UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Propositional Satisfiability: Techniques, Algorithms and Applications

Maria Inês Camarate de Campos Lynce de Faria

(Mestre)

Dissertação para obtenção do Grau de Doutor em
Engenharia Informática e de Computadores

Orientador: Doutor João Paulo Marques da Silva

Outubro de 2004

Resumo

Nos últimos anos assistimos a um progresso notável na área de satisfação proposicional (SAT), com contribuições significativas a nível teórico e prático. As soluções algorítmicas para SAT incluem, entre outras, procura local, procura por retrocesso e técnicas de manipulação de fórmulas. Apesar dos algoritmos conhecidos requererem tempo de execução exponencial no pior caso, as ferramentas de SAT podem actualmente ser usadas para solucionar instâncias de problemas difíceis.

Esta dissertação contribui para uma melhor compreensão das técnicas, dos algoritmos e das aplicações de satisfação proposicional. Em primeiro lugar, introduzimos estruturas de dados eficientes que, apesar de não permitirem um conhecimento exacto sobre o tamanho dinâmico de uma cláusula, são bastante precisas a determinar o número de literais não atribuídos de uma cláusula. Por outro lado, sugerimos o uso de técnicas de pré-processamento baseadas no teste de variáveis para manipular fórmulas proposicionais. Propomos também o uso de retrocesso irrestrito, um algoritmo que combina as vantagens da procura local e da procura por retrocesso. Finalmente, relacionamos dificuldade e estrutura oculta em instâncias não satisfazíveis de 3-SAT, onde dificuldade se traduz no esforço de procura e estrutura oculta se traduz em sub-fórmulas não satisfazíveis e em subconjuntos de variáveis que permitem provar não satisfação.

Palavras Chave

Satisfação Proposicional, Algoritmos de Satisfação, Implementações Eficientes, Teste de Variáveis, Retrocesso Irrestrito, Estrutura Oculta

Abstract

Recent years have seen remarkable progress in propositional satisfiability (SAT), with significant theoretical and practical contributions. Algorithmic solutions for SAT include, among others, local search, backtrack search and formula manipulation techniques. Despite the worst-case exponential run time of all known algorithms, SAT solvers can currently be used to solve hard benchmark problems.

This dissertation contributes to better understanding the techniques, the algorithms and the applications of propositional satisfiability. First, we introduce efficient lazy data structures that, even though not being able to determine exactly the dynamic size of a clause, are quite accurate at predicting the number of unassigned literals in a clause. In addition, we suggest the use of probing-based preprocessing techniques for manipulating propositional formulas. Furthermore, unrestricted backtracking is proposed as a backtracking algorithm that combines both the advantages of local search and backtrack search. Finally, we relate hardness with hidden structure in unsatisfiable random 3-SAT formulas, where hardness is measured as the search effort and hidden structure is measured by unsatisfiable cores and strong backdoors.

Keywords

Propositional Satisfiability, Satisfiability Algorithms, Efficient Implementations, Probing, Unrestricted Backtracking, Hidden Structure

Acknowledgments

Valeu a pena? Tudo vale a pena
Se a alma não é pequena.
Quem quer passar além do Bojador
Tem que passar além da dor.
Deus ao mar o perigo e o abismo deu,
Mas nele é que espelhou o céu.

Fernado Pessoa, Mar Português
In Mensagem, 1934

Finally, it is done! Why am I so sure that I took time enough doing this PhD? Because now I know that I know nothing. And that makes me able to conduct my own research work. I took almost four years to learn this. But I was not alone. And I would like to thank to those who helped me making this dream come true.

To all my family, specially to my parents. They gave me life and they keep me alive. To my brothers, who are my best friends. To my nephews, who make me think that knowledge has to be shared. To my grand parents, who share so much knowledge.

To my supervisor, Professor João Marques Silva, with whom I learnt how to do research work from the very beggining. But I learnt much more than that. I learnt that one should not be afraid of making questions as long as one is not afraid of searching for the truth.

To my work colleagues, in particular to Sofia, Luís, Vasco and Zé Carlos. With them I shared the best and the worst moments of my research work.

To all my friends, who make me feel that life is much more than work.

To all the institutions which have funded my research work:

- Fundação para a Ciência e Tecnologia
- European Union
- TNI-Valiosys
- TransEDA
- Cadence European Laboratories
- IBM
- Intelligent Information Systems Institute
- Universidade Técnica de Lisboa
- Conselho de Reitores das Universidades Portuguesas
- Fundação Calouste Gulbenkian
- Fundação Luso-Americana para o Desenvolvimento
- American Association for Artificial Intelligence
- European Coordinating Committee for Artificial Intelligence
- Cork Constraint Computation Centre

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

What is SAT?

Propositional SATisfiability (SAT) can be simply characterized with a couple of words: propositional logic and computational complexity. SAT looks easy but is hard. Easy on its formulation: SAT problems are encoded in propositional logic, a logic with very limited expressiveness. Hard on solving: finding a solution or proving unsatisfiability has an increasing complexity as the size of the problem increases.

Why is SAT important?

SAT is well-know for its theoretical importance. SAT was the first problem to be proved to be NP-Complete (Cook 1971).

SAT is well-known for having many different applications: automatic test pattern generation, combinational equivalence checking, bounded model checking, planning, graph coloring, software verification, etc.

SAT is well-know for its remarkable improvements in the last decade. SAT solvers are now capable of efficiently solving instances with hundreds of thousands of variables and millions of clauses.

Why do we think this dissertation is relevant?

In this dissertation we have striven to contribute to better understanding the structure of SAT problem instances, the organization of SAT algorithms and the efficient implementation of SAT solvers.

In what follows we start by having introductory sections on Logic and Complexity, two concepts very related with the SAT problem. Afterwards, we give examples of problems that can be formulated as SAT problems, namely combinational equivalence checking, the pigeonhole problem and Latin squares. Next, we give a picture of SAT current research status. Finally, the contributions of our thesis are explained and its organization is described.

1.1 Logic

”I know what you're thinking about,” said Tweedledum; ”but it isn't so, nohow.”
”Contrariwise,” continued Tweedledee, ”if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.”
Lewis Carroll, in Through the Looking-Glass, 1872

If we take logic to be the activity of drawing inferences (conclusions) from a body of knowledge, then no doubt humans have been using logic for as long as they have been thinking. On the other hand, if we take logic to be the analysis of concepts involved in making inferences, and the identification of standards and patterns of correct inference, then logic can be traced only back to the days of Aristotle (384-322 BC), with some parallel development in early Hindu writings. Aristotle was the first to start writing down the ideas and rules of what constitutes a logical process. However, it is not clear that this increase in logical self-consciousness improved the accuracy of reasoning processes for humankind in general.

The heart of Aristotle's logic is the syllogism. The classic example of syllogism is as follows: All men are mortal; Socrates is a man; therefore, Socrates is mortal. The core of this definition is the notion of ”resulting of necessity” (*ex ananks sumbainein*). This corresponds to a modern notion of logical consequence: X results of necessity from Y and Z if it would be impossible for X to be false when Y and Z are true. Aristotle's logical work contains the earliest formal study of logic that we have knowledge of. It is therefore a remarkable work that comprises a highly developed logical theory, one that was able to dominate logic for many centuries. Indeed, the syllogistic form of logical argumentation dominated logic for 2,000 years.

Historically, René Descartes (1596-1650) may have been the first mathematician to have had the idea of using algebra, especially its techniques, for solving unknown quantities in equations, as a vehicle for scientific exploration. However, the idea of a calculus of reasoning was cultivated especially by Gottfried Wilhelm Leibniz (1646-1716). Though modern logic in its present form originates with Boole and De Morgan, Leibniz was the first to have a really distinct plan of a broadly applicable system of mathematical logic. However, this information is in Leibniz's unpublished work, which has only recently been explored.

Logic's serious mathematical formulation began with the work of George Boole (1815-1864) in the mid-1800s. Boole made significant contributions in several areas of mathematics, but was immortalized for his book *"An Investigation of the Laws of Thought"* written in 1854, in which he represented logical expressions in a mathematical form now known as Boolean algebra. Boole's work is so impressive because, with the exception of elementary school and a short time in a commercial school, he was almost completely self-educated. Unfortunately, with the exception of students of philosophy and symbolic logic, Boolean algebra was destined to remain largely unknown and unused for the better part of a century.

In conjunction with Boole, another British mathematician, Augustus De Morgan (1806-1871), formalized a set of logical operations now known as De Morgan laws. However, the rules we now attribute to De Morgan were known in a more primitive form by William of Ockham (also known as William of Occam) in the 14th Century.

Gottlob Frege (1848-1925) in his 1879 *Begriffsschrift* extended formal logic beyond propositional logic to include constructors such as "all", "some". He showed how to introduce variables and quantifiers to reveal the logical structure of sentences, which may have been obscured by their grammatical structure. For instance, "All humans are mortal" becomes "All things x are such that, if x is a human then x is mortal."

Charles Peirce (1839-1914) introduced the term "second-order logic" and provided us with most of our modern logical notation, including the symbols $\forall$ and $\exists$. Although Peirce published his work some time after the *Begriffsschrift*, Frege's contribution was not very well known until many years later. Logicians in the late 19th and early 20th centuries were thus more familiar with Peirce's system of logic (although Frege is generally recognized today as being the "Father of modern logic").

In 1889 Giuseppe Peano (1858-1932) published the first version of the logical axiomatization of arithmetic. Five of the nine axioms he came up with are now known as the Peano axioms. One of these axioms was a formalized statement of the principle of mathematical induction.

In 1938 a young student called Claude E. Shannon(1916-2001) recognized Boolean algebra's relevance to electronics design. In a paper based on his master's thesis at MIT, *"A Symbolic Analysis of Relay and Switching Circuits"*, published in the Transactions of the American Institute of Electrical Engineers, volume 57, pages 713-723, which was widely circulated, Shannon showed how Boole's concepts of TRUE and FALSE could be used to represent the functions of switches in electronic circuits. It is difficult to convey just how important this concept was; suffice is to say that Shannon had provided electronics engineers with the mathematical tool they needed to design electronic circuits, and these techniques remain the cornerstone of electronic design to this day.

In 1948, Brattain, Bardeen and Shockley, working at the Bell Telephone Laboratories, published their invention of the transistor. Bardeen, Shockley and Brattain shared the 1956 Physics Nobel Prize for this invention. Later on, in 1959, the first planar transistor was produced. The revolution in integrated circuits has accelerated the automation of information technology that we enjoy today.

1.2 Complexity

Many of the games and puzzles people play are interesting because of their difficulty: it requires cleverness to solve them. Often this difficulty can be measured mathematically, in the form of complexity.

The complexity of a process or algorithm is a measure of how difficult it is to perform.

Table 1.1: Comparison of time complexity

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f(n) = n!$
10	0.01μ s	0.1μ s	1μ s	3.63 ms
20	0.02μ s	0.4μ s	1 ms	77.1 years
30	0.03μ s	0.9μ s	1 s	$8.4*10^{15}$ years
40	0.04μ s	1.6μ s	18.3 min	
50	0.05μ s	2.5μ s	13 days	
100	0.1μ s	10μ s	$4*10^{13}$ years	
1,000	1.00μ s	1 ms		

The study of the complexity of algorithms, also known as complexity theory, deals with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem). Other resources can also be considered, such as how many parallel processors are needed to solve a problem in parallel.

The time complexity of a problem is the number of steps that it takes to solve an instance of the problem, as a function of the size of the input, using the most efficient algorithm. Table 1.1 has the CPU time required for solving different functions, thus giving a picture of time complexity.

To further understand time complexity intuitively, consider the example of an n size instance that can be solved in n^2 steps. For this example we say that the problem has a time complexity of n^2. Of course, the exact number of computer instructions will depend on what machine or language is being used. To avoid this dependency problem, we generally use the Big O notation. If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most other computers, so this notation allows us a generalization away from the details of a particular computer. In this case, we say that this problem has *polynomial* time complexity. This is also true for all problems having $O(n^x)$ time complexity, with x being a constant.

The space complexity of a problem defines how much memory does it take to solve a problem. For example, consider an n size instance that can be solved using $2n$ memory units. Hence, this problem has $O(n)$ space complexity and therefore linear complexity.

Much of complexity theory deals with decision problems. A decision problem is a problem where the answer is always YES/NO. For example, the problem IS-PRIME is: given an integer, return whether it is a prime number or not. Decision problems are often considered because an arbitrary problem can always be reduced to a decision problem.

Decision problems fall into sets of comparable complexity, called complexity classes. P and NP are the most well-known complexity classes, meaning Polynomial time and Nondeterministic Polynomial time, respectively.

The complexity class P is the set of decision problems that can be solved by a deterministic machine with a number of steps bounded by a power of the problem's size. This class corresponds to an intuitive idea of problems which can be effectively solved even in the worst cases.

The complexity class NP is the set of decision problems with a nondeterministic solution and with the number of steps to verify the solution being bounded by a power of the problem's size. In other words, all the problems in this class have the property that their solutions can be checked effectively in polynomial time. The complexity class Co-NP is the set of decision problems where the NO instances can be checked effectively in polynomial time. The *Co* in the name stands for complement.

The class of P-problems is a subset of the class of NP-problems. The question of whether P is the same set as NP is the most important open question in theoretical computer science. There is even a $1,000,000 prize for solving it (see `http://www.claymath.org/millennium/`). Observe that if P and NP are *not* equivalent, then finding a solution for NP-problems requires an exhaustive search in the worst case.

The question of whether P = NP motivates the concepts of *hard* and *complete*.

A set of problems X is *hard* for a set of problems Y if every problem instance in Y can be transformed easily (i.e. in polynomial time) into some problem instance in X with the same answer. The most important hard set is NP-hard. A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem.

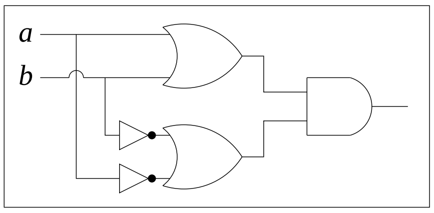


Figure 1.1: Combinational circuit

In general, is easier to show that a problem is NP than to show that it is NP-hard.

Set X is *complete* for Y if it is hard for Y, and is also a subset of Y. The most important complete set is NP-complete. A NP-complete problem is both NP (verifiable in nondeterministic polynomial time) and NP-hard (any other NP-problem can be translated into this problem). SAT is an example of an NP-complete problem (Cook 1971).

1.3 Examples of SAT Encodings

Many problems can be encoded in propositional logic. However, more sophisticated logics are frequently more adequate to represent most of the problems. The only exception is for combinational electronic circuits. Indeed, Shannon's work in 1938 showed how constants TRUE and FALSE could be used to represent the functions of switches in electronic circuits.

For example, given the combinational circuit in Figure 1.1, it is clear that it can be easily encoded in propositional logic as follows:

$$(a \vee b) \wedge (\neg a \vee \neg b)$$

Challenging SAT problems obtained from combinational circuits usually involve verification of properties. Sometimes errors appear only for input traces that seldom occur and hence they may not be easily discovered through simulation. Therefore it is important to be able to formally verify the equivalence between the circuit and the behavioral model.

In what follows we give different examples of problems that can be easily encoded as SAT problems and further effectively solved by state-of-the-art SAT solvers. SAT

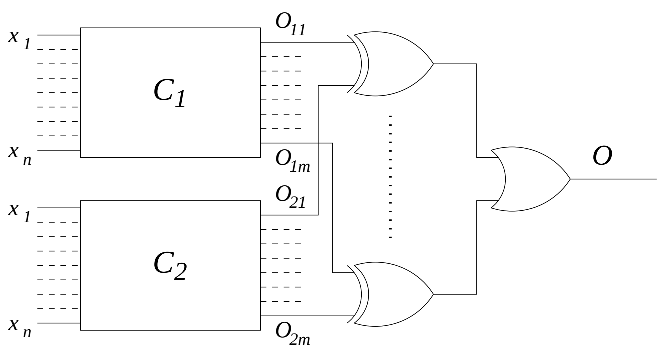


Figure 1.2: Generic miter circuit

formulas are represented, as usual, in Conjunctive Normal Form (CNF). A CNF formula is a conjunction ($\wedge$) of clauses, a clause is a disjunction ($\vee$) of literals, and a literal is a variable (x) or its complement ($\neg x$).

1.3.1 Combinational Equivalence Checking

The combinational equivalence problem consists in determining whether two given digital circuits implement the same Boolean function. This problem arises in a significant number of computer-aided design (CAD) applications, for example when checking the correctness of incremental design changes (performed either manually or by a design automation tool).

Equivalence checking is a Co-NP complete problem. Equivalence checking can be solved using SAT by identifying a counterexample.

SAT-based equivalence checking builds upon a miter circuit. A miter circuit consists of two circuits C_1 and C_2 and also a set of XOR gates and an OR gate. Consider that the outputs of C_1 are $O_{11}, \ldots, O_{1m}$ and the outputs of C_2 are $O_{21}, \ldots, O_{2m}$. Hence, the miter circuit has m XOR gates, and the input of an XOR$_i$ gate is O_{1i} and O_{2i}, where $i = 1, \ldots, m$. Finally, an OR gate links all the outputs of the XOR gates. A generic miter circuit is given in Figure 1.2. The output of the miter is 1 *iff* the two circuits represent different Boolean functions. Hence, adding the objective $O = 1$ to the CNF encoding makes the SAT instance unsatisfiable *iff* C_1 and C_2 are equivalent.

Figure 1.3 gives an example of a miter including the circuit from Figure 1.1, which is

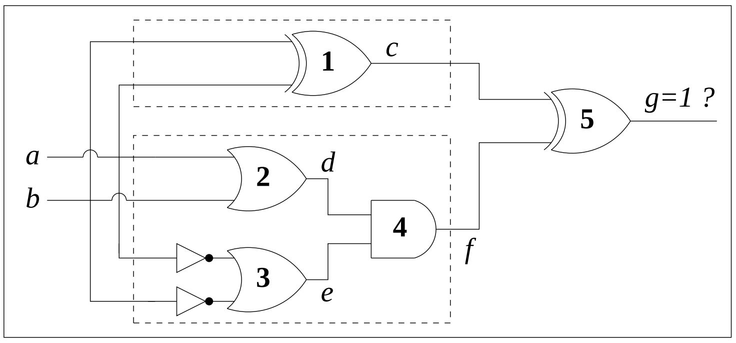
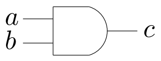
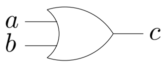


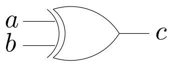
Figure 1.3: Miter circuit



$(a \vee \neg c)(b \vee \neg c)(\neg a \vee \neg b \vee c)$



$(\neg a \vee c)(\neg b \vee c)(a \vee b \vee \neg c)$



$(\neg a \vee \neg b \vee \neg c)(\neg a \vee b \vee c)(a \vee \neg b \vee c)(a \vee b \vee \neg c)$

Figure 1.4: Modeling of combinational gates

supposed to be equivalent to an XOR gate. Clearly, for this miter circuit there is no need to add an OR gate.

To encode the circuit given above, one has first to consider the encoding of the combinational gates AND, OR and XOR in the CNF format given in Figure 1.4. Considering those encodings, the miter circuit may be encoded in a CNF formula by defining a set of clauses for each gate as follows (Tseitin 1968):

1. $(\neg a \vee \neg b \vee \neg c)(\neg a \vee b \vee c)(a \vee \neg b \vee c)(a \vee b \vee \neg c)$

2. $(\neg a \vee b)(\neg b \vee d)(a \vee b \vee \neg d)$

3. $(a \vee e)(b \vee e)(\neg a \vee \neg b \vee \neg e)$

4. $(d \vee \neg f)(e \vee \neg f)(\neg d \vee \neg e \vee f)$

5. $(\neg c \vee \neg f \vee \neg g)(\neg c \vee f \vee g)(c \vee \neg f \vee g)(c \vee f \vee \neg g)$

6. (g)

Observe that the number given for each set of clauses corresponds to a number in a gate (see Figure 1.3). This CNF formula has no solution, meaning that the two circuits are equivalent.

1.3.2 Pigeonhole

The first statement of the pigeonhole principle is believed to have been made by Dirichlet in 1834. The pigeonhole principle states that if n pigeons are put into m holes, and if $n > m$, then at least one hole must contain more than one pigeon. Another way of stating this would be that m holes can hold at most m objects with one object to a hole; adding another object will force you to reuse one of the holes.

Although the pigeonhole principle may seem to be a trivial observation, it can be used to demonstrate unexpected results. For example, for proving that there must be at least two people in Lisbon with the same number of hairs on their heads. Demonstration: A typical head of hair has around 150,000 hairs. It is reasonable to assume that nobody has more than 1,000,000 hairs on their head. There are more than 1,000,000 people in Lisbon. If we assign a pigeonhole for each number of hairs on a head, and assign people to the pigeonhole with their number of hairs on it, there must be at least two people with the same number of hairs on their heads.

Another practical example of the pigeonhole principle involves the situation when there are 15 students that wrote a dictation. John made 13 errors, each of the other students made less than that number. Now prove that at least two students made equal number of errors. Demonstration: To solve this problem, let us pretend that the students are pigeons and put them in 14 holes numbered $0, 1, 2, \ldots, 13$, according to the number of errors made. In hole 0 we put those students who made no errors, in hole 1 those who made exactly 1 error, in hole 2 those who made 2 errors, and so on. Certainly, hole 13 is occupied solely by John. Now apply the pigeonhole principle.

The SAT encoding of this problem is very straightforward. Consider we have $n + 1$ pigeon and n holes. Consider $n*(n+1)$ variables x_{ij}. Each variable x_{ij} means that pigeon i is placed in hole j, where $i = 1, \ldots, n + 1$ and $j = 1, \ldots, n$. Then we have $n + 1$ clauses which say that a pigeon has to be placed in some hole:

$$\forall_i (x_{i1} \vee x_{i2} \vee \ldots x_{in})$$

Then for each hole we have a set of clauses ensuring that only one single pigeon is placed into that hole:

$$\forall_j (x_{1j} \oplus x_{2j} \oplus \ldots \oplus x_{n+1j})$$

This encoding leads to a total of $n * (n+1)$ propositional variables and $(n+1) + n * (n * (n+1)/2)$ CNF clauses.

1.3.3 Latin Squares

The mathematician Leonhard Euler introduced Latin squares in 1783 as a new kind of magic squares.

A Latin square of order n is an n by n array of n symbols in which every symbol occurs exactly once in each row and column of the array. Here are two examples:

Latin square of order 2

a	b
b	a

Latin square of order 3

x	y	z
z	x	y
y	z	x

You can get many more Latin squares by permuting rows, columns, and/or symbols in any combination.

Latin squares were originally mathematical curiosities, but statistical applications were found early in the 20th century, e.g. experimental designs. The classic example is the use of a Latin square configuration to place different grain varieties in test patches. Having multiple patches for each variety helps to minimize localized soil effects.

Similar statements can be made about medical treatments. Suppose that we want to test three drugs A, B and C for their effect in alleviating the symptoms of a chronic disease. Three patients are available for a trial, and each will be available for three weeks. Testing a single drug requires a week. Each patient is expected to try all the drugs and each drug is supposed to be tried exactly in one patient per week. The structure of the experimental units is a rectangular grid (which happens to be square in this case); there is no structure on the set of treatments. We can use the Latin square to allocate treatments. The rows of the square represent patients (P1, P2, P3) and the columns are weeks (W1, W2, W3). For example the second patient (P2), in the third week of the trial (W3), will be given drug B. Each patient receives all three drugs, and in each week all three drugs are tested.

	W1	W2	W3
P1	A	B	C
P2	C	A	B
P3	B	C	A

An incomplete or partial Latin square is a partially filled Latin square such that no symbol occurs more than once in a row or a column. The Latin square completion problem is the problem of determining whether the remaining entries of the array can be filled in such a way that we obtain a complete Latin square. The Latin square completion problem is NP-complete (Colbourn 1984) and exhibits a phase-transition behavior with an easy-hard-easy pattern as a function of the fraction of the symbols already assigned (Achlioptas *et al.* 2000).

Latin square completion problems are naturally represented as a constraint satisfaction problem (CSP), even though efficient SAT-based formulations have also been tried in the past (Gomes & Shmoys 2002). SAT-based encodings for the Latin square completion problem can be distinguished between the minimal and the extended encoding. The extended encoding adds some clauses to the minimal encoding. In both SAT encodings, for each Latin square of order n consider n^3 Boolean variables x_{ijk}, meaning that a symbol k is assigned to cell i, j, where $i, j, k = 1, 2, \ldots, n$.

The minimal encoding includes clauses that represent the following constraints:

1. Some symbol must be assigned to each entry:

   $\forall_{ij} \vee_{k=1}^{n} x_{ijk}$

2. No symbol is repeated in the same row:

   $\forall_{ijk} \wedge_{l=j+1}^{n} (\neg x_{ijk} \vee \neg x_{ilk})$

3. No symbol is repeated in the same column:

   $\forall_{ijk} \wedge_{l=i+1}^{n} (\neg x_{ijk} \vee \neg x_{ljk})$

The total number of clauses of the minimal encoding is $O(n^4)$.

The extended encoding explicitly considers each entry in the array has exactly one symbol, by also including the following constraints:

1. Each symbol much appear at least once in each row:

   $\forall_{ik} \vee_{j=1}^{n} x_{ijk}$

2. Each symbol much appear at least once in each column:

   $\forall_{jk} \vee_{i=1}^{n} x_{ijk}$

3. No two symbols are assigned to the same entry:

   $\forall_{ijk} \wedge_{l=k+1}^{n} (\neg x_{ijk} \vee \neg x_{ijl})$

Similarly to the minimal encoding, the size of the extended encoding is $O(n^4)$.

Experimental results on both encodings reveal that SAT solvers are competitive on solving this problems, as long as the size of a problem instance is manageable. Moreover, SAT solvers perform better on the extended encoding.

1.4 Research on SAT at a Glance

The area of propositional satisfiability has been the subject of intensive research in recent years, with significant theoretical and practical contributions. Algorithmic solutions for SAT include, among others, local search, backtrack search and formula manipulation techniques. In the last decade, several different organizations of local search and backtrack search algorithms for SAT have been proposed, in many cases allowing larger problem instances to be solved in different application domains. While local search algorithms (Selman, Levesque, & Mitchell 1992; Selman & Kautz 1993) have

Table 1.2: Evolution of SAT solvers in the last decade

Instance	Posit' 94	Grasp' 96	Sato' 98	Chaff' 01	Siege'03
ssa2670-136	40.66 s	1.2 s	0.95 s	0.02 s	0.01 s
bf1355-638	1805.21 s	0.11 s	0.04 s	0.01 s	0.01 s
pret150_25	>7200 s	0.21 s	0.09 s	0.01 s	0.01 s
dubois100	>7200 s	11.85 s	0.08 s	0.01 s	0.01 s
aim200-2_0-no-1	>7200 s	0.01 s	0 s	0 s	0.01 s
2dlx_cc_mc_ex_bp_f2_bug005	>7200 s	>7200 s	>7200 s	2.9 s	0.22 s
c6288	>7200 s	>7200 s	>7200 s	>7200 s	6676.17 s

been shown to be particularly useful for random instances of SAT, recent backtrack search algorithms have been used for solving large instances of SAT from real-world applications. Indeed, a large number of very efficient backtrack search SAT solvers have been proposed (Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997; Li & Anbulagan 1997; Zhang 1997; Moskewicz *et al.* 2001; Goldberg & Novikov 2002; Ryan 2004), most of which based on improvements made to the original Davis-Logemann-Loveland (DLL) backtrack search algorithm (Davis, Logemann, & Loveland 1962). These improvements range from new search strategies, to new search pruning and reasoning techniques, and to new fast implementations.

State-of-the-art SAT solvers can now very easily solve problem instances that more traditional SAT solvers are known to be totally incapable of. For example, Table 1.2 gives a picture of how SAT has evolved in the last decade, showing how problem instances have been effectively solved as new solvers appeared. As a result, a thorough understanding of the organization, the strategies, the techniques, and the implementation of state-of-the-art SAT solvers is essential to help focus future SAT research, to help devise new ideas for the next generation of solvers, to be able to solve the next generation of problem instances, and finally to help develop innovative modeling approaches, more capable of exploiting the organization of state-of-the-art SAT solvers.

Despite the significant improvements in state-of-the-art backtrack search SAT solvers, several questions can be asked. Is a well-organized and well-implemented DLL algorithm enough per se, or should the algorithm definitely include additional search techniques? Which search techniques are indeed efficient for solving most problem instances? Which search techniques cooperate effectively and which do not?

Through the last years, significant research has been done in order to clarify these issues. Indeed, the SAT community has increased significantly in the last years, and the SAT community meetings are definitely a proof of such evolution:

- **Workshops:** Sienna 1996, Paderborn 1998, Renesse 2000, Boston 2001.
- **Symposium:** Cincinnati 2002.
- **Conferences:** Santa Margarita Ligure 2003, Vancouver 2004.

Moreover, a SAT Solvers Competition has been run along the meetings of the last three years: 2002, 2003 and 2004 (see `http://www.satlive.org/SATCompetition`). These competitions establish for each year the state-of-the-art SAT solvers in different categories, namely random, handmade and industrial categories.

The SAT community is also alive through the Internet:

- **SAT*Live!*** available from `http://www.satlive.org/`: up-to-date links to the SATisfiability problem.
- **SAT-Ex** available from `www.lri.fr/~simon/satex/satex.php3`: the experimentation web site around the satisfiability problem.
- **SATLIB** available from `http://www.satlib.org/`: a collection of benchmark problems, solvers, and tools to be used for SAT related research.

All these sites are extremely useful to those who do research on SAT. For example, if you go to the SAT*Live!* web page, then you will find out the following:

- **People interested in SAT:** there are about 300 people registered.
- **Software:** the most visited software has 2776 hits (*siege_v4*).

- **Conference Paper:** the most visited conference paper has 1252 hits (*Daniel Jackson, Mandana Vaziri. Finding Bugs with a Constraint Solver. ISSTA'00, Portland, OR, 2000*).

- **Technical Report:** the most visited technical report has 1222 hits (*Inês Lynce and João P. Marques-Silva, Efficient Data Structures for Fast SAT Solvers, Technical Report RT/05/2001, INESC-ID, Portugal, November 2001*).

1.5 Contributions

This PhD thesis contains four main contributions, which correspond to research work that has been developed after my MSc thesis. Moreover, the overview chapter on Satisfiability Algorithms also involved research work, from which resulted three publications (Lynce & Marques-Silva 2002a; 2003a; 2003c). Nonetheless, and even though this work has been fundamental for the success of this PhD work, we do not consider it to be a novel contribution.

All the four main contributions refer to propositional satisfiability algorithms. All of them are quite distinct, but all of them have the same target: to better understand the structure of SAT problem instances as well as the behavior of SAT solvers, with the aim of allowing SAT solvers to efficiently solve the most challenging problem instances.

Next we describe each one of the four main contributions. In addition, we give the most relevant references containing the work we have developed in each one of the four topics.

1. **Efficient implementations of backtrack search SAT solvers**

   Efficient implementations represent the most recent paradigm shift in SAT solvers. Due to the huge size of many of the benchmark problem instances, a careful implementation can make the difference between being or not being able to solve a given problem instance in a reasonable amount of time. Moreover, learning new clauses may increase significantly the clause database, even though this technique is essential for solving hard real-world instances of satisfiability. This motivates the use of very efficient data structures. In the most recent years, they have evolved from intuitive to more sophisticated data structures. In our work (Lynce & Marques-Silva 2002c; 2003b; 2005a), we compare different existing data structures and further suggest new data structures. These new data structures, although not being able to determine exactly the dynamic size of a clause, are quite accurate in predicting the number of unassigned literals in a clause.

2. **Preprocessing formula manipulation techniques**

   These techniques consist of inferring new clauses from the CNF formula, often resulting in the elimination of variables. In the history of SAT, research on this topic has been a constant. "Backtrack search + learning" is like an all purposes recipe, whereas the use of preprocessing formula manipulation techniques implies selecting the most adequate techniques to be applied to a given problem instance. Experimental evidence suggests that there does not exist a preprocessing technique that is useful to help solving all problem instances, or at least a significant number of problem instances (Lynce & Marques-Silva 2001). Another motivation for the use of preprocessing formula manipulation techniques is related with the SAT encodings. Sometimes, the SAT encoding used for a problem is not the most efficient. Furthermore, we may obtain quite different results when using different SAT encodings. The number of variables and clauses is one of the relevant aspects, not only due to the size of the formula but specifically because it may considerably affect the heuristics and the learnt clauses. Conversely, having a smaller number of variables and clauses does not imply a problem instance is easier to solve. For example, for the Latin square completion problem (see Section 1.3.3), the extended encoding, which extends the minimal encoding, outperforms the minimal encoding. In our work (Lynce & Marques-Silva 2003d), we introduce a generic framework for applying formula manipulation techniques, with the aim of having a unified approach for better relating the different techniques and consequently for more efficiently implementing them. This generic framework is based on probing, i.e. on the formulation of hypothetical scenarios, obtained by assigning a value to a variable, and then applying unit propagation. Experimental results reveal that significant improvements can be achieved. Moreover, we envision applying the same techniques during the search, although this will bring an additional overhead.

3. **Unrestricted Backtracking**

   This contribution is motivated by the successful use of search restarts on solving hard real-world problem instances of satisfiability (Gomes, Selman, & Kautz 1998). We started by introducing randomization in the backtrack step, thus resulting in a randomized backtracking (Lynce, Baptista, & Marques-Silva 2001a; Lynce & Marques-Silva 2005b). Afterwards, random backtracking evolved to unrestricted backtracking (Lynce, Baptista, & Marques-Silva 2001b; 2001c; Bhalla *et al.* 2003a; 2003b). Unrestricted backtracking includes different forms of backtracking, apart from the traditional ones, i.e. chronological and non-chronological backtracking. The new forms of backtracking give more freedom to the search, at the cost of being incomplete. Nonetheless, it is possible to establish a variety of completeness conditions to assure that an otherwise incomplete form of search, by including completeness techniques, may also prove unsatisfiability if given enough CPU time. Obtained results suggest that these techniques may be useful for efficiently solving hard real-world instances, in particular for those instances where the use of search restarts is also useful.

4. **Hidden structure in unsatisfiable random 3-SAT**

   Our most recent work (Lynce & Marques-Silva 2004a; 2004b) aims to empirically reveal hidden structure in unsatisfiable random 3-SAT instances, based on the identification of unsatisfiable cores and strong backdoors. These concepts arise from recent developments on SAT research. Unsatisfiable cores result from the effort to extract a proof of unsatisfiability when a given problem instance is found to be unsatisfiable. An unsatisfiable core is a subset of clauses that are still unsatisfiable. Strong backdoors consist of a subset of variables of the formula that once assigned conduct the search to found that the instance is unsatisfiable. In this work we relate the number of nodes required to solve an unsatisfiable random 3-SAT instance with the sizes of unsatisfiable cores and strong backdoors. We conclude that, for unsatisfiable random 3-SAT instances, as the number of nodes required to solve the instance decreases, the percentage of clauses and variables in the unsatisfiable core or strong backdoor, respectively, also decreases.

1.6 Organization

This dissertation has a total of seven chapters. The first chapter is an introduction, and the last chapter describes conclusions and future work. The remaining chapters contain an overview and the main contributions of the dissertation.

After the Introduction, the second chapter overviews satisfiability algorithms. We give a special emphasis to the concepts related to backtrack search algorithms, since most of our contributions are related to backtrack search. Apart from the definitions, we introduce the Davis-Putnam (DP) resolution-based algorithm, followed by the Davis-Logemann-Loveland (DLL) backtrack search algorithm. Afterwards, a detailed description of non-chronological backtracking is given, including conflict-directed backjumping, learning and clause deletion policies. Next, we briefly review branching heuristics and search strategies. Finally, experimental results compare the different backtrack search algorithms, namely chronological and non-chronological backtracking, as well as different clause deletion policies.

Chapter 3 describes efficient implementations for backtrack search SAT algorithms. We start with an overview of four traditional data structures, i.e. assigned literal hiding, counter-based, counter-based with satisfied clause hiding, and satisfied clause and assigned literal hiding. Afterwards, we describe the basic lazy data structures, i.e. Sato's head/tail lists and Chaff's watched literals. Based on the latter, we propose new data structures: head/tail lists with literal sifting and watched literals with literal sifting. All the data structures are then compared, followed by a discussion on the advantages of the new data structures. We conclude that the new data structures, although being lazy, have a more accurate knowledge of the dynamic size of a clause.

Probing-based preprocessing techniques are then introduced in Chapter 4. We start by giving examples on how probing can be used to identify necessary assignments and to infer new clauses. Next, we describe reasoning with probing-based conditions, namely conditions for identifying satisfiability and unsatisfiability-based necessary assignments, and conditions for inferring implication, satisfiability and unsatisfiability-based clauses. Finally, we introduce ProbIt, a probing-based SAT preprocessor, and give results on using the new preprocessor.

Unrestricted backtracking is described in the next chapter. We first introduce random backtracking, that is a special case of unrestricted backtracking. Afterwards, we characterize unrestricted backtracking. Completeness conditions are then discussed, based on a few general results on completeness. Some of these completeness conditions may allow deleting some of the clauses. This avoids an exponential growth of the clause database. Finally, we give experimental results.

Chapter 6 discusses hidden structure in unsatisfiable random 3-SAT formulas. We first define unsatisfiable cores and strong backdoors. Then we describe the behavior of random 3-SAT instances, followed by results that relate hardness with hidden structure. Afterwards, we suggest an algorithm for giving even more accurate results on the size of unsatisfiable cores and strong backdoors. The algorithm confirms the previous results relating hardness with hidden structure in unsatisfiable random 3-SAT instances.

Finally, we conclude the dissertation and suggest future research work.

A final remark regarding the experimental results. Most of the chapters include experimental results using one of our SAT solvers. In chronological order we have CQuest0.5, JQuest, JQuest2 and CQuest. These solvers were developed during this PhD, and therefore for each of our contributions we used the most recent solver available at the time the research work was conducted. All of these solvers are quite competitive for solving hard real-world instances of satisfiability. JQuest, JQuest2 and CQuest entered in the SAT Competition in 2002, 2003 and 2004, respectively, and were among the “top 10” SAT solvers in the industrial category.

2

Satisfiability Algorithms

Over the years a large number of algorithms has been proposed for SAT, from the original Davis-Putnam (DP) procedure (Davis & Putnam 1960), followed by the Davis-Logemann-Loveland (DLL) procedure (Davis, Logemann, & Loveland 1962), to recent backtrack search algorithms (Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997; Li & Anbulagan 1997; Zhang 1997; Moskewicz *et al.* 2001; Goldberg & Novikov 2002; Ryan 2004) and to local search algorithms (Selman, Levesque, & Mitchell 1992; Selman & Kautz 1993), among many others. Local search algorithms can solve extremely large satisfiable instances of SAT. These algorithms have also been shown to be very efficient on randomly generated instances of SAT. On the other hand, several improvements to the DLL backtrack search algorithm have been introduced. These improvements have been shown to be crucial for solving large instances of SAT derived from real-world applications, and in particular for those where local search cannot be applied, i.e. for unsatisfiable instances. Indeed, proving unsatisfiability is often the objective in a large number of significant real-world applications.

In this chapter we start by providing the definitions used throughout this dissertation. Afterwards, we describe both DP and DLL procedures. Moreover, other characteristics of a competitive SAT solver are carefully described, namely non-chronological backtracking with clause learning, accurate branching heuristics and efficient search strategies. Finally, we give empirical results for the different algorithms.

2.1 Definitions

A conjunctive normal form (CNF) formula φ on n binary variables $X = \{x_1, \ldots, x_n\}$ is the conjunction of m clauses $\Omega = \{\omega_1, \ldots, \omega_m\}$ each of which is the disjunction of one or more literals, where a literal is the occurrence of a variable x or its complement $\neg x$. A clause having a variable and its complement is called a *tautology* and is always satisfied regardless the given assignment. A clause with no literals is an *empty clause* and is always unsatisfied. A clause with one literal is called *unit*, with two literals is called *binary* and with three literals is called *ternary*. Moreover, a literal is *pure* if its complement does not occur in the formula. For a CNF formula φ and for each clause ω we can also use set notation. Hence, $\omega \in \varphi$ means that clause ω is a clause of formula φ, and $l \in \omega$ means that l is a literal of clause ω. Often, variables are also denoted by a sequence of alphabetically ordered letters, e.g. $a, b, c, \ldots$ or x, y, z.

A formula φ denotes a unique n-variable Boolean function $f(x_1, \ldots, x_n)$ and each of its clauses corresponds to an implicate of f. Clearly, a function f can be represented by many equivalent formulas. A binary variable can be assigned a truth value v which may assume value 0 (or $false$/F) or 1 (or $true$/T). Also, clauses and formulas may assume values depending on the values of the corresponding literals and clauses, respectively. The SAT problem is concerned with finding an assignment to the arguments of $f(x_1, \ldots, x_n)$ that makes the function equal to 1 or proving that the function is equal to the constant 0.

A *truth assignment* $A_{X'} : X' \subseteq X \rightarrow \{true, false\}$ for a formula φ is a subset of assigned variables X' and their corresponding binary values. For a matter of simplicity, an assignment is also denoted by A. A pair with a variable and a value is denoted by α. Informally, we often refer to α as an assignment and to A as a set of assignments.

An assignment $A_{X'}$ is *complete iff* $|X'| = n$; otherwise it is *partial*. Moreover, $\varphi[A_{X'}]$ denotes formula φ after setting the truth assignment $A_{X'}$ and $\nu(\varphi[A_{X'}])$ denotes the value of formula φ after setting the truth assignment $A_{X'}$. Similarly, $\omega[A_{X'}]$ denotes clause ω after setting the truth assignment $A_{X'}$ and $\nu(\omega[A_{X'}])$ denotes the value of clause ω after setting the truth assignment $A_{X'}$. It will be convenient to represent such assignments as sets of pairs of variables and their assigned values; for example $A = \{\langle x_1, 0\rangle, \langle x_7, 1\rangle, \langle x_{13}, 0\rangle\}$. Alternatively, assignments can also be denoted by $A = \{x_1 = 0, x_7 = 1, x_{13} = 0\}$. Moreover, the truth value v assigned to a variable x is denoted by $\nu(x)$. For the given example, $\nu(x_1) = 0$, $\nu(x_7) = 1$ and $\nu(x_{13}) = 0$.

Evaluating a formula φ for a given truth assignment $A_{X'}$ yields three possible outcomes: $\nu(\varphi[A_{X'}]) = 1$ and we say that φ is satisfied and refer to $A_{X'}$ as a *satisfying assignment*; $\nu(\varphi[A_{X'}]) = 0$ in which case φ is unsatisfied and $A_{X'}$ is referred to as an *unsatisfying or conflicting assignment*; and $\nu(\varphi[A_{X'}]) = U$ indicating that the value of φ is undefined, i.e. cannot be resolved by the assignment. This last case can only happen when $A_{X'}$ is a partial assignment.

An assignment $A_{X'}$ also partitions the clauses of φ into three sets: *satisfied* clauses when $\nu(\omega[A_{X'}]) = 1$; *unsatisfied* clauses when $\nu(\omega[A_{X'}]) = 0$; and *unresolved* clauses when $\nu(\omega[A_{X'}]) = U$. The unassigned literals of a clause are referred to as its *free literals*. In a search context, a clause is said to be *unit* if the number of its free literals is one. Similarly, a clause with two free literals is said to be *binary* and a clause with three free literals is said to be *ternary*. The search process is declared to reach a *conflict* whenever $\nu(\varphi[A_{X'}]) = 0$ for a given assignment $A_{X'}$.

Example 2.1 *Let us consider a CNF formula φ having three clauses ω_1, ω_2 and ω_3:*

$$\omega_1 = (x_1 \vee x_2),\ \omega_2 = (x_2 \vee \neg x_3),\ \omega_3 = (x_1 \vee x_2 \vee x_3)$$

Suppose that the current truth assignment is $A = \{x_1 = 0, x_3 = 0\}$. This implies having clauses ω_1 and ω_3 unresolved and clause ω_2 satisfied. Observe that clauses ω_1 and ω_3 are also unit due to x_2 being the only free literal. Hence, $\nu(\varphi[A]) = U$. Suppose that this assignment is extended with $x_2 = 0$, i.e. $A' = \{x_1 = 0, x_2 = 0, x_3 = 0\}$. Consequently, clause ω_3 becomes unsatisfied. This means that the search reached a conflict, i.e. $\nu(\varphi[A']) = 0$. Also, suppose that in the subsequent search we have the assignment $A'' = \{x_1 = 1, x_3 = 0\}$. Clearly, all the clauses get satisfied and consequently

$\nu(\varphi[A'']) = 1$.

2.2 DP Resolution-Based Algorithm

Resolution provides a complete proof system by refutation (Robinson 1965). Given two clauses $\omega_a = (z \vee x_1 \vee ... \vee x_m)$ and $\omega_b = (\neg z \vee y_1 \vee ... \vee y_n)$, where all x_i and y_j are distinct literals, resolution allows deriving the resolvent clause $\omega_c = (x_1 \vee ... \vee x_m \vee y_1 \vee ... \vee y_n)$, that is, the disjunction of ω_a and ω_b without z and $\neg z$. We consider ω_c to be the result of applying $res(\omega_a, \omega_b, z)$. The resolvent is a logical consequence of the conjunction of the two clauses. Repeated applications of the resolution inference rule will reduce the formula to a set of clauses having only pure literals if the instance is satisfiable or will derive the empty clause if the instance is unsatisfiable.

Resolution was first applied to the SAT problem by Davis and Putnam (Davis & Putnam 1960), and therefore the complete resolution procedure for SAT is often referred to as the Davis-Putnam procedure. In addition, the Davis-Putnam procedure also applies the unit clause rule and the pure literal rule. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfied. In this case, the value of the literal and of the associated variable are said to be *implied*. Also, each implied assignment is associated with an *explanation*, i.e. with the unit clause that implied the assignment. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation(BCP) (Zabih & McAllester 1988). A literal is pure if its complement does not occur in the formula. Clearly, the satisfiability of a formula is unaffected by satisfying those literals. Moreover, no resolvents can be generated by resolving on a pure literal, but all clauses containing a pure literal can be removed without loss. Hence, an important improvement to the basic resolution algorithm is to first imply assignments and remove clauses containing pure literals.

The pseudo-code for the Davis-Putnam procedure is given in Algorithm 2.1. Besides applying the unit clause rule (APPLY_BCP) and the pure literal rule (APPLY_PLR), the resolution rule is applied for eliminating the variables one-by-one (ELIMINATE_VARIABLE),

Algorithm 2.1: Davis-Putnam procedure

```
DAVIS-PUTNAM($\varphi$)
(1)   while TRUE
(2)     if APPLY_BCP($\varphi$) == CONFLICT
(3)       return UNSAT
(4)     APPLY_PLR($\varphi$)
(5)     if ALL_CLAUSES_SAT($\varphi$)
(6)       return SAT
(7)     $x$ = SELECT_VARIABLE($\varphi$)
(8)     ELIMINATE_VARIABLE($\varphi$, $x$)
```

thus adding all possible resolvents to the set of clauses. The idea is to iteratively apply resolution to eliminate one selected variable each time (SELECT_VARIABLE), i.e., resolution between all pairs of clauses containing literals x and $\neg x$. Each step generates a sub-problem with one fewer variable x, but possibly quadratically more clauses, depending on the number of clauses in x and $\neg x$. The procedure stops applying resolution when either the formula is found to be satisfiable or unsatisfiable. A formula is declared to be satisfied when it contains only satisfied clauses or pure literals. A formula is declared to be unsatisfied whenever a conflict is reached. Moreover, conflicts are detected while applying BCP.

Example 2.2 *Consider formula φ having the following clauses:*

$$\omega_1 = (x_1 \vee x_3),\ \omega_2 = (x_2 \vee x_3),\ \omega_3 = (x_3 \vee x_4),\ \omega_4 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Let us start by applying resolution in order to eliminate variable x_3. The obtained formula φ' has three clauses ω', ω'' and ω''':

$$\omega' = res(\omega_1, \omega_4, x_3) = (x_1 \vee \neg x_1 \vee \neg x_2)$$
$$\omega'' = res(\omega_2, \omega_4, x_3) = (\neg x_1 \vee x_2 \vee \neg x_2)$$
$$\omega''' = res(\omega_3, \omega_4, x_3) = (\neg x_1 \vee \neg x_2 \vee x_4)$$

Observe that ω' and ω'' are tautologous clauses and therefore the remaining clause ω''' has only pure literals. Thus formula φ' and consequently formula φ is satisfied.

One can readily conclude that this procedure requires exponential space in general. In practice, many resolvents are generated but only a small part of them is used either to find a solution or to prove unsatisfiability. For this reason, Davis, Longemann and Loveland (Davis, Logemann, & Loveland 1962) replaced the resolution rule with a splitting rule which divides the problem into two smaller subproblems. During each iteration, the procedure selects a variable and generates two sub-formulas by assigning the two values, true and false, to the selected variable. This procedure will be described in the next section.

2.3 DLL Backtrack Search Algorithm

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland (DLL) (Davis, Logemann, & Loveland 1962). The backtrack search algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n problem variables.

Starting from an empty truth assignment, a backtrack search algorithm enumerates the space of truth assignments implicitly and organizes the search to find a satisfying assignment by searching a *decision tree*. Each node in the decision tree specifies an elective assignment to an unassigned variable; such assignments are referred to as *decision assignments*. A *decision level* is associated with each decision assignment to denote its depth in the decision tree; the first decision assignment at the root of the tree is at decision level 1. Assignments made before the first decision, i.e. during preprocessing, are assigned at decision level 0. In general, the notation $x = \nu(x)@\delta(x)$ is used to denote a variable x assigned at decision level $\delta(x)$ with value $\nu(x)$. For each new decision assignment, the decision level is incremented by 1. After each decision assignment, the *unit clause rule* (Davis & Putnam 1960) is applied iteratively, i.e. BCP is applied. Also, each implied assignment is associated with an *explanation*, i.e. with the unit clause that implied the assignment.

Algorithm 2.2 gives the pseudo-code for a DLL-based backtrack search algorithm. Given a SAT problem, formulated as a CNF formula φ, the algorithm conducts a search

Algorithm 2.2: DLL-based backtrack search algorithm

```
SAT($\varphi$)
(1)     $d = 0$
(2)     while Decide($\varphi, d$) == DECISION
(3)         if Deduce($\varphi, d$) == CONFLICT
(4)             $\beta$ = Diagnose($\varphi, d$)
(5)             if $\beta$ == -1
(6)                 return UNSATISFIABLE
(7)             else
(8)                 Backtrack($\varphi, d, \beta$)
(9)                 $d = \beta$
(10)        else
(11)            d = d + 1
(12)    return SATISFIABLE
```

through the space of all possible assignments to the n problem variables. At each stage of the search, a variable assignment is selected with the Decide function. A decision level d is then associated with each selection of an assignment. Implied assignments are identified with the Deduce function, which in most cases corresponds to the straightforward derivation of implications by applying BCP. Whenever a clause becomes unsatisfied, the Deduce function returns a conflict indication which is then analyzed using the Diagnose function. The diagnosis of a given conflict returns a backtracking decision level β, which denotes the decision level to which the search process is required to backtrack to. Afterwards, the Backtrack function clears *all* assignments (both decision and implied assignments) from the current decision level d through the backtrack decision level β. Furthermore, considering that the search process should resume at the backtrack level, the current decision level d becomes β. Finally, the current decision level d is incremented. This process is interrupted whenever the formula is found to be satisfiable or unsatisfiable. The formula is satisfied when all variables are assigned (meaning Decide(φ, d) != DECISION) and therefore all clauses must be satisfied. The formula is unsatisfied when the empty clause is derived, which is implicit when the Diagnose function returns -1 as the backtrack level.

Given the above description, it is usual to decompose the backtrack search algorithm into three main engines:

1. The decision engine DECIDE, which selects a decision assignment at each stage of the search. (`DECISION` is returned unless all variables are assigned or all clauses are satisfied.) Observe that selected variable assignments have no explanation. This engine is the basic mechanism for exploring new regions of the search space.

2. The deduction engine DEDUCE, which identifies assignments that are deemed necessary, usually called implied assignments. Whenever a clause becomes unsatisfied, implying that the current assignment is not a satisfying assignment, we have a *conflict*, and the associated unsatisfying assignment is called a *conflicting assignment*. The DEDUCE function then returns a conflict indication which is then analyzed using the DIAGNOSE function.

3. The diagnosis engine DIAGNOSE, which identifies the causes of a given conflicting partial assignment. The diagnosis of a given conflict returns a backtracking decision level, which corresponds to the decision level to which the search must backtrack. This backtracking process is the basic mechanism for retreating from regions of the search space where satisfying assignments do not exist.

Distinct organizations of SAT algorithms can be obtained by different configurations of this generic algorithm. For example, the original DLL procedure can be captured by this algorithms as follows:

- The decision engine randomly picks an unassigned variable x, since the DLL procedure does not specify the variable upon which to branch.

- The deduction engine implements Boolean constraint propagation. If no conflicts are detected, the pure literal rule is applied: if a variable x only occurs either as a positive or as a negative literal, then all clauses with a literal on x become satisfied.

- The diagnosis engine implements plain backtracking (also called chronological backtracking). The search algorithm keeps track of which decision assignments have been toggled. Given a conflict that occurs at decision level d, the algorithm checks whether the corresponding decision variable x has already been toggled. If not, it

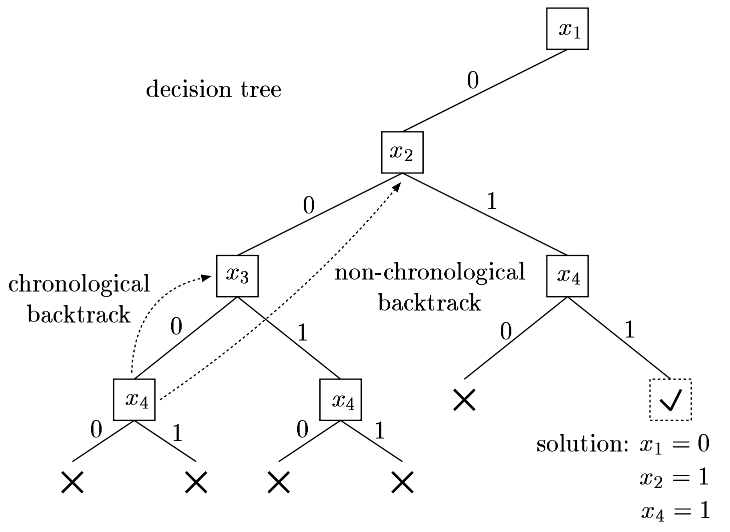


Figure 2.1: Chronological and non-chronological backtracking

erases the variable assignments which are implied by the assignment on x, including the assignment on x, and assigns the opposite value to x. In contrast, if the value of x has already been toggled, the search backtracks to the level corresponding to the most recent yet untoggled decision variable.

2.4 Non-Chronological Backtracking

All of the most efficient recent state-of-the-art SAT solvers (Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997; Li & Anbulagan 1997; Zhang 1997; Moskewicz *et al.* 2001; Goldberg & Novikov 2002; Ryan 2004) utilize different forms of non-chronological backtracking (NCB). Non-chronological backtracking, as opposed to chronological backtracking (CB), backs up the search tree to one of the identified causes of failure, skipping over irrelevant variable assignments.

Example 2.3 *Let us consider the CNF formula φ having the following four clauses:*

$$\omega_1 = (x_2 \vee x_4),\ \omega_2 = (x_1 \vee \neg x_2 \vee x_4),\ \omega_3 = (x_2 \vee \neg x_4),\ \omega_4 = (\neg x_1 \vee x_3)$$

The search tree is illustrated in Figure 2.1. (For a matter of simplicity, the unit clause rule was not considered.) Once both x_1 and x_2 are assigned value 0, there are no possible *assignments for the remaining variables x_3 and x_4 that can satisfy the formula. In this example, chronological backtracking wastes a potentially significant amount of time exploring a region of the search space without solutions, only to discover, after potentially much effort, that the region does not contain any satisfying assignments. On the other hand, non-chronological backtracking is able to extract information from the first two conflicts and then conclude that those conflicts are not related with the value given to x_3 but rather to the value given to x_2.*

Conflict-directed backjumping (Prosser 1993) is one of the most accurate forms of non-chronological backtracking. Nonetheless, the forms of non-chronological backtracking used in state-of-the-art SAT solvers are most related to *dependency-directed backtracking* (Stallman & Sussman 1977), since they are always associated with learning from conflicts (this technique is often called *clause learning* or *clause recording*). Clause learning consists of the following: for each identified conflict, its causes are identified, and a new recorded clause (also called *nogood* or *conflict clause*) is created to explain and subsequently prevent the identified conflicting conditions.

In the next section we start by addressing conflict-directed backjumping (Prosser 1993). Afterwards, we describe the use of conflict-directed backjumping jointly with learning, i.e. dependency-directed backtracking. Finally, we discuss different clause deletion policies.

2.4.1 Conflict-Directed Backjumping

Conflict-directed backjumping (CBJ) (Prosser 1993) can be considered a combination of Gaschnig's backjumping (BJ) (Gaschnig 1979) and Dechter's graph-based backjumping (GBJ) (Dechter 1990).

BJ aims performing higher jumps in the search tree, rather than backtracking to the most recent yet untoggled decision variable. For each value of a variable v_j, Gaschnig's algorithm obtains the lowest level for which the considered assignment is inconsistent. In addition, BJ uses a marking technique that maintains, for each variable v_j, a reference to a variable v_i with the deepest level of the different levels with which any value of v_j was found to be inconsistent. Hence, a backjump from v_j is to v_i. Moreover, if the domain of

v_i is wiped-out, then the search must chronologically backtrack to v_{i-1}.

As an improvement, Dechter's GBJ extracts knowledge about dependencies from the constraint graph. CBJ builds upon this idea and, based on dependencies from the constraints, records the *set* of past variables that failed consistency checks with each variable v. This set (called *conflict set* in (Dechter 1990)) allows the algorithm to perform multiple jumps.

2.4.2 Learning and Conflict-Directed Backjumping

Learning can be combined with CBJ when each identified conflict is analyzed, its causes are identified, and a clause is recorded to explain and prevent the identified conflicting conditions from occurring again during the subsequent search. This technique is called *dependency-directed backtracking* and was proposed in (Stallman & Sussman 1977). Moreover, the newly recorded clause is then used to compute the backtrack point as the *most recent* decision assignment from all the decision assignments represented in the recorded clause. GRASP (Marques-Silva & Sakallah 1996) and relsat (Bayardo Jr. & Schrag 1997) were the first SAT solvers to successfully implement conflict-directed backjumping enhanced with learning.

For implementing learning techniques common to some of the most competitive backtrack search SAT algorithms, it is necessary to properly *explain* the truth assignments given to the propositional variables that are implied by the clauses of the CNF formula. The *antecedent assignment* of x, denoted by $Antec(x)$, is defined as the set of assignments to variables other than x with literals in ω. Intuitively, $Antec(x)$ designates those variable assignments that are directly responsible for implying the assignment of x due to ω. For example, let $\omega = (x_1 \vee \neg x_2 \vee x_3)$ be a clause of a CNF formula φ, and assume the assignment $A = \{x_1 = 0, x_3 = 0\}$. For having $\nu(\omega[A]) = 1$ we must necessarily have $\nu(x_2) = 0$. Hence, we say that the *antecedent assignment* of x_2, denoted by $Antec(x_2)$, is defined as $Antec(x_2) = \{x_1 = 0, x_3 = 0\}$.

In addition, in order to explain other NCB-related concepts, we shall often analyze the *directed acyclic implication graph* created by the sequences of implied assignments

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2, ...\}$
Current Decision Assignment: $\{x_1 = 1@6\}$

$\omega_1 = (\neg x_1 \vee x_2)$
$\omega_2 = (\neg x_1 \vee x_3 \vee x_9)$
$\omega_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$
$\omega_4 = (\neg x_4 \vee x_5 \vee x_{10})$
$\omega_5 = (\neg x_4 \vee x_6 \vee x_{11})$
$\omega_6 = (\neg x_5 \vee \neg x_6)$
$\omega_7 = (x_1 \vee x_7 \vee \neg x_{12})$
$\omega_8 = (x_1 \vee x_8)$
$\omega_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...

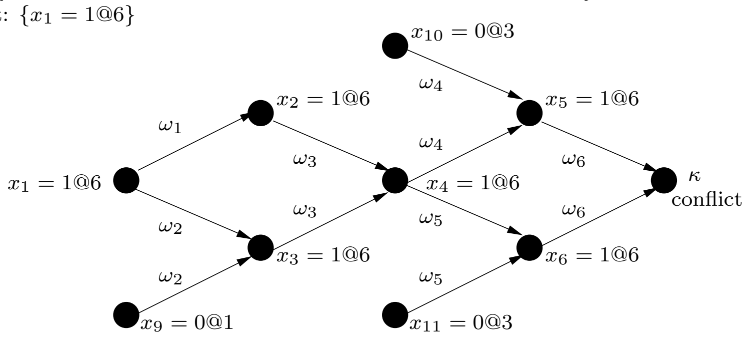


Figure 2.2: Example of conflict diagnosis with clause recording

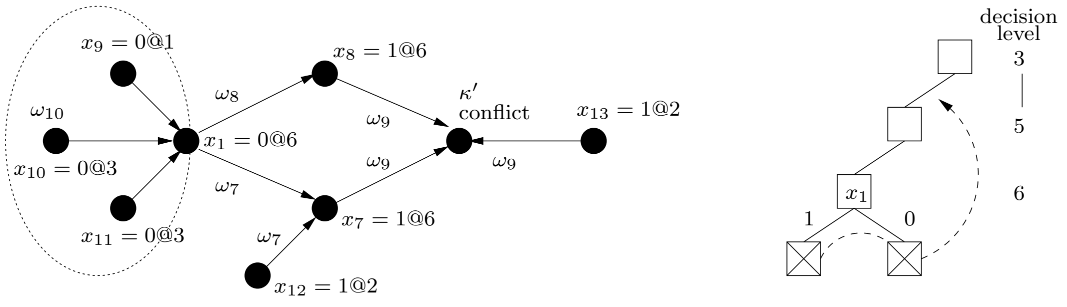
generated by BCP. An implication graph I is defined as follows:

1. Each vertex in I corresponds to a variable assignment at a given decision level $x = \nu(x)@\delta(x)$.

2. The predecessors of vertex $x = \nu(x)@\delta(x)$ in I are the antecedent assignments $Antec(x)$ corresponding to the unit clause ω that caused the value of x to be implied. The directed edges from the vertices in $Antec(x)$ to vertex $x = \nu(x)@\delta(x)$ are all labeled with ω. Vertices that have no predecessors correspond to decision assignments.

3. Special conflict vertices are added to I to indicate the occurrence of conflicts. The predecessors of a conflict vertex κ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignment $Antec(\kappa)$. The directed edges from the vertices in $Antec(\kappa)$ to κ are all labeled with ω.

Next, we illustrate clause recording with the example of Figure 2.2. A subset of the CNF formula is shown, and we assume that the current decision level is 6, corresponding to the decision assignment $x_1 = 1$. This assignment yields a conflict κ involving clause ω_6. By inspection of the implication graph, we can readily conclude that a *sufficient condition* for this conflict to be identified is



(a) Conflicting implication sequence (b) Decision tree

Figure 2.3: Computing the backtrack decision level

$$(x_{10} = 0) \wedge (x_{11} = 0) \wedge (x_9 = 0) \wedge (x_1 = 1)$$

By creating clause $\omega_{10} = (x_{10} \vee x_{11} \vee x_9 \vee \neg x_1)$, we prevent the same set of assignments from occurring again during the subsequent search.

In order to illustrate non-chronological backtracking based on clause recording, let us now consider the example of Figure 2.3, which continues the example in Figure 2.2. This figure illustrates non-chronological backtracking based on clause recording. After recording clause $\omega_{10} = (x_{10} \vee x_{11} \vee x_9 \vee \neg x_1)$, BCP implies the assignment $x_1 = 0$ because clause ω_{10} becomes unit at decision level 6. By inspection of the CNF formula (see Figure 2.2), we can conclude that clauses ω_7 and ω_8 imply the assignments shown, and so we obtain a conflict κ' involving clause ω_9. By creating clause $\omega_{11} = (\neg x_{13} \vee \neg x_{12} \vee x_{11} \vee x_{10} \vee x_9)$ we prevent the same conflicting conditions from occurring again. It is straightforward to conclude that even though the current decision level is 6, all assignments directly involved in the conflict are associated with variables assigned at decision levels less than 6, the highest of which being 3. Hence we can backtrack immediately to decision level 3.

2.4.3 Clause Deletion Policy

Unrestricted clause recording can in some cases be impractical. Recorded clauses consume memory and repeated recording of clauses can eventually lead to the exhaustion of the available memory. Observe that the number of recorded clauses grows with the number of conflicts; in the worst case, such growth can be *exponential* in the number of variables. Furthermore, large recorded clauses are known for not being particularly useful for search pruning purposes (Marques-Silva & Sakallah 1996). Adding larger clauses leads to an additional overhead for conducting the search process and, hence, it eventually costs more than what it saves in terms of backtracks.

As a result, there are three main solutions for guaranteeing the worst case growth of the recorded clauses to be *polynomial* in the number of variables:

1. We may consider *n-order learning*, that records only clauses with n or fewer literals (Dechter 1990).

2. Clauses can be temporarily recorded while they either imply variable assignments or are unit clauses, being discarded as soon as the number of unassigned literals is greater than an integer m. This technique is named *m-size relevance-based learning* (Bayardo Jr. & Schrag 1997).

3. Clauses with a size less than a threshold k are kept during the subsequent search, whereas larger clauses are discarded as soon as the number of unassigned literals is greater than one. We refer to this technique as *k-bounded learning* (Marques-Silva & Sakallah 1996).

Observe that we can combine k-bounded learning with m-size relevance-based learning. The search algorithm is organized so that all recorded clauses of size no greater than k are kept and larger clauses are deleted only after m literals have become unassigned.

More recently, a heuristic clause deletion policy has been introduced (Goldberg & Novikov 2002). Basically, the decision whether a clause should be deleted is based not only on the number of literals but also on its *activity* in contributing to conflict making and on the number of decisions taken since its creation.

2.5 Branching Heuristics

The heuristics used for variable selection during the search, and consequently responsible for the organization of the decision engine, represent a key aspect of backtrack search SAT algorithms. Several heuristics have been proposed over the years, each denoting a trade-off between computational requirements and the ability to reduce the amount of search (Hooker & Vinay 1995). Examples of decision making heuristics include M. Bohm's heuristic (briefly described in (Buro & Kleine-Büning 1992)), the Jeroslow-Wang branching rule (Jeroslow & Wang 1990), MOM's heuristic (Freeman 1995), BCP-based heuristics (Bayardo Jr. & Schrag 1997; Li & Anbulagan 1997) and variable state independent heuristics (Moskewicz *et al.* 2001; Goldberg & Novikov 2002; Ryan 2004). Most heuristics try to constrain the search as much as possible, by identifying at each step decision assignments that are expected to imply the largest number of variable assignments. For example, the one-sided Jeroslow-Wang heuristic (Hooker & Vinay 1995; Jeroslow & Wang 1990) assigns value true to the literal l that maximizes the following function:

$$J(l) = \sum_{l \in C_i} 2^{-n_i}$$

where n_i is the number of free literals in unresolved clause C_i. Hence, preference is given to satisfying a literal that occurs in the largest number of the smallest clauses. MOM's heuristic (Freeman 1995), for example, also gives preference to variables that occur in the smallest clauses, but variables are preferred if they simultaneously maximize their number of positive and negative literals in the smallest clauses. Bohm's heuristic applies a similar reasoning, giving preference to variables that occur more often in the smallest clauses and, among these variables, to those for which both positive and negative literals occur in the smallest clauses.

Other heuristics involve BCP-based probing of each unassigned variable, in order to decide which variable will lead to the largest number of implied assignments (Bayardo Jr. & Schrag 1997; Freeman 1995; Li & Anbulagan 1997). Broadly, probing consists of assigning both values to a variable and then propagating these values. These heuristics are typically integrated in DLL-like SAT solvers. The main goal is to reduce the number of decisions required either to prove satisfiability or unsatisfiability. These kind of techniques are called look-ahead techniques, as opposed to look-back techniques (e.g. learning) that aim to repair wrong decisions in the past. Observe that having an accurate heuristic reduces the number of conflicts and also the number of decisions made during the search. So, one may argue that learning is only necessary when the heuristic being used is not accurate.

Satz (Li & Anbulagan 1997) is a SAT solver that successfully applies a look-ahead heuristic. Algorithm 2.3 illustrates a generic look-ahead branching rule. The sequence of *if*-conditions correspond to the well-known *failed-literal rule* (Crawford & Auton 1993). If forcing an assignment to a variable $x = \nu(x)$ and then performing BCP yields a conflict then $x = 1 - \nu(x)$ is a necessary assignment. Observe that a conflict is reached when the empty clause (denoted by $()$) is derived. Specific configurations lead to different look-ahead techniques. Each configuration determines how to select the candidate variables and how to calculate the weight-based scores (denoted by W), thus allowing to select the next branching variable. For example, satz selects a candidate variable depending on the predicate $\mathrm{PROP}(x, i)$ that returns true *iff* x occurs both positively and negatively in binary clauses and x has at least i binary occurrences in φ. Moreover, satz branches on the candidate variable x such that $H(x)$ is the greatest, where

$$H(x) = \mathrm{W}(\neg x) * \mathrm{W}(x) * 1024 + \mathrm{W}(\neg x) + \mathrm{W}(x)$$

as suggested by Freeman in POSIT (Freeman 1995).

More recently, a different kind of variable selection heuristic (referred to as VSIDS, Variable State Independent Decaying Sum) has been proposed by Chaff authors (Moskewicz *et al.* 2001). One of the reasons for proposing this new heuristic was the introduction of *lazy* data structures (to be described in the next chapter), where the knowledge of the dynamic size of a clause is not accurate. Hence, the heuristics described

Algorithm 2.3: Generic look-ahead branching rule

```
LOOK-AHEAD BRANCHING RULE($\varphi$)
(1)   foreach candidate variable $x$
(2)      $\varphi' = $ APPLY_BCP$(\varphi \cup \{x\})$
(3)      $\varphi'' = $ APPLY_BCP$(\varphi \cup \{\neg x\})$
(4)      if $() \in \varphi'$ and $() \in \varphi''$
(5)         return UNSATISFIABLE
(6)      if $() \in \varphi'$
(7)         $\varphi = \varphi''$
(8)      else if $() \in \varphi''$
(9)         $\varphi = \varphi'$
(10)     else
(11)        W$(x) = $ DIFF$(\varphi', \varphi)$
(12)        W$(\neg x) = $ DIFF$(\varphi'', \varphi)$
(13)  BRANCH(variable with best W-based score)
```

above cannot be utilized.

VSIDS selects the literal that appears most frequently over all the clauses, which means that the metrics only have to be updated when a new recorded clause is created. More than to develop an *accurate* heuristic, the motivation has been to design a *fast* (but dynamically adaptive) heuristic. In fact, one of the key properties of this strategy is the very low overhead, due to being independent of the variable state. Two chaff-like SAT solvers, BerkMin (Goldberg & Novikov 2002) and siege (Ryan 2004), have improved the VSIDS heuristic. BerkMin also measures clauses' age and activity for deciding the next branching variable, whereas siege gives priority to assigning variables on recently recorded clauses.

2.6 Search Strategies

Search strategies are used to implement different organizations of the search process.

The most well-known strategy consists in randomizing the variable selection heuristic used for selecting variables and also the values to assign to them (Bayardo Jr. & Schrag 1997). Although intimately related with randomizing variable selection heuristics, randomization is also a key aspect of search restart strategies (Baptista & Marques-Silva 2000; Gomes, Selman, & Kautz 1998). Randomization ensures with high probability that different sub-trees are searched each time the backtrack search algorithm is restarted. Randomization can be also integrated in the backtrack step of a backtrack search algorithm (Lynce, Baptista, & Marques-Silva 2001a), allowing the search to randomly backtrack after a conflict is found. Random backtracking will be further described in Chapter 5.

Current state-of-the-art SAT solvers already incorporate some of the above forms of randomization (Baptista & Marques-Silva 2000; Moskewicz *et al.* 2001; Goldberg & Novikov 2002). In these SAT solvers, variable selection heuristics are randomized and search restart strategies are utilized. Randomized restarts have been shown to yield dramatic improvements on satisfiable instances that exhibit heavy-tailed behavior (Gomes, Selman, & Kautz 1998). Also, completeness conditions have been established to apply random restarts on solving unsatisfiable instances (Baptista & Marques-Silva 2000).

Algorithm portfolio design (Gomes & Selman 1997) is another search strategy. This work was motivated by observing that the runtime and the performance of stochastic algorithms can vary dramatically on solving the same instance. Nonetheless, randomized algorithms are among the best current algorithms for solving satisfiable computationally hard problems. Hence, the main goal is to improve the performance of algorithms by combining them into a portfolio to exploit stochasticity. Combining different algorithms into a portfolio only makes sense if they exhibit different probability profiles and none of them dominates the others over the whole spectrum of problem instances.

2.7 Experimental Results

The experimental results given below were obtained using the JQuest SAT solver, a Java framework of SAT algorithms. JQuest implements a significant number of the most well-known SAT techniques, and can be used to conduct unbiased experimental evaluations of SAT techniques and algorithms. The idea is to experimentally evaluate the different approaches in a controlled experiment that ensures that only specific differences are evaluated. Besides differing data structures and coding styles, each existing SAT solver implements its own set of search techniques, strategies and heuristics. Hence, a comparison between state-of-the-art SAT solvers hardly guarantees meaningful results.

In order to perform this comparison using the JQuest SAT solver, instances were selected from several classes of instances available from SATLIB web site `http://www.satlib.org/` (see Table 2.1). For each problem instance, we indicate the application domain, the number of variables, the number of clauses and whether the instance is satisfiable. In all cases, the problem instances chosen can be solved with several thousand decisions by the most efficient solvers, usually taking a few tens of seconds, and thus being significantly hard. For this reason, different algorithms can provide significant variations on the time required for solving a given instance. In addition, we should also observe that the problem instances selected are intended to be *representative*, since each resembles, in terms of hardness for SAT solvers, the typical instance in each class of problem instances. For the results shown a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used. The Java Virtual Machine used was SUN's HotSpot JVM for JDK1.4. The CPU time was limited to 1500 seconds.

The first table of results (Table 2.2) shows the CPU time required to solve each problem instance. (Instances that were not solved in the allowed CPU time are marked with —.) For the algorithms considered: **CB** denotes the chronological backtracking search SAT algorithm (based on DLL), **CBJ** denotes the DLL-CBJ SAT algorithm and **CBJ+cr** denotes the CBJ SAT algorithm with clause recording. Moreover, a variety of clause deletion policies were considered, depending on the value of k, where k defines the *k-bounded learning* procedure used (see Section 2.4.3). For instance, `+cr10` means that recorded clauses with size greater than `10` are deleted as soon as they become unresolved (i.e. not satisfied with more than one unassigned literal), whereas `+crAll` means that *all* the recorded clauses are kept.

Table 2.2 reveals interesting trends, and several conclusions can be drawn:

- Clearly, CB and CBJ have in general similar behavior (except for *bf0432-079* and *data encryption standard* instances).

- The CBJ+cr algorithms are in general clearly more efficient than the other algo-

Table 2.1: Example instances

Application Domain	Selected Instance	# Variables	#Clauses	Satisfiable?
Circuit Testing (Dimacs)	bf0432-079	1044	3685	N
	ssa2670-141	4843	2315	N
Inductive Inference(Dimacs)	ii16b2	1076	16121	Y
	ii16e1	1245	14766	Y
Parity Learning(Dimacs)	par16-1-c	317	1264	Y
	par16-4	1015	3324	Y
Graph Colouring	flat200-39	600	2237	Y
	sw100-49	500	3100	Y
Quasigroup	qg3-08	512	10469	Y
	qg5-09	729	28540	N
Blocks World	2bitadd_12	708	1702	Y
	4blocksb	410	24758	Y
Planning-Sat	logistics.a	828	6718	Y
	bw_large.c	3016	50457	Y
Planning-Unsat	logistics.c	1027	9507	N
	bw_large.b	920	11491	N
Bounded Model Checking	barrel5	1407	5383	N
	queueinvar16	1168	6496	N
	longmult6	2848	8853	N
Superscalar Processor Verification	dlx2_aa	490	2804	N
	dlx2_cc_a_bug17	4847	39184	Y
	2dlx_cc_mc_ex_bp_f2_bug006	4824	48215	Y
	2dlx_cc_mc_ex_bp_f2_bug010	5754	60689	Y
Data Encryption Standard	cnf-r3-b2-k1.1	5679	17857	Y
	cnf-r3-b4-k1.2	2855	35963	Y

rithms. Indeed, for almost all the instances CBJ+cr achieves remarkable improvements, when compared with CB or with CBJ. Instances *flat200-39* and *barrel5* are the only exceptions. (For instance *barrel5*, this is only true for CBJ+cr with small values of *k*.)

- Some of the instances that are not solved by CBJ in the allowed CPU time (e.g *ii16b2* and *dlx2_cc_a_bug17*), also need a significant amount of time to be solved by *k-bounded learning* with a small value of *k*.

- For instance *flat200-39*, recorded clauses result in an additional search effort to find a solution.

- From a practical perspective, unrestricted clause recording is *not* necessarily a bad approach.

Table 2.2: CPU time (in seconds)

Instance	**CB**	**CBJ**	**CBJ+cr**						
			`+cr0`	`+cr5`	`+cr10`	`+cr20`	`+cr50`	`+cr100`	`+crAll`
bf0432-079	—	41.74	5.18	2.78	2.97	2.70	1.53	**1.44**	1.45
ssa2670-141	—	—	1.21	0.87	0.81	**0.52**	0.55	0.54	0.56
ii16b2	—	—	—	—	857.61	302.63	158.63	141.41	**141.05**
ii16e1	—	—	20.58	26.75	20.40	12.65	12.89	15.96	**11.86**
par16-1-c	65.92	77.06	19.91	**14.75**	16.07	16.78	18.19	18.08	18.13
par16-4	14.88	20.59	11.51	8.49	8.77	9.34	7.16	7.20	**7.14**
flat200-39	**8.44**	8.75	97.35	255.04	85.19	114.05	67.55	67.00	67.19
sw100-49	—	—	1.94	13.48	1.26	2.18	0.73	0.74	**0.71**
qg3-08	2.29	2.65	**0.86**	0.88	0.91	1.00	1.07	1.30	1.32
qg5-09	13.28	8.61	1.35	1.29	**1.06**	1.17	1.16	1.21	1.15
2bitadd_12	—	—	—	—	—	—	87.68	**50.74**	50.93
4blocksb	—	—	31.23	30.25	39.62	29.66	**16.34**	20.33	31.75
logistics.a	—	—	2.98	1.87	1.62	1.65	**1.61**	1.63	1.68
bw_large.c	—	—	76.03	55.13	**36.41**	38.37	43.71	38.03	38.06
logistics.c	—	—	26.88	7.24	**4.60**	15.40	10.22	10.23	10.25
bw_large.b	8.27	4.78	1.60	**0.59**	0.61	0.64	0.61	0.62	0.62
barrel5	99.49	132.37	171.94	279.31	36.35	**19.80**	23.43	24.00	21.99
queueinvar16	—	—	23.36	21.39	15.74	15.48	**8.05**	8.08	8.12
longmult6	—	—	27.66	**23.63**	26.20	29.16	32.32	31.74	32.02
dlx2_aa	—	—	54.74	36.21	37.96	9.80	**6.43**	6.62	6.60
dlx2_cc_a_bug17	—	—	430.31	—	—	500.25	220.06	**6.48**	6.54
2dlx_..._bug006	—	—	17.29	14.32	3.87	2.27	2.27	2.23	**2.22**
2dlx_..._bug010	—	—	3.32	4.13	3.83	5.31	4.55	2.03	**1.93**
cnf-r3-b2-k1.1	802.90	19.37	3.05	3.13	2.39	2.58	2.40	**2.16**	3.90
cnf-r3-b4-k1.2	405.98	12.62	5.42	5.39	5.48	5.12	4.89	4.45	**3.91**

Table 2.3 gives the results for the number of searched nodes, for each instance and for the different configurations. It is plain from the results that CB and CBJ in general need to search more nodes to find a solution than the other algorithms. This can be explained by the effect of the recorded clauses. Besides explaining an identified conflict, clauses are often re-used, either for yielding conflicts or for implying variable assignments, introducing significant pruning in the search tree. Moreover, other conclusions can be established from the results on the searched nodes:

- For the *par* instances, CB and CBJ have the same or an approximate number of search nodes for these instances. This is explained by the fact that there are none or just a few backjumps during the search.

Table 2.3: Searched nodes

Instance	**CB**	**CBJ**	**CBJ+cr**						
			`+cr0`	`+cr5`	`+cr10`	`+cr20`	`+cr50`	`+cr100`	`+crAll`
bf0432-079	—	98824	3939	1950	2010	1600	**1168**	1188	1188
ssa2670-141	—	—	2882	1988	1480	806	736	**698**	**698**
ii16b2	—	—	—	—	101451	32923	12188	**10573**	**10573**
ii16e1	—	—	20872	24714	17271	13869	8117	8442	**7869**
par16-1-c	52800	52800	11307	7249	7524	**5255**	5364	5364	5364
par16-4	10673	10246	4157	2676	2745	2467	**1919**	**1919**	**1919**
flat200-39	6286	**5427**	139264	287983	51308	40888	26428	25738	25738
sw100-49	—	—	4596	44247	2370	3748	**1450**	**1450**	**1450**
qg3-08	703	703	220	220	242	**214**	222	282	282
qg5-09	1578	1547	373	329	**318**	337	337	337	337
2bitadd_12	—	—	—	—	—	—	21244	**11238**	**11238**
4blocksb	—	—	5559	5007	6205	5009	2618	**2491**	3363
logistics.a	—	—	32872	16999	**14899**	15185	15185	15185	15185
bw_large.c	—	—	6137	5132	**2763**	2878	3000	2783	2783
logistics.c	—	—	55721	18839	**14520**	16444	15441	15441	15441
bw_large.b	1431	1112	293	**128**	195	195	195	195	195
barrel5	24727	24664	90115	141953	14684	8731	10396	12315	**5985**
queueinvar16	—	—	45053	41510	24842	19557	8460	8506	**8083**
longmult6	—	—	7407	5482	5666	5507	5019	4729	**4725**
dlx2_aa	—	—	319120	204995	208725	21036	10062	**10035**	**10035**
dlx2_cc_a_bug17	—	—	446626	—	—	212816	85713	**3383**	**3383**
2dlx_..._bug006	—	—	32297	25259	7775	3288	3227	**3123**	**3123**
2dlx_..._bug010	—	—	10086	19002	14358	13229	8533	3547	**3522**
cnf-r3-b2-k1.1	219037	6273	1168	1138	872	942	825	**667**	1221
cnf-r3-b4-k1.2	57843	2216	1011	1012	1010	943	910	776	**729**

- For instances *logistics.a*, *bw_large.b* and *qg5-09*, the search needs the same number of nodes for increasing values of k, since only small-size clauses are recorded.
- Usually more recorded clauses imply less searched nodes and less time needed to find a solution. (Even though the reduction in the number of nodes is more significant than the reduction in the amount of time, due to the overhead introduced by the management of additional clauses.)

Overall, the effect of clause recording is clear, and in general dramatic. The results clearly indicate that clause recording is an essential component of current state-of-the-art SAT solvers.

Finally, we also evaluate whether a different variable ordering heuristic could have affected the results. For the above results we have applied the variable selection heuris-

Table 2.4: Time and nodes for DLIS

Instance	Time		Nodes	
	CB	**CBJ**	**CB**	**CBJ**
bf0432-079	—	**437.12**	—	**275336**
ssa2670-141	—	**363.53**	—	**244848**
ii16b2	1220.50	**630.05**	53667	**37141**
ii16e1	—	**30.32**	—	**3277**
par16-1-c	41.15	**39.56**	8212	**7906**
par16-4	**130.45**	132.49	13786	**13556**
flat200-39	45.26	**8.91**	9437	**2756**
sw100-49	—	—	—	—
qg3-08	323.92	**184.29**	35120	**30943**
qg5-09	—	—	—	—
2bitadd_12	—	**49.79**	—	**51900**
4blocksb	—	—	—	—
logistics.a	—	—	—	—
bw_large.c	—	—	—	—
logistics.c	—	—	—	—
bw_large.b	49.31	**17.46**	1794	**986**
barrel5	**379.46**	411.72	**25624**	25731
queueinvar16	—	—	—	—
longmult6	—	—	—	—
dlx2_aa	—	—	—	—
dlx2_cc_a_bug17	—	—	—	—
2dlx_..._bug006	—	—	—	—
2dlx_..._bug010	0.63	**0.61**	718	**711**
cnf-r3-b2-k1.1	—	—	—	—
cnf-r3-b4-k1.2	—	—	—	—

tic VSIDS. Given that CB and CBJ do not record clauses, for these algorithms VSIDS corresponds to SLIS (Static Largest Individual Sum). SLIS is a heuristic that selects the literal that appears most frequently in the original clauses; in this case, the metrics are not dynamically changed during the search.

For the results in Table 2.4 we used the DLIS (Dynamic Largest Individual Sum of literals) heuristic (Marques-Silva 1999). The intuition is that CB(J) cooperates poorly with a simple heuristic such as SLIS. On the contrary, VSIDS cooperates effectively with CBJ+cr, because learning allows correcting early wrong variable orderings. For this reason, we decided to experiment DLIS, a more elaborated heuristic. The obtained results are significantly better than those obtained with SLIS (namely for CBJ), but are still far from being competitive with CBJ+cr results.

2.8 Summary

In this chapter we give a perspective on the evolution of SAT algorithms. SAT algorithms can be categorized as incomplete or complete algorithms. Local search and backtrack search are examples of incomplete and complete algorithms, respectively. Although incomplete algorithms are quite efficient, only complete algorithms are able to prove unsatisfiability. Hence, complete algorithms are preferable for solving hard real-world SAT instances, most of which are unsatisfiable.

The most popular complete algorithms in the history of SAT are the resolution-based Davis-Putnam procedure (DP) (Davis & Putnam 1960) and the backtrack search Davis-Logemann-Loveland procedure (DLL) (Davis, Logemann, & Loveland 1962). Broadly, the resolution rule in DP is replaced by the splitting rule in DLL. Even though resolution is a complete proof procedure for SAT, it is not competitive in practice, due to requiring exponential space in general.

The DLL procedure was further developed into different forms of non-chronological backtracking. Conflict-directed backjumping (Prosser 1993) is a sophisticated form of non-chronological backtracking, which can be enhanced with learning (Stallman & Sussman 1977; Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997). Indeed, non-chronological backtracking with learning is currently the most competitive form of backtracking. Experimental results give empirical evidence on the usefulness of using learning jointly with non-chronological backtracking on solving hard real-world SAT instances.

In addition, DLL-based algorithms include different branching heuristics, which define the variable to be selected by the splitting rule. Existing branching heuristics range from the most sophisticated to the most simple ones. Also, different search strategies are currently used to implement different organizations of the search process. Random restarts are an example of a very efficient search strategy.

Other approaches for solving satisfiability problems are particularly efficient for solving instances of specific problem domains. For example, BDD-based approaches are used for solving unsatisfiable hardware verification problems (Uribe & Stickel 1994).

3

Efficient Implementations

Implementation issues for SAT solvers include the design of suitable data structures for storing clauses, variables and literals. The implemented data structures dictate the way BCP and conflict analysis are implemented and have a significant impact on the run time performance of the SAT solver. Recent state-of-the-art SAT solvers are characterized by using very efficient data structures, intended to reduce the CPU time required per each node in the search tree. Traditional SAT data structures are accurate, meaning that is possible to know exactly the value of each literal in the clause. Examples of traditional data structures, also called adjacency lists data structures, can be found in GRASP (Marques-Silva & Sakallah 1996), relsat (Bayardo Jr. & Schrag 1997) and satz (Li & Anbulagan 1997). Conversely, most recent SAT data structures are not accurate, and therefore are called *lazy*. Examples of efficient lazy data structures include the head/tail lists used in Sato (Zhang 1997) and the watched literals used in Chaff (Moskewicz *et al.* 2001).

The main purpose of this chapter is twofold. First, to review existing SAT data structures. Second, to propose new data structures, that aim to be less lazy and so may be preferable for the next generation SAT solvers. Our description of SAT data structures is organized in two main categories: *adjacency lists* data structures and *lazy* data structures. Afterwards, we analyze optimizations that can be applied to most data structures, by special handling of small clauses. Also, we discuss the effect of lazy data structures in accurately predicting dynamic clause size (i.e. the number of unassigned literals in a clause). Experimental results are then provided for comparing the different data structures. Finally, we discuss the most recently proposed data structures, which have been introduced after our work has been developed.

3.1 Adjacency Lists Data Structures

Most backtrack search SAT algorithms represent clauses as lists of literals, and associate with each variable x a list of the clauses that contain a literal in x. Clearly, after assigning a variable x the clauses with literals in x are immediately aware of the assignment of x. The lists associated with each variable can be viewed as containing the clauses that are *adjacent* to that variable. In general, we use the term *adjacency lists* to refer to data structures in which each variable x contains a *complete* list of the clauses that contain a literal in x.

In the following sub-sections, different alternative implementations of adjacency lists are described. In each case, we are interested in being able to accurately and efficiently identify when clauses become satisfied, unsatisfied or unit.

3.1.1 Assigned Literal Hiding

One approach to identify satisfied, unsatisfied or unit clauses consists of extracting from the clause's list of literals all the references to unsatisfied and satisfied literals. These references are added to dedicated lists associated with each clause. As a result, satisfied clauses contain one or more literal references in the list of satisfied literals; unsatisfied clauses contain all literal references in the list of unsatisfied literals; finally, unit clauses contain one unassigned literal and all the other literal references in the list of unsatisfied literals. Algorithm 3.1 has the pseudo-code for these functions.

This data structure is illustrated in Figure 3.1. Whenever a literal is assigned, it is moved either to the satisfied or unsatisfied literals list. In the given example, the ternary clause is identified as unit when only one literal is still unassigned and the other two literals are unsatisfied. Observe that when the search backtracks the same operations are

Algorithm 3.1: Functions for clauses with assigned literal hiding

```
IS_SATISFIED_CLAUSE($\omega$)
(1)    return NUM_SATISFIED_LITERALS($\omega$) $> 0$

IS_UNSATISFIED_CLAUSE($\omega$)
(1)    return NUM_SATISFIED_LITERALS($\omega$) $== 0$ and
(2)     NUM_UNASSIGNED_LITERALS($\omega$) $== 0$

IS_UNIT_CLAUSE($\omega$)
(1)    return NUM_UNASSIGNED_LITERALS($\omega$) $== 1$ and
(2)     NUM_SATISFIED_LITERALS($\omega$) $== 0$
```

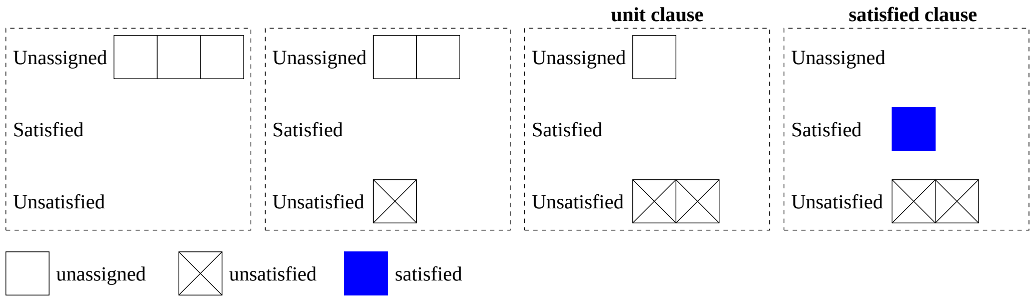


Figure 3.1: Operation of assigned literal hiding data structures

performed on the reverse order.

As will be shown by the experimental results in Section 3.6, this organization of adjacency lists data structure is never competitive with the other approaches.

3.1.2 The Counter-Based Approach

An alternative approach to keep track of unsatisfied, satisfied and unit clauses is to associate literal counters with each clause. These literal counters indicate how many literals are unsatisfied, satisfied and, indirectly, how many are still unassigned.

A clause is satisfied if the counter of satisfied literals is greater than one; is unsatisfied if the unsatisfied literal counter equals the number of literals; finally, it is unit if the unsatisfied literal counter equals the number of literals minus one, and there is still one unassigned literal. When a clause is declared unit, the list of literals is traversed to identify which literal needs to be assigned. Algorithm 3.2 has the pseudo-code for these functions.

The counter-based approach is illustrated in Figure 3.2. Whenever a literal is given a

Algorithm 3.2: Functions for clauses with counter-based approach

```
Is_Satisfied_Clause($\omega$)
(1)    return num_satisfied_literals($\omega$) $> 0$

Is_Unsatisfied_Clause($\omega$)
(1)    return num_unsatisfied_literals($\omega$) $==$ num_literals($\omega$)

Is_Unit_Clause($\omega$)
(1)    return num_satisfied_literals($\omega$) $== 0$ and
(2)     num_unsatisfied_literals($\omega$) $==$ num_literals($\omega$) - 1
```

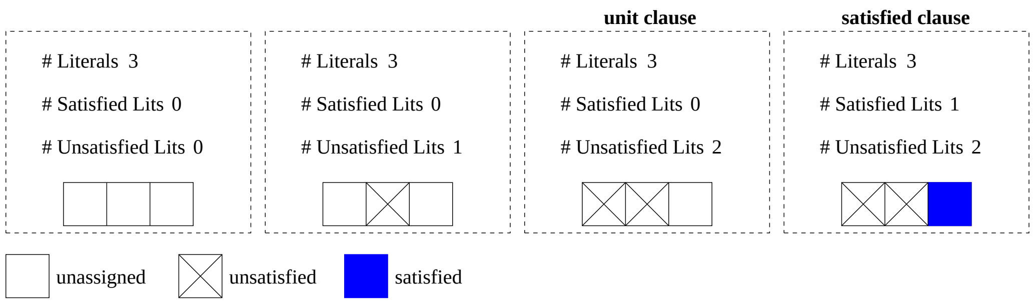


Figure 3.2: Operation of counter-based data structures

value, either the counters for satisfied or unsatisfied literals are updated, depending on the literal being assigned value 1 or 0, respectively. Observe that when the clause is identified as unit, the whole clause is traversed in order to find the remaining unassigned literal. This was not the case for the assigned literal hiding data structure, where identifying a clause as unit implies having *only one* literal on the unassigned literals list. Moreover, in the counter-based approach the counters have to be updated when the search backtracks.

3.1.3 Counter-Based with Satisfied Clause Hiding

A key drawback of using adjacency lists is that the lists of clauses associated with each variable can be large, and will grow as new clauses are recorded during the search process. Hence, each time a variable is assigned, a potentially large list of clauses needs to be traversed. Different approaches can be envisioned to overcome this drawback. For the counter-based approach of the previous section, one solution is to remove from the list of clauses of each variable *all* the clauses that are known to be satisfied. Hence, each time a clause ω becomes satisfied, ω is hidden from the list of clauses of all the variables with literals in ω. The technique of hiding satisfied clauses can be traced back to the work of O. Coudert in Scherzo (Coudert 1996) for the Binate Covering Problem. The motivation for hiding clauses is to reduce the amount of work required each time a variable x is assigned, since in this case only the unresolved clauses associated with x need to be analyzed.

Example 3.1 *Let us consider again formula φ (from Example 2.1) having three clauses ω_1, ω_2 and ω_3:*

$$\omega_1 = (x_1 \vee x_2),\ \omega_2 = (x_2 \vee \neg x_3),\ \omega_3 = (x_1 \vee x_2 \vee x_3).$$

Before starting the search, variable x_2 is associated with clauses ω_1, ω_2 and ω_3. Let us now suppose that variable x_1 is assigned value 1. Then, clauses ω_1 and ω_3 become satisfied and therefore these clauses are hidden from the list of clauses associated with x_2, thus remaining only ω_2 in this list.

3.1.4 Satisfied Clause and Assigned Literal Hiding

One final organization of adjacency lists is to utilize exactly the same data structures as the ones used by Scherzo (Coudert 1996). Not only satisfied clauses get hidden from clause lists in variables, but also unsatisfied literals get removed from literal lists in clauses.

Example 3.2 *Let us consider again formula φ from Example 3.1 but now suppose that variable x_1 is assigned value 0. Then literal x_1 is hidden from clauses ω_1 and ω_3. Hence, we get $\omega_1 = (x_2)$ and $\omega_3 = (x_2 \vee x_3)$.*

The utilization of clause and literal hiding techniques aims reducing the amount of work associated with assigning each variable. As will be shown by the experimental results in Section 3.6, clause and literal hiding techniques are not particularly efficient when compared with the simple counter-based approach described above. Moreover, lazy data structures, described in the next section, are by far more efficient.

3.2 Lazy Data Structures

As mentioned in the previous section, adjacency list-based data structures share a common problem: each variable x keeps references to a potentially large number of clauses, that often increases as the search proceeds. Clearly, this impacts negatively the amount of operations associated with assigning x. Moreover, it is often the case that most of x's clause references do not need to be analyzed when x is assigned, since most of the clauses do not become unit or unsatisfied. Observe that *lazily* declaring a clause to be satisfied does not affect the correctness of the algorithm.

Considering that only unsatisfiable and unit clauses must be identified, then is enough to have two references for each clause. (Although additional references may be required to guarantee clauses' consistency after backtracking.) These references never reference literals assigned value 0. Hence, such references are allowed to move along the clause: whenever a referenced literal is assigned value 0, the reference moves to another literal either assigned value 1 or assigned value `U` (i.e. unassigned). Algorithm 3.3 shows how the value and the position of these two references (REFA and REFB) are enough for declaring a clause to be satisfied, unsatisfied or unit. As already mention, a clause is lazily declared to be satisfied, meaning that some clauses being satisfied are not recognized as so. Again, this aspect does not affect the correctness of the algorithm.

Algorithm 3.3: Functions for clauses with lazy data structures

```
IS_SATISFIED_CLAUSE($\omega$)
(1)    return REFA($\omega$) == 1 or REFB($\omega$) == 1

IS_UNSATISFIED_CLAUSE($\omega$)
(1)    return REFA($\omega$) == 0 and
(2)     POSITION_REFA($\omega$) == POSITION_REFB($\omega$)

IS_UNIT_CLAUSE($\omega$)
(1)    return REFA($\omega$) == U and
(2)     POSITION_REFA($\omega$) == POSITION_REFB($\omega$)
```

In this section we analyze *lazy* data structures, which are characterized by each variable keeping a reduced set of clauses' references, for each of which the variable can be effectively used for declaring the clause as unit, as satisfied or as unsatisfied. The operation of these data structures is summarized in Figure 3.3. For each data structure, we illustrate literal assignment, unit clause identification and backtracking.

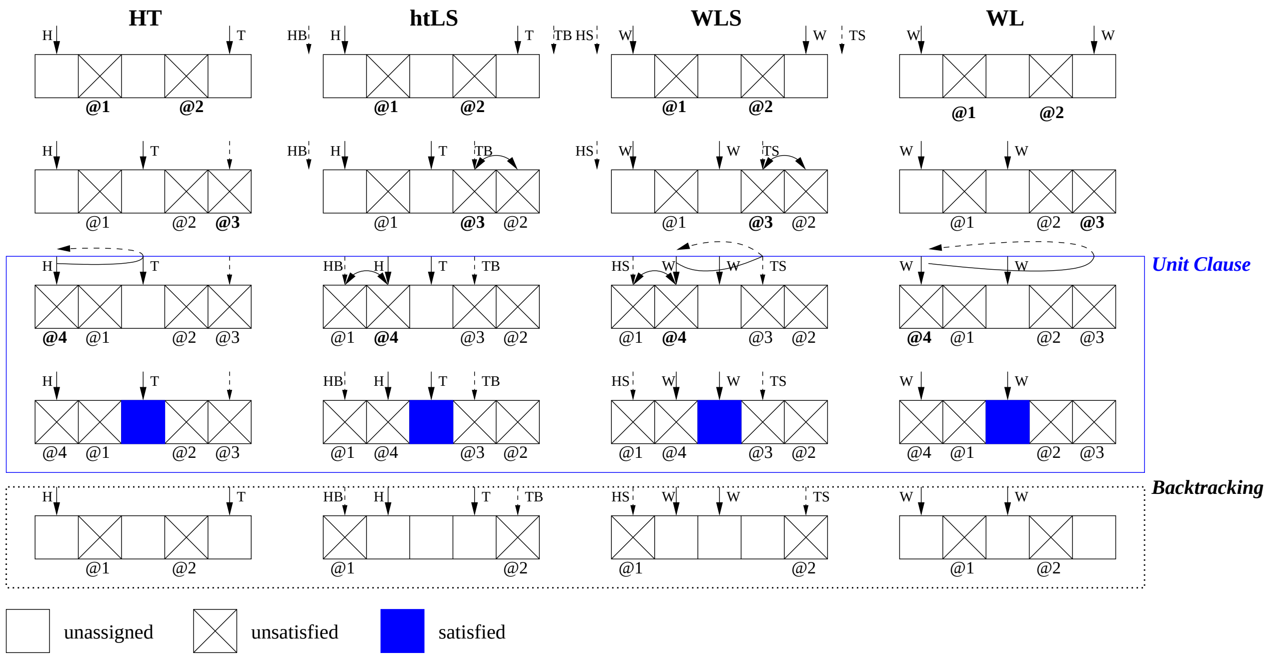


Figure 3.3: Operation of lazy data structures

3.2.1 Sato's Head/Tail Lists

The first lazy data structure proposed for SAT was the *Head/Tail* (H/T) data structure, originally used in the Sato SAT solver (Zhang 1997) and later described in (Zhang & Stickel 2000). As the name implies, this data structure associates two references with each clause, the *head* (H) and the *tail* (T) literal references (see Figure 3.3).

Initially the head reference points to the first literal, and the tail reference points to the last literal. Each time a literal pointed to by either the head or tail reference is assigned, a new unassigned literal is searched for. Both points move towards to the center of the clause. In case an unassigned literal is identified, it becomes the new head (or tail) reference, and a *new* reference is created and associated with the literal's variable. These references guarantee that H/T positions are correctly recovered when the search backtracks. In case a satisfied literal is identified, the clause is declared satisfied. In case no unassigned literal can be identified, and the other reference is reached, then the clause is declared unit, unsatisfied or satisfied, depending on the value of the literal pointed to by the other reference.

When the search process backtracks, the references that have become associated with the head and tail references can be discarded, and the previous head and tail references become activated (represented with a dashed arrow in Figure 3.3 for column HT). Observe that this requires in the worst-case associating with each clause a number of literal references in variables that equals the number of literals.

3.2.2 Chaff's Watched Literals

The more recent Chaff SAT solver (Moskewicz *et al.* 2001) proposed a new data structure, the Watched Literals (WL), that solves some of the problems posed by H/T lists. As with H/T lists, two references are associated with each clause. However, and in contrast with H/T lists, there is *no* order relation between the two references, allowing the references to move in any direction. The lack of *order* between the two references has the key advantage that no literal references need to be updated when backtracking takes place. In contrast, unit or unsatisfied clauses are identified only after traversing *all* the clauses' literals; a clear drawback. The identification of satisfied clauses is similar to H/T lists.

With respect to Figure 3.3, the most significant difference between H/T lists and watched literals occurs when the search process backtracks, in which case the references to the watched literals are not modified. Consequently, and in contrast with H/T lists, there is no need to keep additional references. This implies that for each clause the number of literal references that are associated with variables is kept *constant*.

3.2.3 Head/Tail Lists with Literal Sifting

The problems identified for H/T lists and Watched Literals can be solved with yet another data structure, H/T lists with literal sifting (htLS). This new data structure is similar to H/T lists, but it dynamically rearranges the list of literals, ordering the clause's assigned literals by increasing decision level. Assigned variables are sorted by non-decreasing decision level, starting from the first or last literal reference, and terminating at the most recently assigned literal references, just before the head reference and just after the tail reference. This sorting is achieved by sifting assigned literals as each is visited by the H and T literal references. The sifting is performed towards one of the ends of the literal list.

The solution based on literal sifting has several advantages:

- When a clause either becomes unit or unsatisfied, there is no need to traverse all the clause's literals to confirm this fact. Moreover, satisfied clauses are identified in the same way as for the other lazy data structures.
- As illustrated in Figure 3.3, only four literal references need to be associated with each clause. Besides the Head (H) and Tail (T) references, we also need the Head Back (HB) and Tail Back (TB) references. HB and TB references are kept precisely before the H and T references, respectively. Having four literal references contrasts with H/T lists, that in the worst-case need a number of references that equals the number of literals (even though watched literals just require two references).
- Literals that are assigned at low decision levels are visited only once, and then sifted out of the literal range identified by the H/T references, until the search backtracks to those low decision levels. Hence, literal references never cross over assigned literals, neither when the search is moving forward nor when the search is backtracking.

3.2.4 Watched Literals with Literal Sifting

One additional data structure consists of utilizing watched literals with literal sifting (WLS). This data structure applies literal sifting, but the references to unassigned literals are *watched*, in the sense that when backtracking takes place the literal references are not updated (see Figure 3.3). This data structure keeps two watched literals, and uses two additional references (HS and TS, meaning Head Sifting and Tail Sifting, respectively) for applying literal sifting and keeping assigned literals by decreasing order of decision level. Watched literals are managed as described earlier, and literal sifting is applied as

Table 3.1: Comparison of the data structures

data structures			AL	HT	htLS	WLS	WL
lazy data structure?			N	Y	Y	Y	Y
# literal references	min		L	2C	4C	4C	2C
	max		L	L	4C	4C	2C
# visited literals	when identifying unit/unsat cl^s	min	1	1	1	1	W-1
		max	1	W-1	W-1	W-1	W-1
	when backtracking		L_b	L_b	L_b	0	0

L = number of literals

C = number of clauses

W = number of literals in clause

L_b = number of literals to be unassigned when backtracking

proposed in the previous section.

Similarly to watched literals, the main advantage of the WLS data structure is the simplified backtracking process, whereas the main disadvantage is the requirement to visit all literals between the literal references HS and TS each time the clause is either unit or unsatisfied. Observe that it is easy to reduce the number of literal references to three: two for the watched literals and one for keeping the sifted literals. However, the overhead of literal sifting then becomes more significant.

3.3 A Comparison of the Data Structures

Besides describing the organization of each data structure, it is also interesting to characterize each one in terms of the memory requirements and computational effort. In Table 3.1, we provide a comparison of the data structures described above, where AL, HT, htLS, WLS and WL stand for Adjacency Lists with assigned literal hiding, Head/Tail lists, Head/Tail lists with Literal Sifting, Watched Literals with Literal Sifting and Watched Literals, respectively.

The table indicates which data structures are *lazy*, the minimum and maximum total number of literal references associated with all clauses, and also a broad indication of the work associated with keeping clause state when the search either moves forward (i.e.

implies assignments) or backward (i.e. backtracks).

Even though it is straightforward to prove the results shown, a careful analysis of the behavior of each data structure is enough to establishing these results. For example, when backtracking takes place, the WL data structure updates *no* literal references. Hence, the number of visited literal references for each conflict is 0.

All different data structures clearly exhibit advantages and disadvantages. With the experimental results given in section 3.6 we will try to identify which characteristics are the most important when building an efficient SAT solver.

3.4 Handling Special Cases: Binary/Ternary Clauses

As one final optimization to literal sifting, we propose the special handling of the clauses that are more common in problem instances: binary and ternary clauses. Both binary and ternary clauses can be identified as unit, satisfied or unsatisfied in constant time, thus eliminating the need for moving literal references around. Since the vast majority of the initial number of clauses for most real-world problem instances are either binary or ternary, the average CPU time required to handle each clause may be noticeably reduced. In this situation, the other data structures described in this chapter are solely applied to original large clauses and to clauses recorded during the search process, which are known for having a huge number of literals.

3.5 Do Lazy Data Structures Suffice?

As mentioned earlier, most state-of-the-art SAT solvers currently utilize lazy data structures. Even though these data structures suffice for backtrack search SAT solvers that solely utilize Boolean Constraint Propagation, the *laziness* of these data structures may pose some problems, in particular for algorithms that aim the integration of more advanced techniques for the identification of necessary assignments, namely restricted resolution, two-variable equivalence, and pattern-based clause inference, among other techniques (Groote & Warners 2000; Marques-Silva 2000; Brafman 2001;

Bacchus 2002b). For these techniques, it is essential to know which clauses are binary and/or ternary. As already mentioned, lazy data structures are not capable of keeping precise information about the set of binary and/or ternary clauses. Clearly, this can be done by associating additional literal references with each clause, and as a result by introducing additional overhead. Actually, the use of additional references has been referred in (Gelder 2002).

Consequently, if future SAT solvers choose to integrate advanced techniques for the identification of necessary assignments, they either forgo using lazy data structures, or they apply those techniques to a subset of the total number of binary/ternary clauses. One reasonable assumption is that lazy data structures will indeed be deemed essential, and that future SAT solvers will apply advanced techniques to a *lazy* set of binary/ternary clauses. In this situation, it becomes important to characterize the *laziness* of a lazy data structure in terms of the actual percentage of binary/ternary clauses it is capable of identifying. A data structure that is able to identify the largest percentage of binary/ternary clauses is clearly the best option for the implementation of advanced search techniques.

3.6 Experimental Results

This section evaluates the different SAT data structures described in the previous sections. Afterwards, we also evaluate the accuracy of lazy SAT data structures in estimating the number of satisfied, binary and ternary clauses. Again, we used the JQuest SAT framework, a Java implementation that was built to conduct unbiased experimental evaluations of SAT algorithms and techniques. Using JQuest in these experiments, we guarantee that for a given problem instance and for each data structure the *same* algorithmic organization and the *same* search tree are obtained. In other words, JQuest ensures that the number of decisions required for solving a problem instance is exactly the same regardless of the implemented data structures.

Even though Java yields a necessarily slower implementation, it is also plain that it allows fast prototyping of new algorithms. Moreover, well-devised Java implementations

Table 3.2: Example instances

Application Domain	Selected Instance	# Variables	#Clauses	Satisfiable?
Graph Colouring	flat175-81	525	1951	Y
	flat200-82	600	2237	Y
	sw100-13	500	3100	Y
	sw100-79	500	3100	Y
All-Interval Series	ais10	181	3151	Y
Bounded Model Checking	barrel5	1407	5383	N
	longmult6	2848	8853	N
	queueinvar18	2081	17368	N
Equivalence Checking	c5315_bug	5396	15004	Y
Pigeon Hole(Dimacs)	hole9	90	415	N
Inductive Inference(Dimacs)	ii32e5	522	11636	Y
Parity Learning(Dimacs)	par16-4-c	324	1292	Y
Blocks World	4blocksb	410	24758	Y
Bounded Model Checking(IBM)	bmc-ibm-3	14930	72106	Y
Logistics	facts7hh.13	4315	90646	Y
Planning	sat-bw_large.c	3016	50457	Y
	unsat-bw_large.c	2729	45368	N
Superscalar Processor Verification	dlx2_aa	490	2804	N
	dlx2_cc_a_bug07	1515	12808	Y
	dlx2_cc_a_bug17	4847	39184	Y
	dlx2_cc_a_bug59	4731	37950	Y
	2dlx_cc_mc_ex_bp_f2_bug004	4824	48233	Y
	2dlx_cc_mc_ex_bp_f2_bug006	4824	48215	Y
Circuit Testing (Dimacs)	bf0432-079	1044	3685	N
	ssa2670-141	4843	2315	N

can be used as the blueprint for faster C/C++ implementations. In the case of JQuest, all the proven strategies and techniques for SAT have been implemented: clause recording, non-chronological backtracking, search restarts and variable selection heuristics.

For the results shown below a P-III@833 MHz Linux Red Hat 6.1 machine with 1 GByte of physical memory was used. The Java Virtual Machine used was SUN's HotSpot JVM for JDK1.3. Moreover, results are given for representative instances that were selected from several classes of instances available from SATLIB web site `http://www.satlib.org/`. Table 3.2 characterizes each of the selected instances, namely giving the application domain, the number of variables and clauses and the information about whether the instance is satisfiable.

Table 3.3: Results for the time per decision (tpd, in msec)

			Time ratio wrt minimum tpd									
Instance	#decs	tpd	ALl	ALcb	ALcbsr	ALlsr	HT	WL	htLS	htLS23	wLS	wLS23
flat175-81	1001	3.33	1.99	1.10	2.06	1.88	1.11	1.02	1.09	**1.00**	1.22	1.01
flat200-82	29308	2.13	7.28	3.17	1.78	1.60	1.68	1.23	1.06	**1.00**	1.26	1.13
sw100-13	1816	0.61	1.69	**1.00**	1.84	1.59	1.18	1.03	1.20	1.15	1.28	1.15
sw100-79	1421	0.77	1.71	**1.00**	2.16	1.90	1.21	1.21	1.23	1.22	1.40	1.18
ais10	6380	3.91	8.39	3.39	1.47	1.27	1.88	1.39	**1.00**	1.02	1.21	1.13
barrel5	5940	8.12	3.16	1.62	1.85	1.75	1.35	1.06	1.06	1.02	1.14	**1.00**
longmult6	4807	11.53	6.80	3.03	1.60	1.51	1.36	1.13	1.09	**1.00**	1.23	1.08
queueinvar18	8680	3.17	4.46	2.10	1.46	1.31	1.27	1.23	1.06	**1.00**	1.15	1.03
c5315_bug	28621	1.51	1.58	1.07	1.81	1.77	1.17	1.04	1.16	1.03	1.21	**1.00**
hole9	6072	5.16	7.51	3.00	2.06	1.62	1.45	1.04	1.03	1.03	1.04	**1.00**
ii32e5	1466	1.95	2.72	1.30	3.25	3.67	1.05	1.09	1.33	1.28	1.21	**1.00**
par16-4-c	6167	5.30	7.90	3.44	1.33	1.21	1.80	1.22	1.08	**1.00**	1.20	1.03
4blocksb	6803	15.37	6.34	2.51	2.13	1.73	1.24	1.29	**1.00**	1.17	1.14	1.16
bmc-ibm-3	2559	16.15	1.84	1.09	2.25	2.13	1.21	1.05	1.18	1.07	1.21	**1.00**
facts7hh.13	2241	6.70	2.71	1.36	3.02	2.71	1.42	1.46	1.14	1.03	1.36	**1.00**
sat-bw_large.c	10020	37.97	5.24	2.39	2.55	2.38	1.41	1.25	1.10	**1.00**	1.26	1.01
unsat-bw_large.c	3280	24.09	3.03	1.50	2.62	2.46	1.39	1.31	1.13	1.02	1.30	**1.00**
dlx2_aa	10292	1.02	5.04	2.22	1.97	1.66	1.55	**1.00**	1.04	1.02	1.09	1.01
dlx2_cc_bug07	10314	2.54	4.57	2.00	1.98	1.72	1.25	1.03	1.15	**1.00**	1.17	1.05
dlx2_cc_bug17	7681	2.74	2.55	1.31	1.93	1.73	1.30	1.13	1.09	1.03	1.13	**1.00**
dlx2_cc_bug59	2588	1.87	2.27	1.20	2.03	1.89	1.22	1.13	1.12	1.07	1.18	**1.00**
2dlx_cc_mc...bug004	18481	1.23	2.51	1.30	2.00	1.77	1.27	1.14	1.09	1.03	1.13	**1.00**
2dlx_cc_mc...bug006	29173	1.91	3.33	1.61	2.05	1.77	1.36	1.13	1.09	1.02	1.12	**1.00**
bf0432-079	1038	2.23	1.67	1.04	2.01	1.86	1.16	**1.00**	1.13	1.05	1.18	1.03
ssa2670-141	674	1.31	1.28	**1.00**	1.70	1.57	1.22	1.06	1.22	1.17	1.27	1.12

3.6.1 Lazy vs Non-Lazy Data Structures

In order to compare the different data structures, the following algorithm organization of JQuest is used:

- The VSIDS (Moskewicz *et al.* 2001) (Variable State Independent Decaying Sum) heuristic is used for all data structures. Our implementation of the VSIDS heuristic closely follows the one proposed in Chaff.
- Identification of necessary assignments solely uses Boolean constraint propagation. We should note that, in order to guarantee that the same search tree is visited, the unit clauses are handled in a *fixed predefined* order.
- Conflict analysis is implemented as in GRASP (Marques-Silva & Sakallah 1996). However, only a single clause is recorded (by stopping at the first Unique Implication Point (UIP) as suggested by the authors of Chaff (Moskewicz *et al.* 2001)). Moreover, *no* clauses are ever deleted.
- Search restarts are not applied.

The results of comparing the different data structures are shown in Table 3.3. Observe that in all cases the problem instances chosen are solved with several thousand decisions, usually taking a few tens of seconds. Hence, the instances chosen are significantly hard, but can be solved without sophisticated search strategies. Clearly, using sophisticated search strategies would make even more difficult guaranteeing the same search tree for all data structures considered.

Table 3.3 gives results for all the data structures described above:

ALl	adjacency lists with assigned literal hiding
ALcb	counter-based adjacency lists
ALcbsr	adjacency lists with satisfied clause removal/hiding
ALlsr	adjacency lists with assigned literal and satisfied clause removal/hiding
HT	H/T lists
WL	watched literals
htLS	H/T lists with literal sifting
htLS23	H/T lists with literal sifting and special handling of binary and ternary clauses
wLS	watched literals lists with literal sifting
wLS23	watched literals lists with literal sifting and special handling of binary and ternary clauses

The table of results includes the (constant) number of decisions required to solve each problem instance, and the minimum time-per-decision (tpd) over all data structures (in milliseconds, msec). The results for all problem instances are shown as a ratio with respect to the minimum time-per-decision for each problem instance. For example, instance *bmc-ibm-3* when solved with wLS23 data structure requires 16.15 msec per decision (total CPU time is given by 2559 x 16.15 msec = 41.33 sec), and when solved with HT data structure requires 1.21 x 16.15 msec = 19.54 msec per decision (total CPU time is given by 2559 x 19.54 msec = 50 sec).

From the table of results, several conclusions can be drawn. Clearly, lazy data structures are (in most cases) significantly more efficient than data structures based on adjacency lists. Regarding the data structures based on adjacency lists, the utilization of satisfied clause and assigned literal hiding does not pay off. For the lazy data structures, H/T lists are in general significantly slower than either watched literals or H/T lists with literal sifting. Finally, H/T and WL lists with literal sifting and special handling of binary and ternary clauses tend to be somewhat more efficient than watched literals. This results in part from the literal sifting technique, that allows literals assigned at low decision levels not to be repeatedly analyzed during the search process. In addition, special handling of small clauses may also impact the search time.

Despite the previous results that indicate H/T and WL lists with literal sifting and with special handling of binary and ternary clauses to be in general faster than the watched literals data structure, one may expect the small performance difference between the two data structures to be eliminated by careful C/C++ implementations. This is justified by the expected better cache behavior of watched literals (Moskewicz *et al.* 2001).

3.6.2 Limitations of Lazy Data Structures

As mentioned in Section 3.2, lazy data structures do *not* maintain all the information that may be required for implementing advanced SAT techniques, namely two-variable equivalence conditions (from pairs of binary clauses), restricted resolution (between binary and ternary clauses), and pattern-based clause inference conditions (also using binary and ternary clauses). Even though some of these techniques are often used as a preprocessing step by SAT solvers, their application during the search phase has been proposed in the past (Groote & Warners 2000; Marques-Silva 2000; Bacchus 2002b). The objective of this section is thus to measure the laziness of lazy data structures during the search process. The more lazy a (lazy) data structure is, the less suitable it is for implementing (lazy) advanced reasoning techniques during the search process. As we show below, no lazy data structure provides completely accurate information regarding the number of binary, ternary or satisfied clauses. However, some lazy data structures are significantly more accurate than others. Hence, if some form of *lazy* implementation of advanced SAT techniques is to be used during the search process, some lazy data structures are

Table 3.4: Results for the accuracy of recorded clause identification

	satisfied clauses					binary clauses				ternary clauses			
Instance	AL	HT	WL	wLS	htLS	AL	wLS	HT	htLS	AL	wLS	HT	htLS
flat175-81	291874	73%	80%	62%	89%	9978	10%	19%	93%	11166	3%	37%	86%
flat200-82	148284026	96%	98%	85%	99%	438356	20%	29%	85%	613244	9%	14%	75%
sw100-13	424018	95%	96%	91%	98%	7185	36%	13%	91%	8616	2%	0%	85%
sw100-79	259450	95%	96%	94%	98%	3062	26%	10%	79%	4780	5%	2%	73%
ais10	18519748	98%	98%	83%	99%	43337	31%	20%	75%	74899	10%	9%	68%
barrel5	9005238	90%	95%	73%	99%	251321	1%	78%	98%	168820	1%	50%	92%
longmult6	9892419	88%	93%	70%	95%	109446	8%	75%	96%	45805	9%	8%	77%
queueinvar18	11318602	96%	97%	90%	98%	3927	8%	51%	90%	11486	1%	8%	74%
c5315_bug	24701766	90%	92%	86%	96%	628304	3%	65%	96%	539811	1%	50%	90%
hole9	14775953	84%	93%	53%	98%	22258	10%	17%	72%	62987	4%	1%	64%
ii32e5	128713	99%	99%	99%	100%	1413	4%	14%	70%	1256	0%	4%	50%
par16-4-c	18326757	97%	99%	66%	100%	9454	19%	38%	95%	12131	7%	37%	90%
4blocksb	15442183	92%	93%	81%	96%	191817	12%	48%	89%	196534	7%	16%	72%
bmc-ibm-3	778745	82%	88%	73%	94%	136082	2%	89%	98%	31120	3%	18%	89%
facts7hh.13	493070	89%	94%	86%	96%	16055	8%	62%	90%	14160	3%	52%	84%
bw_large.c	32784773	89%	93%	65%	97%	275761	12%	36%	86%	284054	6%	24%	71%
bw_large.c	2713365	87%	90%	70%	96%	48475	14%	34%	91%	46996	7%	23%	82%
dlx2_aa	14905254	83%	89%	52%	93%	105184	20%	10%	89%	116638	5%	15%	58%
dlx2_cc_bug07	16664430	66%	85%	78%	91%	157500	16%	14%	86%	131612	6%	6%	66%
dlx2_cc_bug17	6359386	95%	96%	86%	98%	44562	13%	10%	87%	49437	8%	2%	75%
dlx2_cc_bug59	586538	94%	93%	90%	95%	6450	13%	3%	74%	13002	5%	1%	55%
2dlx_cc_mc...bug004	8587704	90%	93%	86%	97%	147713	11%	10%	92%	137653	7%	15%	84%
2dlx_cc_mc...bug006	35417574	88%	93%	72%	97%	318105	12%	13%	93%	271931	6%	12%	81%
bf0432-079	200114	89%	92%	79%	98%	7423	4%	23%	90%	6702	2%	26%	91%
ssa2670-141	57588	93%	92%	87%	96%	1595	11%	13%	88%	1646	3%	4%	90%

significantly more adequate than others.

We start by observing that the watched literals data structure is unable to dynamically identify binary and ternary clauses, since there is no order relation between the two references used. Identifying binary and ternary clauses would involve maintaining additional information w.r.t. what is required by the watched literals data structure. Observe that the utilization of two references only guarantees the identification of unit clauses. The lack of order among the two references prevents the identification of binary and ternary clauses. In order to identify all or some of the binary/ternary clauses, either the two references respect some order relation, or more references need to be used.

Table 3.4 includes results measuring the accuracy of each lazy data structure in identifying satisfied, binary and ternary clauses among recorded clauses. The reference values considered are given by the values obtained with adjacency lists data structures, which are the actual exact values. Observe that, as mentioned above, the watched literals data structure can only be used for identifying satisfied clauses. From the results shown, we can conclude that H/T lists with literal sifting provide by far the most accurate estimates of the number of satisfied, binary and ternary clauses, when compared with other lazy data structures. In addition, for satisfied and binary clauses, the measured accuracy is often close to the maximum possible value, whereas for ternary clauses the accuracy values tend to be somewhat lower.

3.7 Recent Advances

The Chaff SAT solver has definitely introduced a new paradigm shift in the history of SAT solvers. Currently, it is generally accepted that a competitive SAT solver must have a very fast implementation. Inspired in Chaff, Jerusat (Nadel 2002) and siege (Ryan 2004) SAT solvers have made significant efforts to build competitive data structures, introducing even more efficient implementations.

Jerusat introduces a new data structure named WLCC (Watched List with Conflicts Counter). This data structure combines the advantages of watched literals with the advantages of literal sifting. The watched literals data structure guarantees having only two references per clause, as well as no need to visit a clause during unassignment. By using data structures with literal sifting, there is a reduction on the number of literals visited during assignment. In addition, the WLCC data structure enables reducing the number of visits to satisfied clauses and allows visiting fewer literals in recorded clauses, by keeping a counter for each assigned variable with the number of conflicts by the time the variable was assigned. This information is used to find out which literals in the clause have been sifted.

The siege SAT solver includes a careful handling of binary clauses. Observe that most available instances have a significant number of binary clauses, which means that an improvement in the implementation of the data structures for binary clauses has a significant impact on the overall performance. The main idea is that having two watched literals does not pay off for binary clauses because all literals have to be watched. Obviously, the effort for finding a non-watched literal is completely unnecessary. The alternative is to maintain a specialized representation for binary clauses, where each literal is associated with a list of literals. For each literal, such list contains the literals that share binary clauses with it. Whenever a literal becomes false, all literals in the corresponding list are implied. Siege's author also suggests a specialized data structure for ternary clauses. Moreover, larger clauses processing is improved by using boundary sentinels, which reduces the computational effort of the search loops when trying to find an unassigned non-watched literal.

3.8 Summary

This chapter surveys existing data structures for backtrack search SAT algorithms and proposes new data structures.

Existing data structures can be divided in two different categories: the traditional *adjacency lists* data structures and the most recent *lazy* data structures. For each clause, the adjacency lists data structures allow to know exactly the number of literals assigned value 1, assigned value 0 or unassigned, and therefore whether a clause is satisfied, unsatisfied or unit. On the other hand, the lazy data structures only allow to know whether a clause is unit or unsatisfied. Clearly, being able to determine whether a clause is unit or unsatisfied suffices to guarantee that unit propagation and conflict detection are applied.

The lazy data structures are by far the most competitive when using backtrack search algorithms *with clause recording*. For the watched literals data structure, which is the most competitive, the main drawback is in the number of operations required to confirm that a clause is unit. Hence, we propose new data structures that are based on literal sifting and therefore do not require so many operations to confirm that a clause is unit. Moreover, we suggest using dedicated data structures for binary and ternary clauses, based on the traditional data structures. Lazy data structures are particularly useful for representing large (recorded) clauses.

The experimental results compare all the different data structures, being a first step on evaluating how advanced SAT techniques perform with lazy data structures. The obtained results indicate that the new data structures, based on literal sifting, may be preferable for the next generation SAT solvers. This conclusion results from these new data structures being in general faster, but mostly due to coping better with the laziness of recent (lazy) data structures. Clearly, this depends on the accuracy of each data structure to identify binary and ternary clauses. As a result, data structures that are unable to gather the information required by advanced SAT techniques may be inadequate for the next generation state-of-the-art SAT solvers.

4

Probing-Based Preprocessing Techniques

Preprocessing techniques are often used for simplifying hard instances of propositional satisfiability. Indeed, in the last years there have been significant contributions in terms of applying different formula manipulation techniques during preprocessing. These techniques can in some cases yield competitive approaches (Gelder & Tsuji 1993; Groote & Warners 2000; Li 2000; Berre 2001; Brafman 2001; Bacchus 2002a; Novikov 2003).

Preprocessing can be used for reducing the number of variables and for drastically modifying the set of clauses, either by eliminating irrelevant clauses or by inferring new clauses. It is generally accepted that the ability to reduce either the number of variables or clauses in instances of SAT impacts the expected computational effort of solving a given instance. This ability can actually be essential for specific and hard classes of instances. Interestingly, the ability to infer *new* clauses *may* also impact the expected computational effort of SAT solvers (Lynce & Marques-Silva 2001). Observe that these new clauses can be useful for reducing the number of variables (and consequently the number of clauses).

This chapter proposes the utilization of *probing-based techniques* for manipulating propositional satisfiability formulas. Probing allows the formulation of hypothetical scenarios, obtained by assigning a value to a variable, and then applying unit propagation. Furthermore, probing-based techniques can build upon a *very simple* idea: a table of *trig-*

gering assignments, which registers the result of applying probing to every variable in the propositional formula.

The new probing-based approach provides a generic framework for applying different SAT preprocessing techniques by establishing reasoning conditions on the entries of the table of assignments. Reasoning conditions allow to implement most existing formula manipulation techniques, as well as new formula manipulating techniques. Interestingly, and to our best knowledge, this new framework represents the first approach to *jointly* apply variable and clause probing.

This chapter is organized as follows. The next section gives the preliminaries, followed by motivating examples using a table of assignments, which records the probing results. Afterwards, we establish reasoning conditions for identifying necessary assignments and inferring new clauses. In Section 4.4, we present *ProbIt*: a probing-based preprocessor for propositional satisfiability. Next, experimental results are presented and analyzed. Finally, Section 4.6 overviews related work and relates existing formula manipulation techniques with the proposed reasoning conditions.

4.1 Preliminaries

In what follows we analyze conditions relating sets of assignments. Given an assignment α [1] and a formula φ, the set of assignments resulting of applying Boolean Constraint Propagation (the iterative appliance of unit propagation) to φ given α is denoted by $\text{BCP}(\varphi, \alpha)$. Obviously, φ is updated with such new assignments. When clear from the context, we use the notation $\text{BCP}(\alpha)$, and the existence of the CNF formula φ is implicit. Without assignment, $\text{BCP}(\varphi)$ denotes plain unit propagation to formula φ, given the existence of unit clauses. Assignment α is referred to as the *triggering assignment* of the assignments in $\text{BCP}(\alpha)$. We may also use the notation $\text{BCP}(A)$ to denote the result of applying unit propagation as the result of *all* assignments in the set of assignments A.

Reasoning conditions are analyzed based on a tabular representation of triggering as-

[1] An assignment α is a pair $\langle l, v\rangle$ that denotes assigning value v to literal l.

Algorithm 4.1: Reasoning conditions under Boolean constraint propagation

```
Apply Reasoning Conditions($\varphi, \tau$)
(1)   if apply_BCP($\varphi$) yields a conflict
(2)      return UNSATISFIABLE
(3)   foreach variable $x_i$ such that $x_i$ is unassigned
(4)      assignments = apply_BCP($\varphi$,$\langle x_i, 0\rangle$)
(5)      store assignments in table of assignments $\tau$
(6)      undo assignments
(7)      assignments = apply_BCP($\varphi$, $\langle x_i, 1\rangle$)
(8)      store assignments in table of assignments $\tau$
(9)      undo assignments
(10)  Use table of assignments $\tau$ to apply Reasoning Conditions
(11)  return $\varphi$
```

signments, i.e. a *table of assignments*, where each row represents a triggering assignment, and each column represents a possible implied assignment. In practice the table of assignments is represented as a sparse matrix and so the memory requirements are never significant. In this table, each 1-valued entry (α_t, α_i) denotes an implied assignment α_i given a triggering assignment α_t. Hence the 1-valued entries of a given row α_t denote the elements of set $\text{BCP}(\alpha_t)$.

The reasoning conditions can be organized in an algorithm that can be used as a preprocessor algorithm. We refer to this algorithm as *reasoning conditions under Boolean constraint propagation*. Algorithm 4.1 gives the pseudo-code for the preprocessing algorithm. For this algorithm, it is implicit that if BCP yields a conflict for a triggering assignment $\langle x_t, v_t\rangle$, then the assignment $\langle x_t, \neg v_t\rangle$ is implied (this procedure corresponds to the well-known *failed-literal rule* (Crawford & Auton 1993)).

4.2 Motivating Examples

This section analyzes a few examples, that motivate the techniques upon which our framework is based, and which allow the identification of necessary assignments and the inference of new clauses.

Example 4.1 *Consider a CNF formula φ defined on a set of variables $X = \{a, b, c, d, e\}$ and having the following clauses:*

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.1: Table of assignments

$$\omega_1 = (a \vee b) \qquad \omega_5 = (\neg a \vee \neg c \vee d)$$

$$\omega_2 = (a \vee \neg b \vee d) \qquad \omega_6 = (\neg a \vee \neg d \vee e)$$

$$\omega_3 = (a \vee \neg c \vee e) \qquad \omega_7 = (c \vee e)$$

$$\omega_4 = (\neg a \vee c)$$

This formula has the table of assignments shown in Figure 4.1. Each line in the table corresponds to the result of applying BCP, given a triggering assignment. For example, given the assignment a=0 (also denoted by $\langle a, 0\rangle$), we can conclude from the table that $\text{BCP}(\langle a, 0\rangle) = \{\langle a, 0\rangle, \langle b, 1\rangle, \langle d, 1\rangle\}$.

4.2.1 Necessary Assignments

The identification of necessary assignments plays a key role in SAT algorithms, where it can be viewed as a form of reduction of the domain of each variable. In SAT algorithms, the most commonly used procedure for identifying necessary assignments is BCP, which identifies necessary assignments due to the unit clause rule in linear time on the number of literals in the CNF formula. However, BCP does not identify all necessary assignments

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.2: BCP($\langle a, 0\rangle$) $\cap$ BCP($\langle a, 1\rangle$) $= \langle d, 1\rangle$

given a set of variable assignments and a CNF formula. Consider for example a formula having two clauses $(x_1 \vee \neg x_2)$ and $(x_1 \vee x_2)$. For any assignment to variable x_2, variable x_1 must be assigned value 1 for preventing a conflict. This conclusion could be easily reached by inferring the unit clause (x_1) as a result of applying resolution between the two clauses. Nevertheless, BCP applied to this CNF formula would not produce this straightforward conclusion.

Let us now consider Figure 4.1, in order to identify necessary assignments that BCP is not able to identify. We will start by checking necessary assignments conditions for *formula satisfiability.*

Example 4.2 *From Figure 4.1, observe that for the two possible assignments to variable a, we always obtain the implied assignment $\langle d, 1\rangle$. Since variable a must assume one of the two possible assignments, then the assignment $\langle d, 1\rangle$ is deemed necessary. This conclusion is represented as* BCP($\langle a, 0\rangle$) $\cap$ BCP($\langle a, 1\rangle$) $= \langle d, 1\rangle$. *This condition is named* variable probing. *Figure 4.2 illustrates this result.*

Example 4.3 *The same conclusion from Example 4.2 could be achieved by considering clause $\omega_2 = (a \vee \neg b \vee d)$. Any assignment that satisfies the formula must also satisfy this clause, and so at least one of the variable assignments that satisfies the clause must*

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.3: $\text{BCP}(\langle a, 1\rangle) \cap \text{BCP}(\langle b, 0\rangle) \cap \text{BCP}(\langle d, 1\rangle) = \langle d, 1\rangle$

hold. Given that in this example the three assignments that satisfy the clause also imply the assignment $\langle d, 1\rangle$, then this assignment is part of any *assignment that satisfies the CNF formula, and so it is a necessary assignment. This conclusion is represented as* $\text{BCP}(\langle a, 1\rangle) \cap \text{BCP}(\langle b, 0\rangle) \cap \text{BCP}(\langle d, 1\rangle) = \langle d, 1\rangle$*. This condition is named* clause probing*. Figure 4.3 illustrates this result.*

The two previous examples concern necessary assignments conditions for *formula satisfiability*, based on variable and clause probing. Next we address necessary assignments conditions for preventing *formula unsatisfiability.*

Example 4.4 *Let us now consider Figure 4.4. First of all, note that the triggering assignment $\langle e, 0\rangle$ implies both $\langle d, 0\rangle$ and $\langle d, 1\rangle$, and hence a conflict is necessarily declared. As a result, the assignment $\langle e, 1\rangle$ is deemed necessary. This condition corresponds to the failed-literal rule.*

Example 4.5 *Let us know consider an alternative approach for assignment $\langle e, 1\rangle$ based on clauses, instead of variables. Another explanation for the same assignment comes from considering clause $\omega_6 = (\neg a \vee \neg d \vee e)$. The assignment $\langle e, 0\rangle$ makes this clause unsatisfied. Hence, a conflict is declared, and the assignment $\langle e, 1\rangle$ is deemed necessary. This result is illustrated in Figure 4.5.*

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.4: $\langle e, 0\rangle$ implies both $\langle d, 0\rangle$ and $\langle d, 1\rangle$

4.2.2 Inferred Clauses

Besides the identification of necessary assignments, the table of assignments can also be used for inferring new clauses. These new clauses not only constrain even more the given formula but also potentially remove variables in the formula, either by inferring unit clauses or allowing to identify equivalent literals.

Example 4.6 *Let us consider the triggering assignment* $\langle a, 1\rangle$ *and the respective implied assignment* $\langle e, 1\rangle$ *(see Figure 4.6). Formally,* $a \rightarrow e$*. Hence, the clause* $(\neg a \vee e)$ *can be inferred.*

Clearly, for each entry in the table of assignments a new binary clause can be created. However, such procedure would allow to infer a huge number of clauses. In practice, our goal is to be selective about which entries to utilize for inferring new clauses.

Example 4.7 *Observe in Figure 4.7 that the triggering assignments* $\langle a, 0\rangle$ *and* $\langle a, 1\rangle$ *imply the assignments* $\langle b, 1\rangle$ *and* $\langle c, 1\rangle$*, respectively (besides other triggering assignments). Since a must be subject to one of the two possible assignments, then one of the assignments in* $\{\langle b, 1\rangle, \langle c, 1\rangle\}$ *must also hold, and so the clause* $(b \vee c)$ *can be inferred.*

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.5: $\langle e, 0\rangle$ makes clause $\omega_6 = (\neg a \vee \neg d \vee e)$ unsatisfied

Example 4.8 *Consider clause $\omega_4 = (\neg a \vee c)$. Each assignment that satisfies clause ω_4 either implies the set of assignments $\{\langle a, 0\rangle, \langle b, 1\rangle, \langle d, 1\rangle\}$ or $\{\langle c, 1\rangle\}$ (see Figure 4.8). Hence, because at least one of the assignments that satisfies ω_4 must hold, the clauses $(b \vee c)$ and $(d \vee c)$ can be inferred. (Clause $(b \vee c)$ could also be inferred, although this clause already exists in the formula.)*

The previous examples illustrate how to infer clauses from *formula satisfiability* requirements. Next, we illustrate the inference of clauses from necessary conditions for preventing *formula unsatisfiability.*

Example 4.9 *First, observe in Figure 4.9 that the assignments in $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ imply the assignments in $\{\langle a, 1\rangle, \langle c, 1\rangle, \langle d, 0\rangle, \langle d, 1\rangle, \langle e, 1\rangle\}$, that denote an inconsistent assignment due to variable d. Hence, the assignments $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ must not hold simultaneously, and so the clause $(\neg a \vee d)$ can be inferred.*

Example 4.10 *Alternatively, observe that the set of assignments $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ unsatisfy clause $\omega_5 = (\neg a \vee \neg c \vee d)$ (see Figure 4.10). As a result, the assignments $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ must not hold simultaneously, and so the clause $(\neg a \vee d)$ can be inferred.*

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.6: Inferring clause $(\neg a \vee e)$

4.3 Reasoning with Probing-Based Conditions

The examples of the previous section illustrate the forms of reasoning that can be performed given information regarding the assignments implied by each triggering assignment. These forms of reasoning include identification of necessary assignments and inference of new clauses, both depending on formula satisfiability and formula unsatisfiability conditions.

In this section we formalize different conditions for identifying necessary assignments and for inferring new clauses. All proposed reasoning conditions result from analyzing the consequences of assignments made to variables and of propagating those assignments with BCP. Moreover, all the established reasoning conditions can be explained as a sequence of resolution steps. Nevertheless, we will give more intuitive explanations whenever they are clear enough to justify a reasoning condition.

4.3.1 Satisfiability-Based Necessary Assignments

The purpose of this section is to describe the identification of necessary assignments based on formula satisfiability conditions. The first condition identifies common implied assignments given the two possible triggering assignments that can be assigned to a vari-

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.7: Inferring clause $(b \vee c)$ (II)

able. This technique is often called *variable probing* and is illustrated in Example 4.2.

Proposition 4.3.1 *Given a CNF formula φ, for any variable x of the formula, the assignments defined by* $\text{BCP}(\langle x, 0\rangle) \cap \text{BCP}(\langle x, 1\rangle)$ *are necessary assignments.*

Proof. Any complete set of assignments to the variables that satisfies the CNF formula must assign either value 0 or 1 to each variable x. If for both assignments to x, some other variable y is implied to the same value v, then the assignment $\langle y, v\rangle$ is deemed necessary. ■

The second condition identifies common implied assignments given required conditions for satisfying each clause. This technique is often called *clause probing* and is illustrated in Example 4.3.

Proposition 4.3.2 *Given a CNF formula φ, for any clause ω of the formula, the assignments defined by* $\bigcap_{l\in\omega} \text{BCP}(\langle l, 1\rangle)$ *are necessary assignments.*

Proof. Any complete set of assignments that satisfies the CNF formula must satisfy all clauses. Hence, assignments that are common to all assignments that satisfy a given clause must be deemed necessary assignments. ■

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.8: Inferring clause $(b \vee c)$ (I)

Interestingly, Proposition 4.3.1 can be seen as a special case of Proposition 4.3.2 if we consider that any formula contains all binary tautological clauses (e.g. $(a \vee \neg a)$).

4.3.2 Unsatisfiability-Based Necessary Assignments

We now proceed describing the identification of necessary assignments based on formula unsatisfiability conditions.

Proposition 4.3.3 *Given a CNF formula φ, if* BCP$(\langle x, v\rangle)$ *yields a conflict, then the assignment $\langle x, \neg v\rangle$ is deemed necessary.*

Proof. Any complete set of assignments to the variables that satisfies the CNF formula must assign either value 0 or 1 to each variable x. Hence, if one of the assignments yields a conflict, then the other assignment is deemed necessary. ■

The previous proposition includes the conditions regarding both the identification of inconsistent assignments to a variable and the identification of unsatisfied clauses. Observe that most BCP algorithms do not distinguish between these two situations, being a conflict declared in both cases.

This condition corresponds to the failed-literal rule and is illustrated in both Examples 4.4 and 4.4. The first corresponds to the identification of inconsistent assignments to

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.9: Inferring clause $(\neg a \vee d)$ (II)

a variable and the latter corresponds to the identification of unsatisfied clauses.

4.3.3 Implication-Based Inferred Clauses

As illustrated earlier, probing can also be used for inferring new clauses. One simple approach for inferring new clauses is to use each entry in the table of assignments (see Example 4.6).

Proposition 4.3.4 *Given a CNF formula φ, if $\langle l_2, 1\rangle \in \mathrm{BCP}(\langle l_1, 1\rangle)$, then the clause $(\neg l_1 \vee l_2)$ is an implicate of φ.*

Proof. Let us refer to Ω as the ordered set of clauses involved in the sequence of implications that allowed to conclude that $\langle l_2, 1\rangle \in \mathrm{BCP}(\langle l_1, 1\rangle)$. Clearly, l_1 must be in the first clause in Ω, and l_2 must be in the last clause in Ω. Hence, if we apply a sequence of resolution steps between clauses in Ω, s.t. the resolution steps are ordered (the first step involves the first and the second clauses and eliminates $var(l_1)$, whereas the last step involves the penultimate and the last clauses and eliminates $var(l_2)$), then the clause $(\neg l_1 \vee l_2)$ is derived. ■

Clearly, this result can yield many irrelevant binary clauses. Hence, as already mentioned, the objective is to be selective about which clauses to actually consider.

	a=0	a=1	b=0	b=1	c=0	c=1	d=0	d=1	e=0	e=1
a=0	1			1				1		
a=1		1				1		1		1
b=0		1	1			1		1		1
b=1				1						
c=0	1			1	1			1		1
c=1						1				
d=0							1			
d=1								1		
e=0		1				1	1	1	1	
e=1										1

Figure 4.10: Inferring clause $(\neg a \vee d)$ (I)

4.3.4 Satisfiability-Based Inferred Clauses

This section describes the inference of clauses based on formula satisfiability conditions.

Proposition 4.3.5 *Given a CNF formula φ, for every pair of literals l_1 and l_2 for which there exists a variable x such that, $\langle l_1, 1\rangle \in \text{BCP}(\langle x, 0\rangle)$ and $\langle l_2, 1\rangle \in \text{BCP}(\langle x, 1\rangle)$ then the clause $(l_1 \vee l_2)$ is an implicate of φ.*

Proof. Clearly, if the two possible truth assignments on x_i either imply $\langle l_1, 1\rangle$ or $\langle l_2, 1\rangle$, then one of these two assignments must hold. In addition, we may prove this result with a simple resolution-based proof. If $\langle l_1, 1\rangle \in \text{BCP}(\langle x, 0\rangle)$, then $(x \vee l_1)$ is an implicate of φ (by using Proposition 4.3.4). If $\langle l_2, 1\rangle \in \text{BCP}(\langle x, 1\rangle)$, then $(\neg x \vee l_2)$ is an implicate of φ (also by using Proposition 4.3.4). Resolution between these two clauses allows to infer $(l_1 \vee l_2)$. ■

Proposition 4.3.6 *Given a CNF formula φ, for any clause $\omega = (l_{t_1} \vee \ldots \vee l_{t_k}) \in \varphi$,* all *clauses of the form, $\{l_{i_j} | \langle l_{i_j}, 1\rangle \in \text{BCP}(\langle l_{t_j}, 1\rangle), l_{t_j} \in \omega, j = 1, \ldots, k\}$ are implicates of φ.*

Proof. Since the original clause ω must be satisfied, any set of literals with size $|\omega|$, where each literal is implied by a different literal in ω, forms an implicate of φ. There is also a simple resolution-based proof. Given clause $\omega = (l_{t_1} \vee \ldots \vee l_{t_k}) \in \varphi$, if it is possible to infer the set of clauses $\{(\neg l_{t_1} \vee l_i), \ldots, (\neg l_{t_k} \vee l_j)\}$ by using Proposition 4.3.4, then clause $(l_i \vee \ldots \vee l_j)$ is inferred by resolution between this set of clauses and clause ω. ■

Propositions 4.3.5 and 4.3.6 are illustrated in Examples 4.7 and 4.8, respectively.

We should observe that the number of clauses that can be created is upper-bounded by the Cartesian product of each set of assignments that results from applying BCP to each triggering assignment. In addition, observe that the previous proposition can yield clauses with duplicate literals. Clearly, simple procedures can be implemented in order to filter out these duplicate literals.

4.3.5 Unsatisfiability-Based Inferred Clauses

Next we describe the inference of clauses based on formula unsatisfiability conditions.

Proposition 4.3.7 *Given a CNF formula φ, for all pairs l_1 and l_2 for which there exists a variable x such that, $\langle x, 0\rangle \in \mathrm{BCP}(\langle l_1, 0\rangle)$ and $\langle x, 1\rangle \in \mathrm{BCP}(\langle l_2, 0\rangle)$ the clause $(l_1 \vee l_2)$ is an implicate of φ.*

Proof. If two assignments imply distinct truth values on a given variable x_i, then the two assignments *must not* hold simultaneously. ■

One additional condition related with unsatisfiability is the following:

Proposition 4.3.8 *Given a CNF formula φ, for each set of assignments $A = \mathrm{BCP}(\langle l_1, v_1\rangle) \cup \ldots \cup \mathrm{BCP}(\langle l_k, v_k\rangle)$ such that there exists a clause $\omega \in \varphi$, with $\omega(A) = 0$, then the clause $(\neg l_1 \vee \ldots \vee \neg l_k)$ is an implicate of φ.*

Proof. If the union of sets of assignments resulting from applying BCP to a set of k triggering assignments unsatisfies a given clause, then the simultaneous occurrence of the k assignments must be prevented by creating a new clause. ■

Propositions 4.3.7 and 4.3.8 are illustrated in Examples 4.9 and 4.10, respectively.

Observe that a stronger condition can be established if the condition $w_j(\mathrm{BCP}(A)) = 0$ is used, at the cost of additional computational overhead. However, this would imply composed application of the BCP operation, which would complicate the algorithms to be proposed in the subsequent sections.

4.4 ProbIt: a Probing-Based SAT Preprocessor

The reasoning conditions described in the previous section were used to implement a SAT preprocessor, ProbIt. This preprocessor is organized as follows:

1. Create the table of assignments by applying BCP to each individual assignment.
2. Apply a restricted set of the reasoning conditions described in the previous sections:
   (a) Identification of necessary assignments, obtained by reasoning conditions from Propositions 4.3.1, 4.3.2 and 4.3.3. These propositions correspond to variable and clause probing, respectively.
   (b) Identification of equivalent variables, obtained by a *restricted* application of reasoning conditions from Proposition 4.3.4.
3. Iterate from 1 while more equivalent variables can be identified.

For the current version of ProbIt, we opted not to infer new clauses during preprocessing. Existing experimental evidence suggests that the inference of clauses during preprocessing can sometimes result in large numbers of new clauses, which can impact negatively the run times of SAT solvers (Gelder & Tsuji 1993). The identification of conditions for the selective utilization of clause inference conditions during preprocessing is the subject of future research work.

As a result, the utilization of Proposition 4.3.4 is restricted to the inference of binary clauses that lead to the identification of equivalent variables. Remember that two-variable equivalence (e.g. $x \leftrightarrow y$) is described by the pair of clauses $(\neg x \vee y) \wedge (x \vee \neg y)$, that can be represented as implications $(x \rightarrow y) \wedge (y \rightarrow x)$ (and also as $(\neg y \rightarrow \neg x) \wedge (\neg x \rightarrow \neg y)$). In ProbIt, rather than inferring new clauses which allow to identify equivalent variables, it is simpler to identify equivalences without having to infer the corresponding clauses. Based on the table of assignments, equivalent variables may be identified as follows:

- If $\langle y, 0\rangle \in \text{BCP}(\langle x, 0\rangle)$ and $\langle y, 1\rangle \in \text{BCP}(\langle x, 1\rangle)$, then $x \leftrightarrow y$.
- If $\langle y, 1\rangle \in \text{BCP}(\langle x, 0\rangle)$ and $\langle y, 0\rangle \in \text{BCP}(\langle x, 1\rangle)$, then $x \leftrightarrow \neg y$.

4.5 Experimental Results

In this section we present experimental results to evaluate the usefulness of the new algorithm. Results are given for different classes of problem instances, that include some of the hardest instances. Benchmark instances can be found on the SATLIB web site `http://www.satlib.org/`.

ProbIt has been integrated on the top of JQuest2. JQuest2 results from an evolution of the JQuest SAT solver. JQuest2 is a competitive Java SAT solver which entered in the second stage of the industrial category in the SAT'2003 Competition (see `http://www.satlive.org/SATCompetition/2003/comp03report/`). JQuest2 is a backtrack search SAT solver, based on efficient data structures, and implementing the most efficient backtrack search techniques, namely clause recording and non-chronological backtracking, search restarts, and adaptive branching heuristics. One of the main objectives of JQuest2 is to allow the rapid prototyping of new SAT algorithms. Since ProbIt is still an evolving preliminary implementation, the utilization of JQuest2 facilitates the evaluation and configuration of ProbIt.

In what follows, we first analyze the improvements on JQuest2 by integrating ProbIt as a preprocessor. Next, experimental results obtained for ProbIt+JQuest2 are compared with results obtained for other state-of-the-art SAT solvers. Afterwards, we analyze the reduction of variables and clauses after applying ProbIt. Finally, we give the numbers of variable equivalences, variable probing and clause probing for specific benchmark instances.

Tables 4.1 and 4.2 give the CPU time in seconds required for solving the different classes of problem instances. For each benchmark suite, the total number of instances is shown. For all experimental results a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used. The CPU time was limited to 5000 seconds. Hence, we

Table 4.1: Improvements on JQuest2

Family	ProbIt	ProbIt+JQuest2	JQuest2
barrel(8)	18.36	700.28	1,118.12
longmult(16)	544.18	1,725.17	4,658.67
queueinvar(10)	48.04	70.96	30.00
miters(25)	166.53	248.11	(2)11,175.65
fvp-unsat-1.0(4)	648.57	1,599.44	549.75
quasigroup(22)	61.21	531.43	735.25

Table 4.2: Comparison with other solvers

Family	ProbIt+JQuest2	2clseq	zChaff
barrel(8)	700.28	1,634.23	487.91
longmult(16)	1,725.17	2,201.37	2,191.08
queueinvar(10)	70.96	83.23	5.52
miters(25)	248.11	170.84	(2)10,537.49
fvp-unsat-1.0(4)	1,599.44	(2)13,545.74	549.75
quasigroup(22)	531.43	3,726.91	348.07

added 5000 seconds for each instance not solved in the allowed CPU time (the number of aborted instances is indicated between parentheses).

In Table 4.1, ProbIt+JQuest2 (JQuest2 with ProbIt integrated as a preprocessor) is compared with the original JQuest2. Moreover, the time required for the preprocessor ProbIt is also given. Table 4.2 compares ProbIt+JQuest2 with other SAT solvers, namely zChaff (Moskewicz *et al.* 2001) and 2clseq (Bacchus 2002a). zChaff is one of the most competitive SAT solvers. On the other hand, 2clseq is also a well-known competitive SAT solver, characterized by efficiently integrating formula manipulation techniques.

From the obtained results, several conclusions can be drawn:

- ProbIt+JQuest2 comes out as the most robust solver on the set of problem instances considered. Despite being implemented in Java, which necessarily yields a slower implementation, ProbIt+JQuest2 performance is indeed comparable to state-of-the-art SAT solvers.

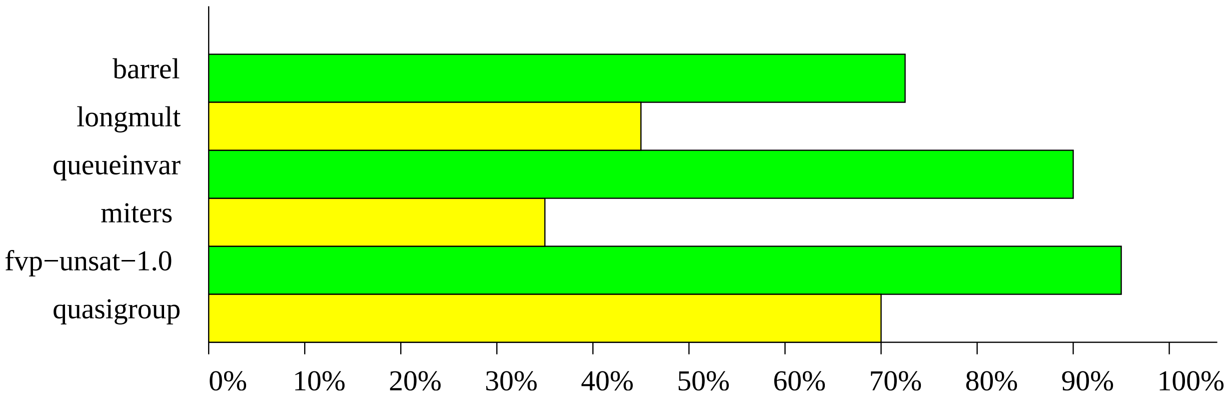


Figure 4.11: Percentage of variables kept after applying ProbIt

- The performance of ProbIt+JQuest2 is comparable to 2clseq in instances where formula manipulation helps on solving an instance. This explains why zChaff performance is not competitive for these instances.

- ProbIt+JQuest2 performance is also comparable to zChaff performance on instances where more sophisticated backtrack search techniques are required. Nonetheless, and when compared to JQuest2 results, ProbIt+JQuest2 may require more time to solve a family of benchmark examples. This can be explained by the time required for applying ProbIt techniques. Clearly, this is a drawback when the number of variables in the CNF formula is not reduced.

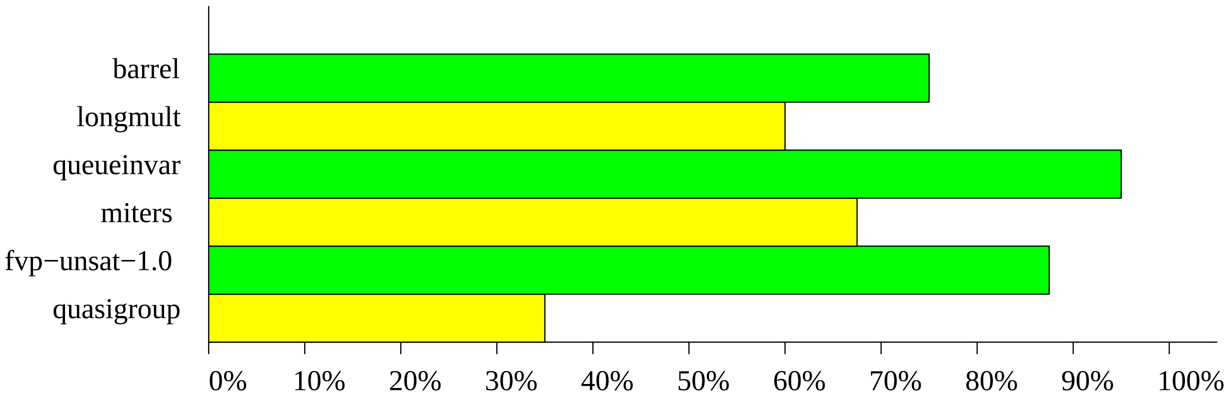


Figure 4.12: Percentage of clauses kept after applying ProbIt

Figures 4.11 and 4.12 give, respectively, the median values for the percentage of variables and clauses that are kept after applying ProbIt. Due to most of the miters' instances

Table 4.3: Results for the fvp-unsat-1.0 instances

Instance	Initial #Vars	% Kept Vars	Initial #Cls	% Kept Cls	Eqs	VP	CP
1dlx_c_mc_ex_bp_f	766	97%	3725	96%	4	0	2
2dlx_ca_mc_ex_bp_f	3186	94%	24640	88%	57	2	2
2dlx_cc_mc_ex_bp_f	4524	91%	41704	85%	118	2	2
9vliw_bp_mc	19148	96%	179492	94%	324	20	26

being solved by ProbIt during preprocessing, only those instances not solved during preprocessing were taken into account for determining the median values.

It is interesting to observe that both the percentage of kept variables and clauses ranges from 35% to 95%. Moreover, the figures suggest a relationship between the percentage of variables and the percentage of clauses that are kept after formula manipulation by ProbIt. In addition, the results indicate that the reduction in the number of variables and clauses is likely to be related with ProbIt's efficiency.

Finally, Tables 4.3 and 4.4 bring some more insights on the use of ProbIt. Table 4.3 gives results for the *fvp-unsat-1.0* instances, whereas Table 4.4 gives results for the *longmult* instances. For all the instances in both tables, we give the initial number of variables and clauses, as well as the percentage of variables and clauses kept after applying ProbIt. Results for instances solved during preprocessing are not presented. Moreover, the number of two-variable equivalences (Eqs) and the number of necessary assignments (found either based on variable probing (VP) or clause probing(CP)) are also given.

The results in Table 4.3 clearly indicate that applying ProbIt to *fvp-unsat-1.0* instances does not pay-off. Also, results from Table 4.1 already indicate that ProbIt is not efficient for these instances. These results may be explained by the reduced number of times the different formula manipulation techniques are applied.

Table 4.4 gives results for a the *longmult* family benchmark, for which ProbIt is very efficient (see Table 4.1). These results clearly indicate that the significant reduction in the number of variables and clauses, due to the number of equivalent variables and necessary assignments identified, reduces the required search effort for solving each problem instance.

Table 4.4: Results for the longmult instances

Instance	Initial #Vars	% Kept Vars	Initial #Cls	% Kept Cls	Eqs	VP	CP
longmult3	1555	35%	4767	39%	527	28	23
longmult4	1966	39%	6069	45%	693	26	11
longmult5	2397	41%	7431	48%	874	31	11
longmult6	2848	43%	8853	52%	1074	32	19
longmult7	3319	44%	10335	54%	1285	29	23
longmult8	3810	45%	11877	57%	1511	26	22
longmult9	4321	46%	13479	59%	1753	23	21
longmult10	4852	46%	15141	61%	2013	20	20
longmult11	5403	46%	16863	62%	2288	17	20
longmult12	5974	47%	18645	64%	2575	13	17
longmult13	6565	46%	20487	64%	2891	10	34
longmult14	7176	45%	22389	63%	3211	9	36
longmult15	7807	44%	24351	62%	3550	6	32

Despite the promising experimental results, probing-based formula manipulation merits additional research work. For instance with a large number of variables, the construction of the table of assignments can become prohibitive, due to the large number of implied assignments that can potentially be identified. Hence, the conditional creation of entries in the table of assignments needs to be addressed. Probing-based formula manipulation can also be more tightly integrated with backtrack search solvers, for example by allowing the application of probing-based techniques after search restarts.

4.6 Related Work

The ProbIt algorithm described in the previous section uses probing as the basis for implementing a number of formula manipulation techniques. In this section we relate ProbIt with previous work in probing techniques and other formula manipulation techniques.

4.6.1 Probing-Based Techniques

The idea of establishing hypotheses and inferring facts from those hypotheses has been studied in SAT and CSP. For CSP, the original reference is most likely the work of Freuder (Freuder 1978) on k-consistency. It can also be related with sigleton arc-consistency techniques (Debruyne & Bessière 1997). In the SAT domain, the notion of probing values and inferring facts from the probed values has been extensively studied in the recent past (Crawford & Auton 1993; Groote & Warners 2000; Berre 2001; Brafman 2001; Dubois & Dequen 2001).

The *failed literal rule* is a well-known and extensively used probing-based technique (see for example (Crawford & Auton 1993)): if the assignment $x = 0$ yields a conflict (due to BCP), then we must assign $x = 1$. This rule is covered by necessary assignments obtained from unsatisfiability conditions (Proposition 4.3.3).

Variable probing is a probing-based technique, which consists of applying the *branch-merge rule* to each variable (Berre 2001; Groote & Warners 2000). The branch-merge rule is the inference rule used in the *Stålmark's Method* (Stålmarck 1989). Common assignments to variables are identified, by detecting and merging equivalent branches. In addition, variable probing is often used as part of look-ahead branching heuristics in SAT solvers (Li & Anbulagan 1997). Observe that variable probing is covered by reasoning conditions established with Proposition 4.3.1.

Clause probing is similar to variable probing, even though variable probing is based on variables and clause probing is based on clauses. Clause probing consists of evaluating clause satisfiability requirements for identifying common assignments to variables. Common assignments are deemed necessary for a clause to become satisfied and consequently for the formula to be satisfied. Observe that clause probing is conceptually similar to the *recursive learning* paradigm for Boolean networks, proposed in (Kunz & Stoffel 1997). These techniques have been applied to SAT in (Marques-Silva & Glass 1999) and more recently in the 2-Simplify preprocessor (Brafman 2001). In our framework, clause probing is captured by Proposition 4.3.2. To the best of our knowledge, no other work proposes the joint utilization of variable and clause probing.

The notion of *literal dropping* considers applying sets of simultaneous assignments for inferring clauses that subsume existing clauses (Dubois & Dequen 2001). For a clause $(l \vee \beta)$, where β is a disjunction of literals, if assigning value 0 to all literals in β yields a conflict, then (β) is an implicate of φ. These techniques and the one described by Proposition 4.3.8 are related as long as both of them infer clauses from unsatisfiability requirements. The work of (Dubois & Dequen 2001) assumes a specific clause and considers BCP of simultaneous sets of assignments. Proposition 4.3.8 allows any k triggering assignments, but considers the separate application of BCP (which may yield fewer implied assignments). Nonetheless, Proposition 4.3.8 proposes more general conditions for inferring clauses, although based on a less powerful unit propagation.

4.6.2 Other Manipulation Techniques

Two-variable equivalence is a well-known formula manipulation technique that has been included in ProbIt, by a *restricted* application of reasoning conditions from Proposition 4.3.4. If x and y are equivalent, then the truth values of x and y must be the same. Consequently, we can replace y by x on *all* occurrences of y, thus eliminating variable y from the CNF formula. Additional two-variable equivalence conditions can be established, namely by the identification of *Strongly Connected Components* (Aspvall, Plass, & Tarjan 1979). It is interesting to observe that the existing strongly connected components in a CNF formula are captured from the construction of the table of assignments and the application of Proposition 4.3.4. Furthermore, sophisticated techniques have been developed to detect chains of biconditionals (Li 2000; Ostrowski *et al.* 2002).

The *2-closure* of a 2CNF sub-formula (Gelder & Tsuji 1993) allows to infer additional binary clauses. The identification of the transitive closure of the implication graph is obtained from the construction of the table of assignments and the application of Proposition 4.3.4: if $\langle y, 1\rangle \in \text{BCP}(\langle x, 1\rangle)$ then create clause $(\neg x \vee y)$.

More recently, a competitive SAT solver – 2clseq – incorporating hyper-resolution with binary clauses has been proposed (Bacchus 2002a). Later, a restricted and more efficient implementation of 2clseq originated a preprocessor (Bacchus & Winter 2003). Given the set of clauses $(\neg l_1 \vee x) \wedge (\neg l_2 \vee x) \wedge ... \wedge (\neg l_k \vee x) \wedge (l_1 \vee l_2 \vee ... \vee l_k \vee y)$, hyper-resolution allows inferring $(x \vee y)$. Once again, observe that this technique is covered by the construction of the table of assignments and the application of Proposition 4.3.4: for the same example, if $\langle y, 1\rangle \in \text{BCP}(\langle x, 0\rangle)$ then create clause $(x \vee y)$.

Compared with existing work, probing-based preprocessing techniques not only naturally capture *all* the above mentioned formula manipulation techniques, but also further allow the development of new techniques. In addition, the proposed unified framework also allows relating and comparing different formula manipulation techniques.

4.7 Summary

This chapter introduces a generic framework for applying probing-based techniques. Probing consists in formulating hypothetical scenarios, obtained by assigning a value to a variable, and then applying unit propagation. The results of probing are then kept on a table of assignments. Afterwards, different reasoning conditions are applied to the table of assignments, allowing either to identify necessary assignments or to infer new clauses.

This new approach integrates for the first time in a single framework most formula manipulation techniques and allows developing new techniques. For example, failed literals are easily detected. Furthermore, variable probing and clause probing techniques are naturally captured by analyzing, respectively, each set of entries corresponding to both literals of a variable or to all the literals in a clause. This framework also allows to establish two-variable equivalences and the 2-closure, among other formula manipulation techniques.

Finally, we introduce ProbIt, a new probing-based formula manipulation SAT preprocessor. ProbIt is a probing-based preprocessor that builds a table of assignments for each hypothetical scenario and further applies a set of reasoning conditions, resulting on the application of different formula manipulation techniques. The obtained experimental results clearly indicate that ProbIt is effective in reducing the required search effort and therefore in increasing the robustness of state-of-the-art SAT solvers.

5

Unrestricted Backtracking

Advanced techniques applied to backtrack search SAT algorithms have achieved remarkable improvements (Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997; Li & Anbulagan 1997; Zhang 1997; Moskewicz *et al.* 2001; Goldberg & Novikov 2002; Ryan 2004), having been shown to be crucial for solving hard instances of SAT obtained from real-world applications.

From a practical perspective, SAT algorithms should be able to prove unsatisfiability, since this is often the objective in a large number of significant real-world applications. Nevertheless, it is also widely accepted that local search can often have clear advantages with respect to backtrack search, since it is allowed to start the search over again whenever it gets *stuck* in a locally optimal partial solution.

The advantages of local search have motivated the study of hybrid approaches. Most of them build on local search algorithms and are able to guarantee completeness at the cost of adding clauses derived from conflicts. Examples of such algorithms are GSAT with dynamic backtracking (Ginsberg & McAllester 1994), *learn*-SAT (Richards & Richards 2000) and also CLS (Fang & Ruml 2004). Conversely, the algorithm proposed by Prestwich (Prestwich 2000) builds on backtrack search but gives some freedom to the search, by adding randomness to the backtrack step. In this algorithm, the backtrack point is randomly picked each time a conflict is identified. The backtrack point corresponds to the decision level of a literal randomly selected from the literals in the just created conflict clause.

By not always backtracking to the most recent untoggled decision variable, Prestwich's algorithm is able to often avoid the characteristic *trashing* of backtrack search algorithms, and so can be very competitive for specific classes of problem instances. We should note, however, that this algorithm is not complete and so cannot establish unsatisfiability.

In this chapter, we propose to select the backtrack point *without restrictions* within a complete backtrack search algorithm. The new algorithm, called *unrestricted backtracking*, avoids thrashing during backtrack search. Moreover, one can think of combining different forms of relaxing the identification of the backtrack point, i.e. different forms of backtracking.

This chapter is organized as follows. We first describe *random backtracking* (Lynce, Baptista, & Marques-Silva 2001a), which is tightly related with the major motivation for this work: to add to the efficient backtrack search algorithms the advantages of local search. Next, we introduce another new form of backtrack search, which is based on the use of heuristic knowledge to select the backtrack point (Bhalla *et al.* 2003a). Afterwards, *unrestricted backtracking* appears as a natural generalization of randomized and heuristic backtracking. Moreover, completeness conditions for the resulting SAT algorithm are established. Finally, we give experimental results and describe related work.

5.1 Randomized Backtracking

The utilization of different forms of randomization in SAT algorithms has seen increasing acceptance in recent years. Randomization is essential in many local search algorithms (Selman & Kautz 1993). Many local search algorithms repeatedly restart the (local) search by randomly generating complete assignments. Moreover, randomization can also be used for deciding among different (local) search strategies (McAllester, Selman, & Kautz 1997).

The efficient use of randomization in local search SAT algorithms has motivated the integration of randomization in the three main engines of a backtrack search algorithm.

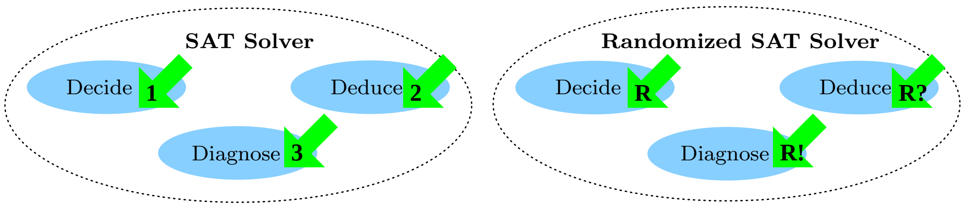


Figure 5.1: Introducing randomization in a backtrack search SAT solver

(These engines are described in Chapter 2.) The introduction of randomization in each of the three main engines is illustrated in Figure 5.1:

1. Randomization can be (and has actually been (Bayardo Jr. & Schrag 1997; Baptista & Marques-Silva 2000; Moskewicz *et al.* 2001)) applied to the decision engine by randomizing the variable selection heuristic. Variable selection heuristics, by being greedy in nature, are unlikely but unavoidably bound to select the wrong assignment at the wrong time for the wrong instance. The utilization of randomization helps reducing the probability of seeing this happening.

2. Randomization can be applied to the deduction engine by randomly picking the order in which implied variable assignments are handled during Boolean Constraint Propagation. Observe that the number of implied variable assignments does not depend on the introduction of randomization. For this engine, randomization only affects the order in which assignments are implied, and hence can only affect which conflicting clause is identified first. Even though randomizing the deduction engine allows randomizing which conflicting clause is analyzed each time a conflict is identified, the number of conflicting clauses is in general not large, and so it is not clear whether randomization of the deduction engine can play a significant role.

3. The diagnosis engine can be randomized by randomly selecting the point to backtrack to. This form of backtracking is referred to as *random backtracking*. The randomized backtracking procedure is outlined in Algorithm 5.1. After a conflict (i.e. an unsatisfied clause ω) is identified, a conflict clause ω_c is created. The conflict clause ω_c is then used for *randomly* deciding which decision assignment x is to be

picked, with $x \in \omega_c$. Then, the search backtracks to the decision level where variable x has been assigned. This contrasts with the usual non-chronological backtracking approach, in which the most recent decision assignment variable is selected as the backtrack point.

Algorithm 5.1: Randomized backtracking

```
Random_Backtracking($\omega$)
(1)    $\omega_c$ = Record_clause($\omega$)
(2)    $x$ = Randomly_pick_decision_assignment_variable($\omega_c$)
(3)    Apply_backtrack_step($x, \omega_c$)
```

After (randomly) selecting a backtrack point, the actual backtrack step can be organized in two different ways. The backtrack step can be *destructive*, meaning that the search tree is erased from the backtrack point down. In contrast, the backtrack step can be *non-destructive*, meaning that the search tree is not erased; only the backtrack point results in a variable assignment being toggled. This approach has much in common with dynamic backtracking (Ginsberg 1993).

The two randomized backtracking approaches differ. Destructive random backtracking is more drastic and attempts to rapidly cause the search to explore other portions of the search space. Non-destructive random backtracking has characteristics of local search, in which the current (partial) assignment is only locally modified.

Either destructive or non-destructive random backtracking can lead to potentially erratic algorithms, since there is no locality on how backtracking is performed. As a result, we propose to only applying random backtracking after every K conflicts; in between non-chronological backtracking is applied. We should note that the value of K is user-defined.

In a situation where $K \neq 1$, the application of random backtracking can either consider the most recent recorded clause or, instead, consider a target set that results from the union of the recorded conflict clauses in the most recent K conflicts. The first solution is characterized by having no memory of past conflicts, whereas the second solution considers equally all identified conflicts.

Finally, observe that randomized backtracking strategies can be interpreted as a generalization of search restart strategies (Gomes, Selman, & Kautz 1998). Search restarts, jointly with a randomized variable selection heuristic, ensure that different sub-trees are searched each time the search algorithm is restarted. However, whereas search restarts always start the search process from the root of the search tree, randomized backtracking randomly selects the point in the search tree from which the search is to be restarted (assuming destructive backtracking is used).

5.2 Heuristic Backtracking

Randomized backtracking is the first algorithm of a new generation of backtrack search algorithms, characterized by having no restrictions on selecting the backtrack point in the search tree. Obviously, this new family of backtrack search algorithms is incomplete, although there are different techniques that can be used to ensure completeness.

After introducing random backtracking, other incomplete forms of backtracking naturally appear. For example, one may envision *heuristic backtracking* (Bhalla *et al.* 2003a). The heuristic backtracking algorithm is quite similar to the random backtracking algorithm. However, the later *randomly* selects the backtrack point in the search tree from the variables in the most recently recorded clause, whereas the former selects the backtrack point *as a function* of variables in the most recently recorded clause. For example, in (Bhalla *et al.* 2003a) three different forms of heuristic backtracking are suggested, based on three different heuristics:

1. One heuristic that decides the backtrack point given the information of the most recently recorded conflict clause. This approach is called plain heuristic backtracking.

2. Another heuristic that is inspired in the VSIDS branching heuristic, used by Chaff (Moskewicz *et al.* 2001). This approach is called VSIDS-like heuristic backtracking.

3. Finally, one heuristic that is inspired in BerkMin's branching heuristic (Goldberg & Novikov 2002). This approach is called BerkMin-like heuristic backtracking.

In all cases the backtrack point is computed as the variable with the largest heuristic metric.

Algorithm 5.2: Unrestricted backtracking

```
UNRESTRICTED_BACKTRACKING($\omega$)
(1)    $\omega_c$ = RECORD_CLAUSE($\omega$)
(2)    $x$ = PICK_ANY_DECISION_ASSIGNMENT_VARIABLE($\omega_c$)
(3)    APPLY_BACKTRACK_STEP($x, \omega_c$)
```

5.3 Unrestricted Backtracking

Random backtracking and heuristic backtracking can be viewed as special cases of unrestricted backtracking (Lynce & Marques-Silva 2002b), the main difference being that in unrestricted backtracking any form of backtrack step can be applied. In random backtracking the backtrack point is computed randomly from the current conflict. In heuristic backtracking the backtrack point is computed from heuristic information, obtained from the current (and past) conflicts. In addition, if search restarts are a special case of random backtracking, then search restarts are also a special case of unrestricted backtracking.

The unrestricted backtracking algorithm is outlined in Figure 5.2. Clearly, this algorithm is quite similar to the random backtracking algorithm (see Fig 5.1). However, random backtracking *randomly* selects the backtrack point in the search tree from the variables in the most recently recorded clause, whereas unrestricted backtracking selects *any* backtrack point. Moreover, observe that in some forms of unrestricted backtracking clause ω_c does not necessarily have to be taken into account, i.e. some forms of unrestricted backtracking may pick a variable x s.t. $x \notin \omega_c$.

Unrestricted backtracking algorithms allow the search to backtrack to *any* point in the current search path whenever a conflict is reached, and for this reason most of the forms of unrestricted backtracking are *incomplete* forms of backtracking. As a consequence of the freedom for selecting the backtrack point in the decision tree, unrestricted backtracking entails a *policy* for applying different backtrack steps in sequence. Each backtrack step can be selected among chronological backtracking, non-chronological backtracking or incomplete forms of backtracking (e.g. search restarts, random backtracking, among many others). More formally, unrestricted backtracking consists of defining a sequence of Backtrack Steps $\{\mathrm{BSt}_1, \mathrm{BSt}_2, \mathrm{BSt}_3, \ldots\}$ such that each backtrack step BSt_i can either be

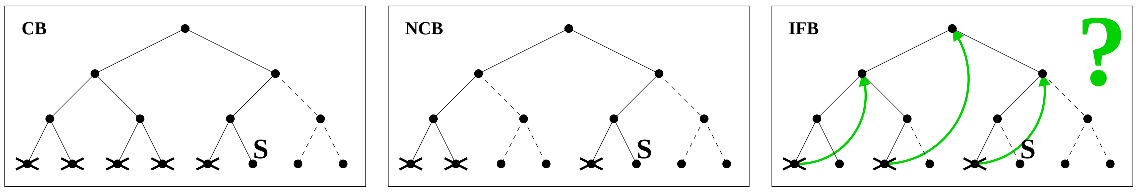


Figure 5.2: Comparing Chronological Backtracking (CB), Non-Chronological Backtracking (NCB) and Incomplete Form of Backtracking (IFB)

a chronological (CB), a non-chronological (NCB) or an incomplete form of backtracking (IFB).

Observe that the definition of unrestricted backtracking allows capturing the backtracking search strategies used by current state-of-the-art SAT solvers (Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997; Li & Anbulagan 1997; Zhang 1997; Moskewicz *et al.* 2001; Goldberg & Novikov 2002; Ryan 2004). Indeed, if the unrestricted backtracking strategy specifies always applying the chronological backtracking step or always applying the non-chronological backtracking step, then we respectively capture the chronological and non-chronological backtracking search strategies. Nonetheless, in this context we consider that an unrestricted backtracking algorithm always entails an incomplete algorithm, meaning that some of the backtrack steps are incomplete.

Unrestricted backtracking is particularly useful for giving a unified representation for different backtracking strategies. Consequently, unrestricted backtracking further allows establishing general completeness conditions for *classes* of backtracking strategies and not only for each individual strategy, as it has often been done in the past (Yokoo 1994; Richards & Richards 2000). We should note that the completeness conditions established to *all* organizations of unrestricted backtracking may obviously be applied to any special case of unrestricted backtracking (e.g. random backtracking and heuristic backtracking).

Even though within unrestricted backtracking any form of backtrack step can be applied (CB, NCB or IFB), unrestricted backtracking is characterized as performing an incomplete form of backtracking at least in some of the backtrack steps. Figure 5.2 compares CB, NCB and IFB. The main goal is to exemplify how incomplete forms of backtracking can lead to incompleteness. The different algorithms are exemplified for a generic search tree, where the search is performed accordingly to a depth-first search, and therefore the dashed branches define the search space not searched so far. Moreover, the search path that leads to the solution is marked with letter **S**. For CB and NCB the solution is easily found. CB makes the search backtrack to the most recent yet untoggled decision variable, whereas in NCB the backtrack point is computed as the *most recent* decision assignment from all the decision assignments represented in the recorded clause, allowing the search to skip portions of the search tree that are found to have no solution. However, since with an incomplete form of backtracking the search backtracks without restrictions, the search space that leads to the solution is simply skipped for IFB. What has to be done in order to assure the correctness and completeness of an incomplete form of backtracking? First, and similarly to local search, we have to assume that variable toggling in an incomplete form of backtracking is *reversible*. For the given example, this means that the solution can be found in a subsequent search, although the solution would have been skipped if variable toggling was not reversible. Clearly, reversible toggling allows to repeat search paths, at the risk of looping. However, techniques to avoid this problem are analyzed in (Lynce & Marques-Silva 2002b) and will be reviewed in what follows of this chapter.

5.4 Completeness Issues

In this section we address the problem of guaranteeing the completeness of SAT algorithms that implement some form of unrestricted backtracking. It is clear that unrestricted backtracking can yield incomplete algorithms. Hence, for each newly devised SAT algorithm, that utilizes some form of unrestricted backtracking, it is important to be able to apply conditions that guarantee the completeness of the resulting algorithm. Broadly, the completeness techniques for unrestricted backtracking can be organized in two classes:

- Marking recorded clauses as non-deletable. This solution may yield an exponential growth in the number of recorded clauses, although in practice this situation does not arise frequently.

- Increasing a given constraint (e.g. the number of non-deletable recorded causes) in

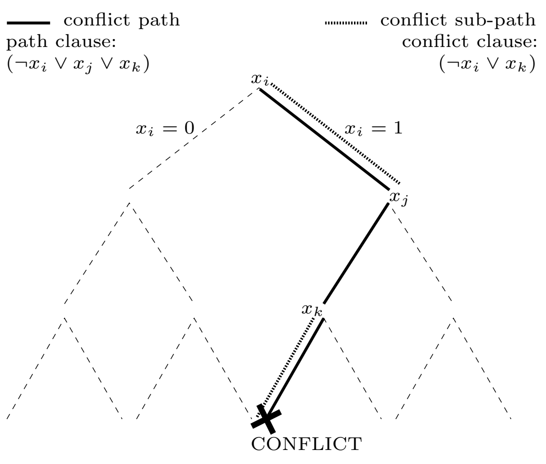


Figure 5.3: Search tree definitions

between applications of different backtracking schemes. This solution can be used to guarantee a polynomial growth of the number of recorded clauses.

The results presented in this section generalize, for the unrestricted backtracking algorithm, completeness results that have been proposed in the past for specific backtracking relaxations. We start by establishing, in a more general context, a few already known results. Afterwards, we establish additional results regarding unrestricted backtracking.

In what follows we assume the organization of a backtrack search SAT algorithm as described earlier in Chapter 2. The main loop of the algorithm consists of selecting a variable assignment (i.e. a *decision assignment*), making that assignment, and propagating that assignment using BCP. In the presence of an unsatisfied clause (i.e. a *conflict*), all the current decision assignments define a *conflict path* in the search tree. (Observe that we restrict the definition of conflict path solely with respect to the decision assignments.) A straightforward conflict analysis procedure consists of constructing a clause with *all* the decision assignments in the conflict path. In this case the created clause is referred to as a *path clause*. Moreover, after a conflict is identified we may also apply a *conflict analysis* procedure (Marques-Silva & Sakallah 1996; Bayardo Jr. & Schrag 1997; Moskewicz *et al.* 2001) to identify a subset of the decision assignments that represent a sufficient condition for producing the same conflict. The subset of decision assignments that is declared to be associated with a given conflict is referred to as a *conflict sub-path*. In this case the created clause is referred to as a *conflict clause*. Clearly, a conflict sub-path is a subset of a conflict path. Also, a conflict clause is a subset of a path clause, considering that these clauses are based on decision assignments.

Figure 5.3 illustrates the definitions of conflict path, path clause, conflict sub-path and conflict clause. We can now establish a few general results that will be used throughout this section.

Proposition 5.4.1 *If an unrestricted backtracking search algorithm does not repeat conflict sub-paths, then it does not repeat conflict paths.*

Proof. *(Sketch)* If a conflict sub-path is not repeated, then no conflict path can contain the same sub-path, and so no conflict path can be repeated. ■

Proposition 5.4.2 *If an unrestricted backtracking search algorithm does not repeat conflict paths, then it is complete.*

Proof. *(Sketch)* Assume a problem instance with n variables. Observe that there are 2^n possible conflict paths. If the algorithm does not repeat conflict paths, then it must necessarily terminate. ■

Proposition 5.4.3 *If an unrestricted backtracking search algorithm does not repeat conflict sub-paths, then it is complete.*

Proof. *(Sketch)* Given the two previous results, if no conflict sub-paths are repeated, then no conflict paths are repeated, and so completeness is guaranteed. ■

Proposition 5.4.4 *If the number of times an unrestricted backtracking search algorithm repeats conflict paths or conflict sub-paths is upper-bounded by a constant, then the backtrack search algorithm is complete.*

Proof. *(Sketch)* We prove the result for conflict paths; for conflict sub-paths, it is similar. Let M be a constant denoting an upper bound on the number of times a given conflict path can be repeated. Since the total number of distinct conflict paths is 2^n, and since each can be repeated at most M times, then the total number of conflict paths the

backtrack search algorithm can enumerate is $M \times 2^n$, and so the algorithm is complete.
■

Proposition 5.4.5 *For an unrestricted backtracking search algorithm the following holds:*

1. *If the algorithm creates a path clause for each identified conflict, then the search algorithm repeats no conflict paths.*
2. *If the algorithm creates a conflict clause for each identified conflict, then the search algorithm repeats no conflict sub-paths.*

In both cases, the search algorithm is complete.

Proof. (Sketch) Recall that the search algorithm always applies BCP after making a decision assignment. Hence, if a clause describing a conflict has been recorded, and not deleted, BCP guarantees that a conflict is declared, without requiring the same set of decision assignments that yields the original conflict. As a result, conflict paths are not repeated. The same holds true respectively for conflict clauses and conflict sub-paths. Since either conflict paths or conflict sub-paths are not repeated, the search algorithm is complete (from Propositions 5.4.2 and 5.4.3). ■

Observe that Proposition 5.4.5 holds *independently* of which backtrack step is taken each time a conflict is identified. Hence, as long as we record a conflict for each identified conflict, *any* form of unrestricted backtracking yields a complete algorithm. Less general formulations of this result have been proposed in the recent past (Ginsberg 1993; Yokoo 1994; Richards & Richards 2000; Fang & Ruml 2004).

The results established so far guarantee completeness at the cost of recording (and keeping) a clause for each identified conflict. In what follows we propose and analyze conditions for relaxing this requirement. As a result, we allow for some clauses to be deleted during the search process, and only require some specific recorded clauses to be kept. We say that a recorded clause is *kept* provided it is prevented from being deleted during the subsequent search.

However, we should note that clause deletion does not apply to chronological backtracking strategies and that, as shown in (Marques-Silva & Sakallah 1999), existing clause deletion policies for non-chronological backtracking strategies do not compromise the completeness of the algorithm.

Proposition 5.4.6 *If an unrestricted backtracking algorithm creates a conflict clause (or a path clause) after every M identified conflicts, then the number of times that the algorithm repeats conflict sub-paths (or conflict paths) is upper-bounded.*

Proof. (Sketch) Clearly, by creating (and recording) a conflict clause (or a path clause) after every M identified conflicts, the number of times a given conflict sub-path (or conflict path) is repeated is upper-bounded. Hence, using the results of Proposition 5.4.4 completeness is guaranteed. ∎

Proposition 5.4.7 *If an unrestricted backtracking algorithm records (and keeps) a* conflict clause *for each identified conflict for which an IFB step is taken, then the backtrack search algorithm is complete.*

Proof. (Sketch) Observe that there are at most 2^n IFB steps that can be taken, because a conflict clause is recorded for each identified conflict for which an IFB step is taken, and so conflict sub-paths due to IFB steps cannot be repeated. Moreover, the additional backtrack steps that can be applied (CB and NCB) also ensure completeness. Hence, the resulting algorithm is complete. ∎

Moreover, we can also generalize Proposition 5.4.4.

Proposition 5.4.8 *Given an integer constant M, an unrestricted backtracking algorithm is complete if it records (and keeps) a* conflict-clause *after every M identified conflicts for which an IFB step is taken.*

Proof. (Sketch) The result immediately follows from Propositions 5.4.5 and 5.4.7. ∎

Observe that for the previous conditions, the number of recorded clauses grows linearly with the number of conflicts where an IFB step is taken, and so in the worst-case exponentially in the number of variables.

Other approaches to guarantee completeness involve increasing the value of some constraint associated with the search algorithm. The following results illustrate these approaches.

Proposition 5.4.9 *Given an unrestricted backtracking strategy that applies a sequence of backtrack steps, if for this sequence the number of conflicts in between IFB steps strictly increases after each IFB step, then the resulting algorithm is complete.*

Proof. (Sketch) If only CB or NCB steps are taken, then the resulting algorithm is complete. When the number of conflicts in between IFB steps reaches 2^n, the algorithm is guaranteed to terminate. ■

We should also note that this result can be viewed as a generalization of the completeness-ensuring condition used in search restarts, that consists of increasing the backtrack cutoff value after each search restart (Baptista & Marques-Silva 2000). Given this condition, the resulting algorithm resembles iterative-deepening. Moreover, observe that in this situation the growth in the number of clauses can be made polynomial, provided clause deletion is applied on clauses recorded from NCB and IFB steps.

The next result establishes conditions for guaranteeing completeness whenever large recorded clauses (due to an IFB step) are opportunistically deleted. The idea is to increase the size of recorded clauses that are kept after each IFB step. Another approach is to increase the life-span of large-recorded clauses, by increasing the relevance-based learning threshold (Bayardo Jr. & Schrag 1997).

Proposition 5.4.10 *Suppose an unrestricted backtracking strategy that applies a specific sequence of backtrack steps. If for this sequence, either the size of the largest recorded clause kept or the size of the relevance-based learning threshold is strictly increased after each IFB step is taken, then the resulting algorithm is complete.*

Proof. (Sketch) When either the size of the largest recorded clause reaches value n, or the relevance-based learning threshold reaches value n, all recorded clauses will be kept, and so completeness is guaranteed from Proposition 5.4.5. ■

Observe that for this last result the number of clauses can grow exponentially with the number of variables. Moreover, we should note that the observation regarding increasing the relevance-based learning threshold was first suggested in (Moskewicz *et al.* 2001).

One final result addresses the number of times conflict paths and conflict sub-paths can be repeated.

Proposition 5.4.11 *Under the conditions of Proposition 5.4.9 and Proposition 5.4.10, the number of times a conflict path or a conflict sub-path is repeated is upper-bounded.*

Proof. (Sketch) Clearly, the resulting algorithms are complete, and so known to terminate after a maximum number of backtrack steps (that is constant for each instance). Hence, the number of times a conflict path (or conflict sub-path) can be repeated is necessarily upper-bounded. ■

5.5 Experimental Results

This section presents and analyzes experimental results that evaluate the efficiency of the techniques proposed in this chapter to solve hard real-world problem instances. Recent examples of such instances are the superscalar processor verification problem instances developed by M. Velev and R. Bryant (Velev & Bryant 1999), available from `http://www.ece.cmu.edu/∼mvelev/`. We consider four sets of instances:

- **sss1.0a** with 9 satisfiable instances;
- **sss1.0** with 40 selected satisfiable instances;
- **sss2.0** with 100 satisfiable instances;
- **sss-sat-1.0** with 100 satisfiable instances.

For all the experimental results presented in this section, a PIII @ 866MHz Linux machine with 512 MByte of RAM was used. The CPU time limit for each instance was set to 200 seconds, except for the *sss-sat-1.0* instances for which it was set to 1000 seconds. Since randomization was used, the number of runs was set to 10 (due to the large number of problem instances being solved). Moreover, the results shown correspond to the median values for all the runs.

In order to analyze the different techniques, we implemented the Quest0.5 SAT solver. Quest0.5 is built on top of the GRASP SAT solver (Marques-Silva & Sakallah 1996), but incorporates restarts as well as *random backtracking*. Random backtracking is applied non-destructively after every *K backtracks*. For Quest0.5 we chose to use the number of backtracks instead of the number of conflicts to decide when to apply random backtracking. This results from how the search algorithm in the original GRASP code is organized (Marques-Silva & Sakallah 1996).

Moreover, for the experimental results shown below, the following configurations were selected:

- **Rst1000+inc100** indicates that restarts are applied after every 1000 backtracks (i.e. the initial cutoff value is 1000), and the increment to the cutoff value after each restart is 100 backtracks. Observe that this increment is necessary to ensure completeness.

- **Rst1000+cr** indicates that restarts are applied after every 1000 backtracks and that **c**lause **r**ecording is applied for each search restart. This strategy ensures completeness.

- **RB1** indicates that random backtracking is taken at every backtrack step. A conflict clause is kept for each random backtrack to guarantee completeness.

- **RB10** applies random backtracking after every 10 backtracks. Again, a conflict clause is kept for each random backtrack to guarantee completeness.

- **Rst1000+RB1** means that random backtracking is taken at every backtrack and that restarts are applied after every 1000 backtracks. A conflict clause is kept for each randomized backtracking and search restart.

- **Rst1000+RB10** means that random backtracking is taken after every 10 backtracks and also that restarts are applied after every 1000 backtracks. Again, a conflict clause

Table 5.1: Results for the SSS instances

Inst	sss1.0a			sss1.0			sss2.0			sss-sat-1.0		
Quest 0.5	Time	Nodes	X	Time	Nodes	X	Time	Nodes	X	Time	Nodes	X
Rst1000+inc100	208	59511	0	508	188798	0	1412	494049	0	50512	8963643	39
Rst1000+cr	161	52850	0	345	143735	0	1111	420717	0	47334	7692906	28
RB1	79	11623	0	231	29677	0	313	31718	0	10307	371277	1
RB10	204	43609	0	278	81882	0	464	118150	0	6807	971446	1
Rst1000+RB1	79	11623	0	221	28635	0	313	31718	0	10330	396551	2
Rst1000+RB10	**84**	**24538**	**0**	**147**	**56119**	**0**	**343**	**98515**	**0**	**7747**	**1141575**	**0**
GRASP	1603	257126	8	2242	562178	11	13298	3602026	65	83030	12587264	82

is kept for each randomized backtracking and search restart.

The results for Quest0.5 on the SSS instances are shown in Table 5.1. In this table, *Time* denotes the CPU time, *Nodes* the number of decision nodes, and X the average number of aborted problem instances. As can be observed, the results for Quest0.5 reveal interesting trends:

- Random backtracking taken at every backtrack step allows significant reductions in the number of decision nodes.
- The elimination of repeated search paths in restarts, when based on keeping a conflict clause for each restart and when compared with the use of an increasing cutoff value, helps reducing the total number of nodes and also the CPU time.
- The best results are always obtained when random backtracking is used, independently of being or not used together with restarts.
- **Rst1000+RB10** is the only configuration able to solve all the instances in the allowed CPU time for *all* the runs.

The experimental results reveal additional interesting patterns. When compared with the results for GRASP, Quest 0.5 yields dramatic improvements. This is confirmed by evaluating either the CPU time, the number of nodes or the number of aborted instances. Furthermore, even though the utilization of restarts reduces the amount of search, it is also clear that more significant reductions can be achieved with randomized backtracking. In addition, the integrated utilization of search restarts and randomized backtracking allows obtaining the best results, thus motivating the utilization of multiple search strategies in backtrack search SAT algorithms.

5.6 Related Work

In the past, different forms of backtracking have been proposed (Stallman & Sussman 1977; Gaschnig 1979; Dechter 1990; Prosser 1993). The introduction of relaxations in the backtrack step is most likely related to dynamic backtracking (Ginsberg 1993). Dynamic backtracking establishes a method by which backtrack points can be moved deeper in the search tree. This allows avoiding the unneeded erasing of the amount of search that has been done thus far. The objective is to find a way to directly *erase* the value assigned to a variable as opposed to backtracking to it, moving the backjump variable to the end of the partial solution in order to replace its value without modifying the values of the variables that currently follow it. Afterwards, Ginsberg and McAllester combined local search and dynamic backtracking in an algorithm which enables arbitrary search movement (Ginsberg & McAllester 1994), starting with *any complete assignment* and evolving by flipping values of variables obtained from the conflicts.

In weak-commitment search (Yokoo 1994), the algorithm constructs a consistent partial solution, but commits to the partial solution *weakly.* In weak-commitment search, whenever a conflict is reached, the *whole* partial solution is abandoned, in explicit contrast to standard backtracking algorithms where the most recently added variable is removed from the partial solution.

Search restarts have been proposed and shown effective for hard instances of SAT (Gomes, Selman, & Kautz 1998). The search is repeatedly restarted whenever a cutoff value is reached. The algorithm proposed is not complete, since the restart cutoff point is kept constant. In (Baptista & Marques-Silva 2000), search restarts were jointly used with learning for solving hard real-world instances of SAT. This latter algorithm is complete, since the backtrack cutoff value increases after each restart. One additional example of backtracking relaxation is described in (Richards & Richards 2000), which is based on attempting to construct a complete solution, that restarts each time a conflict is identified. More recently, highly-optimized complete SAT solvers (Moskewicz *et al.* 2001; Goldberg & Novikov 2002) have successfully combined non-chronological backtracking and search restarts, again obtaining remarkable improvements in solving hard real-world instances of SAT.

5.7 Summary

This chapter proposes the utilization of unrestricted backtracking in backtrack search SAT solvers, a new form of backtracking that allows to select the backtrack point *without restrictions*.

The development of this new generation of backtrack algorithms was motivated by the integration of randomization in backtrack algorithms, due to randomization being successfully used in local search. Adding randomization to the backtrack step has lead to random backtracking, where the backtrack level is randomly selected from the decision levels of the variables in the just recorded clause.

Unrestricted backtracking naturally appears as a generalization of random backtracking. Unrestricted backtracking differs from the traditional forms of backtracking, namely chronological and non-chronological backtracking, to the extend that the search can backtrack to *any* point in the search tree. Clearly, random backtracking is a particular case of unrestricted backtracking.

SAT algorithms including unrestricted backtracking are no longer guaranteed to find a solution. In other words, unrestricted forms of backtrack are incomplete, which motivates defining new strategies for guaranteeing completeness in this context. The new strategies are based either on marking recorded clauses as non-deletable or on increasing a given constraint in between applications of different backtracking schemes.

Experimental results indicate that significant savings in the search effort can be obtained for different organizations of the proposed unrestricted backtrack search algorithm.

6

Hidden Structure in Unsatisfiable Random 3-SAT

The utilization of SAT in practical applications has motivated work on certifying SAT solvers (e.g. see (McMillan 2003)). Given a problem instance, the certifier needs to be able to verify that the computed truth assignment indeed satisfies a satisfiable instance and that, for an unsatisfiable instance, a proof of unsatisfiability can be generated. Certifying a SAT solver for a satisfiable instance is by far easier. Given a truth assignment for a problem instance, the certifier checks whether after setting the assignments all clauses are satisfied. Certifying a SAT solver for an unsatisfiable instance is hard. For an unsatisfiable instance, one has to be able to explain why the instance cannot be satisfied. For instance, one may provide a resolution proof based on an unsatisfiable core (Bruni & Sassano 2001; Zhang & Malik 2003) or a strong backdoor (Williams, Gomes, & Selman 2003). Broadly, an unsatisfiable core is a sub-formula that is still unsatisfiable and a strong backdoor is a subset of variables which define a search subspace that suffices to prove unsatisfiability.

The main goal of this chapter is to make an empirical study on hidden structure in typical case complexity. This work was motivated by the following questions: Is there any relation between the certificate given by a SAT solver and search complexity? It is quite intuitive that for a satisfiable instance there is a relation between hardness and the number of solutions (Selman, Mitchell, & Levesque 1996), although other factors may also play an important role, e.g. the size of minimal unsolvable subproblems (Mammen &

Hogg 1997). Is it also possible to relate a proof of unsatisfiability with search complexity? How does the size of unsatisfiable cores and strong backdoors relate with the hardness of unsatisfiable instances?

To answer these questions, we studied the behavior of random 3-SAT instances. Random 3-SAT is a well-know NP-complete problem (Cook 1971). Random k-SAT formulas are also well-know for exhibiting a phase transition when the ratio of clauses to variables is compared with the search effort (Cheeseman, Kanefsky, & Taylor 1991; Coarfa *et al.* 2003; Selman, Mitchell, & Levesque 1996). The experimental results given in this chapter aim to relate the size of unsatisfiable cores and strong backdoors with the search effort required to solve random 3-SAT unsatisfiable instances. Observe that theoretical work has already been developed in the past (Beame *et al.* 2002; Chvtal & Szemerédi 1988), but our focus is to make an empirical study.

Empirical results have first been obtained using the most recent version of zChaff, available from `http://ee.princeton.edu/∼chaff/zchaff.php`. zChaff is the *only* available solver to integrate extraction of unsatisfiable cores (Zhang & Malik 2003). However, this algorithm has the drawback of giving approximate results, meaning that there is no guarantee about the unsatisfiable core having the smallest number of clauses. Hence, one may doubt of the accuracy of results, i.e. one may argue that more accurate results would lead to different conclusions. Consequently, we have developed a new model and algorithm that can be used to obtain the smallest size unsatisfiable cores and strong backdoors. Results for the new algorithm confirm the conclusions obtained by using zChaff.

This chapter is organized as follows. In the next section we characterize random 3-SAT instances, followed by the definitions of unsatisfiable cores and strong backdoors. In Section 6.4 we give experimental data obtained by running zChaff on random 3-SAT instances. Section 6.5 relates hardness with hidden structure, allowing us to conclude that hard unsatisfiable instances are hard because of the size of unsatisfiable cores and strong backdoors. Afterwards, we introduce a new model and algorithm for computing smallest size unsatisfiable cores and strong backdoors. Additional results for the new algorithm confirm the conclusions previously obtained.

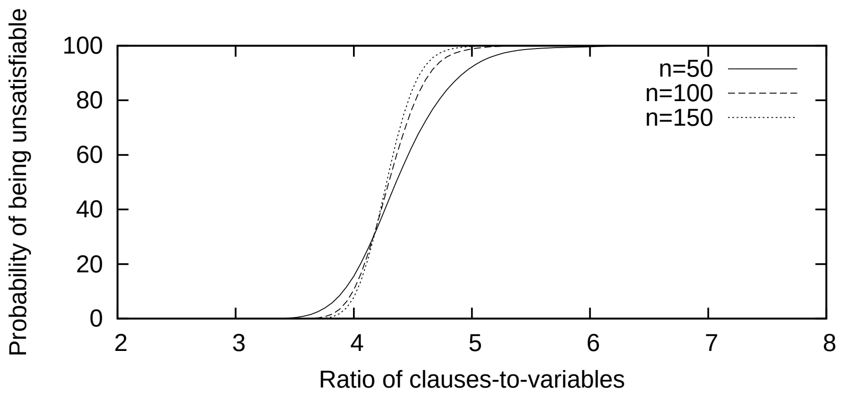


Figure 6.1: Probability of unsatisfiability of random 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio of clauses to variables

6.1 Random 3-SAT

Random 3-SAT instances are obtained by randomly generating clauses with length 3. For an instance with n variables and m clauses, each literal of the m clauses is randomly selected from the $2n$ possible literals such that each literal is selected with the same probability of $1/2n$. Clauses with repeated literals or with a literal and its negation (tautological clauses) are discarded.

Random k-SAT formulas are particularly interesting due to the occurrence of a phase-transition or threshold phenomenon, i.e. a rapid change in complexity when increasing (or decreasing) the ratio of clauses to variables (Crawford & Auton 1993; Selman, Mitchell, & Levesque 1996). For a small ratio, almost all formulas are under-constrained and therefore satisfiable. As the value of m/n increases, almost all instances are over-constrained and therefore unsatisfiable. Experiments strongly suggest that for random 3-SAT there is a threshold at a critical ratio of clauses to variables $m/n \approx 4.3$ such that beyond this value the probability of generating a satisfiable instance drops to almost zero. (Also, a complexity peak also occurs around 4.3.) This behavior is illustrated in Figure 6.1. Experimental data is given for the probability of satisfiability of random 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio of clauses to variables. Each point corresponds to the median value of 500 random 3-SAT instances.

Algorithm 6.1: Computing an unsatisfiable core

```
COMPUTE_UNSAT_CORE(ClauseStack $St$)
(1)    while $St$ is not empty
(2)        Clause $C$ = POP_CLAUSE($St$)
(3)        if $C$ is not marked
(4)            continue
(5)        ClauseSet $CS$ = GET_REASONS_FOR_RECORDING_CLAUSE($C$)
(6)        MARK_CLAUSE_SET($CS$)
```

6.2 Unsatisfiable Cores

Research on unsatisfiable cores can be distinguished between theoretical and experimental work. In the theoretical field, unsatisfiable cores complexity has been analyzed and formal algorithms have been proposed (Aharoni & Linial 1986; Beame *et al.* 2002; Büning 2000; Chvtal & Szemerédi 1988; Davydov, Davydova, & Büning 1998; Fleischner, Kullmann, & Szeider 2002; Papadimitriou & Wolfe 1988). Experimental work includes contributions of Bruni and Sassano (Bruni & Sassano 2001) and Zhang and Malik (Zhang & Malik 2003). Both approaches extract unsatisfiable cores. The first approach proposes an adaptive search guided by clauses hardness. The second approach is motivated by considering that a CNF formula is unsatisfiable if and only if is possible to generate an empty clause by resolution from the original clauses. In this case, the resolution steps are emulated by the creation of nogoods. The unsatisfiable core is given by the set of original clauses involved in the derivation of the empty clause.

Algorithm 6.1 presents the pseudo-code for the algorithm proposed in (Zhang & Malik 2003). In this algorithm, ClauseStack St represents a stack with the recorded clauses, ordered by creation time (the clause on the top is the most recently recorded clause). Also, it is necessary to consider a marking scheme for the clauses. Initially, only the clauses involved in deriving the empty clause are marked. At the end, the marked original clauses correspond to the unsatisfiable core. For each iteration in the algorithm, a new set of clauses is marked (MARK_CLAUSE_SET). In addition, the algorithm keeps a file with all the reasons for creating each recorded clause, i.e. with all the clauses involved in the resolution steps utilized for creating a recorded clause. This file is updated during the search for each new recorded clause. For computing the unsatisfiable core, a breath-first traversal over the file is used, allowing to traverse the marked recorded clauses (obtained with GET_REASONS_FOR_RECORDING_CLAUSE(C)) in the order as they appear in the clause stack.

Definition 6.1 (Unsatisfiable Core) *Given a formula φ, UC is an unsatisfiable core for φ iff UC is a formula s.t. UC is unsatisfiable and UC $\subseteq \varphi$.*

Observe that an unsatisfiable core can be defined as any subset of the original formula that is unsatisfiable. Consequently, there may exist many different unsatisfiable cores, with a different number of clauses, for the same problem instance, such that some of these cores can be subsets of others. Also, and in the worst case, the unsatisfiable core corresponds exactly to the set of original clauses.

Definition 6.2 (Minimal Unsatisfiable Core) *An unsatisfiable core UC for φ is a minimal unsatisfiable core iff removing any clause $\omega \in$ UC from UC implies that UC $-\{\omega\}$ is satisfiable.*

Definition 6.3 (Minimum Unsatisfiable Core) *An unsatisfiable core UC for φ is a minimum unsatisfiable core iff it is a minimal unsatisfiable core of minimum cardinality.*

Example 6.1 *Consider the following formula φ having the following clauses:*

$\omega_1 = (x_1 \vee \neg x_3)$ $\quad \omega_3 = (\neg x_2 \vee x_3)$ $\quad \omega_5 = (x_2 \vee x_3)$

$\omega_2 = (x_2)$ $\quad \omega_4 = (\neg x_2 \vee \neg x_3)$ $\quad \omega_6 = (\neg x_1 \vee x_2 \vee \neg x_3)$

Given these clauses, the following unsatisfiable cores can be identified:

$UC_1 = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6\}$ $\quad UC_4 = \{\omega_1, \omega_3, \omega_4, \omega_5, \omega_6\}$ $\quad UC_7 = \{\omega_2, \omega_3, \omega_4, \omega_5\}$

$UC_2 = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5\}$ $\quad UC_5 = \{\omega_2, \omega_3, \omega_4, \omega_5, \omega_6\}$ $\quad UC_8 = \{\omega_2, \omega_3, \omega_4, \omega_6\}$

$UC_3 = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_6\}$ $\quad UC_6 = \{\omega_1, \omega_2, \omega_3, \omega_4\}$ $\quad UC_9 = \{\omega_2, \omega_3, \omega_4\}$

From the unsatisfiable cores given above, UC_4 and UC_9 are minimal unsatisfiable cores, but only UC_9 is a minimum unsatisfiable core.

Interestingly, the existing experimental work described above (Bruni & Sassano 2001; Zhang & Malik 2003) has very little concern regarding extraction of *minimal* unsatisfiable cores. Nonetheless, the work in (Zhang & Malik 2003) proposes an iterative solution for *reducing* an unsatisfiable core, by iteratively invoking the SAT solver on each computed sub-formula. This solution, albeit capable of reducing the size of computed unsatisfiable cores, does not provide *any* guarantees regarding the unsatisfiable core being either minimal or minimum. However, in some practical applications it may be useful identifying the *minimum* unsatisfiable core of a given problem instance, i.e. the *smallest* number of clauses that make the instance unsatisfiable. We should note that in some cases the size of a minimal unsatisfiable core may be much larger than the size of the minimum unsatisfiable core.

6.3 Strong Backdoors

The notion of backdoor was introduced by Williams, Gomes and Selman in (Williams, Gomes, & Selman 2003). Research on backdoors was motivated by the heavy-tailed phenomenon in backtrack search algorithms (Gomes *et al.* 2000). A backdoor is a special subset of variables that characterizes hidden structure in problem instances.

Backdoor definition depends on a sort of algorithm called *sub-solver*. A sub-solver $\mathcal{S}$ always runs in polynomial time. For example, $\mathcal{S}$ could be a solver that is able to solve 2-SAT instances but rejects k-SAT instances, with $k \geq 3$. Given a partial truth assignment $A'_X : X' \subseteq X \rightarrow \{true, false\}$, a sub-solver $\mathcal{S}$ is able to solve the formula $\varphi[A'_X]$ in polynomial time.

Definition 6.4 (Backdoor) *A nonempty subset Y of the variables set X is a backdoor for φ w.r.t. $\mathcal{S}$ iff for some partial truth assignment $A_Y : Y \rightarrow \{true, false\}$, $\mathcal{S}$ returns a satisfying assignment of $\varphi[A_Y]$.*

Clearly, the definition of backdoor given above only applies to *satisfiable* formulas. Moreover, observe that there may exist many backdoor sets for a given formula. In the worst case, there is only one backdoor that corresponds exactly to the set of all variables. Consequently, it may be interesting to identify minimum and minimal backdoors.

Definition 6.5 (Minimal Backdoor) *A nonempty backdoor set Y for φ w.r.t. $\mathcal{S}$ is a minimal backdoor iff removing any variable $v \in Y$ from Y implies that $Y - \{v\}$ is not a backdoor set.*

Definition 6.6 (Minimum Backdoor) *A nonempty backdoor set Y for φ w.r.t. $\mathcal{S}$ is a minimum backdoor iff it is a minimal backdoor of minimum cardinality.*

Since the definition of backdoor given above only considers satisfiable instances, the work of Williams *et al.* also introduced the definition of strong backdoor for unsatisfiable instances. This definition holds for both satisfiable and unsatisfiable instances.

Definition 6.7 (Strong Backdoor) *A nonempty subset Y of the variables set X is a strong backdoor for φ* w.r.t. *$\mathcal{S}$ iff for all $A_Y : Y \to \{true, false\}$, $\mathcal{S}$ returns a satisfying assignment for $\varphi[A_Y]$ or concludes unsatisfiability of $\varphi[A_Y]$.*

The definition of strong backdoor contrasts with the definition of backdoor to the extent that for a strong backdoor Y no truth assignment is specified. This means that all possible assignments of Y have to be considered. Moreover, minimum and minimal strong backdoors can be defined similarly to minimum and minimal backdoors.

6.4 zChaff on Random 3-SAT

In this section we analyze zChaff's results on random 3-SAT instances. zChaff (Moskewicz *et al.* 2001) is an efficient DLL-based SAT solver enhanced with clause recording (Marques-Silva & Sakallah 1996). We used zChaff for being a state-of-the-art SAT solver integrating the extraction of unsatisfiable cores. Clearly, solving different sub-formulas with *any* complete solver would also allow us to extract unsatisfiable cores, although not so efficiently.

Overall, zChaff's behavior on solving 3-SAT instances is similar to the behavior reported in the literature for a DLL solver (e.g. see (Selman, Mitchell, & Levesque 1996)). Hence, one may conclude that for random instances clause learning is not particularly relevant. This conclusion corresponds indeed to practice: clause learning is very useful for

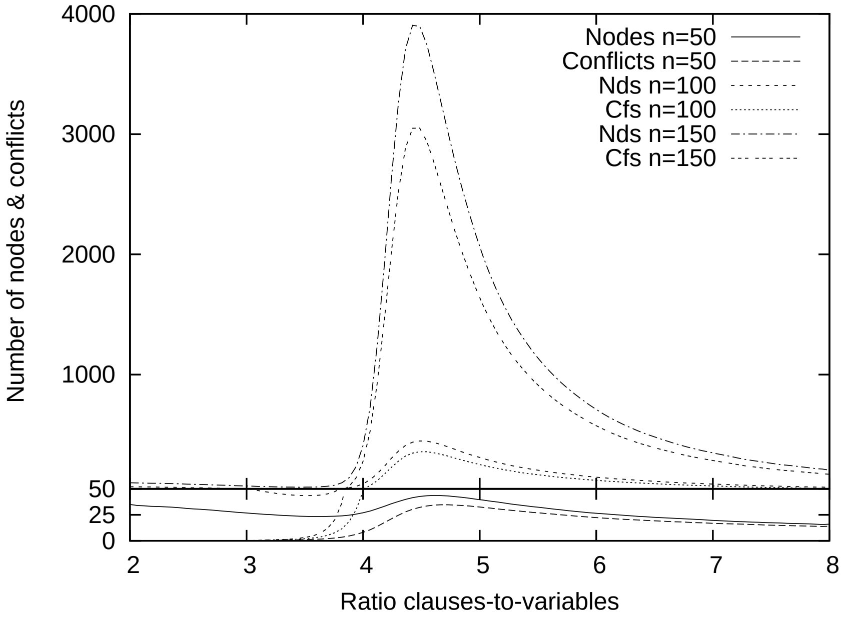


Figure 6.2: Number of nodes and conflicts when using zChaff for solving random 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio of clauses to variables

structured instances that usually come from real-world domains, rather than for random instances.

Figure 6.2 gives experimental results for the number of nodes and conflicts when using zChaff for solving random 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio clauses/variables. Observe that two different scales are used for the plot; otherwise values for 50 variables would not be legible.

By analyzing Figure 6.2, the following conclusions can be drawn:

1. As a whole, there is a correspondence between the number of nodes and the number of conflicts. This means that the more we search, the more conflicts we find. However, there is an exception: when the ratio clauses/variables is reduced. In these cases, there are so few constraints that is trivial to find a solution for them and consequently the number of conflicts is negligible. On the other hand, the number of nodes almost equals the number of variables due to the lack of implications.

2. The graph exhibits a similar shape independently of the number of variables, although as the number of variables increases, the steeper are the curves.

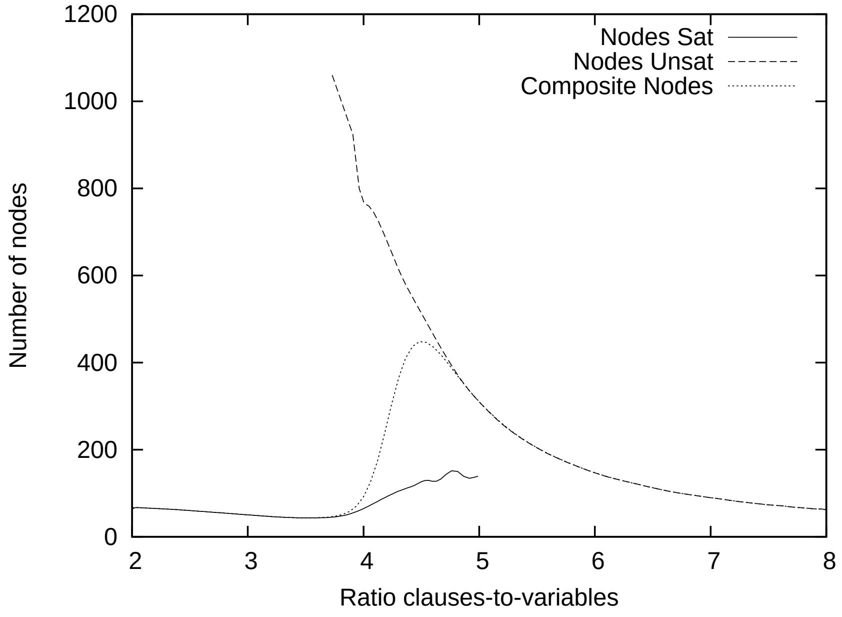


Figure 6.3: Number of nodes when using zChaff for solving satisfiable and unsatisfiable random 3-SAT formulas with 100 variables as a function of the ratio of clauses to variables

3. The maximum value for the number of nodes and conflicts can be observed when the ratio of clauses/variables is ≈ 4.3. Recall that this value corresponds to the ratio of clauses to variables where the probability of generating a satisfiable instance equals the probability of generating an unsatisfiable instance.

Figure 6.3 gives the number of nodes when using zChaff for solving satisfiable and unsatisfiable random 3-SAT formulas with 100 variables as a function of the ratio of clauses to variables. Observe that similar results have been obtained in the past with a DLL solver (Cheeseman, Kanefsky, & Taylor 1991; Selman, Mitchell, & Levesque 1996). Moreover, the graph would exhibit a similar shape independently of the number of variables, although as the number of variables increases the steeper are the curves. Overall, the maximum value for the number of nodes is observed when the ratio of clauses/variables is ≈ 4.3.

The main conclusion is essentially that satisfiable and unsatisfiable sets are quite different when comparing the number of nodes. Most satisfiable instances are very easy to solve. Satisfiable instances with a higher ratio clauses/variables are slightly more difficult

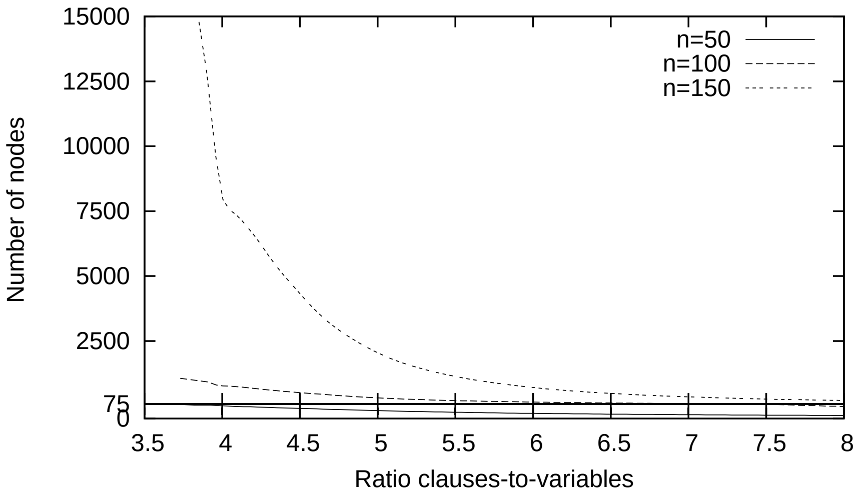


Figure 6.4: Number of nodes when using zChaff for solving unsatisfiable random 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio of clauses to variables

to solve. Unsatisfiable instances with a small ratio clauses/variables are the most difficult to solve. Also, unsatisfiable instances with a larger ratio are still hard.

Figure 6.4 is a plot for the number of nodes of all unsatisfiable instances with 50, 100 and 150 clauses. Once again, we should note that two different scales are used in order to clarify results for 50 variables. Clearly, the same trends are observed for all the instances, i.e. the number of variables does not affect the general trend: unsatisfiable instances with a small ratio of clauses to variables are much harder than unsatisfiable instances with a larger ratio.

6.5 Hardness and Hidden Structure

Early studies on complexity relate hardness of k-SAT instances with the ratio of the number of clauses to the number of variables (Cheeseman, Kanefsky, & Taylor 1991; Crawford & Auton 1993; Gent & Walsh 1996; Selman, Mitchell, & Levesque 1996). Moreover, hardness has often been related with the hidden structure of *satisfiable* instances. For example, in (Selman, Mitchell, & Levesque 1996) hardness is regarded as a function of the number of solutions. Moreover, recent work relates backdoors with hardness of satisfiable instances, based on the concept of *key* backdoors (Ruan, Kautz, & Horvitz 2004).

Theoretical work has already related hardness and hidden structure (Beame *et al.* 2002; Chvtal & Szemerédi 1988). However, little effort has ever been made in order to empirically relate these two aspects. Interestingly, recent empirical work on unsatisfiable cores and strong backdoors has brought some new insights on the topic.

Our first intuition was that hardness and the size of unsatisfiable cores and strong backdoors would be related due to the following reasons:

- Unsatisfiability is proved when the search space is exhausted. For a DLL solver with an *accurate* heuristic, the search space can be reduced to 2^b, where b is the size of the minimum strong backdoor. Also, for a solver with clause recording, the number of steps required to derive the empty clause can be related with the size of the unsatisfiable core. Although a recorded clause may include more than one resolution step, problem instances with small unsatisfiable cores should require less resolution steps to be solved.

- The probability of generating satisfiable instances exhibits a phase-transition (see Figure 6.1), i.e. at a certain value of the ratio of clauses to variables the probability of generating satisfiable instances *quickly* decreases to 0% as we add clauses to the formula. Conversely, the probability of generating unsatisfiable instances *quickly* increases to 100% at a certain value of the ratio of clauses to variables. Hence, unsatisfiable instances with a ratio of clauses to variables m/n above ≈ 4.3, where the probability of generating a satisfiable/unsatisfiable instance is about 50%, are probably unsatisfiable with less than m clauses.

  For example, let us consider the generation of a *typical* unsatisfiable formula φ with n variables and m clauses, where $m/n > 4.3$. Consider that formula φ has a set of clauses $\Omega = \{\omega_1, ..., \omega_p, ..., \omega_m\}$. Suppose that φ is built by adding clauses in Ω one at a time. Moreover, with clauses $\{\omega_1..\omega_{p-1}\}$ ($p \approx m - 4.3n$) the formula is satisfiable but with all the clauses $\{\omega_1..\omega_p\}$ the formula is unsatisfiable. Thus the minimum unsatisfiable core size is $\leq p$. Furthermore, adding clauses $\{\omega_{p+1}, ..., \omega_m\}$ to the formula may only reduce the size of the minimum unsatisfiable core.

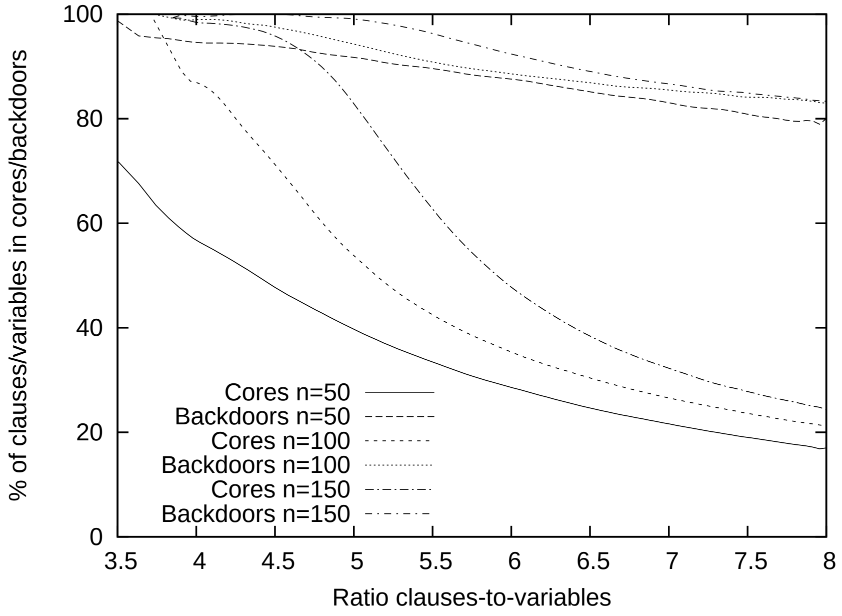


Figure 6.5: Size of unsatisfiable cores and strong backdoors (%) for unsatisfiable random 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio of clauses to variables

Clearly, the same reasoning can be applied to strong backdoors. This allows us to conclude that unsatisfiable cores and strong backdoors sizes are related, to the extent that both sizes decrease with the increasing of the ratio of the number of clauses to variables.

Figure 6.5 shows the evolution on the size of unsatisfiable cores and strong backdoors. More precisely, results indicate the percentage of clauses in the unsatisfiable cores with respect to the total number of clauses and the percentage of variables in the strong backdoors with respect to the total number of variables. Results are given for random unsatisfiable 3-SAT formulas with 50, 100 and 150 variables, as a function of the ratio of the number of clauses to variables.

The size of unsatisfiable cores has been computed by zChaff. The size of strong backdoors has been obtained from the corresponding unsatisfiable cores: for each instance, all variables in the clauses of the unsatisfiable core have been considered to be part of the strong backdoor. This means that each strong backdoor Y for a formula φ has been defined w.r.t. a sub-solver $\mathcal{S}$ that for all assignments $A_Y : Y \rightarrow \{true, false\}$ simply checks that at least one clause is unsatisfied and consequently concludes unsatisfiability of $\varphi[A_Y]$.

With respect to the size of unsatisfiable cores, results in Figure 6.5 clearly confirm our intuition. Observe that the reduction in the size of unsatisfiable cores is not only due to the increasing number of clauses with the increasing ratio of clauses to variables. Indeed, the absolute value for the size of unsatisfiable cores also decreases as a function of the ratio clauses/variables. Hence, one may conclude that harder instances have unsatisfiable cores larger than easier instances with a higher ratio of clauses to variables.

In addition, the relation between hardness and strong backdoors size is also suggested, although not so clearly. One may argue that the sub-solver $\mathcal{S}$ involved in the extraction of the strong backdoor does not favor getting a small strong backdoor. (Using different sub-solvers is a topic for future research work.) Also, one may argue that the size of the obtained strong backdoors is much larger than the size of minimum or even minimal strong backdoors. For this reason, in the next section we address an algorithm for extracting minimal and minimum sizes unsatisfiable cores and strong backdoors.

Figures 6.6, 6.7 and 6.8 also emphasize the correlation between hardness (given by the number of nodes searched by zChaff) and the number of clauses in the unsatisfiable cores, between hardness and the number of variables in the backdoors, and also between the number of clauses in the unsatisfiable cores and the the number of variables in the backdoors, respectively. For Figures 6.6 and 6.7, the regression was done using an exponential function, whereas for Figure 6.8 a polynomial of second order was used. The correlation is more clear in Figures 6.6 and 6.8 than in Figure 6.7, probably due to the reasons we have pointed in the previous paragraph. Figure 6.8 also reflects the plot obtained in Figure 6.7, again suggesting that using another sub-solver $\mathcal{S}$ may allow us to draw stronger conclusions.

6.6 Improving Results Accuracy

The previous plots exhibit a clear trend towards relating hardness with the size of unsatisfiable cores and strong backdoors. However, one may strengthen the obtained

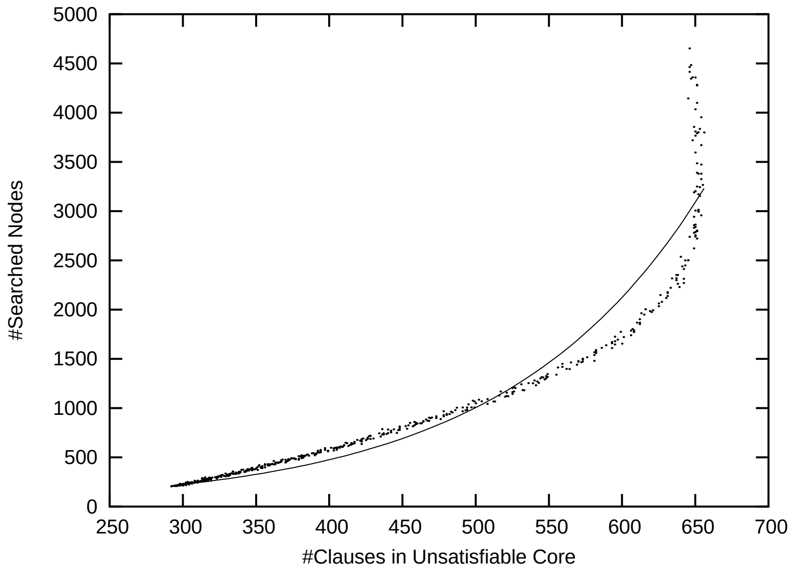


Figure 6.6: Regression on hardness and the number of clauses in the unsatisfiable cores for unsatisfiable random 3-SAT formulas with 150 variables

conclusions with more accurate results. In this section, we provide a model for identifying minimum and minimal unsatisfiable cores and strong backdoors. Experimental results with the new algorithm confirm the conclusions established in the previous section.

Let us start by introducing our model for extracting minimum unsatisfiable cores. First of all, it is clear that we can use a brute-force algorithm for exploring the whole search space while keeping track of the minimum unsatisfiable core. But we can do significantly better: we can emulate hiding each one of the clauses in order to perform the search in all possible subsets of clauses. Also, we can learn from the conflicts.

We assume that each formula φ is defined over n variables, $X = \{x_1, \ldots, x_n\}$, and m clauses, $\Omega = \{\omega_1, \ldots, \omega_m\}$. We start by defining a set S of m new variables, $S = \{s_1, \ldots, s_m\}$, and then create a new formula φ' defined on $n+m$ variables, $X \cup S$, and with m clauses, $\Omega' = \{\omega'_1, ..., \omega'_m\}$. Each clause $\omega'_i \in \varphi'$ is defined from a corresponding clause $\omega_i \in \varphi$ and from a variable s_i s.t. $\omega'_i = \{\neg s_i\} \cup \omega_i$.

Example 6.2 *Consider the CNF formula φ having the variables $X = \{x_1, x_2, x_3\}$ and the clauses $\Omega = \{\omega_1, ..., \omega_6\}$:*

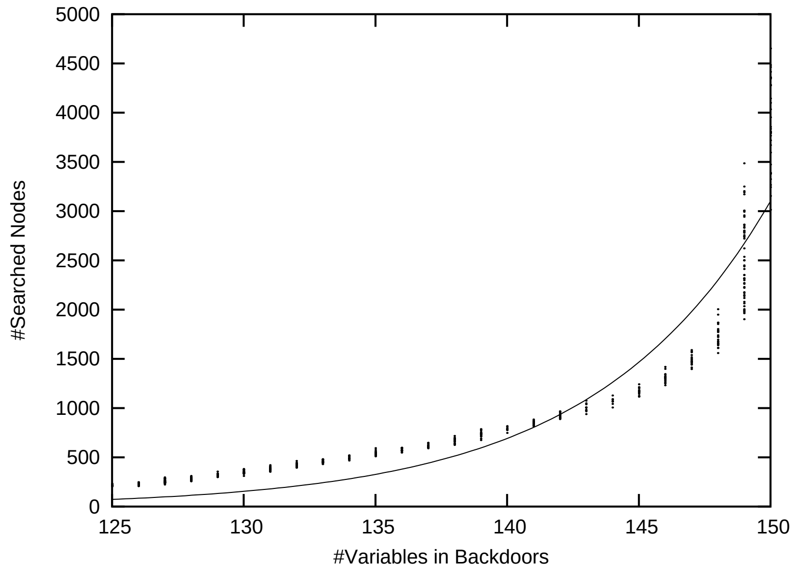


Figure 6.7: Regression on hardness and the number of variables in the backdoors for unsatisfiable random 3-SAT formulas with 150 variables

$$\omega_1 = (x_1 \vee \neg x_3) \qquad \omega_4 = (\neg x_2 \vee \neg x_3)$$
$$\omega_2 = (x_2) \qquad \omega_5 = (x_2 \vee x_3)$$
$$\omega_3 = (\neg x_2 \vee x_3) \qquad \omega_6 = (\neg x_1 \vee x_2 \vee \neg x_3)$$

Given formula φ, the new formula φ' is defined on variables $X \cup S = \{x_1, x_2, x_3, s_1, ..., s_6\}$ and clauses $\Omega' = \{\omega'_1, ..., \omega'_6\}$, such that:

$$\omega'_1 = (\neg s_1 \vee x_1 \vee \neg x_3) \qquad \omega'_4 = (\neg s_4 \vee \neg x_2 \vee \neg x_3)$$
$$\omega'_2 = (\neg s_2 \vee x_2) \qquad \omega'_5 = (\neg s_5 \vee x_2 \vee x_3)$$
$$\omega'_3 = (\neg s_3 \vee \neg x_2 \vee x_3) \qquad \omega'_6 = (\neg s_6 \vee \neg x_1 \vee x_2 \vee \neg x_3)$$

Observe that S variables can be interpreted as *clause selectors* which allow considering or not each clause ω_i. For example, assigning $s_2 = 0$ makes clause ω'_2 satisfied and therefore variable x_2 does not have to be assigned value 1, as it was for the original clause $\omega_2 = (x_2)$. Moreover, φ' is readily satisfiable by setting all s_i variables to 0. Now, for each assignment to the S variables, the resulting sub-formula may be satisfiable or unsatisfiable. For each unsatisfiable sub-formula, the number of S variables assigned value 1 indicates how many clauses are contained in the unsatisfiable core. Observe that this unsatisfiable core may further be reduced if we restrict the core to clauses involved in the derivation

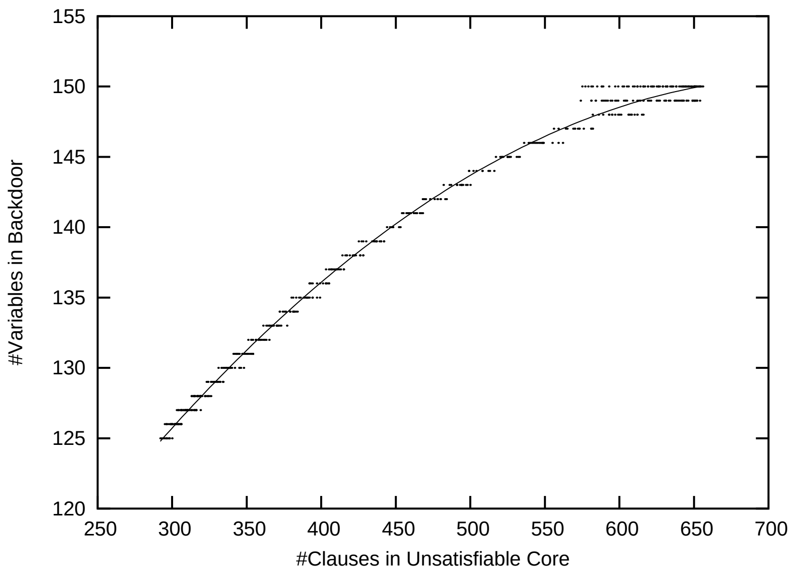


Figure 6.8: Regression on the number of clauses in the unsatisfiable cores and the number of variables in the backdoors for unsatisfiable random 3-SAT formulas with 150 variables

of the empty clause. The *minimum* unsatisfiable core is obtained from the unsatisfiable sub-formula with the *least* number of S variables assigned value 1.

One can adapt a state-of-the-art SAT solver to implement the proposed model. The problem instance variables are organized into two disjoint sets: the S variables and the X variables. Decisions are first made on the S variables (defining the S space) and afterwards on the X variables (defining the X space); hence, each assignment to the S variables defines a potential core. If for a given assignment all clauses become satisfied, then the search simply backtracks to the most recently S variable assigned value 0. Otherwise, each time the search backtracks from a decision level associated with an X variable to a decision level associated with a S variable, we have identified an unsatisfiable core, defined by the S variables assigned value 1. After all assignments to the S variables have been (implicitly) evaluated, the unsatisfiable core with the least number of utilized clauses corresponds to the minimum unsatisfiable core.

The key challenge of the proposed model is the search space. For the original problem instance the search space is 2^n, where n is the number of variables, whereas for the transformed problem instance the search space becomes 2^{n+m}, where m is the number of clauses. Nevertheless, a few key optimizations can be applied. First, the SAT-based algorithm can start with an upper bound on the size of the minimum unsatisfiable core. For this purpose, the algorithm proposed in (Zhang & Malik 2003) can be used. Hence, when searching for the minimum unsatisfiable core, we just need to consider assignments to the S variables which yield smaller unsatisfiable cores. This additional constraint can be modeled as a cardinality constraint. Furthermore, each computed unsatisfiable core can be used for backtracking *non-chronologically* on the S variables, thus potentially reducing the search space. Observe that an unsatisfiable core is computed whenever the search backtracks from the X space to the S space, meaning that there is no solution to the formula given the current S assignments, i.e. the original formula φ was proved to be unsatisfiable.

As usual, clause recording is used to reduce the search space. Interestingly, after a conflict that implies recording a clause that allows backtracking from the X space to the S space, an unsatisfiable core can be easily obtained from the new recorded clause.

Example 6.3 *Given formula φ' from Example 6.2, recording clause $\omega'_7 = (\neg s_2 \vee \neg s_3 \vee \neg s_4)$ means that the unsatisfiable core $\{\omega_2, \omega_3, \omega_4\}$ has been identified.*

Besides the *traditional* clause recording scheme (Marques-Silva & Sakallah 1996), where each new clause corresponds to a sequence of resolution steps, a new clause is recorded whenever *a solution is found.* The new clause contains all the S literals responsible for not selecting the corresponding clause [1], *except* for those clauses that would be satisfied by the X variables in the computed solution.

Example 6.4 *Consider again formula φ' from Example 6.2, and suppose that the current set of assignments is $\{s_1{=}0,\ s_2{=}0,\ s_3{=}1,\ s_4{=}1,\ s_5{=}0,\ s_6{=}1,\ x_1{=}1,\ x_2{=}0,\ x_3{=}0\}$. At this stage of the search, all the clauses are satisfied, and therefore a solution is found. Consequently, a new clause is recorded to avoid finding again the same solution and also to force finding an unsatisfiable core in the future. For this example, a new clause $\omega'_8 = (s_2 \vee s_5)$ is recorded. Observe that clause ω'_1 is satisfied by assigning $x_1 = 1$. The new*

[1] Such S literals are assigned value 1 in a clause that is part of the original specification.

clause means that for finding an unsatisfiable core either clause ω_2 or clause ω_5 has to be part of the formula.

Finally, observe that *minimal* unsatisfiable cores can also be obtained by this algorithm as long as the solver is given any unsatisfiable sub-formula instead of the whole formula.

A similar algorithm can be used to obtain a minimum strong backdoor. Again, the idea is to extract a strong backdoor from the corresponding unsatisfiable core. Besides having additional variables for selecting clauses, we also need a set T of new variables to be used as selectors for variables in the original formula. (Satisfying variable $t_i \in T$ implies variable x_i being part of a strong backdoor.) For each variable x_i a new constraint is added,

$$t_i \leftrightarrow \bigvee_{s \in S_i} s$$

where S_i is the subset of S variables occurring in clauses with x_i or $\neg x_i$. The *minimum strong backdoor* is obtained from the unsatisfiable sub-formula with the *least* number of T variables assigned value 1. With these additional constraints, we guarantee that a variable x_i is part of a strong backdoor *iff* a clause with x_i or $\neg x_i$ is part of a given unsatisfiable core.

Example 6.5 *Given formula φ' from Example 6.2, the CNF clauses to be added w.r.t. variable x_1 would be the following:*

$$(\neg t_1 \vee s_1 \vee s_6) \qquad (\neg s_1 \vee t_1) \qquad (\neg s_6 \vee t_1)$$

The proposed algorithm is able to identify minimum or minimal strong backdoors, depending on the input being either the original formula or an unsatisfiable sub-formula. A key optimization consists in using the size of the smallest strong backdoor extracted so far as a cardinality constraint.

The plot in Figure 6.9 gives the size of *minimal* unsatisfiable cores and strong backdoors as a percentage of clauses and variables in the formula, respectively. Results are restricted to the minimal - and not minimum - unsatisfiable cores and strong backdoors for unsatisfiable random 3-SAT formulas with only 50 variables due to the complexity of

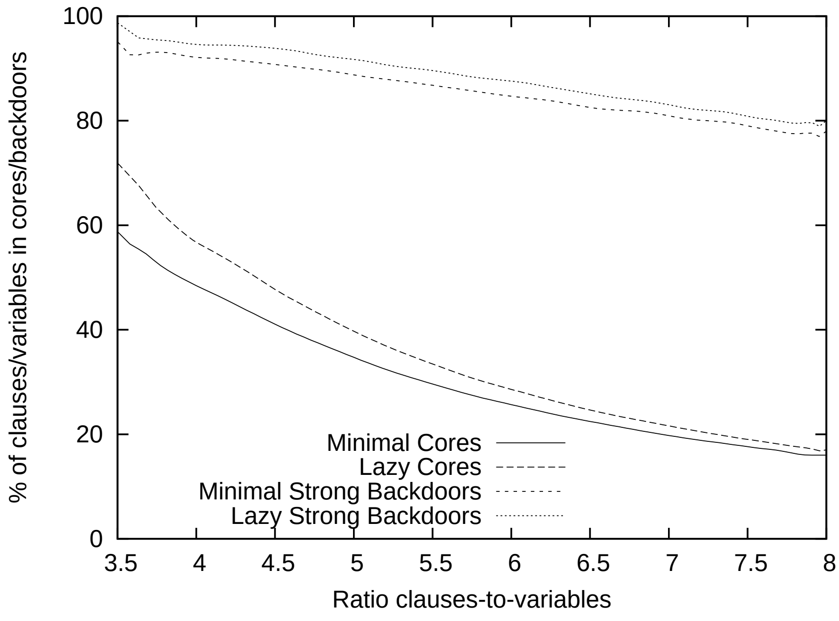


Figure 6.9: Size of minimal unsatisfiable cores and strong backdoors (%) for unsatisfiable random 3-SAT formulas with 50 variables, as a function of the ratio of clauses to variables

this optimization problem. However, we predict that similar figures would be obtained for minimum values and for instances with more variables.

Interesting conclusions may be drawn from Figure 6.9. First of all, it is clear that the values obtained by a lazy approach do not correspond to minimal values. Second, it is possible to relate the values for the lazy approach with the minimal values by an almost constant gap. Finally, this plot confirms that hardness can be related with hidden structure, i.e. hard unsatisfiable random 3-SAT formulas exhibit larger unsatisfiable cores and strong backdoors.

Nonetheless, the relation between hardness and strong backdoors is still not as clear as the relation between hardness and unsatisfiable cores. But we believe that there is an explanation for this: the sub-solver used on the definition of these strong backdoors if far from giving small backdoors. If we consider a sub-solver with unit propagation, then the implications between assignments have to be taken into account, and consequently the number of variables in the strong backdoor is probably reduced. Moreover, for formulas with a larger number of clauses, the number of implications is expected to be larger. Hence, one may expect the size of strong backdoors for formulas with a larger ratio of clauses to variables to be more reduced than the size of strong backdoors for formulas with a small ratio of clauses to variables.

6.7 Summary

Recent advances in propositional satisfiability include studying the hidden structure of a formula, e.g. backbones (Monasson *et al.* 1999) and small worlds (Walsh 1999). Moreover, studying the hidden structure of unsatisfiable formulas aims to explain why a given formula is unsatisfiable. Two empirical successful approaches have recently been proposed: extracting unsatisfiable cores and identifying strong backdoors. An unsatisfiable core is a subset of clauses that defines a sub-formula that is also unsatisfiable, whereas a strong backdoor defines a subset of variables which assigned with all values allow concluding that the formula is unsatisfiable.

The contribution of this chapter is two-fold.

First, we study the relation between the search complexity of unsatisfiable random 3-SAT formulas and the sizes of unsatisfiable cores and strong backdoors. For this purpose, we use an existing algorithm which uses an approximated approach for calculating these values. Experimental results indicate that the search complexity of unsatisfiable random 3-SAT formulas is related with the size of unsatisfiable cores and strong backdoors. However, this algorithm uses a lazy approach which means that the size of the extracted unsatisfiable cores and strong backdoors can be far from being minimum or even minimal.

Second, we introduce a new algorithm that optimally reduces the size of unsatisfiable cores and strong backdoors, thus giving more accurate results. Results obtained with this more accurate approach also suggest that hardness is related with the size of unsatisfiable cores and strong backdoors. Nonetheless, and similarly to the first approach, the relation between hardness and the size of strong backdoors is not as clear as the relation between hardness and the size of unsatisfiable cores. This is probably due to the sub-solver that is being used for extracting strong backdoors. Future research work should definitely consider more accurate sub-solvers with the aim of reducing the size of strong backdoors.

7

Conclusions and Future Work

The last few years have seen enormous progress in research on satisfiability. In 1997, a very well-known paper presented ten challenges in propositional reasoning and search (Selman, Kautz, & McAllester 1997). By the time, these challenges appeared as quite ambitious. During the next years, the improvements in satisfiability were remarkable. Hence, in 2003 two of the authors revisited these challenges, reviewed progress and offered some suggestions for future research (Kautz & Selman 2003).

Despite the worst-case exponential run time of all known algorithms, SAT solvers are now routinely used to solve different benchmark problems. Systematic methods can now easily solve real-world problems with thousands or tens of thousands of variables, while local search methods are known for solving satisfiable random 3-SAT problems with a thousand variables.

Progress in the performance of SAT solvers is often mentioned as the most remarkable progress in the area of SAT. However, we believe that SAT evolved as a whole, and so the progress of SAT solvers is simply part of that evolution. Clearly, it is easier to measure the advances in the performance of SAT solvers, rather than advances in the theoretical foundations. For example, the SAT competitions that have been run on the last years provide a clear picture on the evolution of SAT solvers. From one year to another, many instances that were not solved in one competition become easily solved in the next competition.

The contributions of our thesis allow a better understanding of the efficient implementation of SAT solvers, the structure of SAT problem instances and the organization of SAT algorithms. This dissertation describes four main contributions, which are summarized on the next paragraphs.

The first contribution corresponds to efficient implementations for backtrack search SAT solvers. State-of-the-art SAT solvers are very competitive, not only due to integrating sophisticated techniques and algorithms, but also for being carefully implemented. Observe that a negligible reduction in the time per decision can make the difference between solving or not solving a given problem instance in a reasonable amount of time. Efficient implementations for backtrack search SAT solvers with clause recording are based on *lazy* data structures. These data structures are lazy due to not allowing to know precisely the number of literals in a clause which are assigned value 0, assigned value 1 or not assigned. Nonetheless, unit and unsatisfied clause are always identified, meaning that unit propagation is always applied and conflicts are always identified, respectively.

Probing-based preprocessing formula manipulation techniques are also described in this dissertation. Probing allows establishing different scenarios by assigning both values to a variable and then applying unit propagation. Clearly, probing is an expensive technique, and therefore we decided to restrict probing to preprocessing in a first phase. Preprocessing techniques are expected to simplify ¡the formula, namely by reducing the number of variables and clauses, or by adding important clauses, thus reducing the search effort required to find a solution or prove unsatisfiability. Different preprocessing formula manipulation techniques can be applied based on the result of probing. Furthermore, probing provides a unified framework for applying well-know preprocessing techniques and also new preprocessing techniques.

Another contribution is unrestricted backtracking for satisfiability. Unrestricted backtracking allows to backtrack *without restrictions* in the search tree whenever a conflict is found. The idea is to give more freedom to the search, thus avoiding the characteristic *trashing* of backtrack search algorithms. Such an algorithm combines the advantages of local search with the advantages of systematic search. Unrestricted backtrack algorithms are incomplete, due to not backtracking to the most recent yet untoggled variable in the just recorded clause. Nonetheless, different conditions can be established for guaranteeing completeness, based on keeping some of the recorded clauses.

Another issue discussed in this dissertation is hidden structure in unsatisfiable random 3-SAT formulas. In this context, hidden structure is given by the size of unsatisfiable cores and strong backdoors, which can be extracted from any unsatisfiable formula. Both concepts are related, although unsatisfiable cores refer to clauses and strong backdoors refer to variables. Broadly, an unsatisfiable core is a subset of clauses that is still unsatisfiable, whereas a strong backdoor is a subset of variables which restrict the search to a space where unsatisfiability is proved. Our study allows to relate hardness, i.e. the number of searched nodes, with the size of unsatisfiable cores and strong backdoors. Moreover, we provide an algorithm for extracting small unsatisfiable cores and strong backdoors.

This dissertation has a number of limitations. Some of them are addressed along the dissertation. In a near future, we expect to improve some of the proposed techniques and eliminate the already identified limitations. For example, we should consider more powerful sub-solvers, i.e. allowing more constraint propagation, when identifying strong backdoors.

In addition, and albeit beyond the scope of this dissertation, some of the proposed formal results could be more thoroughly detailed, namely regarding unrestricted backtracking. Moreover, a more extensive and generic (see (Bhalla *et al.* 2003b)) experimental evaluation might be conducted to strongly support our conclusions.

We may also point out more generic limitations. For example, the probing-based techniques and the unrestricted backtracking algorithm are supposed to be useful for solving instances of specific problem domains. We should consider either improving these techniques or characterizing in more detail the classes where the techniques indeed work.

The recent progress in SAT is not a problem to the future of SAT. Conversely, progress brings more progress: answering to an open question opens even more questions. There are currently more open questions than there were some years ago. Also, there is much more interest in SAT than there was some years ago. SAT is now a more competitive research field with a more numerous community.

Our future work will certainly be developed in the context of SAT. We believe that after working on SAT-core issues we are now prepared not only to continue working on SAT but also to start working on SAT-related issues. Indeed, beyond-SAT issues seem to be currently more promising than SAT-core issues. Clearly, SAT foundations have been strengthened in the last years. So, having such strong and clear SAT foundations is a basis for extending these concepts to other areas. SAT is indeed effective for solving hard real-world problems, which has motivated encoding other problem domains as CNF formulas to extend the use of SAT (e.g. graph coloring and scheduling problems) and extending SAT to be used for solving other problems with more sophisticated formulations (e.g. quantified Boolean formulas). We believe that we will be able to make contributions to this field.

Bibliography

[Achlioptas *et al.* 2000] Achlioptas, D.; Gomes, C.; Kautz, H.; and Selman, B. 2000. Generating satisfiable instances. In *Proceedings of the National Conference on Artificial Intelligence*, 256–261.

[Aharoni & Linial 1986] Aharoni, R., and Linial, N. 1986. Minimal non two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory, Series A* 43:196–204.

[Aspvall, Plass, & Tarjan 1979] Aspvall, B.; Plass, M. F.; and Tarjan, R. E. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters* 8(3):121–123.

[Bacchus & Winter 2003] Bacchus, F., and Winter, J. 2003. Effective preprocessing with hyper-resolution and equality reduction. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 183–192.

[Bacchus 2002a] Bacchus, F. 2002a. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of the National Conference on Artificial Intelligence.*

[Bacchus 2002b] Bacchus, F. 2002b. Exploiting the computational tradeoff of more reasoning and less searching. In *Fifth International Symposium on Theory and Applications of Satisfiability Testing*, 7–16.

[Baptista & Marques-Silva 2000] Baptista, L., and Marques-Silva, J. P. 2000. Using randomization and learning to solve hard real-world instances of satisfiability. In Dechter, R., ed., *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, 489–494. Springer Verlag.

[Bayardo Jr. & Schrag 1997] Bayardo Jr., R., and Schrag, R. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, 203–208.

[Beame *et al.* 2002] Beame, P.; Karp, R.; Pitassi, T.; and Saks, M. 2002. The efficiency of resolution and Davis-Putnam procedures. *SIAM Journal on Computing* 31(4):1048–1075.

[Berre 2001] Berre, D. L. 2001. Exploiting the real power of unit propagation lookahead. In *LICS Workshop on Theory and Applications of Satisfiability Testing.*

[Bhalla *et al.* 2003a] Bhalla, A.; Lynce, I.; de Sousa, J.; and Marques-Silva, J. 2003a. Heuristic backtracking algorithms for SAT. In *Proceedings of the International Workshop on Microprocessor Test and Verification.*

[Bhalla *et al.* 2003b] Bhalla, A.; Lynce, I.; de Sousa, J.; and Marques-Silva, J. P. 2003b. Heuristic-based backtracking for propositional satisfiability. In *Proceedings of the Portuguese Conference on Artificial Intelligence.*

[Brafman 2001] Brafman, R. I. 2001. A simplifier for propositional formulas with many binary clauses. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

[Bruni & Sassano 2001] Bruni, R., and Sassano, A. 2001. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *LICS Workshop on Theory and Applications of Satisfiability Testing.*

[Büning 2000] Büning, H. K. 2000. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics* 107(1-3):83–98.

[Buro & Kleine-Büning 1992] Buro, M., and Kleine-Büning, H. 1992. Report on a SAT competition. Technical report, University of Paderborn.

[Cheeseman, Kanefsky, & Taylor 1991] Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. 1991. Where the really hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 331–337.

[Chvtal & Szemerédi 1988] Chvtal, V., and Szemerédi, E. 1988. Many hard examples for resolution. *Journal of the ACM* 35(4):759–768.

[Coarfa *et al.* 2003] Coarfa, C.; Demopoulos, D. D.; Aguire, A. S. M.; Subramanian, D.; and Vardi, M. Y. 2003. Random 3-SAT: The plot thickens. *Constraints* 8(3):243–261.

[Colbourn 1984] Colbourn, C. 1984. The complexity of completing partial latin squares. *Discrete Applied Mathematics* 8:25–30.

[Cook 1971] Cook, S. 1971. The complexity of theorem proving procedures. In *Proceedings of the Third Annual Symposium on Theory of Computing*, 151–158.

[Coudert 1996] Coudert, O. 1996. On Solving Covering Problems. In *Proceedings of the ACM/IEEE Design Automation Conference*, 197–202.

[Crawford & Auton 1993] Crawford, J. M., and Auton, L. 1993. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, 22–28.

[Davis & Putnam 1960] Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery* 7:201–215.

[Davis, Logemann, & Loveland 1962] Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the Association for Computing Machinery* 5:394–397.

[Davydov, Davydova, & Büning 1998] Davydov, G.; Davydova, I.; and Büning, H. K. 1998. An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF. *Annals of Mathematics and Artificial Intelligence* 23(3-4):229–245.

[Debruyne & Bessière 1997] Debruyne, R., and Bessière, C. 1997. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

[Dechter 1990] Dechter, R. 1990. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence* 41(3):273–312.

[Dubois & Dequen 2001] Dubois, O., and Dequen, G. 2001. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

[Fang & Ruml 2004] Fang, H., and Ruml, W. 2004. Complete local search for propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence.*

[Fleischner, Kullmann, & Szeider 2002] Fleischner, H.; Kullmann, O.; and Szeider, S. 2002. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science* 289(1):503–516.

[Freeman 1995] Freeman, J. W. 1995. *Improvements to Propositional Satisfiability Search Algorithms.* Ph.D. Dissertation, University of Pennsylvania, Philadelphia, PA.

[Freuder 1978] Freuder, E. C. 1978. Synthesizing constraint expressions. *Communications of the Association for Computing Machinery* 21:958–966.

[Gaschnig 1979] Gaschnig, J. 1979. *Performance Measurement and Analysis of Certain Search Algorithms.* Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA.

[Gelder & Tsuji 1993] Gelder, A. V., and Tsuji, Y. K. 1993. Satisfiability testing with more reasoning and less guessing. In Johnson, D. S., and Trick, M. A., eds., *Second DIMACS Implementation Challenge.* American Mathematical Society.

[Gelder 2002] Gelder, A. V. 2002. Generalizations of watched literals for backtracking search. In *Seventh International Symposium on Artificial Intelligence and Mathematics.*

[Gent & Walsh 1996] Gent, I. P., and Walsh, T. 1996. The satisfiability constraint gap. *Artificial Intelligence* 81(1-2):59–80.

[Ginsberg & McAllester 1994] Ginsberg, M. L., and McAllester, D. 1994. GSAT and dynamic backtracking. In *Proceedings of the International Conference on Principles of Knowledge and Reasoning*, 226–237.

[Ginsberg 1993] Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

[Goldberg & Novikov 2002] Goldberg, E., and Novikov, Y. 2002. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, 142–149.

[Gomes & Selman 1997] Gomes, C. P., and Selman, B. 1997. Algorithm portfolio design: Theory vs. practice. In *Proceedings of the Thirteenth Conference On Uncertainty in Artificial Intelligence.*

[Gomes & Shmoys 2002] Gomes, C., and Shmoys, D. 2002. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Generalizations.*

[Gomes *et al.* 2000] Gomes, C. P.; Selman, B.; Crato, N.; and Kautz, H. A. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* 24(1/2):67–100.

[Gomes, Selman, & Kautz 1998] Gomes, C. P.; Selman, B.; and Kautz, H. 1998. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*, 431–437.

[Groote & Warners 2000] Groote, J. F., and Warners, J. P. 2000. The propositional formula checker heerhugo. In Gent, I.; van Maaren, H.; and Walsh, T., eds., *SAT 2000.* IOS Press. 261–281.

[Hooker & Vinay 1995] Hooker, J. N., and Vinay, V. 1995. Branching rules for satisfiability. *Journal of Automated Reasoning* 15:359–383.

[Jeroslow & Wang 1990] Jeroslow, R. G., and Wang, J. 1990. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence* 1:167–187.

[Kautz & Selman 2003] Kautz, H., and Selman, B. 2003. Ten challenges *redux*: Recent progress in propositional reasoning and search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming.*

[Kunz & Stoffel 1997] Kunz, W., and Stoffel, D. 1997. *Reasoning in Boolean Networks.* Kluwer Academic Publishers.

[Li & Anbulagan 1997] Li, C. M., and Anbulagan. 1997. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 341–355.

[Li 2000] Li, C. M. 2000. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the National Conference on Artificial Intelligence*, 291–296.

[Lynce & Marques-Silva 2001] Lynce, I., and Marques-Silva, J. P. 2001. The puzzling role of simplification in propositional satisfiability. In *Proceedings of the EPIA Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving*, 73–86.

[Lynce & Marques-Silva 2002a] Lynce, I., and Marques-Silva, J. P. 2002a. Building state-of-the-art SAT solvers. In Harmelen, V., ed., *Proceedings of the European Conference on Artificial Intelligence*, 166–170. IOS Press.

[Lynce & Marques-Silva 2002b] Lynce, I., and Marques-Silva, J. P. 2002b. Complete unrestricted backtracking algorithms for satisfiability. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 214–221.

[Lynce & Marques-Silva 2002c] Lynce, I., and Marques-Silva, J. P. 2002c. Efficient data structures for backtrack search SAT solvers. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 308–315.

[Lynce & Marques-Silva 2003a] Lynce, I., and Marques-Silva, J. P. 2003a. The effect of nogood recording in DPLL-CBJ SAT algorithms. In O'Sullivan, B., ed., *Recent Advances in Constraints*, volume 2627 of *Lecture Notes in Artificial Intelligence.* Springer Verlag. 144–158.

[Lynce & Marques-Silva 2003b] Lynce, I., and Marques-Silva, J. P. 2003b. On implementing more efficient SAT data structures. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 510–516.

[Lynce & Marques-Silva 2003c] Lynce, I., and Marques-Silva, J. P. 2003c. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence* 37(3):307–326.

[Lynce & Marques-Silva 2003d] Lynce, I., and Marques-Silva, J. P. 2003d. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence.*

[Lynce & Marques-Silva 2004a] Lynce, I., and Marques-Silva, J. 2004a. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 305–310.

[Lynce & Marques-Silva 2004b] Lynce, I., and Marques-Silva, J. P. 2004b. Hidden structure in unsatisfiable random 3-SAT: an empirical study. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence.*

[Lynce & Marques-Silva 2005a] Lynce, I., and Marques-Silva, J. P. 2005a. Efficient data structures for backtrack search SAT solvers. *Annals of Mathematics and Artificial Intelligence* 43(1-4):137–152.

[Lynce & Marques-Silva 2005b] Lynce, I., and Marques-Silva, J. P. 2005b. Random backtracking in backtrack search algorithms for satisfiability. *Discrete Applied Mathematics.* To be published in 2005.

[Lynce, Baptista, & Marques-Silva 2001a] Lynce, I.; Baptista, L.; and Marques-Silva, J. P. 2001a. Stochastic systematic search algorithms for satisfiability. In *Proceedings of the LICS Workshop on Theory and Applications of Satisfiability Testing*, 1–7.

[Lynce, Baptista, & Marques-Silva 2001b] Lynce, I.; Baptista, L.; and Marques-Silva, J. P. 2001b. Towards provably complete stochastic search algorithms for satisfiability. In Brazdil, P., and Jorge, A., eds., *Proceedings of the Portuguese Conference on Artificial Intelligence*, volume 2258 of *Lecture Notes in Artificial Intelligence*, 363–370.

[Lynce, Baptista, & Marques-Silva 2001c] Lynce, I.; Baptista, L.; and Marques-Silva, J. P. 2001c. Unrestricted backtracking algorithms for satisfiability. In *Proceedings of the AAAI Fall Symposium Using Uncertainty within Computation*, 76–82.

[Mammen & Hogg 1997] Mammen, D. L., and Hogg, T. 1997. A new look at the easy-hard-easy pattern of combinatorial search difficulty. *Journal of Artificial Intelligence Research* 7:47–66.

[Marques-Silva & Glass 1999] Marques-Silva, J. P., and Glass, T. 1999. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of the ACM/IEEE Design, Automation and Test in Europe Conference*, 145–149.

[Marques-Silva & Sakallah 1996] Marques-Silva, J. P., and Sakallah, K. A. 1996. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, 220–227.

[Marques-Silva & Sakallah 1999] Marques-Silva, J. P., and Sakallah, K. A. 1999. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5):506–521.

[Marques-Silva 1999] Marques-Silva, J. P. 1999. The impact of branching heuristics in propositional satisfiability algorithms. In Barahona, P., and Alferes, J., eds., *Proceedings of the Portuguese Conference on Artificial Intelligence*, volume 1695 of *Lecture Notes in Artificial Intelligence*, 62–74. Springer-Verlag.

[Marques-Silva 2000] Marques-Silva, J. P. 2000. Algebraic simplification techniques for propositional satisfiability. In Dechter, R., ed., *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, 537–542. Springer Verlag.

[McAllester, Selman, & Kautz 1997] McAllester, D.; Selman, B.; and Kautz, H. 1997. Evidence of invariants in local search. In *Proceedings of the National Conference on Artificial Intelligence*, 321–326.

[McMillan 2003] McMillan, K. L. 2003. Interpolation and SAT-based model checking. In *Proceedings of Computer Aided Verification*.

[Monasson *et al.* 1999] Monasson, R.; Zecchina, R.; Kirkpatrick, S.; Selman, B.; and Troyansky, L. 1999. Determining computational complexity from characteristic phase transitions. *Nature* 400:133–137.

[Moskewicz *et al.* 2001] Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, 530–535.

[Nadel 2002] Nadel, A. 2002. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew University of Jerusalem.

[Novikov 2003] Novikov, Y. 2003. Local search for boolean relations on the basis of unit propagation. In *Proceedings of the Design and Test in Europe Conference*.

[Ostrowski *et al.* 2002] Ostrowski, R.; Grégoire, E.; Mazure, B.; and Sais, L. 2002. Recovering and exploiting structural knowledge from cnf formulas. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 185–199.

[Papadimitriou & Wolfe 1988] Papadimitriou, C. H., and Wolfe, D. 1988. The complexity of facets resolved. *Journal of Computer and System Sciences* 37(1):2–13.

[Prestwich 2000] Prestwich, S. 2000. A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 337–352.

[Prosser 1993] Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence* 9(3):268–299.

[Richards & Richards 2000] Richards, E. T., and Richards, B. 2000. Non-systematic search and no-good learning. *Journal of Automated Reasoning* 24(4):483–533.

[Robinson 1965] Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery* 12(1):23–41.

[Ruan, Kautz, & Horvitz 2004] Ruan, Y.; Kautz, H.; and Horvitz, E. 2004. The backdoor key: A path to understanding problem hardness. In *Proceedings of the National Conference on Artificial Intelligence*.

[Ryan 2004] Ryan, L. 2004. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University.

[Selman & Kautz 1993] Selman, B., and Kautz, H. 1993. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 290–295.

[Selman, Kautz, & McAllester 1997] Selman, B.; Kautz, H.; and McAllester, D. 1997. Ten challenges in propositional reasoning and search. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

[Selman, Levesque, & Mitchell 1992] Selman, B.; Levesque, H.; and Mitchell, D. 1992. A new method for solving hard satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, 440–446.

[Selman, Mitchell, & Levesque 1996] Selman, B.; Mitchell, D. G.; and Levesque, H. J. 1996. Generating hard satisfiability problems. *Artificial Intelligence* 81(1-2):17–29.

[Stallman & Sussman 1977] Stallman, R. M., and Sussman, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9:135–196.

[Stålmarck 1989] Stålmarck, G. 1989. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish

Patent 467 076 (Approved 1992), US Patent 5 276 897 (approved 1994), European Patent 0 403 454 (approved 1995).

[Tseitin 1968] Tseitin, G. S. 1968. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part II* 115–125.

[Uribe & Stickel 1994] Uribe, T. E., and Stickel, M. E. 1994. Ordered binary decision diagrams and the davis-putnam procedure. In *Proceedings of the First International Conference on Constraints in Computational Logics*, 34–49.

[Velev & Bryant 1999] Velev, M. N., and Bryant, R. E. 1999. Superscalar processor verification using efficient reductions from the logic of equality with uninterpreted functions to propositional logic. In *Proceedings of Correct Hardware Design and Verification Methods*, LNCS 1703, 37–53.

[Walsh 1999] Walsh, T. 1999. Search in a small world. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1172–1177.

[Williams, Gomes, & Selman 2003] Williams, R.; Gomes, C. P.; and Selman, B. 2003. Backdoors to typical case complexity. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

[Yokoo 1994] Yokoo, M. 1994. Weak-commitment search for solving satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence*, 313–318.

[Zabih & McAllester 1988] Zabih, R., and McAllester, D. A. 1988. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, 155–160.

[Zhang & Malik 2003] Zhang, L., and Malik, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Design and Test in Europe Conference*, 10880–10885.

[Zhang & Stickel 2000] Zhang, H., and Stickel, M. 2000. Implementing the Davis-Putnam method. In Gent, I.; van Maaren, H.; and Walsh, T., eds., *SAT 2000*. IOS Press. 309–326.

[Zhang 1997] Zhang, H. 1997. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, 272–275.