ldots, p_m$$

for all rules (9.2) in Π such that X contains all the literals $p_{k+1}, \ldots, p_l$ but does not contain any $p_{m+1}, \ldots, p_n$. Thus Π^X is a program without negation as failure. We say the X is an answer set of Π if X is an answer set for Π^X.

For instance, the reduct of $\Pi' = p \leftarrow not\ q$ relative to $\{p\}$ consists of one rule $p \leftarrow$. Since set $\{p\}$ is an answer set for this reduct, it is an answer set for Π'. The reduct of Π' relative to $\{p, q\}$ is the empty set. Since set $\{p, q\}$ is not an answer set for this reduct, it is not an answer set for Π'.

The complexity of finding one answer set of a logic program of the general type (9.2) is $\sum_p^2$-complete.

In this thesis we will focus of *non-disjunctive* (normal) logic programs: Logic programs with $|Head| \leq 1$ and with $k = l$ (no *not* operator in the Head). In this case a rule has the form:

$$p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n \quad (9.4)$$

where P is a set of atoms, $p_0 \in P \cup \{\bot\}$, $\{p_1, \ldots, p_n\} \subseteq P$. The case $p_0 = \bot$ (false) corresponds to the case $Head = \emptyset$ cited above. Checking the consistency of a non-disjunctive logic program (i.e., the existence of an answer set if any) is a NP-complete problem; it is "simpler" that the disjunctive case.

For some purposes, the unavailability of disjunctive rules is an essential limitation. Nevertheless, a disjunctive rule of the form

$$p \; ; \neg p \leftarrow \tag{9.5}$$

can be always replaced by a pair of rules similar to (9.4):

$$p \leftarrow not \; \neg p,$$
$$\neg p \leftarrow not \; p$$

Furthermore, the use of classical negation can be always avoided, because a negative literal $\neg p$ can be replaced by a new atom $p^{'}$ provided that the constraint

$$\leftarrow p, \; p^{'}$$

is added to the program, and this is the reason why in (9.4) p_i are atoms.

Using these last two tricks, (general) disjunctive logic programs can be turned into logic programs containing neither disjunction nor $\neg$. At the price of adding rules and/or adding new atoms, it is possible to use system (for non-disjunctie logic programs) that have some computational advantages (at least from a theoretical point of view).

9.2 From SAT formulas to logic programs

As we will see in more details in Chapter 12, there is a simple a modular translation from SAT (in CNF) to a logic program. We anticipate the translation here.

Consider a formula Γ, as a set of clauses. To each clause $C = \{l_1, \ldots, l_l\}$ ($l \geq 0$) in Γ we associate the rule $sat2lp(C) = \bot \leftarrow not \; l_1, \ldots, not \; l_l$. The *translation of* Γ, denoted with $sat2lp(\Gamma)$, is $\cup_{C \in \Gamma} sat2lp(C) \cup \cup_{p \in P} \{p \leftarrow not \; p', p' \leftarrow not \; p\}$, where, for each atom $p \in P$, p' is a new atom associated to p.

All the details that we have presented in Chapter 3, using the Tseitin's clause form transformation [Tse70], can be easily recast as a logic program. We will see in details the transformation in Chapter 12.

Even if related, ASP and SAT are different in many ways. This is the basic reason why can be interesting to evaluate also this other approach for Safe Planning.

As we have seen, there is a modular translation from SAT to ASP such that each solution to a SAT problem is also an answer set of the related logic program, but for the contrary it is not always the case (unless the logic program is "tight"): If we translate a logic program into a propositional formula, in general there exist models of the propositional theory that do not correspond to any answer set of the logic program.

Moreover, SAT reasoning is a classical monotonic reasoning: Adding rules (in terms of facts, clauses) to a formula only cause a reduction in the number of the models (that now are a subset of the models of the formula before having added the new rules). On the other hand, reasoning under answer set semantic is a form of non-monotonic reasoning: Adding a rule to a logic program can result in more solutions (answer sets) as well as different solutions w.r.t. the program without these new rules.

Finally, from a representational point of view, some problems are better represented as logic programs (ex. problems related to graphs), while other are better represented in propositional logic (ex., formal verification).

It is also important to notices that in general the encoding of a problem as logic program results in a more compact formula. On the other hand, very specialized and efficient solver for propositional reasoning exist, and they have had a tremendous boost in the last years; while decision procedures for logic programs (under answer set semantic) is not a very mature area.

In order to possibly overcame the last point, we have chosen an approach to solve logic programs mainly based on reasoning on propositional formulas (SAT-based reasoning).

9.3 From logic programs to propositional formulas

As we will see in more details in Chapter 12, there is also a simple (non-modular) translation from a logic program ro a propositional formula (not in CNF). We anticipate the translation here, and in Chapter 12 we will further see how to convert it to CNF.

If p_0 is an atom or the symbol $\perp$, the *completion of* Π *relative to* p_0 is the formula

$$p_0 \equiv \bigvee (p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n)$$

where the disjunction extends over all rules (9.4) in Π with head p_0. The *completion* $Comp(\Pi)$ of Π consists formulas $Comp(\Pi, p_0)$ for each atom p_0 and the symbol $\perp$.

Chapter 10

From DLL to a decision procedure for ASP

In this Chapter we will see how to turn a decision procedure for SAT (DLL algorithm) to a decision procedure able to find solutions of a logic program under the answer set semantic.

Fages [Fag94] showed that if a program Π is "tight" then its answer sets are in one-to-one correspondence with the models of its completion [Cla78]. If the completion is converted to a set of clauses Γ, state-of-the-art SAT solvers can be used as answer set generators. Since the size of Γ is at most twice the size of Π, and has at most m new variables (where m is the number of rules in the logic program) this is considered a viable and efficient approach. Fages' result was then generalized to include programs with infinitely many rules [Lif96], programs tight "on their completion model" [BEL00], and programs with nested expressions in the bodies of the rules [EV03]. Still, these results do not apply to the whole class of logic programs. It is well known that each answer set corresponds to a model of its completion, but that the vice-versa in general is not true.

Ben-Eliyahu and Dechter [RR96] gave a translation from a class of disjunctive logic programs (which includes the one that we are focusing) to SAT. However, their translation may need $O(n^2)$ new variables and $O(n^3)$ new clauses (where n is the number of atoms in the logic program). Lin and Zhao [LZ03a] recently introduced a translation which needs the introduction of $O(n^2 + m)$ new variables and $O(n \times m)$ new clauses. In practice, the number of variables or clauses in the resulting formula can be prohibitive.[1]

The only reduction to SAT which does not need extra variables was proposed by Lin and Zhao [LZ02]. The drawback of this reduction is that the resulting formula may blow-up in space. Still, system ASSAT based on such reduction outperforms state-of-the-art ASP systems like SMODELS [Nie99, Sim00] and DLV [ea98] on many interesting problems.

In this Chapter, the question that we positively answer is: Is it possible to

[1] Lin and Zhao [LZ02] report that the grounding of a program corresponding to the computation of an Hamiltonian path in a complete graph with 50 nodes, produces a program with 5000 atoms and 240000 rules. For this problem, the new clauses will be more than a billion.

build an efficient SAT-based answer set generator that (i) deals with any (non-disjunctive) logic program, (ii) works on a SAT formula without additional variables (except for those introduced by the clause form transformation), and (iii) is guaranteed to work in polynomial space? We present a procedure, called ASP-SAT (based on the DLL algorithm), having the above three but also others features. In order to show its effectiveness, we integrated ASP-SAT in CMODELS,[2] and run a wide comparative analysis with other state-of-the-art systems. The results show that our procedure has a clear edge over its competitors.

It is well known that if X is an answer set of Π, then X satisfies $Comp(\Pi)$, while the converse is not necessarily true. Lin and Zhao [LZ02] proved that to have a one-to-one correspondence between the answer sets of Π and the models of its completion, we have to consider also the loop formulas (propositional formulas that kept the information why a model of the completion is not an answer set) of Π. To state this formally, we need the following definitions.

The *dependency graph* of a program Π is the directed graph G such that the vertexes of G are the atoms in Π, and G has an edge from p_0 to $p_1, \ldots, p_m$, $p_i \in P, 1 \leq i \leq m$, for each rule (9.2) in Π, $p_0 \neq \bot$. A *loop* of Π is a set L of atoms such that for each pair A, A' of atoms in L there is a path from A to A' in the dependency graph of Π, whose intermediate nodes belong to L.

Given a loop L, we define $R(L)$ to be the set of formulas

$$p_1 \wedge \ldots \wedge p_m \wedge \neg p_{m+1} \wedge \ldots \wedge \neg p_n$$

for all rules (9.2) in Π, with $p_0 \in L$ and $\{p_1, \ldots, p_m\} \cap L = \emptyset$. The *loop formula associated with* L is

$$\bigvee L \supset \bigvee R(L)$$

where $\bigvee L$ denotes the disjunction of the elements in L, and similarly for $\bigvee R(L)$.

Theorem 1 *[LZ02] Let Π be a program, $Comp(\Pi)$ its completion, and $LF(\Pi)$ be the set of loop formulas associated with the loops in Π. For each set of atoms X, X is an answer set of Π iff X is a model of $Comp(\Pi) \cup LF(\Pi)$.*

10.1 SAT-Based Answer Set Solvers

Given Theorem 1, it is clear that if the dependency graph of a program Π has no cycles (in this case we say that Π is *tight*), then the models of $Comp(\Pi)$ are also answer sets of Π. Thus for tight programs answer set systems can use SAT solvers as search engines. CMODELS used this approach to compute answer sets for tight programs.

A SAT-based system working also on non tight programs was proposed by Lin and Zhao [LZ02]. This system, called ASSAT, performs the following steps:

1. Compute $Comp(\Pi)$ and convert it to a set of clauses Γ.

2. Find a model X of Γ by using a SAT solver. Exit with failure if no such model exists.

[2] http://www.cs.utexas.edu/users/tag/cmodels

3. Compute the set of atoms $X^- = X - Cons(\Pi^X)$, where $Cons(\Pi^X)$ is the set of atoms derivable from the reduct of Π relative to X.

4. If $X^- = \emptyset$, then return X.

5. Otherwise, add the clauses corresponding to the loop formulas of all the maximal (under subset inclusion) loops in X^- to Γ, and go to step 2.

In their article Lin and Zhao showed that the above simple procedure can outperform rival systems, often by orders of magnitude. Still ASSAT has the following two drawbacks:

1. ASSAT is not guaranteed to work in polynomial space, since there can be exponentially many loops in a program Π. If we assume that each loop formula cannot be derived from the others and $Comp(\Pi)$, then

   - If Π has an answer set, then ASSAT performance on Π depends on how lucky the system is in generating the right model first. In the best case it generates an answer set first. In the worst case it blows up in space.
   - If Π has no answer set, then ASSAT blows up in space. In fact, adding and keeping already added loop formulas is essential to guarantee that the SAT solver does not return an already computed model, and ultimately to guarantee ASSAT termination.

2. Considering two successive calls of the SAT solver, the computation done for finding the first model is completely discarded. Thus some branches of the search tree may get computed many times.

Further considering the task of computing all answer sets of program Π, there are two ways for doing it in ASSAT:

1. Compute $Comp(\Pi) \cup LF(\Pi)$ and then call a SAT enumerator, i.e., a SAT solver able to return all the models of a propositional formula, e.g., MCHAFF; or

2. In order to avoid the generation of the same model X, modify ASSAT procedure in step 4. by adding one or both of the clauses in $\{\bigvee_{A \in X} \neg A, \bigvee_{A \notin X} A\}$ to Γ and going back to step 2. Indeed, the obvious solution would be to add the clause $\bigvee_{A \in X} \neg A \vee \bigvee_{A \notin X} A$. However, it is well known that logic programs of the kind we focus satisfy the following *anti-chain property*: If X is an answer set, no strict subset or superset of X is an answer set.

The first approach is unfeasible if there are many loop formulas. The second approach is also unfeasible when there are many answer sets.

The above drawbacks can be eliminated if we do not use a SAT solver as a black-box. Instead, we can take advantage that all the state-of-the-art complete SAT solvers are based on the DLL procedure. Given a program Π, we may first compute the completion of Π, and then

- *generate* models of $Comp(\Pi)$, and
- *test* whether the generated models are answer sets of Π.

```
DLL(Γ, S)
  if Γ = ∅ then exit with true;
  if ∅ ∈ Γ then return false;
  if {l} ∈ Γ then return DLL(assign(l, Γ), S ∪ {l});
  A := an atom occurring in Γ;
  DLL(assign(A, Γ), S ∪ {A});
  DLL(assign(¬A, Γ), S ∪ {¬A}).
```

Figure 10.1: The DLL procedure

Consider DLL procedure represented in Figure 10.1. This is the same as in Figure 2.1 and with a similar description. We report again the description for convenience of the reader. Here l denotes a literal; Γ a set of clauses; S an *assignment*, i.e., a consistent set of literals. Given an atom A, $assign(A, \Gamma)$ is the set of clauses obtained from Γ by removing the clauses to which A belongs, and by removing $\neg A$ from the other clauses in Γ. $assign(\neg A, \Gamma)$ is defined similarly. In the initial call to DLL, Γ is the set of clauses of which we want to compute a model, and S is the empty set. DLL$(\Gamma, \emptyset)$ returns FALSE if Γ is not satisfiable, and TRUE otherwise.

Consider a program Π. Given DLL, we can obtain a SAT-based answer set generator for Π, by

1. modifying the first line of DLL in the figure, by substituting "exit with TRUE" with "return *test*(S, Π)", a new function which has to return
   - TRUE, if the set of atoms in S is an answer set of Π, and
   - FALSE, otherwise.
2. defining a function ASP-SAT(Π), that calls DLL$(\Gamma, \emptyset)$, where Γ is the clausified *Comp*(Π).

Notice that *test*(S, Π) just checks whether the set of atoms in S is an answer set of Π. Given that S may be a partial assignment, the question is whether it is possible that the set of atoms in any total assignment extending S (i.e., which is a superset of S) be an answer set. This is not the case, as established by the following proposition. As a side remark, the proposition generalizes the above stated anti-chain property.

Theorem 2 *Let Π be a program, X, X' be two set of atoms satisfying Comp(Π). If $X \subset X'$ then X' is not an answer set.*

ASP-SAT(Π) returns TRUE if and only if Π has an answer set. Moreover, ASP-SAT (i) performs the search on *Comp*(Π) and thus does not introduce any extra variables except for those needed by the clause form transformation; (ii) is guaranteed to work in polynomial space; (iii) can deal with both tight and non tight problems: In the case of tight problems, each generated model of *Comp*(Π) corresponds to an answer set and thus ASP-SAT behaves as a standard SAT solver run on *Comp*(Π).

Compared to ASSAT, ASP-SAT is guaranteed to work in polynomial space and has also the following advantages:

- It is easily modifiable to return all the answer sets: The only thing that we have to do is to modify $test(S, \Pi)$ in order to (i) print the set of atoms in S if it is an answer set, and (ii) return FALSE.
- No computation is ever repeated.

On the other hand, ASSAT advantage over ASP-SAT is that in the former the SAT solver is used as a black-box without any need of even minor modifications.

Compared to other answer set solvers like SMODELS and DLV, ASP-SAT has the advantage of being SAT-based, and thus it can leverage on the great amount of knowledge available in SAT. For instance, we are not aware of any (not SAT-based) answer set solver using the analogous of two-literal watching data structures for pruning the search tree.

Chapter 11

Answer Set solver Cmodels2

11.1 Integration in Cmodels

We have integrated our implementation of ASP-SAT in CMODELS. ASP-SAT was implemented on top of the SIMO system [GMT03], a MCHAFF-like SAT solver. As we saw in Subsec. 2.4.2, SIMO features a two-literal watching data structure, 1-UIP learning and VSIDS heuristics (see [MMZ+01]). However, it does not feature the low level optimizations of MCHAFF, and thus it is within a factor of 3 slower than MCHAFF. Our implementation of ASP-SAT inherits all the above features, and in particular the learning mechanism. It is thus of fundamental importance that each call to $test(S, \Pi)$ does not just return FALSE (resp. TRUE) if the set $atoms(S)$ of atoms in S is not (resp. is) an answer set of Π. Indeed,

- if $atoms(S)$ is not an answer set of Π, we have to return a "reason" for this, i.e., a subset S' of S such that for any total assignment S'', (i) extending S' and (ii) satisfying $Comp(\Pi)$, $atoms(S'')$ is guaranteed to be not an answer set of Π. By Proposition 2, one such set S' is $atoms(S)$. However, in order to take advantage of the backjumping and learning mechanisms in SIMO, it is important that S' be as small as possible. For computing such S', the $test(S, \Pi)$ procedure

  1. computes the loop formulas associated with the loops in $atoms(S) - Cons(\Pi^{atoms(S)})$,
  2. returns a subset of S which falsifies one of the loop formulas computed in the previous step.

  With such a simple procedure, we are able to compute reasons which are often less than 1% of the size of S.

- analogously, if $atoms(S)$ is an answer set of Π and we want to compute all answer sets of Π, we have to return a subset S' of S such that for any assignment S'' extending S', $atoms(S'')$ is guaranteed to be either not an answer set of Π, or an already computed answer set. Our simple implementation returns $atoms(S)$.

The reason returned by $test(S, \Pi)$ will be learned (i.e., added to the set of input clauses) and eventually deleted according to some criteria from the SAT literature. The added clauses avoid the repetition of the same mistakes in different part of the search tree.
At the same time we modify SIMO in two ways:

1. To call the $test(S, \Pi)$ routine whenever a set of literals S satisfying $Comp(\Pi)$ is generated.

2. While branching, SIMO will first choose among the literals whose atom occurs in the negation as failure operator *not*; when all such atoms are assigned, branching will first try to assign the remaining atoms to FALSE. The intuition behind this few-lines modification in SIMO code, is that we should first get to a set of clauses corresponding to a program Π without negation as failure, and then we should try to satisfy the remaining set of clauses by assigning the fewest possible atoms to true.

To appreciate the behavior of our procedure, assume to have a program Π consisting of the two rules[1]

$$A_i \leftarrow A_{i+1} \qquad\qquad A_{i+1} \leftarrow A_i$$

for each $i \in \{0, 2, \ldots, 2k\}$. $Comp(\Pi)$ has 2^k models, while the only answer set is the empty set:

- The naive implementation of $test(S, \Pi)$ which returns $atoms(S)$, may still generate 2^k satisfying assignments.

- The implementation of $test(S, \Pi)$ which returns the subset of S falsifying one of the loop formulas in $atoms(S) - Cons(\Pi^{atoms(S)})$, may generate at most k satisfying assignments.

- With the modification in the heuristics, the first satisfying assignment that is generated has all the atoms assigned to false, corresponding to the only answer set of the program.

The integration of ASP-SAT in CMODELS posed some challenges related to CMODELS expressivity. In fact CMODELS uses LPARSE[2] as front-end and thus its input may contain cardinality expressions (also called "constraint literals" in LPARSE manual[3]) and choice rules – two constructs widely used in answer set programming.[4] Operationally, CMODELS performs the following steps:

1. Simplifies the given LPARSE program, performing operations similar to those involved in the operation of SMODELS.

2. Eliminates cardinality expressions by introducing auxiliary atoms and rules. Eliminates choice rules in favor of *nested expressions* in the sense of [Lif99]. This is done using a procedure defined in [PV05].

[1] In this paragraph, for simplicity, we assume that the clauses corresponding to the reasons returned by $test(S, \Pi)$ are stored and never deleted.

[2] LPARSE is a grounder for logic programs. It has a nice and rich input language, and it grounds in output the input problem.

[3] `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`

[4] The input can also contain general weight expressions ("weight literals") However, optimize statements (see LPARSE manual) are not allowed.

3. Verifies that the resulting program with nested expression is tight: the definition of tightness is generalized to such programs in [EV03].

4. Forms the program's completion (see [LT84] for the definition of completion of a program with nested expressions) and calls a SAT solver.

As for CMODELS, the integration implied simply calling our procedure instead of the SAT solver. As for our procedure, we had to take into account that programs with nested expression do not satisfy Proposition 2. For instance, the program

$$A \leftarrow \textit{not not } A \tag{11.1}$$

(corresponding to the translation of the choice rule "$\{A\} \leftarrow$") has two answer sets: $\emptyset$, $\{A\}$. The violation of Proposition 2 implied two modifications in our procedure. Consider a program Π, resulting from the last of the above steps.

The first modification, is that we have to guarantee that each set S of literals in $\textit{test}(S, \Pi)$ is total. Assuming that the input set of clauses is satisfiable, SIMO always returns total assignments but in the signature of the set of clauses resulting after SIMO preprocessing. However, SIMO (like most of the SAT solvers) removes tautological clauses in the pre-processing. Tautological clauses can naturally arise during the completion process, and removing them may cause the generation of partial (w.r.t. the signature of the input program) satisfying assignments. This is not a problem if Π does not have nested expressions (see Proposition 2); it may be a problem otherwise. In fact, the completion of the program (11.1) is $A \equiv \neg\neg A$, which, once translated in a set of clauses and after the removal of tautological clauses, corresponds to the empty set of clauses. In order to be able to "generate" both the assignment $\{A\}$ and $\{\neg A\}$, we had to modify SIMO pre-processing in order to keep tautological clauses.

The second modification involves the function $\textit{test}(S, \Pi)$. Indeed, it has to consider loop formulas as defined in [LL03] for nested programs. Further, in the case $\textit{atoms}(S)$ is an answer set and we are interested in finding all answer sets of Π (since any superset or subset of the atoms in S can be an answer set of Π), $\textit{test}(S, \Pi)$ has to return the entire set S as reason.

11.2 Experimental Results

In order to evaluate the effectiveness of our procedure, called CMODELS2, we comparatively tested it against other state-of-the-art systems on a variety of benchmarks. The benchmarks we considered are both *basic* and *non basic* logic programs, the former being the ones without cardinality constraints and choice rules. The systems we considered are SMODELS version 2.27, ASSAT version 1.52, DLV release of 2003-05-16.[5] It worths remarking that while SMODELS, ASSAT and CMODELS2 use LPARSE as preprocessor, and thus can be run on the same problems, DLV does not. This explains why DLV appears only in few tables. Further, ASSAT only can deal with basic programs. Finally, for DLV we have to mention that it is a system specifically designed for disjunctive logic programs, and that very different results can be obtained depending on the specific encoding being used.

[5] A detailed description of these solvers is given in Chapter12 as well as their URL.

		Basic programs			Non basic programs	
#b	#s	SMODELS	ASSAT	CMODELS2	SMODELS	CMODELS2
8	i-1	12.32	0.80	1.19	0.81	0.47
11	i-1	71.78	2.97	4.19	2.97	1.01
8	i	40.87	0.89	2.18	1.56	1.40
11	i	71.42	3.17	4.52	3.41	1.16
8	i+1	23.35	0.96	0.97	4.99	0.31
11	i+1	107.48	3.54	3.33	5.21	0.75

Table 11.1: Blocks world: "#b" is the number of blocks. Finding one solution.

All the tests were run on a Pentium IV PC, with 1.8GHz processor, 512MB RAM DDR 266MHz, running Linux. For SMODELS, ASSAT and CMODELS2, the time taken by LPARSE is not counted.[6] Further, each system was stopped after 3600 seconds of CPU time on an instance, or when it exceeded all the available memory. In the tables, these cases are denoted with "TIME" and "MEM" respectively. Otherwise, the tables report the CPU times in seconds needed by each solver to solve the problem, or a "$-$" to denote an abnormal exit of the program. Finally, we run many more examples than those showed: for lack of space we report only the results for the instances in which at least one of the systems solved it and in more than 1 second.

11.2.1 Finding one answer set

We start our analysis considering blocks world planning problems, encoded as both basic and non basic logic programs, the latter formulation due to Erdem ([Esr02]). The results are represented in Table 11.1. In the table, (i) the numbers in the column "#b" represent the number of blocks; (ii) an "i" in the "#s" (standing for "number of steps") column means that the instance corresponds to the problem of finding a plan in "i" steps, where "i" is the minimum integer for which a plan exists. Thus, the instances with "i" and "$i+1$" in the "#s" column admit at least one answer set, while those with "$i-1$" do not have answer sets.

These blocks world planning problems are tight on their completion models [BEL00], and thus every model of the completion corresponds to an answer set. As it could be expected, SAT-based systems like ASSAT and CMODELS2 perform (sometimes significantly) better than SMODELS, both on basic and non basic programs. On basic programs ASSAT performs slightly better than CMODELS2, and this corresponds to the fact that, on average, MCHAFF is better than SIMO.

We also considered Hamiltonian circuit problems on complete graphs, using both the basic encoding of Niemela ([Nie99]), and the non basic encoding in the "benchmark problems for answer set programming systems"[7]. These problems are particularly interesting because they are non tight and have exponentially many loops. Thus, one would expect these problems to be difficult for ASSAT,

[6] Adding the times of LPARSE will not change the picture for DLV when compared to CMODELS2.

[7] `http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm`

	Basic program				Non basic programs	
	SMODELS	ASSAT	DLV	CMODELS2	SMODELS	CMODELS2
np30c	11.70	1.14	22.08	0.69	0.36	0.36
np40c	62.89	41.81	97.96	1.63	2.48	0.87
np50c	219.56	14.51	314.46	3.37	8.39	1.79
np60c	594.46	48.80	770.07	5.81	20.47	3.41
np70c	1323.61	291.60	1679.12	8.22	39.41	5.87
np80c	2354.28	32.51	3407.35	14.20	75.36	9.18
np90c	TIME	779.06	TIME	22.23	122.53	14.19
np100c	TIME	–	TIME	28.63	185.52	20.76
np120c	TIME	–	TIME	53.33	418.15	41.84

Table 11.2: Complete graphs. npXc corresponds to a graph with "X" nodes. Finding one solution.

but also for CMODELS2 in the case it generate and then reject (exponentially) many candidate answer sets. The results are in Table 11.2. As can be observed, on this test set CMODELS2 performs best, being faster (sometimes by orders of magnitude) than all the other solvers both on basic and non basic problems.

Non tight real-world problems are shown in Table 11.3. The problems correspond to the bounded model checking (BMC) of asynchronous concurrent systems, as described in [KI03].[8] As for the blocks world, these problems are about proving a certain property in a given number of steps, represented as the last number in the instance name. The problems in the first five rows do not have answer sets, while the remaining (obtained by incrementing the number of steps) do. Here the results are mixed, even though on average CMODELS2 performs better. Of particular interest are the number of sets that CMODELS2 generates before finding an answer set (shown in parentheses). As can be observed, there is no obvious correlation between the number of generated sets and the CPU times. Indeed, given the relatively small time needed to check a set of literals, a failure in the search tree due to a *test* invocation is not very different from a failure due to the generation of the empty clause. Thus, the number of invocation to *test* is not a better indication of CMODELS2 performances than the number of CMODELS2 backtracks.

The problems in Table 11.4 are real-world non tight problem related to checking requirements in a deterministic automaton, and are described in [SEM03].[9] Two types of problems are considered and encoded in logic programs. The first type is called IDFD and the results on such problems are reported in the first two rows of the table. The second type of problem is called "Morin", and the results are shown on the last three rows. As can be seen, CMODELS2 times out on one instance that is easily solved by all the other solvers. This is due to the dimension of the related propositional formula. On the other hand, for any other solver, there are one/two instances on which CMODELS is at least 1 order of magnitude faster. Interestingly, ASSAT blows up in memory on one instance

[8]Some of the benchmarks and their generator are available at `http://www.tcs.hut.fi/~kepa/experiments/boundsmodels/`

[9]These benchmarks are available at `http://www.fmi.uni-stuttgart.de/szs/research/projects/synthesis/benchmarks030923.html`

BMC	SMODELS	CMODELS2
dp-10.i-02-b11	382.72	(73)450.38
dp-10.s-02-b8	15.24	(76)13.88
dp-12.s-O2-b9	336.03	(232)135.12
dp-8.i-O2-b9	8.08	(94)12.64
dp-8.s-O2-b7	1.19	(26)2.12
dp-10.i-O2-b12	445.47	(350)139.18
dp-10.s-O2-b9	28.87	(46)18.22
dp-12.s-O2-b10	971.50	(237)90.64
dp-8.i-O2-b10	5.05	(66)6.76
dp-8.s-O2-b8	1.76	(13)2.28

Table 11.3: Bounded Model Checking Problems. Finding one solution.

	SMODELS	ASSAT	DLV	CMODELS2
mutex4	33.92	(0)0.62	840.60	(0)0.68
phi4	0.24	(168)2.98	1.44	TIME
mutex2	0.09	(88)1.78		(0)0.12
mutex3	229.57	MEM		(0)24.16
phi3	2.87	(704)236.91		(57)3.91

Table 11.4: Checking requirements in a deterministic automaton. Finding one solution. DLV was not run on the last 3 instances.

(and also on other instances, not showed here because all the other systems timeout on them).

Summing up, the 4 tables show the performances on 45 problems: CMODELS2

- times out on 1 problem, while the other systems do not conclude on at least 3 problems;
- performs better than all the three solvers on 30 problems, and on 26 it is at least a factor of 2 faster; and,
- except for the problem on which it times out, CMODELS2 is either the top performer or within a factor of 2 from it.

11.2.2 Finding all answer sets

We also considered the problem of generating all the answer sets. We run the same experiments (the ones that admit solutions) for all domains, but the complete graphs. In this domain, all the solvers were unable to find all the answer sets within the timeout for all the benchmarks we used. In order to evaluate also this category, we generated smaller instances. Tables 11.5 - 11.8 report the results, and they have the same meaning as the tables for the problem of computing one answer set. There is only one more column in each table, "#sol", that indicate the number of answer sets found.

			Basic program		Non basic program	
#b	#s	#sol	SMODELS	CMODELS2	SMODELS	CMODELS2
8	i	28	75.38	2.98	5.29	4.64
11	i	2	171.39	4.88	10.79	2.68
8	i+1	3374	586.98	103.30	39.03	217.59
11	i+1	263	888.11	58.76	57.04	110.16

Table 11.5: Blocks world: "#b" is the number of blocks. Finding all solutions

		Basic program			Non basic program	
	#sol	SMODELS	DLV	CMODELS2	SMODELS	CMODELS2
np8c	5040	1.10	3.35	4.68	0.38	4.36
np9c	40320	10.52	31.79	111.52	3.60	170.19
np10c	362880	111.17	330.71	TIME	38.07	TIME

Table 11.6: Complete graphs. npXc corresponds to a graph with "X" nodes. Finding all solutions.

In Table 11.5 there are results for blocks world problems. Also here, as it could be expected, SAT-based answer set solver CMODELS2 performs (sometimes significantly) better than SMODELS in all the problems proposed, but in the non basic programs with i+1 steps. In this domain, each check for loops fails, and each model of completion is also an answer set.

In Table 11.6 there are results for complete graphs. As we anticipated before, this was the only domain in which we could not run the same experiments as for the problem of finding one answer set (if any). We present results for instances with "8", "9" and "10" nodes. Starting from "11" nodes, none of the solvers is able to find all the solutions within the timeout. In this domain, SMODELS and DLV are much faster than CMODELS2, both on basic and non basic programs. Here the point is that, in order to find all the answer sets, CMODELS2 has to check (and most of the times reject) a very high number of propositional model. Moreover, the dimension of the SAT formula grows quite fast.

The analysis on BMC problems can be found in Table 11.7. In this domain, the timing of the solvers are comparable, SMODELS being slightly better than CMODELS2, of around a factor of two. Moreover, SMODELS can solve (with the meaning here of "finding all solutions") one (out of five) more problem than

BMC	#sol	SMODELS	CMODELS2
dp-10.i-O2-b12	12600	1892	TIME
dp-10.s-O2-b9	17280	115.54	292.44
dp-12.s-O2-b10	?	TIME	TIME
dp-8.i-O2-b10	360	42.22	76.18
dp-8.s-O2-b8	720	5.83	10.03

Table 11.7: Bounded Model Checking Problems. Finding all solutions.

	#sol	SMODELS	DLV	CMODELS2
mutex4	?	TIME	TIME	TIME
phi4	134	37.54	48.21	TIME
mutex2	28	0.11		0.49
mutex3	?	TIME		TIME
phi3	18	9.81		16.85

Table 11.8: Checking requirements in a deterministic automaton. Finding all solutions.

CMODELS2. Both SMODELS and CMODELS2 can not solve the biggest problem in the suite.

Finally, in Table 11.8 there are the results on real-world non tight problem related to checking requirements in a deterministic automaton. The results of the solvers here are quite similar, but for the phi4 benchmark in the IDFD category (we noticed a similar "abnormal" result for it also when we considered the problem of finding one answer set). For this problem, SMODELS and DLV perform much better than CMODELS2. For two of the problems presented, none of the solvers that can tackle the problem is able to return all the solutions within the time limit.

Summing up about the feature of finding all the solutions, some considerations can be drawn. Here the results are less in favor to CMODELS2 when compared in particular to SMODELS, and especially on non basic programs. For these programs, SMODELS takes advantages from the (more compact) representation of the problem, while CMODELS2 does not. We believe that the main reason is because of our very naive implementation of $test(S, \Pi)$ that, if $atoms(S)$ is an answer set, just returns the entire set of literals S. This will be investigated in the near future.
Nevertheless, CMODELS2 is the first SAT-based answer set solver that can find all the answer sets of a logic program.

11.3 Summing Up

We have presented a SAT-based procedure that (i) can deal with any logic program (ii) works on a SAT formula without additional variables, (iii) is guaranteed to work in polynomial space. Further, we have evidenced that ASP-SAT can be easily modified in order to generate all the answer sets. We have shown how to implement ASP-SAT on top of a MCHAFF-like solver, and discussed the modifications needed in the case of non basic programs. The experimental evaluation shows that CMODELS2, has a significant edge over other state-of-the-art systems when the problem is to find one answer set, and is competitive when solvers have to find all solutions. Still, we believe that there is a lot of space for improvements, especially in the heuristics, and in the way reasons are computed.

Chapter 12

On the relation between SAT and ASP

ASP and SAT are highly related. As we have seen in the last Chapters, there is modular translation from SAT to ASP (such that a "solution" of the logic program is also a "solution" of the related propositional theory), but in general, unless the logic program is "tight", there is no such a modular translation (some of the "solutions" of the propositional theory are not answer sets of the related logic program).

In order to have a 1-1 correspondence between ASP and SAT we have to add "loop formula" to the initial propositional theory. The number of these loop formula can be exponential (as proved in[LR04]), but this is a viable way, considering the loop formula one by one selectively, as used by two emerging SAT-based ASP solvers ASSAT and CMODELS2.

In several paper (among others [Sim00, FLP01]), has been noticed and argued that the SAT-based solvers (ASSAT and CMODELS2) and the other state-of-the-art solvers (like SMODELS and DLV) are "similar". But this has never been proved formally before.

In this Chapter, we study the relation between Answer Set (AS) and propositional satisfiability (SAT) solvers, both from a theoretical and a "practical", implementation oriented, point of view.

On the theoretical side, given a tight logic program Π, we first define ASP, a complete recursive procedure for computing the answer sets of Π. Besides its simplicity, the main feature of ASP is that its pruning strategies are exactly (resp. very similar to) the ones used by SMODELS [Sim00, SNT02], (resp. DLV [Fab02, LPF+02]) for solving a logic program Π. Thus, ASP is a simple, recursive presentation of SMODELS (and, to a certain extent, of DLV) for tight programs. Then, if $lp2sat(\Pi)$ is the standard conversion to SAT based on Clark's completion and Tseitin's [Tse70] clausal form transformation, we show that ASP(Π) and DLL($lp2sat(\Pi)$) are equivalent, i.e., that they explore search trees with the same branching nodes. Vice-versa, given a set of clauses Γ, if $sat2lp(\Gamma)$ is its "naive" conversion to a tight program, we show that also DLL(Γ) and ASP($sat2lp(\Gamma)$) are equivalent. As a consequence, results related to DLL can be easily shown to hold also for ASP, and vice-versa. For instance, we have that

- *sat2lp*(PHP^n_{n-1}) are exponentially hard for ASP, where PHP^n_m are the pigeonhole formulas from [Hak85];
- deciding the "best" literal to branch on, is both NP-hard and co-NP hard for ASP, and in PSPACE for tight programs; and
- when considering *sat2lp*(Γ) programs, where Γ are randomly generated 3-SAT formulas with a ratio of clauses to variables above 5.6, ASP has an exponential behavior.

These are just a few of the many results that are already known for DLL (DLL is one of the most studied procedure in automated reasoning), and that –thanks to our results– can be easily shown to hold also for ASP, and thus also for SMODELS (and DLV).

The above mentioned theoretical results show that ASP(*sat2lp*()) and DLL(), (resp. ASP() and DLL(*lp2sat*()) on tight programs) are, to a certain extent, the same procedure. However, on the practical side, there is not a unique way of realizing an ASP-based (resp. a DLL-based) AS (resp. SAT) solver. Current state-of-the-art AS solvers are different for (*i*) the more-or-less optimized implementation; and/or (*ii*) the "look-ahead" strategy used while descending the search tree; and/or (*iii*) the "look-back" strategy used for recovering from a failure in the search tree; and/or (*iv*) the heuristic used for selecting the next literal to branch on. In SAT the same holds, but to a less extent. Indeed, current state-of-the-art SAT solvers can be divided in two main categories:[1]

- "look-ahead" solvers, featuring a rather sophisticated look-ahead based on "failed literal", a simple look-back (essentially backtracking) and a heuristic based on the information gleaned during the look-ahead phase. These solvers are best for dealing with "small but relatively difficult" instances, typically randomly generated according to the "Fixed Clause Length" model (FCL).
- "look-back" solvers, featuring a simple but efficient look-ahead (essentially unit-propagation with 2 literal watching), a rather sophisticated look-back based on "learning" and a constant time heuristic based on the information gleaned during the look-back phase. These solvers are best for dealing with "large but relatively easy" instances, typically encoding real-world problems.

Right now, in order to advance the state-of-the-art, SAT developers focus either on randomly generated problems and develop a look-ahead solver, or on real-world problems and develop a look-back solver. This is the result of various SAT competitions, of the development of systems tuned for specific classes, and of various papers assessing the benefits of the different search strategies in different domains.

In AS programming (ASP) the situation is different. Considering randomly generated programs, thanks to the richer structure that programs have if compared to formulas, there are various ways to generate them, each somehow

[1] The phrases "small but relatively difficult" and "large but relatively easy" are used just to convey the basic intuitions about the instances. However, consider that in the SAT2003 competition, instances in the random and industrial categories had, on average, 442 and 42703 variables respectively [LS03].

corresponding to the SAT FCL model (see [LZ03b]). Considering real-world benchmarks, the ones that are most used in SAT and in AS have different characteristics: While in SAT most of them have tens or hundreds of thousands of variables (see, e.g., [LS03]), in AS most of them have far fewer variables. As a consequence, it is not clear whether the combination of the reasoning strategies that currently dominate in SAT, are bound to dominate also in ASP, this despite of the tight relation (proved in this paper) that exists between AS and SAT solvers. In order to answer this question, we conducted an extensive experimental evaluation. Our results point out that on randomly generate problems, look-ahead solvers are best; while for real-world problems, there is not a single winning combination of look-ahead and look-back strategies and heuristic. A closer analysis of the results reveals that large problems are best tackled by look-back solvers, while the others are best dealt with a combination of look-ahead and look-back strategies. Given that the majority of the AS problems that are currently available have a relatively small number of variables, we expect that future development (resp. comparison) of AS solvers for (resp. on) real-world problems should feature (resp. reward) a combination of look-ahead and look-back techniques. On the other hand, we also expect that as soon as more large (but relatively easy) problems will become available, look-back solvers will start to prevail over the others. Using the terminology in [GMTZ01], our comparison is "fair" because all the reasoning strategies are realized on a common platform (thus, our experimental evaluation is not biased by the differences due to the quality of the implementation) and is "significant" because our solver implements current state-of-the-art procedures for look-ahead and look-back, and we considered a wide variety of benchmarks.

12.1 On the relation between ASP and SAT

Let P be a set of atoms. If p is a an atom, $\overline{p}$ is the *negation* of p (to be intended as negation as failure *not*), and $\overline{\overline{p}}$ is p. We will also use the logical symbols $\bot$ and $\top$ (standing for FALSE and TRUE respectively), and assume that $\overline{\bot} = \top$ and $\overline{\top} = \bot$. Atoms, their negations, and the symbols $\bot$, $\top$ form the set of *literals*. If S is a set of literals, we define $\overline{S} = \{\overline{l} : l \in S\}$.

12.1.1 Answer Set Programming

We report in this subsection, for convenience of the reader, some notation we will use in this Chapter. They have been already introduced in Chapter 9.

A *rule* is an expression of the form

$$p_0 \leftarrow p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n \tag{12.1}$$

where $p_0 \in P \cup \{\bot\}$, and $\{p_1, \ldots, p_n\} \subseteq P$ $(0 \leq m \leq n)$. If r is a rule (12.1), $head(r) = p_0$ is the *head* of r, and $body(r) = \{p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\}$ is the *body* of r. A *(logic) program* is a finite set of rules.

Consider a program Π, and let X be a set of atoms. In order to give the definition of an answer set we consider first the special case in which the body of each rule in Π contains only atoms. Under these assumptions, we say that

- X is *closed* under Π if for every rule (12.1) in Π, $p_0 \in X$ whenever $\{p_1, \ldots, p_m\} \subseteq X$, and that
- X is an *answer set* for Π if X is the smallest set closed under Π.

Now we consider the case in which Π is an arbitrary program. The *reduct* Π^X of Π relative to X is the set of rules

$$p_0 \leftarrow p_1, \ldots, p_m$$

for all rules (12.1) in Π such that $X \cap \{p_{m+1}, \ldots, p_n\} = \emptyset$. X is an *answer set* for Π if X is an answer set for Π^X.

12.1.2 Propositional reasoning

A *clause* is a finite set of literals different from $\bot, \top$, and a *formula* is a finite set of clauses. An *interpretation* is a set of atoms. An interpretation X *satisfies* a formula Γ if for each clause C in Γ, $C \cap (X \cup \{\overline{p} \mid p \notin X\}) \neq \emptyset$. If X satisfies φ then we also say that X is a *model* of φ and that φ is *satisfiable*.

12.1.3 Translation from ASP to SAT

Consider a program Π. It is well known that if X is an answer set of Π then X satisfies the completion of Π, and that the converse is true if the program is "tight" [Fag94]. The standard definition of the completion of a program Π is not directly representable as a set of clauses. However, we can use well known linear time conversion methods and obtain an equi-satisfiable set of clauses. Here, in particular, we use Tseitin's conversion [Tse70].

If p_0 is an atom, the *translation of* Π *relative to* p_0, denoted with $lp2sat(\Pi, p_0)$, consists of

1. for each rule $r \in \Pi$ of the form (12.1) and whose head is p_0, the clauses:

$$\begin{gathered}\{p_0, \overline{p}_1, \ldots, \overline{p}_m, p_{m+1}, \ldots, p_n\},\\ \{n_r, \overline{p}_1, \ldots, \overline{p}_m, p_{m+1}, \ldots, p_n\},\\ \{\overline{n}_r, p_1\}, \ldots, \{\overline{n}_r, p_m\}, \{\overline{n}_r, \overline{p}_{m+1}\}, \ldots, \{\overline{n}_r, \overline{p}_n\},\end{gathered}$$

   where n_r is a newly introduced atom,[2] and

2. the clause $\{\overline{p}_0, n_{r_1}, \ldots, n_{r_q}\}$ where $n_{r_1}, \ldots, n_{r_q}$ $(q \geq 0)$ are the new symbols introduced in the previous step.

The *translation of* Π *relative to* $\bot$, denoted with $lp2sat(\Pi, \bot)$, consists of a clause

$$\{\overline{p}_1, \ldots, \overline{p}_m, p_{m+1}, \ldots, p_n\},$$

one for each rule in Π of the form (12.1) with head $\bot$. The *translation of* Π, denoted with $lp2sat(\Pi)$, is $\cup_{p \in P \cup \{\bot\}} lp2sat(\Pi, p)$.

A program Π is *tight* if there exists a function λ from atoms to ordinals such that, for every rule (12.1) in Π whose head is not $\bot$, $\lambda(p_0) > \lambda(p_i)$ for each $i = 1, \ldots, m$. The following theorem is an easy consequence of Fages' and Tseitin's results.

[2] In the definition of $lp2sat(\Pi, p_0)$, we can replace the first clause with the clause $\{p_0, n_r\}$: The resulting formula would be equivalent, and the results in the paper would still hold.

Theorem 3 *Let Π be a program. If X is a an answer set of Π then there exists a unique model Y of lp2sat(Π) such that $Y \cap P = X$. If Y is a model of lp2sat(Π) and Π is tight, then $Y \cap P$ is an answer set of Π.*

12.1.4 Translation from SAT to ASP

We review here the translation introduced in Chapter 9.

Consider a formula Γ. To each clause $C = \{l_1, \ldots, l_l\}$ $(l \geq 0)$ in Γ we associate the rule $sat2lp(C) = \bot \leftarrow \overline{l}_1, \ldots, \overline{l}_l$. The *translation of* Γ, denoted with $sat2lp(\Gamma)$, is $\cup_{C \in \Gamma} sat2lp(C) \cup \cup_{p \in P} \{p \leftarrow \overline{p'}, p' \leftarrow \overline{p}\}$, where, for each atom $p \in P$, p' is a new atom associated to p.

Theorem 4 *Let Γ be a formula. sat2lp(Γ) is tight. Further, if Y is a model of Γ then there exists a unique answer set X of sat2lp(Γ) such that $X \cap P = Y$. If Y is an answer set of sat2lp(Γ), then $Y \cap P$ is a model of Γ.*

12.2 On the relation between ASP and DLL

In this section, we restrict our attention to tight logic programs. Consider a tight logic program Π. In the last few years, several systems for answer sets computation have become available, including SMODELS, DLV, ASSAT and CMODELS.[3] While the first two are native procedures working directly on Π, the last two (i) compute the completion of Π; (ii) convert it to a set of clauses; and (iii) exploit the efficiency of SAT solvers, most of which are based on the Davis-Logemann-Loveland procedure (DLL). Though apparently different, we show that these two approaches are strongly related (if using Tseitin's clausal form transformation).

An *assignment* is a set of literals. Given an assignment S, we say that a *literal l is assigned by S* if $\{l, \overline{l}\} \cap S = \emptyset$.

12.2.1 ASP

Given a program Π, both SMODELS and DLV search for answer sets by extending an assignment S till either S becomes inconsistent (in which case backtracking occurs) or each atom is assigned by S (in which case $S \cap P$ is an answer set). SMODELS and DLV differ in many ways, e.g., for the language considered. Still, at an high-level of abstraction and when considering tight logic programs, SMODELS and DLV are similar to the procedure ASP-REC represented in Figure 12.1. In the figure,

- Π is a program, initially set to the program of which we want to determine the existence of answer sets; S is an assignment, initially set to $\{\top\}$.
- p denotes an atom, r a rule, and l a literal.

[3] All the systems work also on non-tight programs. See, `http://www.tcs.hut.fi/Software/smodels`, `http://www.dbai.tuwien.ac.at/proj/dlv`, `http://assat.cs.ust.hk`, `http://www.cs.utexas.edu/users/tag/cmodels.html`, respectively.

function ASP-REC(Π,S)
1 $\langle \Pi, S \rangle$:= *simplify*(Π, S);
2 **if** $(\{l, \bar{l}\} \subseteq S)$ **return** FALSE ;
3 **if** $(\{p : p \in P, \{p, \overline{p}\} \cap S \neq \emptyset\} = P)$ **return** TRUE ;
4 p := *ChooseLiteral*(S);
5 **return** ASP-REC(p-*assign*(p, Π)), $S \cup \{p\}$) or
6 ASP-REC(p-*assign*($\overline{p}, \Pi$)), $S \cup \{\overline{p}\}$);

function *simplify*(Π,S)
7 **if** ($r \in \Pi$ and *body*(r) $= \emptyset$ and *head*(r) $\notin S$)
return *simplify*(p-*assign*(*head*(r), Π), $S \cup \{$*head*(r)$\}$);
8 **if** $(\{p, \overline{p}\} \cap S = \emptyset$ and $\nexists r \in \Pi$: *head*(r) $= p$)
return *simplify*(p-*assign*($\overline{p}, \Pi$), $S \cup \{\overline{p}\}$);
9 **if** ($r \in \Pi$ and *head*(r) $\in S$ and *body*(r) $\neq \emptyset$ and
$\nexists r' \in \Pi, r' \neq r$: *head*(r') $=$ *head*(r))
return *simplify*(p-*assign*(*body*(r), Π), $S \cup$ *body*(r));
10 **if** ($r \in \Pi$ and $\overline{\textit{head}(r)} \in S$ and *body*(r) $= \{l\}$)
return *simplify*(p-*assign*($\bar{l}, \Pi$)), $S \cup \{\bar{l}\}$);
11 **return** $\langle \Pi, S \rangle$;

Figure 12.1: The algorithm of ASP-REC for tight logic programs.

- p-*assign*(l, Π) returns the program obtained from Π by (i) deleting the rules r such that $\bar{l} \in$ *body*(r); and (ii) deleting l from the body of the other rules in Π.
- If $l_1, \ldots, l_n$ are the literals in a set S', p-*assign*(S', Π) is p-*assign*($l_1, \ldots,$ p-*assign*(l_n, Π) $\ldots$).
- For simplicity, *ChooseLiteral*(S) returns the first, according to a fixed total order on P, unassigned atom by S.

The computation of ASP-REC(Π, S) proceeds as follows (in the following, we say that a set of atoms X *extends an assignment* S if $S \cap P \subseteq X$ and $\overline{S} \cap X = \emptyset$:

- Line 1: The program Π is simplified and the assignment S is extended by the routine *simplify*(Π, S), explained below.
- Line 2: if S is inconsistent, no answer set extending S exists, and FALSE is returned,
- Line 3: if each atom $p \in P$ is assigned, then (i) $S \cap P$ is an answer set of the initial program, and (ii) TRUE is returned.
- Lines 4-6: if none of the above applies, an atom p is selected (line 4), an answer set extending $S \cup \{p\}$ is searched first (line 5), and, upon failure, we search for an answer set extending $S \cup \{\overline{p}\}$ (line 6).

The simplification steps of the procedure *simplify*(Π, S) encode the following facts:

function DLL-REC(Γ,S)
1 $\langle \Gamma, S \rangle$:= *unit-propagate*(Γ, S);
2 **if** $(\{l, \bar{l}\} \subseteq S)$ **return** FALSE ;
3 **if** $(\{p : p \in P, \{p, \overline{p}\} \cap S \neq \emptyset\} = P)$ **return** TRUE ;
4 p := *ChooseLiteral*(S);
5 **return** DLL-REC(*s-assign*(p, Γ)), $S \cup \{p\}$) or
6 DLL-REC(*s-assign*($\overline{p}, \Gamma$)), $S \cup \{\overline{p}\}$);

function *unit-propagate*(Γ,S)
7 **if** $(\{l\} \in \Gamma)$ **return** *unit-propagate*(*s-assign*(l, Γ), $S \cup \{l\}$);
8 **return** $\langle \Gamma, S \rangle$;

Figure 12.2: The algorithm of DLL-REC.

- Line 7: if there exists a rule r whose body is empty, then every answer set extending S includes the head of r.
- Line 8: if an unassigned atom p is not the head of any rule, then every answer set extending S does not include p,
- Line 9: if there is only one rule with head p, and $p \in S$, then each answer set extending S, also extends $S \cup$ *body*(r).
- Line 10: if there is a rule with head p and whose body consists of a single literal l, then if $\overline{p}$ is in S, then every answer set extending S also extends $S \cup \{\bar{l}\}$.

When no further simplification is possible, the pair $\langle \Pi, S \rangle$ is returned by *simplify*(Π, S) (line 11).

Notice that the rules used by ASP-REC to extend the assignment S in lines (7)-(10) are exactly the same used in SMODELS (procedure *expand*, see [Sim00] pagg. 17, 32-37)[4] and are very similar to those used by DLV (procedure $DetCons$, see [Fab02], pagg. 40-43).

In the following, ASP is the procedure which, given a program Π, returns ASP-REC($\Pi, \{\top\}$).

Theorem 5 *Let* Π *be a tight logic program.* ASP(Π) *returns* TRUE *if* Π *has an answer set, and* FALSE *otherwise.*

12.2.2 DLL

DLL-REC is at the basis of most of the state-of-the-art SAT solvers, and is presented in Figure 12.2. We report here again the algorithm and also how *unit-propagate*() is implemented for convenience of the reader. Here, DLL-REC is slightly modified (we will see shortly the reason). In the figure,

[4]More in details, *expand* calls two functions, $AtLeast$ and then $AtMost$, the first of which corresponds to *simplify*, and the second allowing to derive additional negative literals. However, in the case of tight programs, $AtMost$ does not allow to derive any additional literal (see [Sim00], pag. 44).

- Γ is a set of clauses, initially set to the formula of which we want to determine the satisfiability; S is an assignment, initially set to the empty set.
- p and l denote an atom and a literal respectively.
- *s-assign*(l, Γ) returns the formula obtained from Γ by[5] (i) deleting the clauses $C \in \Gamma$ with $l \in C$, and (ii) deleting $\bar{l}$ from the other clauses $C \in \Gamma$ with $|C| \geq 2$.

The name of *s-assign*(l, Γ) here is a bit misleading, because this is the Simplify() procedure presented in past parts. But in this Chapter, following the notation of the original paper, we refer to *simplify*() as the simplification procedure of ASP-REC. Moreover, the definition of *s-assign*(l, Γ) which does not consider the *unit* clauses (i.e., consisting of a single literal) in Γ, allows us to have a termination condition for FALSE similar to that of the procedure ASP (see line 2 in Figure 12.1). This will make easier the formal statement of our results in the next subsection. We call DLL the procedure which, given a formula Γ, returns DLL-REC$(\Gamma, \emptyset)$.

Theorem 6 *Let* Γ *be a formula.* DLL(Γ) *returns* TRUE *if* Γ *has a model, and* FALSE *otherwise.*

Notice that Theorem 6 is very similar to Lemma 1 in Chapter 2.

12.2.3 ASP, DLL and their relation

Our goal is to prove that ASP and DLL computations are tightly related when given a program Π and *lp2sat*(Π) respectively, or *sat2lp*(Γ) and a formula Γ respectively. The first problem is that both translations (*lp2sat* and *sat2lp*) introduce additional atoms not in P: In the following we assume that both ASP and DLL operate in the signature of the input program and formula respectively (i.e., in the possibly extended signature). However, we still assume that *ChooseLiteral*(S) returns the first atom in P which is unassigned by S: Notice that once all the atoms in P are assigned, also the atoms introduced by *lp2sat* (resp. *sat2lp*) will be assigned by *unit-propagate* (resp. *simplify*).

Given this, one possibility for achieving our goal would be to consider the *search trees* corresponding to the assignments generated by the two procedures, and try to prove that they are the same. However, this is not the case:

- Both *lp2sat* and *sat2lp* introduce additional atoms not in P and also these atoms get assigned, and
- The order followed by *simplify* and *unit-propagate* to assign literals may differ.

However, if we do not take into account the above differences, we have that the two procedures generate the "same" search tree. In order to formally state this result we introduce the following definitions.

[5] We assume that *s-assign*(l, Γ) does not eliminate $\bar{l}$ from $\{\bar{l}\}$ because this allows us to have a termination condition for FALSE analogous to the one used in ASP (see line 2 in Figure 12.1).

Consider a program Π and a formula Γ, possibly with atoms not in P. We say that a set of literals S in the signature of Π (resp. Γ) is a *branching node of* ASP(Π) (resp. *of* DLL(Γ)) if there is a call to ASP-REC(Π', S) (resp. DLL-REC(Γ', S)), following the invocation of ASP(Π) (resp. DLL(Γ)). If *proc* is ASP(Π) or DLL(Γ), we define

$$Br(proc) = \{S \cap (P \cup \overline{P}) : S \text{ is a branching node of } proc\}.$$

Finally, we say that ASP(Π) and DLL(Γ) are *equivalent* if $Br(\text{ASP}(\Pi)) = Br(\text{DLL}(\Gamma))$.

Theorem 7 *For each tight program* Π, ASP(Π) *and* DLL(*lp2sat*(Π)) *are equivalent. For each formula* Γ, DLL(Γ) *and* ASP(*sat2lp*(Γ)) *are equivalent.*

The idea underlying the proof is that the atoms in P assigned by *simplify* correspond exactly to those assigned by *unit-propagate* on the translation of the program. Then, *ChooseLiteral* is guarantee to return the same atom in the two procedures.

Theorem 7 establish a strong relation between ASP and DLL: To a certain extent, ASP(*sat2lp*()) and DLL(), ASP() and DLL(*lp2sat*()) are the same procedure (at least on tight programs). Further, the results hold independently from the specific heuristic used by ASP and DLL, as long as they are guaranteed to return the same literal at every point of the two search trees. Because of this, similar results would hold if we enhance ASP and DLL with more powerful look-ahead techniques based on *simplify* and *unit-propagate* respectively. For instance, SMODELS before branching, for every unassigned literal l in the program, checks whether assigning l would "fail", i.e., in our terminology, if *simplify*(*p-assign*(l, Π), $S \cup \{l\}$) returns (as second argument) an inconsistent set of literals: if this is the case, we can safely assign $\overline{l}$ before branching. However, if l fails, then also branching on l would fail, and the tree generated by ASP extended with such "failed literal" strategy corresponds to the tree generated by ASP with a specific heuristic. Using the same heuristic in DLL (i.e., using a similar "failed literal" strategy based on *unit-propagate*) would lead to an equivalent search tree.

Thanks to these results, we can directly derive lower/upper bounds, average case results for ASP, which are already known for DLL, such as:

- *sat2lp*($PHIP^n_{n-1}$) *are exponentially hard for* ASP. Consider the pigeonhole formulas PHP^n_m. It is well known that any resolution proofs of the unsatisfiability of PHP^n_{n-1} is exponential [Hak85]. Given that each DLL search tree corresponds to a regular tree resolution, we can immediately conclude that ASP(*sat2lp*(PHP^n_{n-1})) is exponential.

- *In* ASP, *deciding the optimal literal to branch on, is both NP-hard and co-NP hard, and in PSPACE for tight programs.* Define a literal l as *optimal for a formula* Γ if there exists a minimal search tree of DLL(Γ) whose root is labeled with l. Liberatore [Lib00] showed that deciding whether a literal is optimal is both NP-hard and co-NP hard, and in PSPACE. The definition of *optimal for a program* Π is analogous, and, given our results, the derivation of the above mentioned bounds for ASP, are easy.

- *sat2lp(k-SAT) are asymptotically exponentially hard, in probability, when* $d \geq 0.7 \times 2^k$. If we consider randomly generated SAT formulas with m

clauses and k literals per clause, (as done, e.g., in [FLP01, SNT02, WS04]), with probability tending to one as the number of the number of variables n tends to infinity, ASP has an exponential behavior (in n) if the *density* $d = m/n$ is $d \geq 0.7 \times 2^k$. This result easily follow again from our results and the results in [CS88].

Since, in their SAT formulation, the first and third result hold for any proof system based on resolution, and since ASP (on tight programs) and DLL search trees correspond to a form of resolution, these results hold also if ASP and DLL were using "learning" look-back strategies (as, in AS, does SMODELS-CC [WS04] and CMODELS2 [GML04]).

There are many other results in the SAT literature studying the proof-complexity of DLL and/or resolution that are applicable also to ASP. See, e.g., [Mon04] for a study on the average complexity of coloring randomly generated graphs with DLL, and [ABM01], which derives exponential lower bounds on the running time of DLL on randomly generated 3-SAT formulas also for densities significantly below the satisfiability threshold $d \approx 4.23$. The first result applies also to ASP when run on a program Π being the standard tight formulation of a graph coloring problem:[6] *lp2sat*(Π) is the standard formulation in SAT considered in [Mon04]. Analogously for the second.

12.3 On the relation between AS and SAT solvers

Given the results established in the previous Section, it is easy to show that the numbers n_{ASP} and n_{DLL} of literals assigned by an AS solver based on ASP and a SAT solver based on DLL respectively, if run on a tight program Π and *lp2sat*(Π) respectively (or on *sat2lp*(Γ) and a formula Γ respectively) are such that $min(n_{\text{ASP}}, n_{\text{DLL}}) \leq max(n_{\text{ASP}}, n_{\text{DLL}}) \leq 2 \times min(n_{\text{ASP}}, n_{\text{DLL}})$ (this of course assuming that the two solvers use "equivalent" look-ahead, look-back and heuristic). Because of this, we can expect that the combinations of reasoning strategies that work best in SAT, should also work best in AS. However, as we already anticipated in the introduction, this is only partially the case. We now report about an extensive experimental comparison that we have conducted on a wide variety of programs, and using the combinations of reasoning strategies that, along the years, proved to be more effective in ASP or in SAT. When deciding how to implement the system, we had two possibilities: Either develop a native AS solver based on ASP, or start from a SAT solver and evaluate the performances on the translation to SAT of each program. We opted for the second. In particular, we started from CMODELS2 [GML04] because (*i*) its front-end is LPARSE [Sim00]; (*ii*) its back-end SAT solver already incorporates some of the state-of-the-art strategies (like lazy simplification based on literal watching, 1-UIP learning, VSIDS heuristics) that we want to test; and (*iii*) can be also run on non-tight programs. There is no other publicly available system (be it an AS or a SAT solver) having the above features, and that we know of. Given this, the reasoning strategies that we considered are:

- Look-ahead: Basic unit-propagation (denoted with "U"); and unit-propagation+failed-literal (denoted with "F").

[6] See, e.g., the formulation in `http://www.tcs.hut.fi/~ini/papers/niemela-iclp04-tutorial.ps.gz`.

	PB	# VAR	ulv	flv	flu	fbu	ulp
1	4	300	**0.41**	0.52	0.85	0.66	21.79
2	4.5	300	TIME	TIME	81.92	**22.53**	TIME
3	5	300	448.21	485.36	8.27	**4.72**	452.75
4	5.5	300	73	38.61	2.26	**1.7**	38.48
5	6	300	20.94	12.01	0.95	**0.81**	7.83

Table 12.1: Performances on tight randomly generated logic programs.

- Look-back: Basic backtracking (denoted with "B"); and backtracking+1-UIP learning from [MMZ+01] (denoted with "L").
- Heuristic: VSIDS from [MMZ+01] (denoted with "V"); unit (given an unassigned atom p, while doing failed literal on p we count the number $u(p)$ of unit-propagation caused, and then we select the atom with maximum $1024u(p) \times u(\overline{p}) + u(p) + u(\overline{p})$. This heuristic is denoted with "U".

We start our analysis considering 4 combinations of reasoning strategies: ulv, flv, flu and fbu, where the first, second and third letter denote the look-ahead, look-back and heuristic respectively, used in the combination. ulv is a standard look-back, "zChaff"-like, solver, similar to SMODELS-CC and CMODELS2. fbu is a standard look-ahead solver. flv and flu have both a powerful look-ahead and look-back but different heuristic. All the tests were run on a Pentium IV PC, with 2.8GHz processor, 1024MB RAM, running Linux. The timeout has been set to 600 seconds of CPU time for random problems, and 3600 for real-world problems.

Tables 12.1, 12.2, 12.3 and 12.4 show the results on a wide variety of benchmarks.[7] In particular, benchmarks (1-5) in Table 12.1 are programs being the translation of randomly generated 3-SAT instances with "# VAR" variables and a ratio of clauses to variables as in the column "PB". The number in the column PB for instances (24-28) in Table 12.3 has the same meaning, being instances randomly generated according to the methodology proposed in [LZ03b]. For each ratio, we randomly generated 10 instances and show the median results. All the other benchmarks are real-world problems. For all problems but Table 12.1 the number of variables after LPARSE of each is shown in the column "# VAR".

For each row, the best result is in bold, and the results within a factor of 2 from the best, are underlined.

The first observation is that on random benchmarks, learning does not help, and fbu has an edge. These results confirm what we know from the SAT literature. Considering the non-random, the situation is far less clear. Indeed, look-back solvers perform quite well on a wide variety of benchmarks, but not as well as one would expect. In particular, the performances on the bounded model checking (BMC) instances (problems (9-17) in Table 12.2) of ulv (resp. flv and flu) are worse (resp. better) than expected: In SAT, BMC instances are the benchmarks where look-back (resp. solvers with powerful look-ahead) give

[7] We considered far more benchmarks than those we show. Most of the benchmarks that we considered are from `http://www.tcs.hut.fi/Software/smodels/tests/`, `http://www.tcs.hut.fi/~kepa/tools/boundsmodels/`.

	PB	# VAR	ulv	flv	flu	fbu	ulp
6	queens21	925	786.14	1864.49	384.87	47.33	**0.24**
7	queens24	1201	TIME	TIME	TIME	368.76	**0.28**
8	queens50	5101	TIME	TIME	TIME	TIME	**347.98**
9	d*12*i*9	1186	**223.93**	383.66	353.53	TIME	2910.96
10	k*i*29	3199	415.54	204.87	**44.14**	589.45	1329.53
11	k*s*29	3169	353.69	1028.77	**59.99**	TIME	509.29
12	m*3*i*10	1933	16.23	32.23	26.71	16.55	**6.19**
13	m*4*i*12	3475	1063.15	**867.49**	TIME	3229.09	TIME
14	m*4*s*8	1586	17.02	27.59	421.30	327.55	**13.79**
15	q*i*17	2201	1539.96	505.15	**259.05**	816.26	TIME
16	e*3*i*15	7832	479.28	TIME	7.15	**6.87**	TIME
17	e*4*i*13	6447	87.63	567.27	20.02	**19.41**	104.45
18	bw-large.d9	9956	1.02	5.84	2.69	2.75	**1.01**
19	bw-large.e9	12260	**0.98**	1.91	1.92	1.93	1.03
20	bw-large.e10	13482	**1.29**	7.51	5.03	4.95	1.55
21	p1000	14955	**0.48**	37.86	15.41	15.23	3.69
22	p3000	44961	**8.86**	369.27	144.12	142.83	223.62
23	p6000	89951	**99.50**	TIME	583.55	578.98	2549.50

Table 12.2: Performances on non-randomly generated logic programs. (6-8) are queens; (9-17) are bounded model checking; (18-20) are blocks-world; (21-23) are 4-colorability;

	PB	# VAR	ulv	flv	flu	fbu	ulp
24	4	300	265.43	218.48	41.97	**31.05**	77.41
25	4.5	300	TIME	TIME	190.73	**135.11**	TIME
26	5	300	TIME	TIME	136.67	**99.75**	439.71
27	5.5	300	TIME	TIME	129.29	**78.63**	TIME
28	6	300	TIME	TIME	107.34	**65.83**	591.3

Table 12.3: Performances on non-tight randomly generated logic programs.

	PB	# VAR	ulv	flv	flu	fbu	ulp
29	bw-basic-i	5301	**2.16**	15.54	6.07	5.79	2.54
30	bw-basic-i-1	4760	**1.64**	4.92	2.47	2.44	1.86
31	bw-basic-i+1	5842	2.49	24.27	22.01	19.71	**2.41**
32	np50c	7452	**1.66**	544.45	17.74	17.84	2.38
33	np60c	10742	**2.83**	1611.32	44.12	44.12	4.77
34	np70c	14632	**4.69**	TIME	97.44	97.89	5.91

Table 12.4: Performances on non-randomly generated non-tight problems. Problems (29-31) are blocks-world; (32-34) are Hamiltonian Circuit on complete graphs.

their best (resp. their worst). This is indeed explained if we look at the number of variables "# VAR" of these instances, which is in the order of a few thousands. Indeed, for such "# VAR" it still make sense to perform an aggressive look-ahead at each branching node. On the other hand, as "# VAR" increases, this is no longer the case, as the results on the 4 colorability instances (lines 21-23) show in Table 12.2.

We implemented a 5th combination of reasoning strategies (called ulp) which is similar to flu, except that (*i*) we first select a pool of 10 "most watched" variables, and (*ii*) we perform failed literal and score accordingly only the variables in the pool. The results are shown in the last column. As it can be seen, ulp performs well on some benchmarks in which flu is not competitive, this despite our non optimal pooling criteria.[8] This is not surprising, given that we know from SAT that performing failed literal on a restricted pool of most promising variables is often a better idea than doing failed literal on all the unassigned variables. (As a side remark, SMODELS has criteria which allows it not to try literals that will not fail). We expect that better performances for ulp can be obtained by using more sophisticated pooling criteria. Thus, given the relatively low number of variables in most instances, we believe that if the goal is to develop a general purpose AS solver, an ulp-based solver is, at the moment, the way to go. Nonetheless, as soon as the "# VAR" in challenges benchmarks increase, we except that ulv-based solvers will become the best performing on real-world benchmarks, as in SAT.

[8]Our pooling criteria is motivated by the fact that we are using a solver with literal watching. While this data structure is good for fast unit-propagation, does not allow to properly select the "most promising" variables using better criteria which take into account all the occurrences of a variable. To have such pooling criteria without sacrificing the efficiency of unit propagation, the obvious solution is to have an additional data structure to be used just for pooling.

Part V

Safe Planning via SAT-related optimization problems

Chapter 13

Introduction

In this Chapter we investigate the approach of performing Safe Planning via reduction to some interesting problems related to the SAT problem. We now briefly see the definitions of these problems. The problems we are interested in are:

1. Max-SAT [dGLMS03]: Given an unsatisfiable SAT instance, find how many clauses can be satisfied at most (at the same time).

2. Min-One [ARMS00]: Given a satisfiable SAT instance, find the satisfiable assignment with the highest number of variables set to 1 (true).

(problems Min-SAT and Max-One are defined similarly).

SAT is a *decision* (or *yes/no*) problem. This means that, in order to answer this problem, it is sufficient that the solver returns "yes" if the formula is satisfiable, and "no" if the formula is unsatisfiable. In SAT, a very important class of problems are randomly generated problems. As showed in previous Chapters and Sections, these problems usually have a classical "phase transition" with respect to the ratio between clauses and variables (each clause having a fixed number of literals). For low ratio (under-constrained region), almost all the problems are "satisfiable" (it is quite easy to find a satisfiable assignment). After a region (for "medium" ratio) in which there is a transition between "almost all" satisfiable and "almost all" unsatisfiable problems, for "high" ratio (over-constrained region) almost all the problems are unsatisfiable (and it is quite easy to show it).

The Max-SAT and Min-One problems introduced above are the related "optimization" problems to SAT on the under- and over-constrained regions respectively. It is not enough for them a yes/no answer, but they need some kind of "optimization" w.r.t. the number of clauses or the number of variables.

Consider now in particular a Min-SAT problems, and the Safe Planning via Propositional Logic approach presented in Chapter 3. In order to define the new approach, all the considerations made for the former approach are still valid. Moreover, we can associate to each propositional variable A_i (that, we remember, corresponds to the ground action A at time i) a "weight" to the variable. In this way we can construct a functional cost defined on the (ground) actions of the planning problem. The compiler outputs a propositional formula (exactly the one we saw in Chapter 3), plus a functional cost that must be

minimized (using some informations on top of the CNF formula, for details refer to Chapter 15). Now "propositional formula+ functional cost" represents the optimization problem (Min-One in our case). The problem is weighted if at least one weight is > 1.

A solution returned by a (Weighted) Min-ONE solvers has the following properties.

- the plan is safe
- the plain is minimal w.r.t. the number of actions in the plan, if the problem is un-weighted, or w.r.t. a functional cost defined on the actions, if the problem is weighted.

Chapter 14

From optimization functions to CNF formulas

This section is devoted to showing some state-of-the-art encodings for optimization functions to CNF formulas. We will focus on optimizations related to functions of the type

$$\sum_i a_i x_i$$

where a_i are numeric (integer) constants and x_i are variables ranging over $\{0,1\}$.

Moreover, in this Chapter we will focus our attention to functions with general $a_i = 1$ (also called *cardinality constraints*): Nonetheless, when possible we will show how to extend the encoding to deal with general coefficients. For each encoding, we will also show pros and cons of the encoding, some complexity issues, and an example.

The final goal of each encoding is a bits-based representation of the optimization function.

14.1 Naive encoding

This is probably the simplest encoding one can think of and it uses (full) adders in order to implement the encoding.

A full adder is a circuit showed in Figure 14.1 where x and y are the inputs that are to be added, b is the sum of the bits and C is the carry. b in defined as $x \oplus y$ and C as $x \wedge y$.

In order to generalize to n bits, using $x_0 \ldots x_{n-1}$, we can put "in cascade" several full adders in order to build the encoder. In the end, the bits that characterize the sum of the cardinality constraint are the outputs of the "most external" full adders, called $b_0, b_{log_2(n-1)}$; this means that $\sum_{i=o}^{n-1} x_i := \sum_{k=0}^{log_2(n-1)} 2^k b_k$.

From the complexity-side, the number of new variables that are to be added is: $(n-1) + (n-2) + \ldots + 1$, a quadratic number of new variables.

We will not show here examples regarding how the encoding works: It is sufficient to say that the encoding of the sum of 4 bits (or integer variables

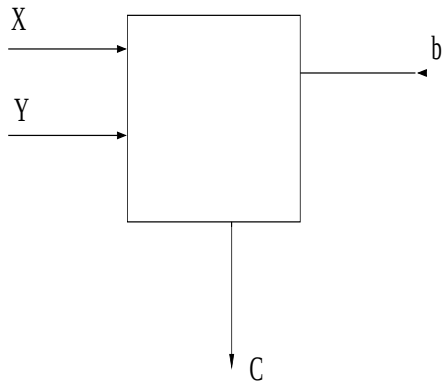


Figure 14.1: Full adder.

ranging over $\{0, 1\}$) with unitary coefficients consists of a CNF formula with 42 clauses.

Pros

- easy to implement

Cons

- could be a naive approach

In fact, some preliminary analysis using this encoding have shown that it leads to non-positive results in terms of CPU time.

For these reasons, the experimental evaluation showed in Chapter 16 will not include this encoding.

14.2 Warners's encoding

This encoding was presented in [Waa96].
It is characterized by a *binary* representation of the coefficients a_i as:

$a_i = \sum_{k=0}^{M-1} a_k^i 2^k$

where, given $a_{max} = max_i\ a_i$, M is the maximum natural number such that $M \leq 1 + log_2(a_{max})$.

We associate a propositional letter $\overline{p}_k^i$ with each coefficient a_k^i, and transform a_i as follow

$$trans(a_i) = \bigwedge_{k \in B_{a_i}} \overline{p}_k^i \wedge \bigwedge_{k \notin B_{a_i}} \neg \overline{p}_k^i \tag{14.1}$$

Furthermore, we associate a propositional letter p_{x_i} with the (binary) variable x_i. Recall that B_{a_i} is the set of indexes j for which $a_k^i = 1$.

Moreover, given I a set of indexes, we introduce the sets $\lfloor I \rfloor \subseteq I$ and $\lceil I \rceil \subset I$, such that the following holds:

$$\lfloor I \rfloor \cup \lceil I \rceil = I,\ \lfloor I \rfloor \cap \lceil I \rceil = \emptyset,\ |\lfloor I \rfloor| \geq |\lceil I \rceil|,\ |\lfloor I \rfloor| - |\lceil I \rceil| \leq 1,$$

The bits that represent the sum are denoted with $\{p_k^i\}_{k=0,\ldots,M_I}$, where $M_I = M + log_2(|I|)$, this means that

$$\sum_i a_i x_i = \sum_{k=0}^{M_I} p_k^i 2^k \tag{14.2}$$

The transformation of $\sum_i a_i x_i$ over the set of indexes I is defined recursively like:

$$trans(\textstyle\sum_{i \in I} a_i x_i) =$$

$$\begin{cases} trans(\sum_{i \in \lfloor I \rfloor} a_i x_i) \ \wedge \ trans(\sum_{i \in \lceil I \rceil} a_i x_i) \ \wedge \ T^+(p_k^I, p_k^{\lfloor I \rfloor}, p_k^{\lceil I \rceil}) & \text{if } |I| \geq 2 \\ trans(a_i) \ \wedge \ T^*(p_k^i, a_k^i, x_i) & \text{if I} = \{2\} \end{cases}$$

with operator $T^+(.)$ that defines the addiction of two numbers in binary notation and $T^*(.)$ that defines the multiplication of a number in binary notation with a binary number.

In more details, the transformation $T^+(p_k^U, p_k^V, p_k^W)$ with U = V ∪ W, V and W non-empty, is given by:

$$(p_0^U \leftrightarrow (p_0^V \leftrightarrow \neg p_0^W)) \wedge \tag{14.3}$$

$$(c_{01}^U \leftrightarrow (p_0^V \wedge p_0^W)) \wedge \tag{14.4}$$

$$\bigwedge_{k=1,\ldots,M_U} (p_k^U \leftrightarrow (p_k^V \leftrightarrow p_k^W \leftrightarrow c_{k-1,k}^U)) \wedge \tag{14.5}$$

$$\bigwedge_{k=1,\ldots,M_U-1} (c_{k+,k+1}^U \leftrightarrow (p_k^U \wedge p_k^W) \vee (p_k^V \wedge c_{k-1,k}^U) \vee (p_k^V \wedge c_{k-1,k}^U)) \wedge \tag{14.6}$$

$$(c_{M_U-1,M_U}^U \leftrightarrow p_{M_U}^U) \tag{14.7}$$

The transformation operator $T^*(.)$ is given by:

$$T^*(p_k^i, \overline{p}_k^i, p_{x_i}) = \bigwedge_{k=0}^{M_i} (p_k^i \leftrightarrow (\overline{p}_k^i \wedge p_{x_i})) \tag{14.8}$$

Till now we have been quite accurate, but a number of simplifications are possible by relaxing same requests.

Consider (14.3): We need for p_0^U to be true that either p_0^V or p_0^W is true we require equivalence. It is possible to relax the equivalence as an implication

$$((p_0^V \leftrightarrow \neg p_0^W) \rightarrow p_O^U) \tag{14.9}$$

p_0^U might be true when both p_0^V and p_0^W are false.

We can also replace the equivalences by implications in expression (14.4) to (14.7).

Moreover, by unit resolution between (14.1) and (14.8) we obtain

$$\bigwedge_{k \in B_{a_i}} (p_k^i \leftrightarrow p_{x_i}) \wedge \bigwedge_{k \notin B_{a_i}} \neg p_k^i \tag{14.10}$$

which has the CNF

$(\neg p_k^i \vee p_{x_i}) \wedge (p_k^i \vee \neg p_{x_i}), \forall k \in B_{a_i}$
$\neg p_k^i, \forall k \notin B_{a_i}$.

From the complexity point of view, for the general case, the number of new added variables var_m is $\theta(m\ log_2(a_{max}))$, and (more precisely) $var_m \leq 2m(2 + log_2(a_{max}))$; the number of newly introduced clauses cl_m has a similar complexity, and precisely $cl_m \leq 7m(3 + 2log_2(a_{max}))$.
Assuming that a_{max} is bounded, the procedure adds a *linear* number (over m) of variables and clauses.

Pros

- linear-time transformation
- same optimizations (described in [Waa96]) that lead to a less number of (variables and) clauses introduced by the encoding are possible

Cons

- the "theory" behind it is not simple
- maybe not so easy to implement

In order to clarify the above encoding, we report here the encoding of the sum of 4 bits.

Example 9 *We start by transforming each x_i into a propositional formula:*
$trans(x_1) = (p_0^1 \leftrightarrow p_{x_1}) \wedge \neg p_1^1 = (p_0^1 \vee \neg p_{x_1}) \wedge (\neg p_0^1 \vee p_{x_1}) \wedge \neg p_1^1$
$trans(x_2) = (p_0^2 \leftrightarrow p_{x_2}) \wedge \neg p_1^2 = (p_0^2 \vee \neg p_{x_2}) \wedge (\neg p_0^2 \vee p_{x_2}) \wedge \neg p_1^2$
$trans(x_3) = (p_0^3 \leftrightarrow p_{x_3}) \wedge \neg p_1^3 = (p_0^3 \vee \neg p_{x_3}) \wedge (\neg p_0^3 \vee p_{x_3}) \wedge \neg p_1^3$
$trans(x_4) = (p_0^4 \leftrightarrow p_{x_4}) \wedge \neg p_1^4 = (p_0^4 \vee \neg p_{x_4}) \wedge (\neg p_0^4 \vee p_{x_4}) \wedge \neg p_1^4$

Now, each of the sum $x_1 + x_2$ and $x_3 + x_4$ is translated in a CNF too:
$trans(x_1 + x_2) =$
$(p_0^{1,2} \leftrightarrow (p_0^1 \leftrightarrow p_0^2)) \wedge (c_{01}^{1,2} \leftrightarrow (p_0^1 \wedge p_0^2)) \wedge (p_1^{1,2} \leftrightarrow (p_1^1 \leftrightarrow p_1^2 \leftrightarrow c_{01}^{1,2})) \wedge (c_{12}^{1,2} \leftrightarrow (p_1^1 \wedge p_1^2) \vee (p_1^1 \wedge c_{01}^{1,2}) \vee (p_1^2 \wedge c_{01}^{1,2})) \wedge (p_2^{1,2} \leftrightarrow c_{12}^{1,2}) =$

$(p_0^{1,2} \vee p_0^1 \vee \neg p_0^2) \wedge (\neg p_0^{1,2} \vee p_0^1 \vee p_0^2) \wedge (p_0^{1,2} \vee \neg p_0^1 \vee p_0^2) \wedge (\neg p_0^{1,2} \vee \neg p_0^1 \vee \neg p_0^2) \wedge$
$(c_{01}^{1,2} \vee \neg p_0^1 \vee \neg p_0^2) \wedge (\neg c_{01}^{1,2} \vee p_0^1) \wedge (\neg c_{01}^{1,2} \vee p_0^2) \wedge$
$(\neg p_1^{1,2} \vee p_1^1 \vee p_1^2 \vee c_{01}^{1,2}) \wedge (p_1^{1,2} \vee \neg p_1^1 \vee p_1^2 \vee c_{01}^{1,2}) \wedge (p_1^{1,2} \vee p_1^1 \vee \neg p_1^2 \vee c_{01}^{1,2}) \wedge (p_1^{1,2} \vee p_1^1 \vee p_1^2 \vee \neg c_{01}^{1,2}) \wedge (p_1^{1,2} \vee \neg p_1^1 \vee \neg p_1^2 \vee \neg c_{01}^{1,2}) \wedge (\neg p_1^{1,2} \vee \neg p_1^1 \vee p_1^2 \vee \neg c_{01}^{1,2}) \wedge (\neg p_1^{1,2} \vee negp_1^1 \vee \neg p_1^2 \vee c_{01}^{1,2}) \wedge (\neg p_1^{1,2} \vee \neg p_1^1 \vee p_1^2 \vee \neg c_{01}^{1,2}) \wedge$
$(c_{12}^{1,2} \vee \neg p_1^1 \vee \neg p_1^2) \wedge (c_{12}^{1,2} \vee \neg p_1^1 \vee \neg c_{01}^{1,2}) \wedge (c_{12}^{1,2} \vee p_1^2 \vee \neg c_{01}^{1,2}) \wedge (\neg c_{12}^{1,2} \vee p_1^1 \vee p_1^2) \wedge$
$(\neg c_{12}^{1,2} \vee p_1^1 \vee \neg c_{01}^{1,2}) \wedge (\neg c_{12}^{1,2} \vee p_1^2 \vee \neg c_{01}^{1,2}) \wedge$

$(c_{12}^{1,2} \vee \neg p_2^{1,2} \wedge (\neg c_{12}^{1,2} \vee p_2^{1,2})$

$trans(x_3 + x_4) =$
$(p_0^{3,4} \leftrightarrow (p_0^3 \leftrightarrow p_0^4)) \wedge (c_{01}^{3,4} \leftrightarrow (p_0^3 \wedge p_0^4)) \wedge (p_1^{3,4} \leftrightarrow (p_1^3 \leftrightarrow p_1^4 \leftrightarrow c_{01}^{3,4})) \wedge (c_{12}^{3,4} \leftrightarrow (p_1^3 \wedge p_1^4) \vee (p_1^3 \wedge c_{01}^{3,4}) \vee (p_1^4 \wedge c_{01}^{3,4})) \wedge (p_2^{3,4} \leftrightarrow c_{12}^{3,4})$

with the set of CNF clauses following the same lines as in trans($x_1 + x_2$).
Finally, considering $U = \{1, 2, 3, 4\}$, $trans(x_1 + x_2 + x_3 + x_4) =$
$(p_0^U \leftrightarrow (p_0^{1,2} \leftrightarrow \neg p_0^{3,4})) \wedge (c_{01}^U \leftrightarrow (p_0^{1,2} \wedge \neg p_0^{3,4})) \wedge (p_1^U \leftrightarrow (p_1^{1,2} \leftrightarrow p_1^{3,4} \leftrightarrow c_{01}^U)) \wedge (c_{12}^U \leftrightarrow (p_1^{1,2}) \wedge (p_1^{1,2} \wedge c_{01}^U) \vee (p_1^{3,4} \wedge c_{01}^U))(p_2^U \leftrightarrow (p_2^{1,2} \leftrightarrow p_2^{3,4} \leftrightarrow c_{12}^U) \wedge (c_{23}^U \leftrightarrow (p_2^{1,2} \wedge p_2^{3,4}) \vee (p_2^{1,2} \wedge c_{12}^U) \vee (p_2^{3,4} \wedge c_{12}^U)) \wedge (c_{23}^U \leftrightarrow p_3^U)$
(the correspondent CNF clauses are not reported here).
It is sufficient to say that the encoding of this part is composed by 37 clauses.

The Warners encoding is composed by all the CNF clauses we have seen in this example.

⋆

What we have presented now is the "accurate" translation. Using (some of) the tricks described in the seminal paper we can substantially reduce the number of variables and clauses needed by the encoding.

14.2.1 Extension to general coefficients

There is no need to extend the procedure to work with general coefficients because the method presented already can do it. On the other hand, it is quite simple to adapt the method in the case of unitary coefficients (in order to be specific for this class, just considering $a_{max} = 1$).

14.3 Bailleux and Boufkhad encoding

This encoding was presented in [BB03].

It does not use a binary representation of integers, but a unary representation. The value of an integer x such that $0 \leq x \leq n$ is represented by 1 x times followed by 0 n - x times. The integer variable v with domain $0 \ldots n$ is represented by a set $V = \{v_1, v_2, \ldots, v_n\}$ of n propositional variables. Each possible value of v is encoded as a complete instantiation of V, as described above. If $v = x$, then $v_1 = 1, v_2 = 1, \ldots, v_x = 1$ and $v_{x+1} = 0, \ldots, v_n = 0$.

Consider now three set of variables:

- $X = \{x_1, \ldots, x_n\}$: The set of *input variables*
- $B = \{b_1, \ldots, b_n\}$: The set of *output variables*
- a set L of variables called *linking variables*

These sets of variables can be described by a binary tree built as follow. We start from an isolated node labeled by the integer n and we proceed iteratively: To each terminal node labeled by $m > 1$, we connect two children labeled by $\lfloor m/2 \rfloor$ and m - $\lfloor m/2 \rfloor$, respectively. This procedure produces a binary tree

with n leaves labeled by 1. Next, each variable in X is allocated to a leaf in a bijective way. The set B of output variables is allocated to the root node. To each internal node labeled by an integer m, a set of m new variables is allocated which will be used to represent a unary value belonging to $1 \ldots m$. The union of the set of variables allocated to the internal nodes is the set L of linking variables.

We will now define a set of clauses that ensures that in any complete instantiation of the variables, the set of variables related to any non-leaf node r with children p and q encode the unary representation of $\alpha + \beta$, where α and β are the integers encoded by the sets of variables related to p and q.

Let r be an internal node related to children p and q. Let $R = \{r_1, \ldots, r_m\}$ be the set of variables related to r, $P = \{p_1, \ldots, p_{m_1}\}$ be the set of variables related to p, and $Q = \{q_1, \ldots, q_{m_2}\}$ be the set variables related to q

The following conjunction of clauses is related to the node r:

$$\bigwedge C_1(\alpha, \beta, \sigma) \wedge C_2(\alpha, \beta, \sigma) \tag{14.11}$$

where $0 \leq \alpha \leq m_1$, $0 \leq \beta \leq m_2$, $0 \leq \sigma \leq m$ and $\alpha + \beta = \sigma$, and with the following notations:

$$p_0 = q_0 = r_0 = 1, p_{m_1+1} = q_{m_2+1} = r_{m+1} = 0 \tag{14.12}$$

$$C_1(\alpha, \beta, \sigma) = \neg p_\alpha \vee \neg q_\beta \vee r_\sigma \tag{14.13}$$

$$C_2(\alpha, \beta, \sigma) = p_{\alpha+1} \vee q_{\beta+1} \vee \neg r_{\sigma+1} \tag{14.14}$$

Notice that $C_1(\alpha, \beta, \sigma)$ is the CNF representation of the relation $\sigma \geq \alpha + \beta$ and $C_2(\alpha, \beta, \sigma)$ is the CNF representation of the relation $\sigma \leq \alpha + \beta$.

The obtained formula is simplified according to the values in (14.12). Notice that each clause includes at most three literals.

The bits that encode the sum are the bits in B following the unary representation of this encoding.

There are two main Lemmas that show the "good" properties of the encoding w.r.t. unit-propagation.

Lemma 1 (forward propagation)
Let $\phi(X)$ be a sum of n input variables. If p input variables are fixed to 1, q input variables are fixed to 0, and all the other variables of the sum are free then the partial instantiation I_B of B obtained after unit propagation in $\phi(X)$ is pre-unary and such that $\min(I_B)$ = p and $\max(I_B)$ = n - q.

Lemma 2 (backward propagation)
Let $\phi(X)$ be a sum with n input variables. If:

- p input variables are fixed to 1, q input variables are fixed to 0 ($p+q < n$), the remaining input variables being free,
- and the output variables s_{p+1} to s_{n-q} are all fixed to the same value ϵ (0 or 1)

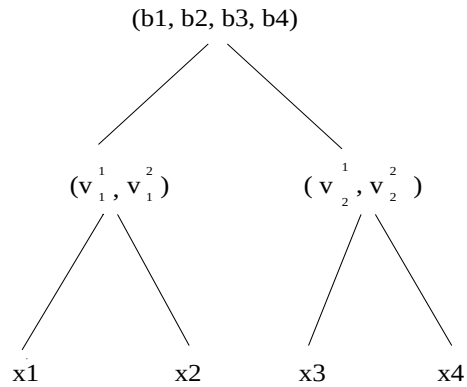


Figure 14.2: Example of Boufkhad with 4 bits.

then all the input variables remaining free are instantiated to ϵ by unit propagation.

We will not prove these Lemmas in details, because they are already proved in [BB03]. Nevertheless we will use these Lemmas to give the related intuitions when we will optimize the encoding (in Subsec. 14.3.1), and when we will adapt it to work with general coefficients (in Subsec. 14.3.2).

Pros

- The proposed encoding is efficient with respect to unit propagation
- It is simple to implement
- The encoding seems to have a "compact" representation

Cons

- For high input/output variables n, the number of clauses generated could not be manageable because it grows quadratically

From the complexity side, the binary tree used to specify the encoding has $\theta(nlog_2(n))$ levels. Each of these levels requires n linking variables. Let us consider, for simplicity, that n is a power of 2. For each node related to a set of m linking (or output) variables, there are less than $2m^2$ clauses. Let us number the levels from 1 to l, where l is the number of the root level. For any i such that $1 \leq i \leq l$, the level $l - i$ includes 2^i nodes, each related to $n/2^i$ variables. Then the level $n - i$ includes less than $2^i(2(n/2^i)^2) = 2n^2/2^i$ clauses. So the encoding includes $O(n^2)$ clauses. Given that the root node requires $\omega(n^2)$ clauses, so the encoding requires $\theta(n^2)$ clauses.

We make the same example as in Section 14.2, regarding the encoding of the sum of the four bits $x_1 + x_2 + x_3 + x_4$ in CNF.

Example 10 *Consider Fig. 14.2. The tree has x_1, x_2, x_3, x_4 as input variables, $v_1^1, v_1^2, v_2^1, v_2^2$ as linking variables and b_1, b_2, b_3, b_4 as input variables. Following the same schema as in Example 9,*

$trans(v_1^1 + v_1^2) =$
$(\neg v_1^1 \vee x_1 \vee x_2) \wedge (v_1^1 \vee \neg x_1) \wedge (v_1^1 \vee \neg x_2) \wedge (v_1^2 \vee \neg x_1 \vee \neg x_2) \wedge (\neg v_1^2 \vee x_1) \wedge (\neg v_1^2 \vee x_2)$
$trans(v_2^1 + v_2^2) =$
$(\neg v_2^1 \vee x_3 \vee x_4) \wedge (v_2^1 \vee \neg x_3) \wedge (v_2^1 \vee \neg x_4) \wedge (v_2^2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg v_2^2 \vee x_3) \wedge (\neg v_2^2 \vee x_4)$
$trans(b_1 + b_2 + b_3 + b_4) =$
$(\neg b_1 \vee v_1^1 \vee v_1^2) \wedge (b_1 \vee \neg v_1^1) \wedge (b_1 \vee \neg v_1^2) \wedge (b_2 \vee \neg v_2^2) \wedge (\neg b_2 \vee v_1^1 \vee v_2^2) \wedge (\neg b_2 \vee v_2^1 \vee v_1^2) \wedge (b_2 \vee \neg v_1^1 \neg v_1^2) \wedge (b_2 \vee \neg v_2^1)$ $(\neg b_3 \vee \neg v_1^1) \wedge (b_3 \vee \neg v_1^1 \vee \neg v_2^2) \wedge (\neg b_3 \vee v_2^1 \vee v_1^2) \wedge (b_3 \vee \neg v_2^1 \neg v_1^2) \wedge (\neg b_3 \vee v_1^2)$ $(\neg b_4 \vee v_2^1 \vee v_2^2) \wedge (b_4 \vee \neg v_2^1) \wedge (b_4 \vee \neg v_2^2)$

$\star$

14.3.1 Optimizing the encoding

Following the properties of Lemma 1 and Lemma 2, the encoding takes information preferentially from the input variables. There is the possibility to improve the encoding imposing some conditions on the values of the output variables. Starting from the unary representation of the integers, we know that:

$$if\ b_i = 1, for\ each\ j < i, b_j = 1 \tag{14.15}$$

$$if\ b_i = 0, for\ each\ j > i, b_j = 0 \tag{14.16}$$

The two constraints together indicate that there exists a relation of the type $b_j \rightarrow b_i(\neg b_j \vee b_i)$ for $j > i$. The relation behind it is that if $b_j = 0$ the clause is satisfied no matter the value of b_i, but if $b_j = 1$ also b_i has to be set to 1 as well in order to satisfy the clause and in syncrone with the semantic of the encoding. Similar consideration can be done when we fix b_i.

It is sufficient a simple example with 4 bits to show that, giving some values assigned to some x_i, new unit propagations get assigned with these additional constraints that would not have been assigned without them.

Example 11 *The encoding is composed by all the clauses in the "basic" version in the Example 10, plus the following clauses:*
$(b_2 \rightarrow b_1) \wedge (b_3 \rightarrow b_2) \wedge (b_4 \rightarrow b_3) \equiv (\neg b_2 \vee b_1) \wedge (\neg b_3 \vee b_2) \wedge (\neg b_4 \vee b_3)$ $\star$

How can these new clauses add reasoning? This is quite simple to see: Suppose b_2 gets assigned to TRUE using the basic form of the encoding, some of clauses including b_2 are simplified, others are satisfied, but no other simplifications can be performed. Otherwise, if we add these further clauses, once b_2 is assigned to TRUE also b_1 can be assigned to TRUE by unit propagation leading to other simplifications in the clauses where b_1 appears.

Note
The additional clauses (and the additional reasoning) added by this modification do not hit the correctness of Lemma 1 and Lemma 2.

14.3.2 Extension to general coefficients

A simple way to extend the encoding to general coefficients (the same can be applied also to the naive encoding) is to consider, for each i, $a_i x_i$ as $\sum_{j=1}^{a_i} x_{i,j}$,

to have a_i different x_i variables with coefficient 1. Now we can simply run the algorithm on a problem with $\sum_i a_i x_i$ input variables with unitary coefficients. Moreover, we know that all the a_i variables related to the same original variable x_i are the same, even if they have different "name". So, we can safely add the equivalences:

$$x_{i,j} \leftrightarrow x_{i,j+1} \ for \ j = 1, \ldots, a_i - 1. \tag{14.17}$$

Still. the good properties of the encoding w.r.t. unit propagation remain unchanged. We re-formulate the Lemmas in Section 14.3

Lemma 1a (forward propagation with general coefficients)
Let $\phi(X)$ be a sum with n input variables, with coefficients $a_1, \ldots, a_n$. If p input variables are fixed to 1, q input variables are fixed to 0, and all the other variables of the sum are free then the partial instantiation I_B of B obtained after unit propagation in $\phi(X)$ is pre-unary and such that $\min(I_B) = \sum_{i=1}^{p} a_i x_i$ and $\max(I_B) = \sum_{i=n-q}^{n} a_i x_i$.

In the above Lemma we have seen each variable associated with the related coefficient. There is an alternative formulation that is more related to the particular translation of the problem with general coefficients to a problem with unitary coefficients explained in this section.

Lemma 1b (forward propagation with general coefficients) Let $\phi(X)$ be a sum with n input variables, with coefficients $a_1, \ldots, a_n$. If $q_{lb} = \sum_{i=1}^{p} a_i x_i$ input variables are fixed to 1, $q_{ub} = \sum_{i=n-q}^{n} a_i x_i$ input variables are fixed to 0, and all the other variables of the sum are free then the partial instantiation I_B of B obtained after unit propagation in $\phi(X)$ is pre-unary and such that $\min(I_B) = \sum_{i=1}^{p} a_i x_i = q_{lb}$ and $\max(I_B) = \sum_{i=n-q}^{n} a_i x_i = q_{ub}$.

Lemma 2a (backward propagation with general coefficients)
Let $\phi(X)$ be a sum with n input variables with coefficients $a_1, \ldots, a_n$. If:

- p input variables are fixed to 1, q input variables are fixed to 0 ($p+q < n$), the remaining input variables being free,
- and the output variables from $x_{q_{lb}+1}$ to $x_{q_{ub}-1}$ are all fixed to the same value ϵ (0 or 1)

then all the input variables remaining free are instantiated to ϵ by unit propagation.

As for the Lemma 1, there is an alternative formulation.
Lemma 2b (backward propagation with general coefficients)
Let $\phi(X)$ be a sum with n input variables. If:

- $q_{lb} = \sum_{i=1}^{p}$ input variables are fixed to 1, $q_{ub} = \sum_{i=n-q}^{n} a_i x_i$ input variables are fixed to 0 ($p + q < n$), the remaining input variables being free,
- and the output variables from $x_{q_{lb}+1}$ to $x_{q_{ub}-1}$ are all fixed to the same value ϵ (0 or 1)

then all the input variables remaining free are instantiated to ϵ by unit propagation.

Chapter 15

The OPTSAT solver

The OPTimal SATisfiability (OPTSAT) solver is based on propositional satisfiability for solving optimization problems related to SAT. The schema of the system is presented in Figure 15.1.
The OPTSAT main module takes in input a propositional formula in CNF format ϕ and a functional cost F (that has to be minimized). The functional cost is then encoded in a CNF formula ϕ_{CNF} (via the encoder). The encoder also outputs the "preferential" variables $b_0, \ldots, b_m$, that are fed to the SAT solver with the extended CNF formula $\phi \wedge \phi_{CNF}$ (ϕ must be modified if we deal with a Max(Min)-SAT problem).

The informations needed by the SAT solver are given via file in CNF format improved with an header that describe the type of optimization we want to perform on the formula (.optsat file). The header of the .optsat file looks like:
c
c min/max SAT/ONE
c
c WEIGHT p
c $w_1 w_2 \ldots w_p$ 0
c
p cnf n m
...
...

Where:

- the 2 bits min/max and SAT/ONE are used to define the type of problem we want to solve
- p is the number of variables that characterize the functional cost
- $w_1 \ldots w_p$ is the list of weights associated with the variables that define the functional cost. The list is 0-terminated.
- "p cnf n m" is the header of the propositional CNF formula.

If a problem is a Min(Max)-SAT problem, $p = m$, if it is a Min(Max)-ONE problem $p = n$. If the problem is un-weighted, $w_1 = w_2 = \ldots = w_p = 1$, otherwise $w_i \geq 1$, $w_i \in N$ (is a natural number), $1 \leq i \leq p$.

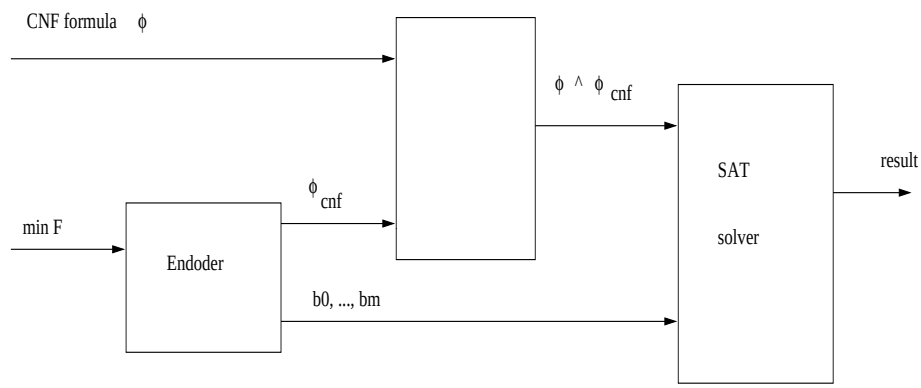


Figure 15.1: The OPTSAT system.

Given the CNF formula ϕ defined in the input file, and ϕ_{CNF} the CNF formula that results from the encoding using one of the algorithm described in Chapter 14, the optimization problem boils down (under some assumption that we will describe later) to check the satisfiability of the improved propositional formula:

$$\phi_{opt} := \phi^{'} \wedge \phi_{CNF}$$

where $\phi^{'}$ is ϕ if we are dealing with a Min(Max)-ONE problem, otherwise $\phi^{'}$ is the formula ϕ in which each clause $C_i \in \phi$ is modified as $\neg s_i \vee C_i$ with s_i a newly introduced variable. The newly introduced variables $s_1, \ldots, s_p$ are called *clause selectors*. Intuitively, a clause C_i is (not) in the best solution if the correspondent s_i is assigned to FALSE (TRUE).

The functional cost is defined on the input variables of ϕ for the Min(Max)-ONE problem, and on the s_i variables for the Min(Max)-SAT problem.

The main property of the approach we are proposing (w.r.t. all the other approaches that use "branch and bound" or "branch and cut" algorithms) is that the first satisfying assignment the (back-end) SAT solver finds is the "optimal" one: But if we just run a SAT solver on ϕ_{opt}, we would probably find a solution very quickly, but unfortunately, the solution will not be optimal.

In order to guarantee[1] the property, we have to impose to some variables a particular order and value: This is done by modifying the heuristic of the underlined solver.

What are the variables we have to impose their values? In each encoding we have presented, there is a bunch of variables that represent the value of the related optimization function. For Boufkhad encoding 14.3, the variables are simply the output variables of the encoding (denoted as the set $B = \{b_1, \ldots, b_n\}$). For Warners encoding (in Sec. 14.2), these variables are the p_k^i variables (easily follow from Equ. 14.2). For the naive encoding, they are the bits $b_0, \ldots, b_{log_2(n-1)}$, with n the number of bits/input variables to be added.

[1] We do not have yet a formal statement about the correctness of the approach, but we give here precise intuition for it.

Consider now that, independently from the encoding, b_b being the most significant bit (MSB) and b_0 (Warners and naive encoding) or b_1 (Boufkhad encoding) being the least significant bits (LSB).

The algorithm has to assign, in order, from the variables b_b (MSB) to b_0 or b_1 (LSB). The variable b_i has to be assigned to TRUE if we are dealing with a Max (SAT or ONE) problem, or to FALSE if we are working with a min (SAT or ONE) problem. Consider the "max" case, and the Boufkhad encoding. Assigning b_b to TRUE means to check if the result of the optimization problem is b; in general assigning b_i to TRUE means to check if the result of the optimization problem is i. To be noticed that, given the particular nature of the encoding, as soon as the first $i = 1 \ldots b$ does not lead to any contradiction (and a solution to ϕ_{opt} is found), the solution to the problem is found. For the "min" case, and for similar reason, the variables are set to FALSE. In the "max" case, and using the Warners or naive encoding, setting a variable b_i to TRUE means to check if the solution is between 2^i and $2^{i+1} - 1$. When a contradiction is found, is known that the solution is within the complementary bound; in any case a large portion of the search space is cut.

In order to implement the above described procedure, we have modified again the DLL algorithm described in Sec. 2.3. The resulting algorithm, called OPTSAT, is described in Fig 15.2.

function OPTSAT(ϕ_{opt}, μ, *pref*, *ismax*)
1 **if** $\{\} \in \phi_{opt}$ **then return** FALSE;
2 **if** $\phi_{opt} = \emptyset$ **then return** ComputeSolution(μ;)
3 **if** $\{l\} \in \phi_{opt}$ **then return** DLL(Simplify(l,ϕ_{opt}),$\mu \wedge l$);
4 $l :=$ ChooseLiteralPref(ϕ_{opt}, *pref*, *ismax*);
5 **return** DLL(Simplify(l,ϕ_{opt}),$\mu \wedge l$) or
DLL(Simplify($\bar{l}$,ϕ_{opt}),$\mu \wedge \bar{l}$)

Figure 15.2: The OPTSAT algorithm.

From Fig. 15.2, we can see that the DLL algorithm is modified not only in one of the basic cases when the formula is satisfiable (as happened in the last 2 Parts), but also in the heuristic. The algorithm takes is input the propositional formula ϕ_{opt}, the stack of assigned literals μ (initially empty), the list $pref = \{LSB, \ldots, b_b\}$, and the boolean variable $ismax$ that is TRUE if the we are dealing with a "max" case, and FALSE if we are dealing with a "min" case.

When the formula is satisfiable, function ComputeSolution(μ) takes in input the satisfying assignment μ and just counts the number of input variables in μ that are assigned to TRUE (or, depending how the problem with general coefficients is dealt, summing the weight of the variables assigned to TRUE).

In Fig. 15.3, the algorithm refers to the new implementation of the function CHOOSELITERAL (called here CHOOSELITERALPREF) that chooses the next variable to work on, taking into account the order on the variables that the approach uses. Let, given a list a, a.size() be the number of elements in a, and $a[i]$ the i-th element of a.

The procedure looks for "preferential" variables (in a given order) not yet assigned. If there exists such a variable in this status, it is assigned according to the type of problem ("min" or "max") we are dealing. If all the "preferential"

```
function ChooseLiteralPref(φ_opt, pref, ismax)
    for (i = (pref.size() - 1); i >= 0; i--)
        if pref[i] = UNDEF then return (ismax ? pref[i] : ¬pref[i]);
    l := ChooseLiteral(φ_opt);
    return l
```

Figure 15.3: The ChooseLiteralPref function.

variables are assigned, a "default" heuristic is called in order to choose the next variable.

The ChooseLiteralPref presented here (and the implementation in OPTSAT) is not optimal because the *pref* list is scan every time the procedure is called. A simple way to optimize the computation would be to maintain a pointer to the last assigned variable in the list.

15.1 Propositional solvers tested

In order to decide the satisfiability of the back-end CNF formula resulting from our approach, we have modified and used two SAT solvers: SIMO and zChaff. While SIMO is our home made SAT solver (described in Section 2.4), zChaff [MFM05] is the winner of the last SAT 2004 Competition [LS05] on "real-world " CNF formulas.

(The new version of) zChaff and SIMO are quite similar, but zChaff can rely on efficient low level optimizations of the code (e.g., profiling-based implementation in order to minimize the missing in cache), and very recent advancements in all the principal components of DLL-based solvers.

In order to test several heuristics (and optimizations), we have implemented some well-known state-of-the-art heuristics (ChooseLiteral) in SIMO:

- VSIDS from [MMZ+01] (denoted with "V");
- unit (given an unassigned atom p, while doing failed literal on p we count the number $u(p)$ of unit-propagation caused, and then we select the atom with maximum $1024u(p) \times u(\overline{p}) + u(p) + u(\overline{p})$. This heuristic is denoted with "U").
- unit with pool, where (i) we first select a pool of 10 "most watched" variables, and (ii) we perform failed literal and score accordingly only the variables in the pool.[2] This heuristic is denoted with "P"). The idea of this heuristic comes from [LA97]

zChaff uses an heuristic that is an evolution of the VSIDS heuristic.

[2] Our pooling criteria is motivated by the fact that we are using a solver with literal watching. While this data structure is good for fast unit-propagation, does not allow to properly select the "most promising" variables using better criteria which take into account all the occurrences of a variable. To have such pooling criteria without sacrificing the efficiency of unit propagation, the obvious solution is to have an additional data structure to be used just for pooling.

Chapter 16

Experimental analysis

In this Chapter, we first present the benchmarks used for testing the optimization problems; than we present the state-of-the-art systems we evaluated; and finally some comparative experimental evaluation we conducted.

16.1 Benchmarks

In order to test the efficiency of the approach, we have tested our system (as well as other systems we will see in details in Section 16.2), both on randomly generated benchmarks and on "real-world" problems (CNF formulas arising from real applications). In the next two subsections we will see in details the benchmarks used.

In the description, we focus on the Max-SAT and Max(Min)-One problems. Moreover we will focus on the un-weighted version of these problems.

It is important to note that we have used different benchmarks for solving the two problems: This is because, in general, the Max-SAT problem needs formulas to be unsatisfiable in order to be "significant", while the Min(Max)-One problem needs formulas to be satisfiable.

16.1.1 Randomly generated problems

Random propositional benchmarks are generated using the model of generation proposed in [SLM92]. Given three parameters:

- the number of propositional variables n
- the number of clauses m
- the number k of literals that appear in each clause

a formula is generated by randomly choosing k different variables in the pool of n, negating each variable with probability 0.5. The k literals together form a clause that is added to the formula. The procedure continues until m clauses are generated.

The ratio $r = m/n$ characterizes the difficulty and the nature of the problem (for each k). There are values of r (given k) for which the benchmarks are expected to be "harder" than other ratios. These particular ratios occur in a

"region" where the probability that a formula is satisfiable drops (more or less rapidly depending on k) from near 1 to near 0.

As long as these benchmarks are randomly generated, for each triple $< n, m, k >$ several (100 for Max-SAT, 10 for Min(Max)-ONE) benchmarks are generated.

We will see in more details in the experimental section (16.3) the precise data we have tested: As a general intuition we concentrated in a region near to the phase transition (where the correspondent propositional formula is supposed to be harder), because we were interested in studying the behavior of the algorithms in this region. Study the problems in this area means, e.g. for the Max-SAT problems, that most of the problems are unsatisfiable and the results (number of satisfiable clauses) is the number of clauses minus 1.

In general, the problems in this area are not the hardest that can be generated for Max-SAT. Usually, the higher is the constraindeness r of the problem, the harder the formula.

16.1.2 Real-world problems

For real-world problems. we have tested well-known problems that come from several application domains. In particular, for Max-SAT we have used:

bmc 30 instances from CMU described in [BCCZ99];

miters 25 instances encoding problems of Combinational Equivalence Checking described, e.g., in [SG99].

For the Min(Max)-One problem:

Beijing'96 some of the 16 instances from the Beijing'96 SAT competition, a mixture of scheduling [CB94], planning and circuit verification problems;

DES 24 Data Encryption Standard problems, see [MM00, Li00];

Planning planning problems generated by BlackBox [KS98];

QGroup Quasi group problems.

16.2 Rival systems

Beyond testing different options in OPTSAT, we have also tested other state-of-the-art systems, and compared the result w.r.t. the results of our system. The systems we have considered are, for the Max-SAT problems:

bf a DLL-based solver developed by Borchers and Furman [BF99];

sz a DLL-based solver developed by Shen and Zhang [SZ04];

MaxSolver a DLL-based solver developed by Xing and Zhang [XZ05];

PBS a Pseudo-Boolean Optimization solver. developed by Aloul [ARMS00]

At the moment. we have taken into account only the BF solver: This is because, SZ can work only on Max-SAT problems with two literal per clause (Max-2SAT); MaxSolver can only deal with problems that contain up to 200 variables and 1000 clauses, otherwise it incurs in memory overflow;[1] and PBS (beyond that is best suited for the Min(Max)-One problem) was not competitive (especially on randomly generated problems).
The systems we have considered for the Max-SAT problems, are

PBS;

opbdp a solver developed by P. Barth [Bar95].

16.3 Results

In this section we will see some (preliminary) results about the analysis we have conducted on our system OPTSAT and on the rival systems. The analysis will be divided in two parts: The analysis for Max-SAT[2] and for Min(Max)-ONE problem.

In the Tables, "b", "bm" and "w" indicate the Boufkhad, Boufkhad modified and Warners (without any optimizations) encoding respectively. #CNF is the number of clauses that are fed to the SAT solver. optz and opts indicate that the OPTSAT solver is using the SAT solvers zChaff and SIMO respectively. Time out ("T", "TIME") is set to 600 sec for each problem (but the real-world Max-SAT where is 1800 sec), and memory limit ("MEM") is set to 900 MB.

Heuristic "U" has not been shown because clearly not competitive w.r.t. the others, as well as Warners encoding with opts.

16.3.1 Analysis on Max-SAT problem

In the Table 16.1:

#var and #cl are the number of propositional variables and clauses of the starting propositional formula respectively.

ub (upper bound) is the result (ub = i means that the Max-SAT result is #cl - i)

#ist is the number of instances that have the correspondent result.

line marked by "T" is the number of instances that reached the time limit.

The other columns being the CPU time of the various versions of the solver. A dash line "–" indicates that the solver can not solve the instance(s) with the correspondent value of "ub" within the time limit.

In Table 16.1, the analysis on randomly generated benchmarks for the Max-SAT problem. We have generated problems with 1500 and 2000 variables, with 1800 and 2400 clauses respectively, 100 instances per point. The benchmarks we have generated contain 2 literals for clause.

[1] Z. Xing, personal communications

[2] We do not consider here the Min-SAT problem. This is because usually the analysis on this type of problem is performed on the Max case.

#var ub	#cl #ist	#CNF b bf	#CNF bm optz b	#CNF w optz bm	optz w	opts b V	opts b P	opts bm V	opts bm P
1500	1800	3279104	3280903	66567					
0	24	3.94	5.2	5.24	26.76	6.52	9.64	6.35	9.41
1	33	7.96	5.55	5.49	27.93	6.90	10.21	6.70	9.96
2	22	8.29	5.90	5.80	28.31	7.31	11.07	7.07	10.83
3	14	29.27	7.50	6.78	29.90	8.56	14.48	8.57	13.54
4	6	131.44	42.82	13.95	46.16	28.42	28.48	31.61	26.43
5	1	–	569.10	107.74	145.85	288.67	137.50	239.49	134.57
T		1	0	0	0	0	0	0	0
2000	2400	5814208	5816607	93281					
0	13	9.89	9.88	9.73	50.71	12.40	18.35	11.84	17.64
1	30	14.12	10.55	10.20	51.11	13.22	19.44	12.51	18.59
2	29	15.05	11.34	10.80	52.87	14.09	21.14	13.22	20.13
3	13	76.88	17.00	12.65	55.45	15.68	25.52	14.64	25.19
4	10	180.59	143.18	34.30	142.33	124.91	67.86	105.46	74.72
5	1	–	351.25	135.53	273.13	473.47	294.76	334.09	261.04
T		12	4	2	2	4	1	1	0

Table 16.1: Randomly generated benchmarks. 2SAT problems.

All the versions of OPTSAT presented perform better than bf; in particular optz with "bm" encoding outperforms bf up to one order of magnitude and solve more instances. The efficiency seems to rely on the additional clauses (and reasoning) that the encoding add w.r.t. the "plain" "b". Quite interesting are also the performances of optz with Warners encoding. From the comparison between the columns related to opts "b" and "bm" we can see that heuristic "p" performs better than heuristic "v".

We have also run some experiments with randomly generated problems with three literals per clause: We do not report any result here because the study is still at an early stage. Preliminary results point out that bf performs better than OPTSAT, and probably some specific domain reasoning should be added in order for OPTSAT to become competitive also on this domain.

In Table 16.2, the analysis on real world problems. The problems bar* (barrel), long* (longmult) and que* (queueinvar) are unsatisfiable bounded model checking problems (described in [BCCZ99]). The c* are encoding problems of Combinational Equivalence Checking described, in [SG99];

Again, result using heuristic "u" has not been shown because not competitive w.r.t. the other version. We also do not show any results for "bm" encoding encoding: The results are very similar to the one for "b". We do not report results on opts w for lack of space and because they are always worse in comparison with optz w. A comparison between "v" and "p" is outlined using "b" encoding.

In order to tackle the problems of these domains, we had to modify bf in order to enhance its static limits on #var and #cl.

In the Table, column "ist" refers to the instances, and the second column #CNF reports the number of clauses of ϕ_{opt}, using "b" / "w" encodings respectively. A dash line "–" here indicates that the number of clauses generates exceeded the memory limit imposed.

For the bmc problems, in the easiest instances of each sub-domain (bar*, long* and que*) bf performs better than any option of OPTSAT. Among the OPTSAT's options, optz with "b" heuristic is the best, and the VSIDS heuristic "v" is slightly better than "p" (even there are no significant differences). When

ist	#CNF	bf	optz b	optz w	opts b V	opts b P
bar2	27631 / 5950	0.02	0.05	0.13	0.06	0.06
bar3	906000 / 35519	0.23	1.46	6.36	1.75	1.92
bar4	$> 4\ 10^6$ / 82124	0.65	6.92	40.41	8.72	9.2
bar5	– / 202480	21.42	MEM	260.97	MEM	MEM
long0	$> 1\ 10^6$ / 46721	0.39	2.53	9.56	3.07	3.20
long1	$> 5\ 10^6$ / 91626	22.26	10.07	51.52	12.43	12.92
long2	– / 129998	57.11	MEM	117.47	MEM	MEM
long7	– / 393514	–	MEM	1061.61	MEM	MEM
long8	– / 435148	–	MEM	1729.87	MEM	MEM
que2	166157 / 14418	0.04	0.3	0.56	0.34	0.37
que4	930987 / 36234	0.17	1.51	6.55	1.80	2.11
que6	$> 5\ 10^6$ / 89636	0.47	8.65	49.86	12.07	13.71
que8	$> 5\ 10^6$ / 89852	0.62	9.80	50.36	12.92	15.24
que10	– / 208933	2.21	MEM	409.09	MEM	MEM
que14	– / 365920	–	MEM	1145.27	MEM	MEM
que16	– / 237459	–	MEM	517.46	MEM	MEM
c432	$> 1\ 10^6$ / 44264	131.06	2.33	9.99	2.81	6.97
c432-s	$> 1\ 10^6$ / 44615	157.25	2.38	10.64	2.98	6.53
c499	$> 3\ 10^6$ / 70417	TIME	6.49	29.91	77.11	743.55
c499-s	$> 3\ 10^6$ / 70856	TIME	6.46	33.78	32.49	651.97
c880	$> 6\ 10^6$ / 98411	TIME	12.65	56.86	MEM	MEM
c880-s	$> 6\ 10^6$ / 98411	TIME	12.64	56.77	MEM	MEM
c1355	– / 136707	TIME	MEM	138.80	MEM	MEM
c1355-s	– / 137147	TIME	MEM	142.95	MEM	MEM
c1908	– / 194731	TIME	MEM	237.09	MEM	MEM
c1908-s	– / 195055	TIME	MEM	229.22	MEM	MEM
c1908.b	– / 194839	1705.69	MEM	232.01	MEM	MEM

Table 16.2: Real world problems. Max-SAT problem.

the hardness of the instances increase, bf and optz start to incur in segmentation faults (denoted with "–") and memory overflow (due to the very high number of propositional clauses generated with the encoding); Warners encoding gives his best and (in conjunction with optz) can solve within the time limit all the instances presented. This is because, thanks to the size of the formula ϕ_{opt}, the solver does not incur in any memory problems on these instances.

For the miters (c* in the Table) instances, OPTSAT solver performs much better than bf. In particular, optz with Boufkhad encoding performs best, outperforming bf up to three orders of magnitude in the "smaller" instances. And, given the results in the last two columns, the "V" heuristic is much powerful on these benchmarks than "P" heuristic. optz using Warners encoding is about a factor of 5 slower than the same version using Boufkhad, but as soon as the dimension of the problem causes the Boufkhad encoding to exceed the memory limit, "optz w" is the only version that it is able to solve the "bigger" instances presented (but the last row), and however if outperforms bf also in the last instance presented.

16.3.2 Analysis on Min(Max)-ONE problem

In this Subsection we report on some results that we have obtained on Min(Max)-ONE problem, using our solver, PBS and opbdp. PBS has been run with the

#var	#cl	#CNF b	#CNF bm	#CNF w	PBS	opbdp	optz b	optz bm	optz w
200	50	42938	43137	7019	0.19	0.76	0.17	0.17	0.96
200	100	42988	43187	7069	47.12	16.91	4.73	3.76	TIME
200	150	43038	43237	7119	16.87	14.54	7.63	10.04	148.30
200	200	43088	43287	7169	26.46	45.95	5.84	6.37	TIME

Table 16.3: Randomly generated benchmarks. Min-ONE 2SAT problems.

#var	#cl	#CNF b	#CNF bm	#CNF w	PBS	opbdp	optz b	optz bm	optz w
200	50	42938	43137	7019	0.08	0.29	0.16	0.16	1.2
200	100	42988	43187	7069	40.35	24.17	1.42	1.68	107.96
200	150	43038	43237	7119	35.66	45.4	6.67	6.8	267.41
200	200	43088	43287	7169	18.66	16.66	8.44	8.59	165.22

Table 16.4: Randomly generated benchmarks. Max-ONE 2SAT problems.

option "-D 1" as suggested by the authors.[3]

Among the OPTSAT options, we have used only the optz because it almost always leads to the best results (w.r.t. opts).

In Tables 16.3, 16.4 16.5 and 16.6 the analysis on randomly generated Min-ONE and Max-ONE problems using the same model generation we introduced in Subsec. 16.1.1. The 2SAT instances we have run contain 200 variables, and 50, 100, 150, and 200 clauses respectively, while the 3SAT instances contain 120 variables, and 120, 240, 360 and 480 clauses respectively. For each ratio, 10 instances were generated.

For Min-ONE 2SAT benchmarks (Table 16.3), optz with "b" and "bm" encodings perform best (especially on problems with 100 clauses), but for the case with 50 clauses where PBS is faster than optz (but these problems are quite easy to solve). "w" encoding is not effective on this domain. For Min-ONE 3SAT benchmarks (Table 16.5), opbdp seems to be the best performing solver overall, while optz with "b" encoding performs best than opbdp on problems with 120 and 240 clauses. optz with "w" encoding and PBS are not competitive on this domain but PBS with problems with 480 clauses.

For the Max-ONE benchmarks (Tables 16.4 and 16.6) the situation is similar w.r.t. the Min-ONE problem, "w" encoding seems to have less negative results on this domain in the 2SAT instances.

In Tables 16.7 and 16.8 the analysis on real-world benchmarks, for Min-ONE and Max-ONE problem respectively. For Min-ONE, optz (with "b" or "bm" encodings) performs better than PBS and opbdp on Beijing (2bit*) and much better on QGroup (qg*) problems. On planning problems (sat*, huge-r and *blocks*) result are mixed. On DES problems, PBS performs better than optz and much better than opbdp, even if optz with "w" is quite competitive. For Max-ONE, the picture is very similar.

We also tried to (further) optimize the encodings presented. We modified the ChooseLiteralPref function in order to preferentially choose among the variables contained in the original formula ϕ. The results were not positive: The performances are often worse, and increase only on same problems in the case of the "u" heuristic (that always had the worst results, and even this enhancement

[3]The suggestion is in the README file down-loadable with the binary of the solver.

#var	#cl	#CNF b	#CNF bm	#CNF w	PBS	opbdp	optz b	optz bm	optz w
120	120	16064	16183	4463	14.61	4.50	0.55	0.46	11.39
120	240	16184	16303	4583	TIME	30.52	22.66	90.62	TIME
120	360	16304	16423	4703	TIME	3.81	85.7	79.6	TIME
120	480	16424	16543	4823	0.64	0.08	5.73	5.15	5.84

Table 16.5: Randomly generated benchmarks. Min-ONE 3SAT problems.

#var	#cl	#CNF b	#CNF bm	#CNF w	PBS	opbdp	optz b	optz bm	optz w
120	120	16064	16183	4463	75.61	37.45	12.01	7.00	70.70
120	240	16184	16303	4583	TIME	62.92	109.02	87.07	TIME
120	360	16304	16423	4703	TIME	2.65	187.66	231.27	TIME
120	480	16424	16543	4823	0.6	0.08	5.09	4.86	7.56

Table 16.6: Randomly generated benchmarks. Max-ONE 3SAT problems.

did not help it to reach results comparable with "b" and "bm" heuristics). We also optimized the "w" encoding trying to minimize the number of variables and clauses needed following some of the suggestions in the original paper. As a results, we obtained a reduction of variables and clauses needed by the encoding of about a factor of 2, and often this leads to a similar reduction in the CPU time.

instance	#var	#res	PBS	opbdp	optz b	optz bm	optz w
2bitcomp-5	125	39	8.02	0.95	1.78	1.82	10.69
2bitmax-6	252	61	TIME	120.5	60.59	48.64	183.08
3blocks	283	56	0.14	296.92	6.75	5.09	0.14
4blocks	758	66	0.25	TIME	4.14	8.72	7.46
4blocksb	410	66	52.57	TIME	TIME	TIME	TIME
huge-r	927	118	0.23	0.88	14.93	14.57	0.22
sat-bw-large.b	1087	131	0.18	1.78	15.16	11.31	0.22
sat-bw-large.c	3016	136	34.97	TIME	MEM	MEM	3.56
sat-logistics.a	828	135	MEM	TIME	14.66	14.06	11.52
sat-logistics.b	843	138	TIME	TIME	364.34	200.21	TIME
cnf-r1-b1-k1.1	307	36	7.6	86.13	1.14	1.22	0.88
cnf-r1-b3-k1.1	851	86	0.05	137.18	3.94	4.72	0.14
cnf-r2-b3-k1.1	1495	141	0.05	67.14	13.41	12.85	0.25
cnf-r3-b1-k1.1	1461	119	6.88	TIME	289.75	200.91	12.97
qg1-8	512	49	TIME	TIME	64.03	105.59	89.24
qg2-8	512	64	TIME	TIME	41.75	38.97	31.02

Table 16.7: Real world problems. Min-ONE problem.

instance	#var	#res	PBS	opbdp	optz b	optz bm	optz w
2bitcomp-5	125	85	8.02	0.46	0.82	0.83	2.24
2bitmax-6	252	146	TIME	48.16	216.23	47.35	249.24
3blocks	283	63	0.11	301.55	20.04	11.63	7.82
4blocks	758	66	0.24	TIME	35.05	55.11	17.13
4blocksb	410	66	52.64	TIME	TIME	TIME	TIME
huge-r	927	118	0.23	0.87	302.8	135.75	0.77
sat-bw-large.b	1087	136	0.16	1.74	216.22	147.18	0.42
sat-bw-large.c	3016	265	35.07	TIME	MEM	MEM	TIME
sat-logistics.a	828	135	MEM	TIME	61.74	60.57	4.7
sat-logistics.b	843	138	TIME	TIME	385.36	546.13	14.17
cnf-r1-b1-k1.1	307	52	7.62	84.93	4.53	4.42	1.01
cnf-r1-b3.k1.1	851	87	0.05	137.7	29.02	28.29	0.32
cnf-r2-b3-k1.1	1496	141	0.05	68.7	140.71	91.99	0.24
cnf-r3-b1-k1.1	1461	119	6.88	TIME	TIME	TIME	435.02
qg1-8	512	64	TIME	TIME	36.89	21.24	47.31
qg2-8	512	64	TIME	TIME	33.54	21.96	36.65

Table 16.8: Real world problems. Max-ONE problem.

Chapter 17

Conclusions

In this thesis we have dealt with the problem of **Safe Planning**. Safe Planning is the task of generating/validating plans that not only achieve the goal, but verify also a set of other user-defined properties, e.g., safety properties. This is particularly important in the context of space applications, where autonomy and safety are indeed two crucial properties any system has to guarantee. The task has been studied via the integration of

- Planning, a research area in Artificial Intelligence (AI) aiming at the construction of systems – called planners – that enable a robot to autonomously synthesize a series of actions that will achieve its goals.
- Model Checking, a research area in Computer Science devoted to the definition of procedures for the automatic verification of programs and specifications.

two push-button technologies.

In the literature, the problem of Safe Planning has been solved starting from a planning problem described in STRIPS language, and a set of safety properties expressed in LTL, and rely on a compilation into a SAT formula. Then, the formula is fed to a SAT solver in order to check if the plan is "safe".

In this thesis we have advanced the state-of-the-art of Safe Planning in several directions, both "theoretical" and "practical".

On the theoretical side, we have introduced three new approaches for implementing Safe Planning

Safe Planning via Temporal Reasoning where the problem of Safe Planning is compiled into a Temporal Reasoning problem via a formula in Separation Logic

Safe Planning via Answer Set Programming where the problem of Safe Planning is compiled into a logic program to be solved under the Answer Set semantic

Safe Planning via SAT Optimization where the problem of Safe Planning is compiled into an optimization problem composed by a propositional formula and a functional cost to be minimized defined on a subset of

the variables of the propositional formula (the actions of the planning problem)

On the "practical", implementation oriented side, we have designed, implemented and experimented four systems:

SIMO (Satisfiability Internal Modulo Object-oriented), an efficient new generation decision procedure for propositional satisfiability. SIMO is the evolution of the SIM solver.

Cmodels2 (Computing models), a decision procedure, based on propositional satisfiability, for finding AS of logic programs.

TSAT++ (Temporal SATisfiability), a decision procedure , based on propositional satisfiability, for solving formula expressed in SL.

OPTSAT (OPTimal SATisifability), a decision procedure for optimization problems related to propositional satisfiability.

The implemented systems have shown performances competitive and often superior w.r.t. state-of-the-art systems working in the same area implemented by other groups in several universities and research institutions.

17.1 Future work

There is other, further work that should be done in this research area as a result of the thesis.

Surely, would be interesting and useful to develop other types of compilers and solvers for other logics. In particular, we can think about of having arithmetic constraints "more expressive" than the ones we can deal with Separation Logic, and we can simply think of other types of "optimizations" rather the one introduced in the last part.

Moreover, until now the decision procedures have been tested using the challenging benchmarks that are known in the literature for the underlined logic. It would be useful to evaluate the approaches starting from a "real" planning problem in STRIPS and with safety properties to be maintained by the plan. This task could give an indication about the most promising approach.

Bibliography

[ABC+02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210. Springer-Verlag, July 27-30 2002.

[ABM01] D. Achlioptas, P. Beame, and M. Molloy. A sharp threshold in proof complexity. In *Proc. STOC*, pages 337–346, 2001.

[ACG99] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In S. Biundo and M. Fox, editors, *Proceedings of the 5th European Conference on Planning (Durham, UK)*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 1999.

[ACG+04] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, Massimo Idini, and Marco Maratea. TSAT++: An open platform for satisfiability modulo theories. In *Proceedings of PDPAR, Pragmatics of Decision Procedures in Automated Reasoning, Cork (Ireland)*. Elsevier, 2004. To appear in the ENTCS series.

[ACGM04] A. Armando, C. Castellini, E. Giunchiglia, and M.Maratea. A SAT-based decision procedure for the boolean combination of difference constraints. In *8th International Conference on Theory an Applications of Satisfiability Testing. Selected Revised Papers.*, Lecture Notes in Computer Science. Springer Verlag, 2004. To appear.

[ACGM05] A. Armando, C. Castellini, E. Giunchiglia, and M.Maratea. The SAT-based approach to separation logic. *Accepted to the Journal of Automated Reasoning (JAR)*, 2005.

[ACGT01] L. Carlucci Aiello, A. Cesta, E. Giunchiglia, and P. Traverso. Merging planning and verification techniques in for 'safe planning' in space robotics. In *Proc. 6th Int. Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, 2001.

[ARMS00] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Pbs: A backtrack-search pseudo-boolean solver and optimizer. In Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors, *SAT2000*. IOS Press, 2000.

[Bac02] F. Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *American Association for Artificial Intelligence*. AAAI Press, 2002.

[Bar95] Peter Barth. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. MPI-I-95-2-003, 1995.

[BB92] M. Buro and H. Buning. Report on a SAT competition. Technical Report 110, University of Paderborn, Germany, November 1992.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, LNCS. Springer, 2003.

[BB04] Clark W. Barrett and Sergey Berezin. CVC lite: A new implementation of the cooperating validity checker category b. In *16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114, pages 515–518. Springer-Verlag, 2004.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund Clarke, M.Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. of the 36th Conference on Design Automation (DAC '99)*, pages 317–320. ACM Press, 1999.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, 1999.

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.E. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 1994.

[BCRZ99] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety proeprties of a power pc microprocessor using symbolic model checking without BDDs. In *Proc. of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 60–71, Berlin, 1999. Springer.

[BEL00] Babovich, Erdem, and Lifschitz. Fages' theorem and answer set programming. In *Non-monotonic Reasoning: Proceedings of the Ninth International Workshop*, 2000.

[Ber97] Michel Berkelaar. The *lp_solve* solver for mixed integer-linear programming, 1997. Version 2.2. Available at `http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml`.

[BF99] B. Borchchers and J. Furman. A two-phase exact algorithm for max-sat and weighted max-sat. *Jornal of Combinatorial Optimization*, 2:299–306, 1999.

[BLS02] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU logic formulas via boolean and pseudo-boolean encodings. In *Proceedings of International Workshop on Constraints in Formal Verification*, sep 2002. Associated with International Conference on Principles and Practice of Constraint Programming, Ithaca, NY (USA).

[BM96] Bayardo, Jr., Roberto J. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 298–304, Menlo Park, August 4–8 1996. AAAI Press / MIT Press.

[Bry92] R. E. Bryant. Symbolic Boolean manipolation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS97] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 203–208, Menlo Park, July 27–31 1997. AAAI Press.

[CB94] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI94*, Washington, 1994. Morgan Kaufmann Publishers.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In J. C. Godskesen, editor, *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, 2002.

[CFG+01] Fady Copty, Limor Fix, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. CAV-01*, 2001.

[CGGT97] Cimatti, Giunchiglia, Giunchiglia, and Traverso. Planning via model checking: a decision procedure for AR. In *Lecture Notes in Computer Science*, volume 1348, 1997.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking. *Proc. Int'l Summer School on Deductive Program Design*, 1994.

[CGT03] C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147:85–117, 2003.

[Cla78] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[CLR98] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms.* MIT Press, 1998.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[CS88] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35:759–768, 1988.

[D'A92] M. D'Agostino. Are Tableaux an Improvement on Truth-Tables? *Journal of Logic, Language and Information*, 1:235–252, 1992.

[DD04] Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k-sat formulae. In Enrico Giunchiglia and Armando Tacchella, editors, *6th International Conference on Theory an Applications of Satisfiability Testing. Selected Revised Papers.*, volume 2919 of *Lecture Notes in Computer Science*, pages 486–501. Springer-Verlag, 2004.

[dGLMS03] Simon de Givry, Javier Larrosa, Pedro Mesenguer, and Thomas Schiex. Solving max-sat as weigthed csp. In *9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, LNCS. Springer, 2003.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[DLM+01] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems, July, 2001. Presented at Linux Clusters: The HPC Revolution.

[dMRSR04] Leonardo de Moura, Harald Ruess, Natarajan Shankar, and John Rushby. The ics decision procedures for embedded deduction. In *Proceedings of IJCAR, International Joint Conference on Automated Reasoning, Cork, Ireland*, jul 2004.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[ea98] Thomas Eiter et al. The KR system dlv: Progress report, comparisons and benchmarks. In L.; Cohn, A. G.; Schubert and eds. Shapiro, S. C., editors, *KR'98: Principles of Knowledge Representation and Reasoning*, page 406'417. Morgan Kaufmann, 1998.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publisher B.V., 1990.

[EMW97] Michael Ernst, Todd Millstein, and Daniel Weld. Automatic SAT-compilation of planning problems. In *Proc. IJCAI-97*, 1997.

[Esr02] Erdem Esra. *Theory and applications of answer set programming.* PhD thesis, University of Texas at Austin, 2002. PhD thesis.

[EV03] Erdem E. and Lifschitz V. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Fab02] Wolfgang Faber. *Enhancing efficiency and expressiveness in ASP systems.* PhD thesis, University of Wien, 2002. PhD thesis.

[Fag94] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725, pages 355–367. Springer-Verlag, 2003.

[FLP01] W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for asp. In *Proc. IJCAI*, 2001.

[Fre95] Jon W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[GGT00] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning*, 2000. To appear.

[Giu00] Enrico Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, 2000.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman and Company, New York, 1979.

[GL88a] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *in Logic Programming: Proceedings of the Fifth International Conference and Symposium*, 1988.

[GL88b] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GL95] Enrico Giunchiglia and Vladimir Lifschitz. Dependent fluents. In *Proc. IJCAI-95*, pages 1964–1969, 1995.

[GML04] E. Giunchiglia, M. Maratea, and Y. Lierler. SAT-based answer set programming. In *American Association for Artificial Intelligence.* AAAI Press, 2004.

[GMS98] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proc. AAAI*, 1998.

[GMT03] Enrico Giunchiglia, Marco Maratea, and Armando Tacchella. (In)Effectiveness of look-ahead techniques in a modern SAT solver. In *9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, pages 842–846, 2003.

[GMTZ01] Enrico Giunchiglia, Marco Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. *Lecture Notes in Computer Science*, 2083:347–363, 2001.

[GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. *Satisfiability Problem: Theory and Applications*, pages 19–153, 1997.

[GT99] F. Giunchiglia and P. Traverso. Planning as model checking. In *Lecture Notes in Computer Science, Proc. of the 5th European Conference on Planning*, 1999.

[GW00] Jan Friso Groote and Joost P. Warners. The propositional formula checker heerhugo. In Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors, *SAT2000. Highlights of Satisfiability Research in the Year 2000*, volume 24, pages 101–125. IOS Press, 2000.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[HM04] Marijn Heule and Hans Van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead sat solver. In *8th International Conference on Theory an Applications of Satisfiability Testing. Selected Revised Papers.*, Lecture Notes in Computer Science. Springer Verlag, 2004. To appear.

[HS00] Holger H. Hoos and Thomas Stützle. Local Search Algorithms for sat: an Empirical Evaluation. In Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors, *SAT2000. Highlights of Satisfiability Research in the Year 2000.* IOS Press, 2000.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[KI03] Heljanko K. and Niemelä I. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3:519–550, 2003.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363, 1992.

[KS98] Henry Kautz and Bart Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, 1998.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[Li00] Chu Min Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, 2000.

[Lib00] Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116:315–326, 2000.

[Lif96] Vladimir Lifschitz. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.

[Lif99] Vladimir Lifschitz. Answer set planning. In *Proceedings of the 1999 international conference on Logic programming*, pages 23–37. Massachusetts Institute of Technology, 1999.

[LL03] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. ICLP-03*, 2003.

[LPF+02] N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, , and A. Polleres. The dlv system. In *Proc. of JELIA*, pages 537–540, 2002.

[LR04] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 2004. To appear.

[LS03] D. Le Berre and L. Simon. The essentials of the sat'03 competition. In *Proc. of the 6th International Conference on the Theory and Applications of Satisfiability Testing (SAT'03). Selected revised papers*, volume 2919 of *LNCS*, May 2003.

[LS05] D. LeBerre and L. Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In *8th International Conference on Theory an Applications of Satisfiability Testing. Selected Revised Papers.*, Lecture Notes in Computer Science. Springer Verlag, 2005. To appear.

[LSB02] Shuvendu K. Lahiri, Sanjit A. Seshia, and Bryant Bryant. Modeling and verification of out-of-order microprocessors in UCLID. *Lecture Notes in Computer Science*, 2517:142–155, 2002.

[LT84] John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[LZ02] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *American Association for Artificial Intelligence*. AAAI Press, 2002.

[LZ03a] F. Lin and J. Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proc. IJCAI*, 2003.

[LZ03b] F. Lin and Y. Zhao. Asp phase transition: A study on randomly generated programs. In *Proc. ICLP-03*, 2003.

[MAAS02] I. Markov, F. Aloul, A.Ramani, and K. Sakallah. Solving difficult SAT instances in the presence of simmetry. In *Proc. of DAC*, 2002.

[MFM05] Y. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *8th International Conference on Theory an Applications of Satisfiability Testing. Selected Revised Papers.*, Lecture Notes in Computer Science. Springer Verlag, 2005. To appear.

[MM93] D. S. Moore and G. P. McCabe. *Introduction to the Practice of Statistics.* W. H. Freeman and Co., 1993.

[MM00] Massacci and Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24, 2000.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[MNPW98] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 1998.

[Mon04] Remi Monasson. On the analysis of backtrack procedures for the coloring of random graphs. *Complex Networks*, 2004.

[MT99] Victor Marek and Miroslaw Truszczynski. Stable models as an alternative programming paradigm. In *The Logic Programming Paradigm: a 25.Years perspective*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[Nie99] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[OC00] A. Oddi and A. Cesta. Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 108–112, Berlin, 2000.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[PG86a] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[PG86b] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. 18th Ann. IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pra77] V. R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, September 1977.

[Pre93] Daniele Pretolani. *Satisfiability and Hypergraphs.* PhD thesis, Università di Pisa, 1993.

[Pre02] S. Prestwich. A SAT approach to query optimization in mediator systems. In *SAT 2002*, 2002.

[Pro93a] Patrick Prosser. Domain filtering can degrade intelligent backjumping search. In *Proc. IJCAI*, pages 262–267, 1993.

[Pro93b] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[PV05] Ferraris P. and Lifschitz V. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[Rob65] Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RR96] Ben-Eliyahu R. and Dechter R. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1996.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a cooperating validity checker. In J. C. Godskesen, editor, *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, 2002.

[SBH02] L. Simon, D. Le Berre, and E. A. Hirsch. The SAT2002 Competition, 2002.

[SEM03] Stefănescu Esparza and Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *Proc. of CONCUR*, volume 2761 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2003.

[SFM00] B. Smith, M.S. Feather, and N. Muscettola. Challenges and methods in testing the remote agent planner. In *Lecture Notes in Computer Science, Proc. 5th Int.nl Conf. o Artificial Intelligence Planning and Scheduling*, 2000.

[SG99] J. P. M. Silva and T. Glass. Combinational Equivalence Checking Using Satisfiability and Recursive Learning. In *DATE*, 1999.

[Sht00] Ofer Shtrichmann. Tuning SAT checkers for bounded model checking. In *Proc. of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 480–494, Berlin, 2000. Springer.

[Sim00] Patrick Simons. *Extending and implementing the stable model semantics.* PhD thesis, Helsinky University, 2000. PhD thesis.

[SK93] Bart Selman and Henry Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI '93)*, pages 46–51, San Francisco, 1993. Morgan Kaufmann Publishers.

[SK98] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proc. AAAI*, 1998.

[SK00] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[SKM97] Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In *Proc. IJCAI-97*, pages 50–54, 1997.

[SLM92] B. Selman, H. Levesque., and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[SNT02] P. Simons, I. Niemelä, and S. Timo. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical report, University of Michigan, April 1996.

[SS98] Mary Sheeran and Gunnar Stalmarck. A tutorial on stalmarck's proof procedure for propositional logic. In *Proc. of the 2nd International Conference on Formal Methods in Computer Aided Design (FMCAD '98)*, volume 1522 of *LNCS*, pages 82–99, Berlin, 1998. Springer.

[SSB02] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding separation formulas with SAT. *Lecture Notes in Computer Science*, 2404:209–222, 2002.

[SW83] Jörg Siekmann and Graham Wrightson, editors. *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer-Verlag, 1983.

[SW98] David Smith and Daniel Weld. Conformant graphplan. In *Proc. AAAI-98*, pages 889–896, 1998.

[SZ04] H. Shen and H. Zhang. Study of Lower Bound Functions for Max-2SAT. In *American Association for Artificial Intelligence*. AAAI Press, 2004.

[TCG00] P. Traverso, A. Cesta, and E. Giunchiglia. Interactive autonomy for space applications. In *Proc. Intl. Workshop on Planning and Scheduling for Space*, 2000.

[TP03] Ioannis Tsamardinos and Martha Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151:43–89, 2003.

[Tse70] G. Tseitin. On the complexity of proofs in propositional logics. *Seminars in Mathematics*, 8, 1970. Reprinted in [SW83].

[Urq95] Alasdair Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[US94] T. E. Uribe and M. E. Stickel. Ordered Binary Decision Diagrams and the Davis-Putnam Procedure. In *Proc. of the 1st International Conference on Constraints in Computational Logics*, 1994.

[VB01] M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. In *DAC*, 2001.

[vM00] Hans van Maaren. Preface. In Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors, *SAT2000. Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.

[Waa96] J. P. Waarners. A linear-time transformation of linear inequalities into cnf. Technical report, ???, 1996.

[WBCG00] Poul F. Williams, Armin Biere, Edmund M. Clarke, and Anublav Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In *Proc. of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 124–138, Berlin, 2000. Springer.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER version 0.42. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 141–145, New Brunswick, New Jersey, USA, July/August 1996. Springer.

[WS04] J. Ward and J.S. Schlipf. Answer set programming with clause learning. In *Proc. of the 7th Int'l Conf. on Logic Programming and Nonmonotonic Reasoning*, 2004.

[WW99] Steven Wolfman and Daniel Weld. The LPSAT-engine & its application to resource planning. In *Proc. IJCAI-99*, 1999.

[XZ05] Z. Xing and W. Zhang. An efficient exact algorithm for (weighted) maximum satisfiability. *Artifical Intelligence*, 2005. To appear.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July13–17 1997. Springer.

[ZMMM01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.