UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Algebraic Simplification Techniques for Propositional Satisfiability

Maria Inês Camarate de Campos Lynce de Faria
Graduated

Dissertation for the degree of
Master of Science in Electrical and Computer Engineering

Research Advisors: Prof. João Emílio Segurado Pavão Martins
Prof. João Paulo Marques da Silva

Committee: Prof. João Emílio Segurado Pavão Martins
Prof. Carla Pedro Gomes
Prof. João Paulo Marques da Silva
Prof. José João Henriques Teixeira de Sousa

March 2001

UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Técnicas de Simplificação Algébrica para Solvabilidade Proposicional

Maria Inês Camarate de Campos Lynce de Faria
Licenciada

Dissertação para obtenção do grau de
Mestre em Engenharia Electrotécnica e de Computadores

Orientadores: Prof. João Emílio Segurado Pavão Martins
Prof. João Paulo Marques da Silva

Júri: Prof. João Emílio Segurado Pavão Martins
Prof. Carla Pedro Gomes
Prof. João Paulo Marques da Silva
Prof. José João Henriques Teixeira de Sousa

Março 2001

Algebraic Simplification Techniques for Propositional Satisfiability

Maria Inês Camarate de Campos Lynce de Faria

Degree of Master of Science in Electrical and Computer Engineering

Research advisors
Prof. João Emílio Segurado Pavão Martins
Full Professor
Prof. João Paulo Marques da Silva
Assistant Professor

Abstract

Propositional Satisfiability (SAT) is fundamental in solving many application problems in Artificial Intelligence and in other fields of Computer Science and Engineering. The ability to reduce either the number of variables or clauses in instances of SAT impacts the expected computational effort of solving a target problem instance. In the past, it has already been shown that large simplifications can be obtained using algebraic simplification techniques. On the other hand, intelligent backtrack search algorithms for SAT have empirically been shown to be highly effective in pruning the amount of search, applying strategies for non-chronological backtracking and procedures for clause recording. In this thesis we propose integrating some algebraic simplification techniques — namely two-variable equivalence and clause inference — within intelligent backtrack search algorithms. When integrated in SAT algorithms, algebraic simplification techniques target the simplification of a problem instance, reducing the associated search cost. With the obtained experimental results we show the importance of the proposed techniques, demonstrating that they are indeed applied, both during preprocessing as well as during the search, for a wide variety of real-world problem instances.

Keywords

Propositional Satisfiability, Algebraic Simplification, Two-variable Equivalence, Reasoning, Non-Chronological Backtrack, Clause Recording

Técnicas de Simplificação Algébrica para Solvabilidade Proposicional

Maria Inês Camarate de Campos Lynce de Faria

Curso de Mestrado em Engenharia Electrotécnica e de Computadores

Orientadores
Prof. João Emílio Segurado Pavão Martins
Professor Catedrático
Prof. João Paulo Marques da Silva
Professor Auxiliar

Resumo

Os algoritmos de solvabilidade proposicional (SAT) são fundamentais para a resolução de inúmeras aplicações em Inteligência Artificial, bem como noutras áreas de Ciência da Computação e de Engenharia. É de prever que a capacidade de reduzir tanto o número de variáveis como o número de cláusulas em instâncias de SAT afecte o desempenho computacional na resolução dessas instâncias. No passado, já foi demonstrado experimentalmente que podem ser obtidas grandes simplificações usando técnicas de simplificação algébrica. Por outro lado, foi também demonstrado experimentalmente que algoritmos de procura com retrocesso inteligente para SAT são muito eficientes, pelo facto de efectuarem cortes na árvore de procura, através do uso de estratégias para retrocesso não cronológico e de procedimentos para a geração de cláusulas. Nesta tese propomo-nos integrar algumas técnicas de simplificação algébrica — nomeadamente equivalência de variáveis e inferência de cláusulas — em algoritmos de procura com retrocesso inteligente. Quando integradas em algoritmos de SAT, as técnicas de simplificação algébrica permitem a simplificação das instâncias, reduzindo o custo de procura associado. Com os resultados experimentais obtidos, ilustramos a importância das técnicas propostas, demonstrando que podem ser aplicadas tanto em pré-processamento como em procura para uma vasta gama de instâncias de problemas de SAT do mundo real.

Palavras-Chave

Solvabilidade Proposicional, Simplificação Algéblica, Equivalência de Variáveis, Raciocínio, Retrocesso Não Cronológico, Geração de Cláusulas

Acknowledgments

Deus quer, o homem sonha, a obra nasce.

Fernado Pessoa, O Infante,
In Mensagem, 1934

Eles não sabem, nem sonham,
que o sonho comanda a vida,
que sempre que um homem sonha
o mundo pula e avança
como bola colorida
entre as mãos de uma criança.

António Gedeão, Pedra Filosofal,
In Movimento Perpétuo, 1956

I would like to thank those that make me dream while being awake:

To *all* my family, for always being so close, even when I am more distant, not only during this work, but throughout each day of these twenty five years. In particular to my father, for his words of wisdom in the most decisive moments.

To Professor João Pavão Martins, for his support since my early days as a teaching assistant at Instituto Superior Técnico.

To Professor João Paulo Silva, with whom I learned how to do research work, for the constant encouragement, specially when my own motivation was not enough.

To my work colleagues.

To all my friends.

To my students.

Contents

List of Figures

List of Tables

1

Introduction

> "If you have to choose disagreed friends to sit at the same table, you can enter inside the kingdom of computational complexity. The most powerful computers can reveal inefficient." [12]

These lines were taken from an article recently published in a Portuguese well-known weekly newspaper. The author wanted to illustrate how common it is to face such kind of problems, and how surprisingly they can easily become complicated. In this case, if you invite A you also have to invite B, but it will be a disaster putting him together with C. On the other hand, inviting D implies an invitation to E, who will not come without F. And sometimes the alphabet is not enough to describe the social complexities...

This sort of problems can be encoded using propositional logic. Propositional logic is concerned with propositions and their relationships. The notion of a proposition here is intuitive and cannot be defined precisely. Roughly speaking, a proposition is a condition of the world about which we want to say something. The condition need not be true in order for us to talk about it. (In fact, we might want to say that it is false.) Propositional logic is also called Boolean logic because propositions have to be true or false.

Propositional Satisfiability (SAT) can be viewed as a special case of finite-domain Constraint Satisfaction Problem (CSP). CSP determines whether a set of constraints over a set of discrete variables can be satisfied. In SAT every variable can only assume the values true or false. Problems that are often formulated as CSP or SAT for the purposes of benchmarking include, among others, graph coloring, the n-queens problem, the tower of Hanoi, blocks world, quasigroup problems and many problems from real-world application domains.

Maybe these problems look simple at first sight. However, propositional satisfiability is a cornerstone in the theory of computational complexity [9]. Even if SAT had never been solved mechanically, it would still be of vital importance in the history of computing. It was the first decision problem shown to be NP-complete. NP-complete are a class of (decision) problems conjectured to require exponential time to be solved, in contrast with those that can be solved in polynomial time. But more than its privileged position in the complexity theory, the main aim of SAT interest lies in the ability to solving real-world instances in practice.

SAT is fundamental in solving many application problems in Artificial Intelligence — not only constraint satisfaction, but also automated reasoning [22], planning [37], robotics [18], vision interpretation [67], diagnosis [16] — and even in more areas — database querying [80], computer-aided design [62], computer manufacturing [23], and integrated circuit design [71]. As a result, real-world instances come from many of these fields.

There has been an explosion in SAT research in recent years. The strong relationship between the theory, the algorithms and the applications, motivated a growing interest and recent advances. It still is an open research area and nothing seems to announce an end to it.

Of course one may wonder why now if these problems have always been there. But sum it with the advances in technology, which allowed solving some problems and centering the attention in others more difficult to solve and therefore even more challenging.

At first, those who had empirical understanding of a specific problem domain would solve the problems of that area. By treating them separately, it was difficult to identify their common characteristics.

However, there has been made an effort, abstracting from specific problems details, to formalize all of them into conventional formulas. In practice, propositional logic began to

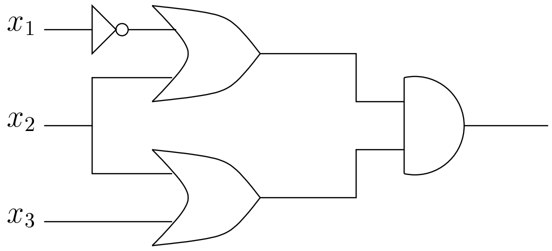


Figure 1-1: Example of a combinatorial circuit.

be used as a framework for knowledge representation and problem solving. Consequently, analyzing those problems as a whole made possible a unified perspective. Based on their structure, new techniques were proposed that benefit the entire domain.

Despite many years of effort, it remains an open question: "Can satisfiability problems be solved efficiently in practice?". The fact is that a first negative answer to a concrete problem can become affirmative later on. So we tend to propose a more ambitious formulation: "What has to be done in order to efficiently solve an instance of SAT?" To such a question we have different answers depending on the properties of the concrete problem instance.

Recent years have seen the proposal of several effective approaches for solving propositional satisfiability. These approaches include, among others, local search and its variations [72, 73], backtrack search improved with different search pruning techniques [6, 58, 87], backtrack search with randomization and restarts [29], continuous formulations [74] and algebraic manipulation [30, 41, 77]. These different approaches have allowed efficiently solving different classes of SAT instances.

To illustrate what has been explained so far, let us start by a naive approach, introducing a specific problem and expressing it by means of propositional logic. One important application of Boolean constraints is for modeling logic circuits, and therefore we present a combinational circuit in figure 1-1. This circuit can be modeled by the propositional formula: $(\neg x_1 \lor x_2) \land (x_2 \lor x_3)$.

Variables x_1, x_2 and x_3 represent Boolean variables, which can be assigned true or false. We are looking for a satisfying assignment — a truth assignment to the variables

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

Table 1.1: Truth-table for the combinatorial circuit.

such that the formula becomes satisfied. A possible approach to find a solution is to construct the truth table (see table 1.1), assigning all possible values and choosing the variable assignments that make the final formula true.

The conclusion is there are five satisfying assignments. Everything seems clear at this point. Nonetheless, observe that the number of possible assignments 2^n depends on the number of variables n. Using search strategies and considering instances from real-world problems, a high number of variables significantly affects the cost of the search process.

Obviously the use of a truth table is not the best choice to find a solution to an instance of the SAT problem. You blindly give values to the variables, next to the constraints, and finally choose the variable assignments that make all the constraints true.

On the other hand, doing it in an incremental way, using search strategies, would be more efficient, as long as we stop searching when we find a solution. Consider a given formula and the different variable assignments. The combination of values the variables can assume define the search space. The solution will be a path from the initial state to a state that satisfies the original formula.

Another alternative is to use algebraic simplification techniques and conjugate these techniques with others. In concrete terms, the work developed in this thesis consists in the integration of simplification techniques in a search-based SAT algorithm.

Algebraic simplification techniques target the simplification of a problem instance, reducing the search cost associated. Search algorithms for SAT already incorporate some techniques to deduce information after the splitting phase. Using simplification rules it is possible to prune the search space, but remember that the search cost is not the only side of the coin. The other side is the effort made to obtain simplifications. In the end, we want to conclude we save more time in search than loose in simplifications.

Algebraic simplification is directly related with problem formulation. Notice that the same problem, with no loss of information, can be specified in a variety of formulas. When encoding a real-world problem, one usually structures the problem in simpler ones. This often results in a number of redundancies in the final SAT instance. Finding equivalences between variables can reduce the dimension of the problem and consequently the required amount of search. In some instances these differences will not significantly affect the efficiency of the approach.

Experimental results have already demonstrated that large simplifications can be obtained using the proposed techniques [50]. Nevertheless, they have only been applied in the form of a preprocessing engine. The next natural step is to incorporate the techniques in specific search algorithms. However, for backtrack search algorithms that implement some form of learning, special details must be considered when applying simplification techniques that replace variables at each step of the search process.

This thesis is organized as follows. Chapter 2 provides the underlying framework to understand the main topics focused throughout the dissertation. Apart from the introduction of definitions and terminology, we briefly review theoretical aspects and discuss propositional satisfiability. Next, we present the existing simplification rules for proposition satisfiability.

Chapter 3 gives more insight into algebraic simplification. We review previous work and explain with more detail the simplification techniques that were applied in the work developed in this dissertation.

The fourth chapter focuses on complete search algorithms for propositional satisfiability. Starting with the Davis-Putnam procedure, a generic backtrack search algorithm is presented. Branching heuristics and necessary assignments are also discussed, since they play an important role in search algorithms. Then, we explain more deeply intelligent backtrack search, namely the fundamental concepts associated: clause recording and non-chronological backtracking. We conclude the chapter by presenting GRASP as an example of a SAT solver incorporating several search-pruning techniques.

Next we describe the integration of algebraic simplification techniques in a SAT solver in Chapter 5. The obtained results are presented and analyzed in Chapter 6. Finally, in Chapter 7, we conclude and suggest further work.

2

Fundamental Concepts

This chapter introduces the underlying framework necessary to describe the main topics covered in this thesis. With this work we are going to propose the integration of algebraic simplification in backtrack search algorithms, and therefore we provide a simple introduction to related concepts.

We start with some definitions and the associated notation used throughout the dissertation. Next, we introduce Propositional Satisfiability (SAT). We describe the most relevant aspects and highlight open research areas. Finally, we provide some insight on existing simplification rules that are usually applied to Propositional Satisfiability.

2.1 Definitions and Notation

Let us start by introducing some definitions and the corresponding notation used throughout the dissertation.

A CNF formula φ on n binary variables $x_1, \ldots, x_n$ is a conjunction of m clauses $\omega_1, \ldots, \omega_m$. A clause ω is a disjunction of literals. A literal l is a complemented or uncomplemented Boolean variable. In other words, a literal is a variable x or its negated form $\neg x$. Sometimes, we will also refer to a formula as a "clause database", since a formula is a set of clauses. Similarly, a clause can be referred to as a set of literals.

Example 2.1 *The CNF formula* $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_2)$ *on variables* x_1, x_2, x_3 *and* x_4 *has three clauses* $\omega_1 = (x_1 \vee \neg x_2)$, $\omega_2 = (\neg x_1 \vee \neg x_3 \vee x_4)$, *and* $\omega_3 = (x_2)$.

The first one has two literals, the second has three literals and the last one has one literal.

Let $U = \{x_1, \ldots, x_n\}$ be a set of n Boolean variables. Considering U is obtained from a CNF formula φ on binary variables, we can also denote $U = vars(\varphi)$. A (partial) truth assignment for U is a (partial) function $A : U \rightarrow \{true, false\}$. For brevity, we sometimes refer to truth value $true$ as T or 1 and $false$ as F or 0. A can therefore be represented by a set of variable/value pairs. An assignment for a formula for a formula φ on n binary variables is complete if all the variables have corresponding values ($\mid A \mid= n$). Otherwise is partial.

The unassigned literals of a clause are referred to as its *free literals.* A clause containing just one free literal is called a *unit clause.* If it contains two free literals, it is a binary clause. A clause with exactly three free literals is a ternary clause.

Example 2.2 *Consider again the example formula* $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_2)$ *and an empty assignment. Hence,* $\omega_3 = (x_2)$ *is a unit clause,* $\omega_1 = (x_1 \vee \neg x_2)$ *is a binary clause and* $\omega_2 = (\neg x_1 \vee \neg x_3 \vee x_4)$ *is a ternary clause. Take now a possible partial assignment* $A = \{(x_1, 1), (x_2, 0)\}$ *for the CNF formula* $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_2)$. *So,* ω_2 *becomes a binary clause (*x_3 *and* x_4 *are its free literals).*

Let φ be a formula, $U = vars(\varphi)$ and ω a clause in φ. A truth assignment A for U satisfies ω if and only if at least one literal l in ω is true under A, i.e., a clause ω is *satisfied* if at least one of its literals assumes value 1. On the opposite, a clause ω is *unsatisfied* if all of its literals assume value 0. An unsatisfied clause is also called an empty clause. A clause that is neither satisfied nor unsatisfied is *unresolved.* A tautologous clause is a clause that contains both a literal and its complement. Obviously it is always satisfied.

Example 2.3 *Assume again the assignment* $A = \{(x_1, 1), (x_2, 0)\}$ *for the formula* $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_2)$. *Considering* $\omega_1 = (x_1 \vee \neg x_2)$, *both literals in* ω_1 *are assigned value 1. For* $\omega_3 = (x_2)$, x_2 *is assigned value 0. Hence,* A *causes* $\omega_1 = (x_1 \vee \neg x_2)$ *to be satisfied,* $\omega_2 = (\neg x_1 \vee \neg x_3 \vee x_4)$ *to be unresolved and* $\omega_3 = (x_2)$ *to be unsatisfied.*

Considering a formula φ and $U = vars(\varphi)$, we say that $A : U \rightarrow \{true, false\}$ satisfies φ if and only if it satisfies every clause in φ. This means a formula φ is satisfied if there is an assignment A of truth values to the variables which satisfies all of its clauses. If the formula is true for every possible assignment to the variables, it is called a tautology. Conversely, if it is false for every possible assignments it is said to be *unsatisfiable*. Otherwise it is *satisfiable*.

Therefore, evaluating a formula φ for a given truth assignment A yields three possible outcomes:

- $\varphi_{|A} = 1$: the formula is satisfied and it is a satisfying assignment or solution;

- $\varphi_{|A} = 0$: the formula is unsatisfied and it is an unsatisfying assignment;

- $\varphi_{|A} = X$: the formula is neither satisfied nor unsatisfied; it has satisfied clauses and unresolved clauses. Observe that this is only possible if A is a partial assignment.

2.2 Propositional Satisfiability

Propositional satisfiability (SAT) is the problem of finding an assignment to the variables in a propositional formula that makes the formula true, or proving that no such assignment exists. Much research in SAT considers problems in Conjunctive Normal Form (CNF). This is so common that we usually assume, when working in the SAT domain, that formulas are in CNF.

Algorithms for SAT can basically be subdivided in two categories.

First, there are the complete algorithms. Such algorithms find one or more satisfying solutions or prove that the formula under consideration is unsatisfiable. Examples of such algorithms are usually variants of the Davis-Putnam procedure [14, 15], that will be described in a specific section. Despite its simplicity and age, this procedure remains one of the best complete procedures for satisfiability. Some state-of-the-art complete algorithms are GRASP [58], rel_sat [6], SATO [87] and satz [43].

Second, there are the incomplete algorithms. These cannot prove unsatisfiability. If the search terminates with a satisfying assignment, the formula is proved to be satisfiable and we have a solution. Most incomplete algorithms find a solution (or perhaps several solutions) in favorable cases, but give up or do not terminate in other cases. In such cases one does not know whether an instance has no solution or the algorithm did not search hard enough. The two probably best known local search algorithms for SAT are GSAT [4] and the more recent WalkSAT [5]. GSAT starts with a randomly generated truth assignment, and evolves by changing the variable assignment which gives the largest increase in the number of clauses satisfied. WalkSAT essentially adds restarts and a heuristic to choose between variables from unsatisfied clauses. Other approaches to local search for SAT include, among others, genetic algorithms [44], neural networks [75], simulated annealing [76] and tabu search [2].

The main interest of local search algorithms relies on the fact that there are instances that are so difficult to solve that a complete algorithm cannot reach a conclusion in a reasonable amount of time. This result suggests the challenge of extending these incomplete methods to complete ones. Such implementation certainly requires some propositional reasoning to be carried out and remembered in a complete procedure.

Evaluating satisfiability algorithms [53] is a difficult task. Interestingly (and also comprehensibly), authors proposing different approaches for solving SAT usually resort to different problem instances to claim the effectiveness of those different approaches.

The first difficulty faced when evaluating SAT algorithms refers to the instances origin. One can distinguish between empirical characterization based on random distributions of problem instances and experimental evaluation given instances from real-world applications. These two areas should be considered separately, since studies on one of them cannot be generalized to the other one.

Empirical studies are important for understanding the complexity of the problem. For example, special techniques have been developed for solving restriction of SAT to instances where all clauses have k literals, denoted k-SAT. Of special interest are 2-SAT and 3-SAT: 3 is the smallest value of k for which k-SAT is NP-complete [9], while 2-SAT is solvable in linear time [1]. It is also expected for randomly generated problem instances to have a practical impact in solving real-world instances. Nonetheless, real-world instances are often large, hard and structured, and most state-of-the-art SAT solvers are currently unable to solve some instances in a reasonable period of time.

Analyzing different SAT solvers, we can conclude that the performance of SAT algorithms critically depends upon the care taken in the implementation. Tableau [13] and a recent version of SATO [88] are examples of algorithms which introduce significant improvements due to specific data structures and implementation details.

Apart from the techniques associated with SAT solvers, there are other areas for improving SAT algorithms. If a decision problem is NP-complete, we can in theory translate it into SAT. And the fact is that many problems of practical interest have been solved by encoding them into SAT (e.g. finding a key for cryptographic algorithms [61] and assigning a channel for cellular mobile networks [69]). In a near future, we expect more decision problems to be translated to CNF formulas. Henceforth, more research in AI domains of representation and problem formulation is required.

Furthermore, many of the algorithms developed for SAT are being extended to languages which are richer than the purely propositional, namely modal logic [27, 28] and first-order logic [89].

2.3 Simplification Rules for Propositional Satisfiability

In this section we describe efficient operations that can be carried out to simplify a CNF formula. By simplification of a formula we mean in general the removal of variables and clauses, or the addition of clauses that are likely to contain useful information.

Basically, we review well-known formula simplification techniques, which are usually included in the different SAT solvers. Based on these techniques, we present advanced simplification techniques in the next chapter.

2.3.1 Resolution

Resolution is an inference rule for propositional logic and can be regarded as a general technique for deriving new clauses.

Given two clauses $w_a = (v \vee x_1 \vee ... \vee x_m)$ and $w_b = (\neg v \vee y_1 \vee ... \vee y_n)$, where all x_i and y_j are distinct, resolution allows deriving the resolvent clause $w_c = (x_1 \vee ... \vee x_m \vee$

$y_1 \vee ... \vee y_n$), that is, the disjunction of w_a and w_b without v and $\neg v$. The resolvent is a logical consequence of the conjunction of the two clauses.

The resolution proof procedure is the process of repeatedly generating resolvents from original clauses and previously generated resolvents until either an empty clause is derived or until no more resolvents can be created.

Resolution provides a complete system for proof by refutation [70]. Unfortunately, this procedure requires exponential space in general, due to the generation of an exponential number of clauses in the general case. Nonetheless, the order in which clauses are resolved can have a critical effect on the required effort needed, because it affects the number and size of the created clauses. Observe that applying resolution we need to generate a very large number of redundant and often irrelevant conclusions in the worst case.

As a consequence of general resolution being computationally too expensive, some strategies have been developed. The main goal is to avoid useless work by not performing certain unnecessary deductions. In practice, we have to choose carefully each pair of clauses to resolve. The idea is to produce shorter and shorter sentences, decreasing the number of necessary resolvents to reach a conclusion.

Developed strategies indirectly identify which resolution operations to apply. In other words, they consist in particular ways of applying *selective* resolution. Considering selective forms of resolution, the main advantage is only simplified formulas can be derived. Restrictions aim to shorten the amount of time needed to compute a resolution derivation. One limits the number of possible resolvents to choose from at each resolution step.

A common strategy to perform selective resolution consists in identifying a *set of support* [40]. Consider a given formula $\mathcal{F}$ split into two sets of clauses $\mathcal{F}_{SS}$ and $\mathcal{F}_{SAT}$. Consider $\mathcal{F}_{SS}$ is the set of support. Every resolution combines a clause from $\mathcal{F}_{SS}$ with another from $\mathcal{F}_{SAT}$. This implies that at least one of the clauses being resolved at every step is selected from a set of support $\mathcal{F}_{SS}$.

In order to guarantee the procedure completeness, we have to choose the set of support $\mathcal{F}_{SS}$ so that the remainder of the clauses $\mathcal{F}_{SAT}$ are jointly satisfiable. Due to this requirement, this kind of resolution can be useful if a large portion of the given formula is easily determined to be satisfiable.

2.3.2 The Unit Clause Rule

The unit clause rule is applied whenever a unit clause $w = (l_i)$ is identified. In this case, all clauses containing l_i are declared satisfied, and literal $\neg l_i$ is removed from the clauses containing it. This results from assigning l_i value 1, thus satisfying all clauses with literal l_i, and setting to zero all literals $\neg l_i$. Observe that a unit clause has exactly this way for being satisfied.

Following the same reasoning, an unresolved clause is unit if it has exactly one unassigned literal. Take for example the clause $w = (l_1 \vee l_2 \vee \neg l_3)$ and the current assignment $l_1 = 0$ and $l_3 = 1$. In this case, we must assign $l_2 = 1$ to satisfy w.

Basically, this strategy applies resolution where one of the clauses is a unit clause. Remember that this is in agreement with the idea of preferring inferences that produce shorter clauses, in order to decrease the number of generated resolvents that lead to a conclusion.

2.3.3 The Pure Literal Rule

The pure literal rule was first mentioned in [21] as a simplification technique for formulas in conjunctive normal form. It is reported that *"variables in partial state can be immediately determined"*. A variable is considered to be in full state if it occurs both complemented and uncomplemented; otherwise, it is in partial state.

The algorithm consists in first detecting the literals whose occurrences are either all positive or all negative. Those literals are named pure. Next the pure literals become satisfied, assigning the variables with the correspondent truth value. Due to such assignments, all clauses containing them become satisfied.

2.3.4 Subsumption

Subsumption [70] is another simplification technique.

If the literals in one clause are a subset of those in another clause, then the smaller clause is said to *subsume* the larger one. Simply speaking, the smaller clause is more specific than the larger one.

Any assignment of values to variables that satisfies the smaller clause also satisfies the larger one. So, the larger clause can be dropped without changing the set of solutions.

For example, consider the clauses $w_a = (x_1 \vee \neg x_2 \vee x_3)$ and $w_b = (x_1 \vee x_3)$. In this case, clause w_b subsumes clause w_a, and therefore clause w_a can be eliminated.

Subsumption is of particular importance in resolution algorithms because resolution tends to produce large clauses. Subsumption helps keep the clause database small, and consequently helps keep the search space small.

Subsumption can only allow removing clauses, but will not induce additional simplifications. For this reason, when integrated with other simplification techniques, subsumption is only to be used after *all* techniques have been applied and yield no further simplification of the formula.

2.3.5 Two-variable Equivalence

Two-variable equivalence is a well-known formula simplification procedure. Once determined $x \equiv y$, a formula can be simplified, replacing all occurrences of y by x. As a consequence, the number of variables in the formula decreases.

Let us consider an example, using the pair of clauses $w_a = (\neg x \vee y)$ and $w_b = (x \vee \neg y)$. For any assignment that satisfies the two clauses, the truth values of x and y must be the same. For this reason, variables x and y are said to be equivalent. Therefore one can replace y by x on all occurrences of y. This reduces the number of variables in the CNF formula and usually causes clauses to be subsumed. Furthermore, clauses that originate a pair of equivalent variables become tautologous clauses.

3

Algebraic Simplification

The ability to reduce either the number of variables or clauses in SAT instances impacts the expected computational effort of solving a given instance. This ability can actually be essential for specific and hard classes of instances.

The objective of this chapter is to describe several simplification techniques that can effectively simplify instances of the satisfiability problem.

We start by reviewing algorithms that incorporate simplification techniques (section 3.1). This description of previous work is based on well-known simplification techniques — resolution, the unit clause rule, the pure literal rule, subsumption and two-variable equivalence — that were explained in the last chapter.

Next, the techniques developed in [50] are explained. We extend the existing formula simplifications, namely generalizing the pure literal rule (section 3.2), analyzing sequences of binary clauses (section 3.3), defining a support-set variable equivalence (section 3.4) and generalizing inference of binary clauses (section 3.5).

Finally, in section 3.6, we discuss the integration of algebraic simplification techniques in SAT algorithms, not only as a preprocessing engine, but also as an extension to the usual boolean constraint propagation in backtrack search SAT algorithms.

3.1 Previous Work

This section covers different algorithms that proposed special techniques, applying various inference rules to simplify the formula under consideration as much as possible. We will start by a brief reference to the simplifications used in the Davis-Putnam procedure, followed by the definition of *strongly connected components*, proposed in [1]. Next, we address Van Gelder and Tsuji's work [25], and then the algorithm of Warners and van Maaren [85]. Furthermore, we present simplification rules incorporated in Heerhugo [30] and EqSatz [41], two state-of-the-art SAT solvers. Finally, we introduce the recent polynomial-time preprocessor Compactor[1] from Chu Min Li, which is adapted from James Crawford's Compact algorithm[2].

Since early days, algorithms proposed for satisfiability have been associated with algebraic simplification techniques. The Davis-Putnam procedure [14, 15], which was the first complete algorithm proposed for satisfiability, incorporated both resolution, the unit clause rule and the pure literal rule. Despite the simplicity and age of the Davis-Putnam procedure, it has established the basis for the algorithms proposed later on. Therefore, such algorithms have only small variations.

Another simplification technique consists in identifying strongly connected components. Start by remembering that the pair of clauses $w_a = (\neg x \vee y)$ and $w_b = (x \vee \neg y)$ represents a two-variable equivalence. Observe that clauses of length two can also be viewed as implications; i.e., $(x \vee y)$ is equivalent to $(\neg x \rightarrow y)$ and $(\neg y \rightarrow x)$. Moreover, additional two-variable equivalence conditions can be established, as clarified by example 3.1.

Example 3.1 *Let us consider the sub-formula* $\varphi_a = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee x)$*, which basically indicates* $(x \rightarrow y) \wedge (y \rightarrow z) \wedge (z \rightarrow x)$ *(and* $(\neg y \rightarrow \neg x) \wedge (\neg z \rightarrow \neg y) \wedge (\neg x \rightarrow \neg z)$*). Clearly, the value of variables* x*,* y *and* z must *be the same, i.e.* x*,* y *and* z *are equivalent. Hence, all but one of these variables can be eliminated and replaced by the remaining*

[1] Available for download from http://www.laria.u-picardie.fr/ cli/my_compact.c.

[2] Available for download from http://www.cirl.uoregon.edu/crawford/ntab.tar, including both the SAT solver Tableau [13] and the preprocessor Compact.

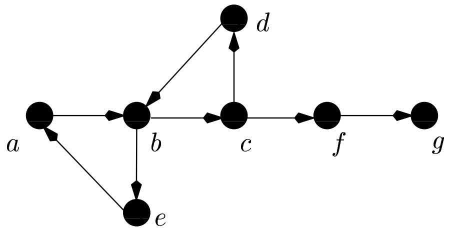


Figure 3-1: Example of implication graph.

variable. Notice that selective resolution allows deriving the same conclusion.

Given a CNF formula, it is computationally inexpensive to identify *chains* of binary clauses that cause a given set of variables to be equivalent. Basically, by representing binary clauses in terms of an *implication graph* (defined below), we just need to identify the *strongly connected components* (SCCs) of the implication graph, as originally used by Aspvall, Plass and Tarjan [1] as part of a linear time algorithm for 2CNFSAT. Observe that the identification of SCCs in a directed graph can be done in linear time [78].

Definition 3.1 (Implication Graph) *For a CNF formula φ, the implication graph is defined as a directed graph $G = (V, E)$. The set of vertices, V, is given by every variable x_j in φ and its complement $\neg x_j$. The set of edges E is obtained from the binary clauses of φ. Each binary clause $(l_1 \vee l_2)$ corresponds to two edges in E, $(\neg l_1, l_2)$ and $(\neg l_2, l_1)$, denoting the two implications captured by the binary clause.*

Example 3.2 *Let us illustrate the construction of the implication graph and SCC identification on the following example CNF formula:*

$(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee b) \wedge (\neg b \vee e) \wedge (\neg e \vee a) \wedge (\neg c \vee f) \wedge (\neg f \vee g)$

The implication graph for this CNF formula is shown in Figure 3-1[3]*. The SCC for the implication graph is composed of vertices $\{a, b, c, d, e\}$. Hence, the five variables can be deemed equivalent and replaced by a single variable, e.g. a.*

One of the first attempts to integrate two-variable equivalence in the resolution of SAT problems was made by Van Gelder and Tsuji [25]. The algorithm applies reasoning in

[3] *By definition, clause $(\neg a \vee b)$ yields two edges in the implication graph, (a, b) and $(\neg b, \neg a)$. For our purposes in this example, it suffices to represent only one of the edges, relating the positive literals.*

the form of certain resolution operations. In concrete, the proposed algorithm applies subsumption and identifies equivalent literals, using the technique of strongly connected components. Experimental results, obtained by the authors on randomly generated instances and several DIMACS benchmarks [36], indicate that, for large problems, the time investment in reasoning returns a profit in reduced searching, and the profit increases with increasing problem size.

After the work of Van Gelder, Warners and Van Maaren proposed another algorithm [85] to solve a class of hard satisfiability problems[4]. In a first phase, the formula is translated into a conjunction of (nested) equivalences, using a polynomial time recognition algorithm. A formula in conjunction of equivalences form (CoE) is a conjunction of *equivalency clauses*, which are defined as a (nested) equivalence of literals or its negation. CoE formulas are solvable in polynomial time [46], as opposed to formulas in CNF which are in general NP-complete [9]. In the next phase, as a consequence of having already identified the existing equivalences, the search-space is considerably reduced. This second part of the algorithm is an extension of the Davis-Putnam procedure.

Some state-of-the-art SAT solvers also apply similar techniques, namely HeerHugo [30] and EqSatz [41]. HeerHugo [30] removes implication cycles, motivated by the occurrence of clauses of length two. Moreover, HeerHugo applies subsumption and selective resolution whenever derives a new clause. EqSatz [41] defines six inference rules to solve satisfiability problems involving *equivalency clauses*, from which we can obtain variable equivalences. An equivalency clause of length k can be written as $l_1 \leftrightarrow l_2 \leftrightarrow \ldots \leftrightarrow l_k$, thus being equivalent to 2^{k-1} CNF clauses. *Equivalency reasoning* consists in the application of these rules in a formula.

More recently, Compactor was developed by Chu Min Li to simplify a formula during preprocessing. Basically, it performs operations that some SAT solvers apply during search:

1. Addition of resolvents of length < 4. For example, if $\omega_a = (x_1 \vee x_2 \vee x_3)$ and

[4] In concrete, these instances arise from the parity learning problem on 32 bits, and are available in the DIMACS suite of SAT benchmarks [36].

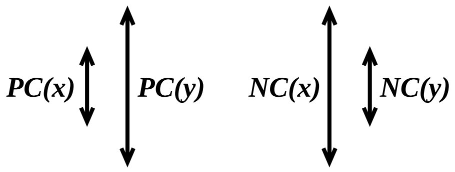


Figure 3-2: Clause sets for generalizing the Pure-Literal Rule.

$\omega_b = (\neg x_1 \vee x_2 \vee x_4)$ are clauses from the initial formula φ, clause $\omega_c = (x_2 \vee x_3 \vee x_4)$ is added into φ.

2. Elimination of pure literals.

3. Detection of failure literals, i.e., if assigning literal l value 1 leads to a contradiction by unit resolution, then set l to false.

Compactor operations are very similar to those from Compact. The main differences concern the implementation details, since Compactor has more concern for efficiency.

3.2 Generalizing the Pure-Literal Rule

In this section we describe generalizations to the pure literal rule. By applying the pure literal rule, 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 pure literal rule is useful for eliminating useless variables. Moreover, it is possible to obtain extensions with two variables, establishing relations between sets of clauses.

Consider a variable x and the sets of clauses where x occurs either as a positive or negative literal. For this purpose $PC(x)$ and $NC(x)$ denote, respectively, the set of clauses where variable x occurs as a positive and as a negative literal. Suppose we have the situation described by figure 3-2. Hence, the assignment $x = 0$ and $y = 1$ satisfies all clauses in $PC(y) \cup NC(x) \supseteq PC(x) \cup NC(y)$.

Similarly, generalizations can be obtained for other relations between sets of clauses. Let φ be a CNF formula, and let x_j and x_k be two of its variables. Then, according to the relation between $PC(x_j)$, $NC(x_j)$, $PC(x_k)$ and $PC(x_k)$ the following conclusions can be drawn:

1. If $PC(x_k) \subseteq PC(x_j)$ and $NC(x_j) \subseteq NC(x_k)$, then φ is satisfiable if and only if $\varphi - PC(x_j) - NC(x_k)$ is satisfiable.

2. If $PC(x_j) \subseteq PC(x_k)$ and $NC(x_k) \subseteq NC(x_j)$, then φ is satisfiable if and only if $\varphi - NC(x_j) - PC(x_k)$ is satisfiable.

3. If $PC(x_j) \subseteq NC(x_k)$ and $PC(x_k) \subseteq NC(x_j)$, then φ is satisfiable if and only if $\varphi - NC(x_j) - NC(x_k)$ is satisfiable.

4. If $NC(x_k) \subseteq PC(x_j)$ and $NC(x_j) \subseteq PC(x_k)$, then φ is satisfiable if and only if $\varphi - PC(x_j) - PC(x_k)$ is satisfiable.

5. If $PC(x_j) \subseteq NC(x_k)$ and $NC(x_j) \subseteq PC(x_k)$, then φ is satisfiable if and only if $\varphi - NC(x_j) - PC(x_j)$ is satisfiable.

6. If $PC(x_j) \subseteq PC(x_k)$ and $NC(x_j) \subseteq NC(x_k)$, then φ is satisfiable if only if $\varphi - NC(x_j) - PC(x_j)$ is satisfiable.

In cases 1. through 4. variables x_j and x_k are eliminated from the CNF formula. In cases 5. and 6. variable x_j is eliminated from the CNF formula.

3.3 Analyzing Sequences of Binary Clauses

Besides the identification of SCCs, an implication graph (see definition 3.1) can also be used for identifying unsatisfiable instances or for implying the value of a variable.

For the implication graph G of a CNF formula φ, the following holds:

1. If there exists a path p, with $l_j \xmapsto{p} \neg l_j$, then formula φ can only be satisfied with $\neg l_j$ assigned value 1.

2. If an SCC containing l_j and $\neg l_j$ is identified, then φ is unsatisfiable.

An example of the previous result is shown in Figure 3-3, representing the implication graph for the set of clauses $(\neg a \vee b) \wedge (\neg b \vee \neg a) \wedge (a \vee c) \wedge (\neg c \vee a)$. As can be concluded, both a and $\neg a$ are in the same SCC. Hence the problem instance is unsatisfiable.

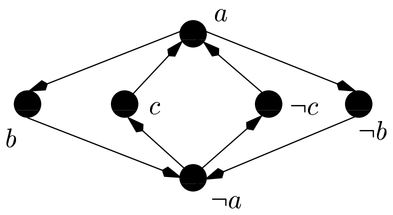
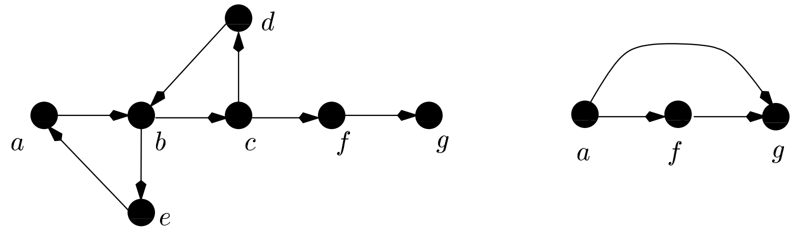


Figure 3-3: Implication graph for identifying an unsatisfiable instance.



(a) Original implication graph (b) Resulting implication graph

Figure 3-4: Example of implication collapsing.

Another technique for deriving information from sequences of binary implications is *implication collapsing*. Let us consider the sub-formula $\varphi_a = (\neg x \vee y) \wedge (\neg y \vee z)$. Then, we can also derive the implication $(\neg x \vee z)$. Basically, after identifying the SCCs of the implication graph, implication collapsing can be achieved by computing the *transitive closure* of the resulting graph. For some of the algebraic manipulation techniques described below, implication collapsing may prove useful in enabling further formula simplifications. For the example implication graph of Figure 3-4-a, and after eliminating the SCC, we can identify the transitive closure of the resulting implication graph, obtaining the graph shown in Figure 3-4-b. The resulting graph indicates the creation of a new binary clause $(\neg a \vee g)$.

3.4 Support Set Variable Equivalence

In many practical situations, a sub-formula φ_s of a CNF formula φ actually describes a Boolean function $x_i = f(y_1, \ldots, y_k)$. For example, the sub-formula $\varphi_a = (y_1 \vee \neg x) \wedge (y_2 \vee \neg x) \wedge (\neg y_1 \vee \neg y_2 \vee x)$ describes the Boolean function $x = y_1 \wedge y_2$. More interestingly, if we have two sub-formulas $\varphi_a = (y_1 \vee \neg x) \wedge (y_2 \vee \neg x) \wedge (\neg y_1 \vee \neg y_2 \vee x)$ and $\varphi_b =$

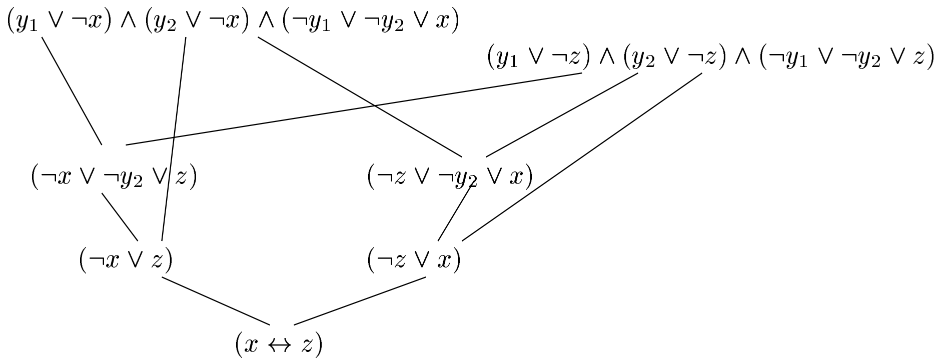


Figure 3-5: Applying resolution to derive two-variable equivalence.

$(y_1 \vee \neg z) \wedge (y_2 \vee \neg z) \wedge (\neg y_1 \vee \neg y_2 \vee z)$, then we can conclude that $x = y_1 \wedge y_2$ and $z = y_1 \wedge y_2$. Hence, x and z are indeed *equivalent* and we can replace z with x. Observe that, by selected resolution operations, we could easily derive the clauses $(x \vee \neg z) \wedge (\neg x \vee z)$, obtaining the same conclusion, as demonstrated by figure 3-5. In general, if $x_i = f(y_1, \ldots, y_k)$ and $x_j = f(y_1, \ldots, y_k)$, then $x_i \equiv x_j$ and either x_i or x_j can be replaced, respectively, by x_j or x_i.

The given example suggests a pattern-matching approach for identifying a set E of variables that can be expressed as a Boolean function f of some other set S of variables (i.e., *the support set*) and thus replace the variables in set E by a single variable. This approach is simply too time consuming for sets S of arbitrary size, and so we restrict S to be of size 2, i.e. we only consider Boolean functions of two variables.

If the support set S is restricted to be of size 2, it becomes feasible to enumerate all possible Boolean functions of 2 variables, and determine the *irredundant* CNF formulas associated with each Boolean function. This information is shown in Table 3.1. (Observe that the missing six boolean functions are either constant 0 or 1, the actual function of a *single* variable or of its complement.) We can now apply a straightforward pattern matching algorithm to a CNF formula, and identify sets E of variables with a common two-variable support set. These variables can then be replaced by a single variable. Clearly, after variable replacement we can apply *all* the techniques described in previous sections for further simplifying the formula.

In what regards the complexity of the pattern matching approach for a given CNF

Table 3.1: Two-variable CNF formulas

Boolean function	CNF formula
$x \equiv f_1(a,b) = a \wedge b$	$(a \vee \neg x) \wedge (b \vee \neg x) \wedge (\neg a \vee \neg b \vee x)$
$x \equiv f_2(a,b) = a \vee b$	$(\neg a \vee x) \wedge (\neg b \vee x) \wedge (a \vee b \vee \neg x)$
$x \equiv f_3(a,b) = a \leftrightarrow b$	$(a \vee b \vee x) \wedge (a \vee \neg b \vee \neg x) \wedge (\neg a \vee \neg b \vee x) \wedge (\neg a \vee b \vee \neg x)$
$x \equiv f_4(a,b) = \neg a \wedge b$	$(\neg a \vee \neg x) \wedge (b \vee \neg x) \wedge (a \vee \neg b \vee x)$
$x \equiv f_5(a,b) = a \wedge \neg b$	$(a \vee \neg x) \wedge (\neg b \vee \neg x) \wedge (\neg a \vee b \vee x)$
$x \equiv f_6(a,b) = \neg(a \wedge b)$	$(a \vee x) \wedge (b \vee x) \wedge (\neg a \vee \neg b \vee \neg x)$
$x \equiv f_7(a,b) = \neg(a \vee b)$	$(\neg a \vee \neg x) \wedge (\neg b \vee \neg x) \wedge (a \vee b \vee x)$
$x \equiv f_8(a,b) = \neg(a \leftrightarrow b)$	$(a \vee b \vee \neg x) \wedge (a \vee \neg b \vee x) \wedge (\neg a \vee b \vee x) \wedge (\neg a \vee \neg b \vee \neg x)$
$x \equiv f_9(a,b) = a \vee \neg b$	$(\neg a \vee x) \wedge (b \vee x) \wedge (a \vee \neg b \vee \neg x)$
$x \equiv f_{10}(a,b) = \neg a \vee b$	$(a \vee x) \wedge (\neg b \vee x) \wedge (\neg a \vee b \vee \neg x)$

formula, it depends on the number of clauses which can potentially match a specific clause pattern. Suppose we are trying to match a clause pattern 2B/1T. Assuming a CNF formula φ has b binary clauses and t ternary clauses, an exact limit value to the pattern matching 2B/1T is $\frac{b \times (b-1)}{2} \times t$.

Other researchers [30] have observed the existence of variable equivalences based on support sets, but on arbitrary instances of SAT (i.e. not in CNF format). Moreover, the underlying approach of [30, 77] is significantly more time consuming (indeed exponential in the worst-case) than the one proposed above. Another interesting result, is that the equivalence reasoning conditions proposed by Chu Min Li in [41] are also superseded by support set equivalences and selective resolution. Observe, in particular, that the utilization of function f_3 on two variables $x = (a \leftrightarrow b)$ and $y = (a \leftrightarrow b)$, for deriving the equivalence between x and y, corresponds to Li's inference rule (5) [41], the other rules being superseded by selective resolution, unit-clause rule and two-variable equivalence.

Despite the potential interest of identifying support sets of variables in a CNF formula, in the next section a further generalization is proposed, that subsumes variable equivalences based on two-variable support sets.

3.5 Generalized Inference of Binary Clauses

In this section we propose to study subsets of clauses for inferring binary clauses, which not only identify support set equivalences, but also provide more general conditions for deriving binary clauses. In what follows, all proposed conditions can be explained by resorting to resolution. However, it is in general extremely hard to decide to which clauses the resolution operation should be applied, being computationally infeasible to apply the resolution operation to all possible pairs of clauses. The objective of studying sets of clauses is to indirectly select to which sets of clauses the resolution operation should be applied.

Moreover, the reasoning technique to be described below is categorized in terms of how many binary clauses and ternary clauses are required for explaining the inferred binary clause. Clearly, the size and number of k-ary clauses involved could be made arbitrary, but the computational overhead could become prohibitive. In general each proposed condition is classified as being of the form mB/nT, meaning that m binary clauses and n ternary clauses are involved.

Let us start by considering an illustrative example. Let $\varphi_c = (y_1 \vee \neg x) \wedge (y_2 \vee \neg x) \wedge (\neg y_1 \vee \neg y_2 \vee z)$ be a sub-formula. The application of the resolution operation between the three clauses allows deriving $(\neg x \vee z)$. Similarly, for the sub-formula $\varphi_d = (y_1 \vee \neg z) \wedge (y_2 \vee \neg z) \wedge (\neg y_1 \vee \neg y_2 \vee x)$, the resolution operation allows deriving the clause $(x \vee \neg z)$. It is interesting to observe that we have just illustrated a different approach for proving x equivalent to z for the first example of the previous section.

In general, let us consider a sub-formula of the form $\varphi_s = (l_1 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_2 \vee l_3 \vee l_4)$, where $l_1, \ldots, l_4$ are any literals. The application of resolution allows deriving the binary clause $(l_1 \vee l_4)$. Since 2 binary clauses and 1 ternary clause are involved in deriving the resulting binary clause, we say that 2B/1T reasoning was applied. The other form of 2B/1T reasoning is the following. Let the sub-formula be $\varphi_t = (l_4 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_2 \vee l_3 \vee l_4)$. Then, application of resolution allows deriving the binary clauses $(l_3 \vee l_4)$ and $(l_1 \vee l_4)$.

Additional forms of mB/nT reasoning can be established. Due to efficiency concerns, our analysis will be restricted to 2B/1T, 1B/1T, 1B/2T, 3B/1T and 0B/4T.

We start with 0B/4T reasoning, since it is also useful for indirect identification of support sets. Suppose the following set of four ternary clauses:

$$(l_1 \vee l_2 \vee l_3) \wedge (\neg l_1 \vee \neg l_2 \vee l_3) \wedge (l_1 \vee \neg l_2 \vee \neg l_4) \wedge (\neg l_1 \vee l_2 \vee \neg l_4)$$

Then the binary clause $(l_3 \vee \neg l_4)$ can be inferred, again by selective resolution. Observe that with $l_1 = a$, $l_2 = b$, $l_3 = x$ and $l_4 = y$, we obtain $(x \vee \neg y)$. Similarly, with $l_1 = a$, $l_2 = b$, $l_3 = y$ and $l_4 = x$, we obtain $(\neg x \vee y)$. Considering simultaneously the two sets of clauses, we establish the equivalence between x and y, as would be the case with support set two-variable equivalence based on function f_3 (see Table 3.1) or using Li's inference rule (5) [41].

Another form of 0B/4T reasoning is the following. Let us consider the following set of clauses:

$$(l_1 \vee l_2 \vee \neg l_3) \wedge (l_1 \vee l_2 \vee \neg l_4) \wedge (l_1 \vee l_2 \vee \neg l_5) \wedge (l_3 \vee l_4 \vee l_5)$$

Then the binary clause $(l_1 \vee l_2)$ can be inferred.

Next we examine the remaining cases 1B/1T, 1B/2T and 3B/1T. Let us start with 1B/1T reasoning. Suppose the pairs of clauses $(l_1 \vee l_2) \wedge (\neg l_1 \vee l_2 \vee l_3)$. Then the binary clause $(l_2 \vee l_3)$ can be derived. For 3B/1T reasoning, let us consider the set of clauses $(l_1 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_1 \vee \neg l_4) \wedge (l_2 \vee l_3 \vee l_4)$. Then, the unit clause (l_1) can be derived. (Notice that this case is an exception, since a unit clause is inferred.)

Finally, we analyze 1B/2T reasoning. The first set of clauses considered is $(l_1 \vee l_2 \vee \neg l_3) \wedge (l_1 \vee l_2 \vee \neg l_4) \wedge (l_3 \vee l_4)$. Then the binary clause $(l_1 \vee l_2)$ can be derived. The second set of clauses considered is $(l_1 \vee l_2) \wedge (\neg l_1 \vee l_3 \vee l_4) \wedge (l_2 \vee l_3 \vee \neg l_4)$. In this case, the binary clause $(l_2 \vee l_3)$ can be derived.

All the unit/binary clause inference rules described above are summarized in Table 3.2. Moreover, and from the previous examples and claims, we can readily conclude that support set variable equivalence (described in the previous section) is superseded by 2B/1T

Table 3.2: Rules for Inferring Binary/Unit Clauses

Clause Pattern	Inferred Clause(s)
$(l_1 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_2 \vee l_3 \vee l_4)$	$(l_1 \vee l_4)$
$(l_4 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_2 \vee l_3 \vee l_4)$	$(l_3 \vee l_4), (l_1 \vee l_4)$
$(l_1 \vee l_2 \vee l_3) \wedge (\neg l_1 \vee \neg l_2 \vee l_3) \wedge (l_1 \vee \neg l_2 \vee l_4) \wedge (\neg l_1 \vee l_2 \vee l_4)$	$(l_1 \vee l_4)$
$(l_1 \vee l_2 \vee \neg l_3) \wedge (l_1 \vee l_2 \vee \neg l_4) \wedge (l_1 \vee l_2 \vee \neg l_5) \wedge (l_3 \vee l_4 \vee l_5)$	$(l_1 \vee l_2)$
$(l_1 \vee l_2) \wedge (\neg l_1 \vee l_2 \vee l_3)$	$(l_2 \vee l_3)$
$(l_1 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_1 \vee \neg l_4) \wedge (l_2 \vee l_3 \vee l_4)$	(l_1)
$(l_1 \vee l_2 \vee \neg l_3) \wedge (l_1 \vee l_2 \vee \neg l_4) \wedge (l_3 \vee l_4)$	$(l_1 \vee l_2)$
$(l_1 \vee l_2) \wedge (\neg l_1 \vee l_3 \vee l_4) \wedge (l_2 \vee l_3 \vee \neg l_4)$	$(l_2 \vee l_3)$

and 0B/4T reasoning. As a result, is sufficient for an algebraic simplification system to implements the restricted forms of mB/nT reasoning described above.

From an implementation perspective, the simple approach currently used consists of searching, in the CNF formula, for the mB/nT patterns mentioned above. For a larger number of patterns, more elaborated techniques could be implemented.

3.6 Integration in SAT Algorithms during Preprocessing

When considered in the context of SAT algorithms, there are two main ways for using the algebraic simplification techniques described in the previous sections:

- As a preprocessing engine, to be used by *any* SAT algorithm.
- As an extension to the usual *boolean constraint propagation* in backtrack search SAT algorithms.

Whereas the first approach can be readily utilized with any SAT algorithm, the second approach can be challenging depending on the organization of the backtrack search SAT algorithm considered. For backtrack search algorithms that implement some form of learning (e.g. rel_sat [6], GRASP [58] and SATO [87]), simplification techniques that replace variables at each step of the search process require a careful accounting of when to maintain and/or eliminate already created clauses.

Table 3.3: Example Instances

Application Domain	Instance	Satisfiable?
Circuit Testing (DIMACS) [36]	bf1355-075	N
	bf2670-001	N
	ssa2670-130	N
	ssa2670-141	N
Bounded Model Checking [7]	barrel6	N
	barrel7	N
	longmult5	N
	longmult6	N
	queueinvar12	N
	queueinvar16	N
Combinational Equivalence Checking [39]	c1908	N
	c1908_bug	Y
	c2670	N
	c2670_bug	Y
Superscalar Processor Verification [83]	dlx2_cc_bug02	Y
	dlx2_cc_bug08	Y

In face of these difficulties, and with the objective of evaluating the simplification ability of the proposed techniques, in previous work [50] these techniques were solely applied during preprocessing. Experimental results, obtained on representative problem instances, indicate that large simplifications can be observed.

We start by analyzing possible ways for utilizing the proposed techniques. Afterwards, these techniques are evaluated according to two main metrics:

- The ability to effectively simplify the original formula.
- The effective reduction in the amount of search, when using formula simplification techniques within a preprocessing engine for backtrack search SAT algorithms.

In this section, we will only evaluate the ability to effectively simplify the original formula.

With the objective of evaluating the formula simplification ability of the proposed techniques, several representative problem instances from different real-world applications were used. The problem instances are shown in Table 3.3, and can be obtained from the different authors [7, 36, 39, 83].

The first experiment consisted in applying the proposed formula simplification techniques to the different problem instances. Table 3.4 shows the obtained results, namely the

Table 3.4: Formula Simplification

Instance	Initial Formula		Final Formula	
	Variables	Clauses	Variables	Clauses
bf1355-075	2180	6778	722	3075
bf2670-001	1393	3434	411	1252
ssa2670-130	1359	3321	440	1295
ssa2670-141	986	2315	283	883
barrel6	2306	8931	519	2209
barrel7	3523	13765	805	3467
longmult5	2397	7431	1483	5180
longmult6	2848	8853	1824	6376
queueinvar12	1112	7335	1049	9884
queueinvar16	1168	6496	1088	7114
c1908	1917	5096	658	2248
c1908_bug	1919	5100	659	2250
c2670	2703	6756	1220	3725
c2670_bug	2708	6696	1162	3553
dlx2_cc_bug02	1515	12808	1486	13965
dlx2_cc_bug08	1515	12808	1486	13890

number of variables and clauses before and after applying the simplification techniques[5]. As can be concluded, for most of the instances considered, large reductions in the number of variables and clauses can be achieved. In some cases (e.g. *barrel6* and *barrel7*) the final number of variables is one fourth of the original number. Similar reductions can be observed in the number of clauses. Nevertheless, for some instances (e.g. *queueinvar16* and *dlx2_cc_bug08*) the amount of simplification in the number of variables is negligible. In these cases, it is interesting to observe an increase in the number of clauses, due to the application of clause inference techniques (see Section 3.5).

Despite the promising results obtained, and given the amount of simplification achieved on most problem instances, the next natural step is to incorporate the same algebraic simplification techniques into backtrack search SAT algorithms, to be applied during the search. Starting from such motivation, our work consists in the implementation of some of the described techniques in a non-chronological backtrack search SAT algorithm, namely the identification of two-variable equivalence, with the additional inference of clauses using 2B/1T and 0B/4T reasoning.

[5]For obtaining these preliminary results, the formula simplification techniques were implemented with a Perl script, with no concern for efficiency.

4

Complete Search for Propositional Satisfiability

In this chapter we describe complete search algorithms for propositional satisfiability. We start by describing *backtrack search algorithms* in section 4.1, and based on their structure we present *intelligent backtrack search algorithms* in section 4.2.

Complete algorithms, using backtrack search, find one or more satisfying assignments or prove that the formula under consideration is unsatisfiable. Examples of such algorithms are usually variants of the Davis-Putnam procedure, which is described in detail in section 4.1.1. Furthermore, starting from the *Davis Putnam procedure*, we sketch a *generic backtrack search SAT algorithm* in section 4.1.2, which captures the organization of several of the most competitive SAT solvers.

Even having many characteristics in common, backtrack algorithms usually differ in the way they select which variable to set at each iteration. As a result, in section 4.1.3 we briefly discuss some of the most used *branching heuristics*. Other differences can be identified from variations to the algorithm used for inferring *necessary assignments*, discussed in section 4.1.4.

Intelligent backtrack search algorithms represent an improvement to traditional backtrack search SAT algorithms, providing *clause recording* (see section 4.2.1) and *non-chronological backtrack* (see section 4.2.2). Clause recording allows recording the cause of failure thus preventing similar mistakes from being made later on, while non-chronological backtrack backs up the search tree to one of the deemed causes of failure, skipping over irrelevant variable assignments. These techniques can contribute decisively for efficiently solving a large number of classes of instances of SAT, mainly because the number of non-chronological backtracks can be a significant percentage of the total number of backtracks and the jumps in the decision tree can save a large amount of search work. Moreover, we emphasize the description of the *GRASP SAT algorithm* in section 4.2.3, as an example of an intelligent backtrack search algorithm, which will be subsequently used for integrating algebraic simplification techniques.

4.1 Backtrack Search Algorithms

Given a problem, the process of identifying a sequence of actions in order to achieve a goal is called *search* [70]. The idea is to maintain and extend a set of partial solution sequences. One has to generate these sequences and keep track of them using suitable data structures.

Search algorithms for SAT are based on splitting. 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. Splitting insures that a search for a solution terminates with a result.

When searching for a solution to a SAT problem, at the end of each iteration, the algorithm has to check whether any clause has became unsatisfied by the variable assignments made up to that point. In such situation we have a *conflict*. The resulting algorithm, called *backtrack search*, then backtracks to try something else. Backtrack often saves a huge amount of time.

A backtrack search algorithm for SAT 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 for a satisfying assignment by maintaining a *decision tree*. For each node we have two branches, one for each possible assignment. We associate a branch of the tree with a sequence of variable assignments.

The root of the tree corresponds to the initial formula. The leaf nodes correspond to sub-formulas that are either satisfiable or unsatisfiable. In the first case the formula is declared to be satisfiable, while in the second one the search backtracks, if possible. If the whole search space has already been traversed, which means backtrack is no more possible, the formula is declared to be unsatisfiable.

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. Moreover, a decision level $\delta(x)$ is associated with each assigned variable x, that denotes the decision level at which the variable is assigned. The notation used to denote that x is assigned with truth value v at decision level d will be $x = v@d$.

Backtrack algorithms differ in the way they select which variable to set at each iteration, i.e., the decision assignments. The criteria used to decision assignments within a formula φ may be as simple as choosing the first remaining variable in φ, or it may be quite sophisticated [49]. The name *heuristic* is used to refer to the rules we believe will generate solutions without exhaustive search.

We usually distinguish between *chronological backtrack* and *non-chronological backtrack*. Chronological backtrack backtracks to the most recent choice point, exploring one branch of the search tree completely before backtracking and exploring the other. On the other hand, non-chronological backtrack backtracks more steps. In this case, the search can jump back to the source of the conflict, while potentially skipping irrelevant levels on the search tree. In the AI domain, this last technique is often called *conflict-directed backjumping* or simply *backjumping*.

4.1.1 The Davis-Putnam Procedure

Resolution provides a complete system for proof by refutation [70]. We know that repeated applications of the resolution inference rule will find a solution if one exists, or will prove its inexistence. It was first applied to the SAT problem and is often referred to as the Davis-Putnam procedure [15].

The original Davis-Putnam procedure was based on a resolution rule that eliminated the variables one-by-one and added all possible resolvents to the set of clauses. The idea is to iteratively apply resolution to eliminate one variable each time, i.e., resolution between all pairs of clauses containing variable x and $\neg x$. Each step generates a sub-problem with one fewer literal, but possibly quadratically more clauses, depending on the number of clauses in x and $\neg x$. We stop applying resolution when either an empty clause is derived, which means the instance is unsatisfiable, or only satisfied clauses or with pure literals are obtained, which means the instance is satisfiable.

Example 4.1 *Consider the formula φ has the following clauses:*

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

Let us start by applying resolution in order to eliminate variable x_3. The obtained formula φ' has three clauses: $\omega_i = (x_1 \vee \neg x_1 \vee \neg x_2)$, $\omega_{ii} = (\neg x_1 \vee x_2 \vee \neg x_2)$ and $\omega_{iii} = (\neg x_1 \vee \neg x_2 \vee x_4)$. Observe that ω_i and ω_{ii} are tautologous clauses and therefore the remaining clause ω_{iii} has only pure literals. Thus φ' and consequently φ is satisfied.

Unfortunately, this procedure requires exponential space in general. In practice, many resolvents may be generated even when a satisfying assignment can be found easily by other methods. Davis, Longemann and Loveland [14] therefore 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.

Let us describe the Davis-Putnam-Loveland procedure, which in much of the literature (including this thesis), is often albeit somewhat inaccurately called the "Davis-Putnam" or "DP" procedure. The algorithm is a backtracking depth-first search through (partial) truth assignments, augmented by the tautology rule, the pure literal rule and the unit clause rule. All these rules are the application of restricted forms of resolution. Remember that, even resolution being impractical for real-world problem instances, application of restricted forms, called *selective* resolution, has been successful [68]. This can be successful because the remaining formula must be smaller than the original. Nevertheless, neither rule but the splitting rule is required for the soundness or completeness of the procedure.

Let us explain once more the effects of applying the splitting rule. Whenever we make a decision assignment, the sub-formula with the unresolved clauses from the previous decision level is reviewed. Those clauses containing the literal which is true for the variable assignment are erased from the formula, and those clauses which contain the literal which is false have that literal removed. Nonetheless, erasing satisfied clauses can be ignored if the DP procedure is terminated when every variable has been assigned a truth value.

Applying the tautology rule, we remove tautologous clauses from the formula. Because of its cost, and considering that all tautologous clauses can be identified at the start of the search, the tautology rule is usually deleted in modern SAT solvers inspired in the DP procedure.

The pure literal rule was introduced in the original Davis-Putnam procedure. It was called *Affirmative-Negative Rule* and was applied between the *Rule for the Elimination of One-Literal Clauses* and the *Rule for Eliminating Atomic Formulas.*

This technique was defined with the following statement:

> If an atomic formula p occurs in a formula F in conjunctive normal form only affirmatively, or if p occurs only negatively, then all clauses which contain p may be deleted.

Clearly, the satisfiability of a formula is unaffected by assigning those variables. Applying this rule, we are trying to accelerate the search process, assigning as many variables as possible for each decision level, and thus reducing the search space.

Notice that no resolvents can be generated by resolving on a pure literal, but all clauses containing a pure literal can be removed without loss. So, an important improvement to the basic resolution algorithm is to first remove clauses containing pure literals. Probably this is the reason why it was introduced with the original Davis-Putnam procedure, which used resolution instead of splitting.

In search algorithms for SAT, the pure literal rule is usually not considered because of its cost. The results presented in Appendix A of this thesis also lead to such conclusion. A possible explanation for this fact is that pure literals do not occur often enough or are not relevant to determine the (un)satisfiability of a formula in order to compensate the computational effort. As will be explained also in Appendix A, the algorithm to identify pure literals requires the traversal of many variables and respective clauses. Moreover, the associated data structures require additional memory resources.

The unit clause rule was originally described in the Davis-Putnam procedure [15] with the following words:

> If an atomic formula p appears as a clause in formula F in conjunctive normal form, then one may modify F by striking out all clauses that contain p affirmatively and deleting all occurrences of $\neg p$ from the remaining clauses, thus obtaining a formula F' which is consistent if and only if F is.

In fact, the unit clause rule, when applicable, allows identifying which variable to assign. If some clause contains only one unassigned variable, then select that variable and assign it a value that satisfies the clause containing it.

Observe that the unit clause rule is not necessary for the soundness or completeness of the search procedure. Nevertheless, it contributes greatly to the efficiency of the procedure. Improving the selection of the variable to assign often results in much faster backtracking. Moreover, the unit clause rule is a useful simplification technique that can be combined with other strategies.

Boolean Constraint Propagation (BCP) [86] consists on the iterated application of the unit clause rule, inducing the derivation of implications. BCP extends the current truth assignment to the variables contained in all unit clauses. Sometimes, this action is also called *unit propagation*.

4.1.2 Generic Backtrack Search Algorithm

Starting from the DP procedure, we can sketch a generic backtrack search SAT algorithm, as described in figure 4-1. For this generic algorithm we will assume that

```
SAT(φ)
{
    d = 0;
    while (Decide(φ, d) == DECISION) {
        if (Deduce(φ, d) == CONFLICT) {
            β = Diagnose(φ, d);
            if (β == −1)
                return UNSATISFIABLE;
            else {
                Backtrack(φ, d, β);
                d = β;
            }
        }
        d = d + 1;
    }
    return SATISFIABLE;
}
```

Figure 4-1: Generic Backtrack Search Algorithm

we only want to check if the problem has a solution, and not explicitly obtain it. This SAT algorithm captures the organization of several of the most competitive algorithms [6, 24, 58, 87].

Despite the figure not mentioning it, a preprocess procedure can precede backtrack search [6, 43, 58]. It can be applied to reduce the size of the formula under consideration, using different simplification techniques. If during this preprocessing phase the empty clause is not derived, a backtrack search algorithm is them invoked.

Given a SAT problem, formulated as a CNF formula φ, the algorithm conducts a search through the space of the possible assignments to the problem variables. At each stage of the search, a variable assignment is selected with the *Decide* function. A *decision level* d is associated with each selection of an assignment. Implied necessary assignments are identified with the *Deduce* function, which in most cases corresponds to straightforward derivation of implications [6, 58]. 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. The *Backtrack* function clears all implied assignments that result from each assignment selection, from the current decision level d through the backtrack decision level β. Further, since the search process should resume at the backtrack level, the current decision level d becomes β. Finally, the current decision level d is incremented.

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

1. The decision engine `Decide()`, which selects a decision assignment at each stage of the search. 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 necessary 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 process can provably backtrack to. This backtracking process is the basic mechanism for retreating from regions of the search space where satisfying assignments cannot be identified.

Distinct organizations of SAT algorithms can be obtained by different configurations of this generic algorithm. For instance, the Davis-Putnam procedure described in the previous section can be captured by the generic algorithms as follows:

- The decision engine randomly picks an unassigned variable x, since the DP 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.
- The diagnosis engine implements chronological backtrack. First, implied assignments at level d are erased, including the assignment on x. In case the decision variable x has not yet been toggled, x is assigned to the opposite value. Otherwise, a conflict is detected, and the search backtracks to level $d-1$.

4.1.3 Branching Heuristics

As referred in the last paragraph, one approach for improving backtrack search algorithms consists of using heuristic knowledge in the choice of the variable used for splitting. In most cases, it has a very significant impact on the overall efficiency of the algorithm. Furthermore, early mistakes can be very costly. Some problems that are typically easy to solve can occasionally take over a significant number of nodes to solve. Based on this fact, a strategy of randomization and restarts can often be effective at tackling such early mistakes [29].

The most simple heuristic is to randomly select one of the yet unassigned variables, and to assign it a randomly chosen value. Moreover, randomization can also be applied within other branching heuristics, to broke ties between several variables having the same maximum heuristic value.

Other branching heuristics take into account the dynamic information provided by the backtrack search algorithm. This information can include, for example, the number of literals of each variable in unresolved clauses and the relative sizes of the unresolved clauses that contain literals in a given variable.

MOM's Heuristic

A popular and low cost branching heuristic is MOM's (Maximum number of Occurrences in Minimum size clauses) [20, 24, 63, 86]. Intuitively, the objective of this heuristic is to select variables that exploit the power of BCP and increase the chance to reach an empty clause. Moreover, Mom's heuristic suggests that the larger the number of binary occurrences of a variable is, the higher it is the probability of being a good branching variable.

Nonetheless, in some cases computing the literal that occurs most often is considered too expensive. Thus, we apply a variant and simply branch on the first literal of a clause of the shortest size. Another alternative consists in taking longer clauses into account with exponentially smaller weights, which means weights will cause bigger clauses to be less weighted.

Let $f^*(l)$ be the number of occurrences of a literal l in the smallest non-satisfied clauses. Henceforth, we can assume that a *good* variable to select is one that maximizes the function,

$$[f^*(x) + f^*(\neg x)] * 2^k + f^*(x) * f^*(\neg x) \tag{4.1}$$

assuming k is chosen to be sufficiently large.

Jeroslow-Wang Heuristics

The Jeroslow-Wang heuristic [35] estimates the contribution each literal is likely to make to satisfying the clause set. The idea is to choose the variable and value which, in some sense, maximize the chance of satisfying the remaining clauses. Each literal is scored as follows: for each clause ω the literal appears in, the value $2^{-|\omega|}$ is added to the literals score, where $\mid \omega \mid$ is the number of literals in ω. The splitting rule is then applied to the literal with the highest score.

Formalizing the Jeroslow-Wang heuristic $J(l)$, for a given literal l we have:

$$J(l) = \sum_{l \in \omega \wedge \omega \in \varphi} 2^{-|\omega|} \tag{4.2}$$

The one-sided Jeroslow-Wang (JW-OS) branching heuristic selects the assignment that satisfies the literal with the largest value $J(l)$. The two-sided Jeroslow-Wang (JW-TS) heuristic identifies the variable x with the largest sum $J(x) + J(\neg x)$, and assigns to x

value true, if $J(x) \geq J(\neg x)$, and value false otherwise.

Hooker and Vinay have investigated the motivation behind the scoring function used in the Jeroslow-Wang heuristic [31]. They propose both the *satisfaction hypothesis*, that it is best to branch into subproblems that are more likely to be satisfiable, and the *simplification hypothesis*, that it is best to branch into simpler problems with fewer and shorter clauses after unit propagation. At last, they reject the satisfaction hypothesis in favor of the simplification hypothesis. Using the simplification hypothesis they proposed the described above two-sided Jeroslow-Wang rule (as an alternative to the one-sided), which performs better than the original Jeroslow-Wang rule.

One of the best current implementations of the Davis-Putnam procedure, *satz* [43], uses a branching heuristic based on maximizing the amount of unit propagation. This adds further support to Hooker and Vinay's simplification hypothesis. Moreover, satz uses another hypothesis called *constraint hypothesis*, which assumes that a branching rule works better when it creates subproblems with more and stronger constraints so that a contradiction can be found earlier.

BOHM's Heuristic

Using BOHM's heuristic [8], each selected literal gives preference to satisfying small clauses (when assigned value true) or to further reducing the size of small clauses (when assigned value false). The idea is when faced with several choices of extending a partial solution, it is more efficient to choose the one that offers the fewest alternatives. That is, nodes with fewer successors should be generated early in the search tree, and nodes with more successors should be considered later.

At each step of the backtrack search algorithm, the BOHM heuristic selects a variable with the maximal vector $(H_1(x), H_2(x), \ldots, H_n(x))$ in lexicographic order. Each $H_i(x)$ is computed as follows:

$$H_i(x) = \alpha\ max(h_i(x), h_i(\neg x)) + \beta\ min(h_i(x), h_i(\neg x)) \tag{4.3}$$

where $h_i(x)$ is the number of unresolved clauses with i literals that contain literal x. The values of α and β are chosen heuristically, and the values usually suggested are $\alpha = 1$ and $\beta = 2$.

Literal Count Heuristics

Literal count heuristics are those that only take into account the number of literals in unresolved clauses of a given variable at each step of the backtrack search algorithm.

Let us consider C_P and C_N the number of unresolved clauses in which a given variable x appears as a positive or negative literal. Starting from these numbers, and considering them either individually or combined, one can obtain different heuristics:

- DLCS — Dynamic Largest Combined Sum (of literals) — selects the variable with the largest sum $C_P + C_N$, and assigns to it value true, if $C_P \geq C_N$, or value false, if $C_P < C_N$.
- DLIS — Dynamic Largest Individual Sum (of literals) — selects the variable with the largest individual value of C_P or C_N, and assigns to it value true, if $C_P \geq C_N$, or value false, if $C_P < C_N$.
- RDLIS — variation of DLIS, that consists in *randomly* selecting the value to be assigned to a given selected variable, instead of comparing C_P with C_N, in order to prevent making too many bad decisions for a few specific instances.

The designation dynamic derives from the fact that C_P and C_N are computed during the search.

4.1.4 Necessary Assignments

We have seen how backtrack algorithms can differ in the way they select which variable to set at each iteration, using distinct branching heuristics. Furthermore, other differences can be identified from variations to the algorithm used for inferring necessary assignments. While branching heuristics refer to variations in the decision engine, necessary assignments affect the deduction engine.

Necessary Assignments can be obtained by using different forms of *value probing*. The idea of probing consists in identifying assignments that are deemed necessary, usually called implied necessary assignments. In detail, it can be viewed as a form of reduction of the domain of each variable. On the other hand, these techniques are basically a mechanism for identifying suitable resolution operations.

In this section, we are going to explain three procedures used for identifying necessary assignments: Boolean Constraint Propagation (BCP) [86], Recursive Learning (RL) [39] and Stålmark's Method (SM) [77].

Boolean Constraint Propagation (BCP) probes by assigning each unassigned variable from unit clauses (that is, applying the unit clause rule), and testing to see whether all relations simplify to true. However, BCP does not identify *all* necessary assignments given a set of variable assignments and a CNF formula, as explained by example 4.2.

Example 4.2 *Consider the formula $\varphi = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$. For any assignment to variable x_2, variable x_1 must be assigned value 1 for preventing a conflict. Nevertheless, BCP applied to this CNF formula would not produce this straightforward conclusion.*

Other techniques, namely Recursive Learning (RL) [39] and Stålmark's Method (SM) [77], have been proposed with the goal of identifying necessary assignments. The idea is to derive conclusions starting from trying an assignment X, $X \in \{0,1\}$, to a variable x and obtain conclusions related to other variable assignments. The notation to the conclusions will be $C(x = X_1) = \{x = X_1, y = X_2, z = X_3\}$, denoting that assigning $x = X_1$ implies $y = X_2$ and $z = X_3$ as necessary assignments.

Recursive Learning consists in the recursive evaluation of clause satisfiability requirements for identifying common assignments to variables. Common assignments are deemed necessary for the clause to become satisfiable and consequently for the instance of SAT to be satisfiable.

RL can be seen under this perspective, as in example 4.3, or alternatively as a result of application of restricted forms of resolution, as in example 4.4.

Example 4.3 *Consider the formula φ having the following clauses: $\omega_1 = (x_1 \vee x_2)$,*

$\omega_2 = (\neg x_1 \vee x_3)$ *and* $\omega_3 = (\neg x_2 \vee x_3)$.

Let us try to satisfy clause ω_1*, either by assigning* $x_1 = 1$ *or* $x_2 = 1$.

If we assign $x_1 = 1$*, we obtain the sequence of implications* $(x_1 = 1) \Rightarrow (x_3 = 1)$*, which means* $C(x_1 = 1) = \{x_1 = 1, x_3 = 1\}$*. On the other hand, if we assign* $x_2 = 1$*, we obtain the sequence of implications* $(x_2 = 1) \Rightarrow (x_3 = 1)$*, that is* $C(x_1 = 1) = \{x_2 = 1, x_3 = 1\}$*. Therefore, we have* $C(x_1 = 1) \cap C(x_2 = 1) = \{x_3 = 1\}$*, concluding that every way of satisfying clause* $\omega_1 = (x_1 \vee x_2)$ *implies* $x_3 = 1$*. Hence,* $x_3 = 1$ *is a necessary assignment.*

Example 4.4 *Consider the formula* φ *from example 4.3, having the following clauses:* $\omega_1 = (x_1 \vee x_2)$, $\omega_2 = (\neg x_1 \vee x_3)$ *and* $\omega_3 = (\neg x_2 \vee x_3)$.

Applying resolution to eliminate variable x_1*, from clauses* ω_1 *and* ω_2 *we obtain clause* $\omega_4 = (x_2 \vee x_3)$*. Similarly, applying resolution to eliminate variable* x_2*, from clauses* ω_3 *and* ω_4 *we derive the unit clause* (x_3)*, hence deriving the necessary assignment* $x_3 = 1$ *using BCP.*

Stållmark's Method was proposed by G. Stållmark [77] and is protected by a software patent. It is available in solver by prover technologies and recently has also been incorporated in the SAT solver Heerhugo [30]. This method is defined by the recursive application of the branch-merge rule to each variable.

Observe that many search trees possess equivalent branches which can be merged to reduce the search effort. Therefore, our goal is to derive equivalence conclusions, either $(x_1 \equiv x_2)$ or $\neg(x_1 \equiv x_2)$. Alternatively, if we conclude that both x and $\neg x$ must hold, a contradiction is detected.

SM is illustrated by figure 4-2. Intuitively, we try to assign a variable x with both truth values true and false, obtaining C_x and $C_{\neg x}$. Identifying common assignments to C_x and $C_{\neg x}$, one can derive necessary assignments $C_x \cap C_{\neg x}$.

Similarly to RL, SM can be seen under the perspective of applying the branch-merge rule to each variable, as in example 4.5, or alternatively as a result of applying restricted forms of resolution, as in example 4.6.

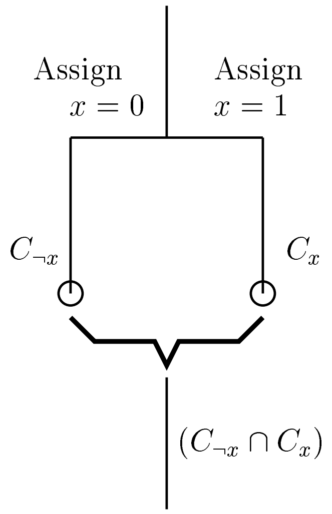


Figure 4-2: Illustration of the branch-merge rule.

Example 4.5 *Consider the formula φ having the following clauses: $\omega_1 = (x_1 \vee x_2)$, $\omega_2 = (\neg x_1 \vee x_3)$, $\omega_3 = (\neg x_2 \vee x_4)$ and $\omega_4 = (\neg x_3 \vee x_4)$.*

Let us try to assign variable x_1 with both truth values.

If we assign $x_1 = 0$, we obtain the sequence of implications $(x_1 = 0) \Rightarrow (x_2 = 1) \Rightarrow (x_4 = 1)$, which means $C(x_1 = 0) = \{x_1 = 1, x_2 = 1, x_4 = 1\}$. On the other hand, if we assign $x_1 = 1$, we obtain the sequence of implications $(x_1 = 1) \Rightarrow (x_3 = 1) \Rightarrow (x_4 = 1)$, that is $C(x_1 = 1) = \{x_1 = 1, x_3 = 1, x_4 = 1\}$. Therefore, we have $C(x_1 = 0) \cap C(x_1 = 1) = \{x_4 = 1\}$, concluding that any assignment to variable x_1 implies $x_4 = 1$. Hence, $x_4 = 1$ is a necessary assignment.

Example 4.6 *Consider the formula φ from example 4.5, having the following clauses: $\omega_1 = (x_1 \vee x_2)$, $\omega_2 = (\neg x_1 \vee x_3)$, $\omega_3 = (\neg x_2 \vee x_4)$ and $\omega_4 = (\neg x_3 \vee x_4)$.*

Applying resolution to ω_1 and ω_2 we obtain $\omega_5 = (x_2 \vee x_3)$. Applying resolution once more to ω_3 and ω_5 we obtain $\omega_6 = (x_3 \vee x_4)$. From ω_5 and ω_6 we derive the unit clause (x_4), hence deriving the necessary assignment $x_4 = 1$ using BCP.

A less powerful variant of the branch-merge rule is often employed in SAT solvers. Sometimes it is used to detect possible variable fixings (if assuming $\neg x$ a contradiction is derived, then x is a necessary assignment), and early closing of branches (when both assignments x and $\neg x$ lead to a contradiction).

Note that RL and SM can be applied using recursion of arbitrary depth. Moreover, both are based on similar reasoning, despite RL being based on clauses and SM on variables. Finally we should observe that these methods can be integrated into backtrack search algorithms.

4.2 Intelligent Backtrack Search Algorithms

The standard Davis-Putnam procedure performs chronological backtracking, and therefore backtracks to the most recent choice point.

Chronological backtracking can be improved by adapting some of the well-known techniques from the constraint satisfaction community like conflict-directed backjumping and nogood learning. Conflict-directed backjumping backs up the search tree to the cause of failure, skipping over irrelevant variable assignments. Nogood learning records the cause of failure to prevent similar mistakes being made down other branches.

Observe that, since chronological backtracking uses no knowledge of the the causes of conflicts, it can easily yield large sequences of conflicts *all* of which result from essentially the same variable assignments. In contrast, non-chronological backtracking strategies attempt to identify the variable assignments causing a conflict and backtrack directly so that at least one of those variable assignments is modified.

Nonetheless, some argue against the benefits of intelligent backtrack search algorithms. Is Chu Min Li's credo [43] (and also many others) that the purpose of intelligent backtrack is only to correct bad choices of branching variables. The fact is the more effective the order in which variables are assigned is, the less need there is for backjumping or simply for chronological backtracking. Good search orders tend to cause the sources of conflicts to being one level back. However, even good branching heuristics cannot eliminate mistakes, and so non-chronological backtrack is expected to be able to correct some of them. Moreover, experimental results obtained from a large number of benchmarks, indicate that these techniques can contribute decisively for efficiently solving a large number of classes of instances of SAT.

4.2.1 Clause Recording

Given a set of variable assignments, that is identified as representing a sufficient condition for yielding an identified conflict, clause recording consists in the creation of a new clause that prevents the same assignments from occurring simultaneously again during the search.

Despite the new clauses being created directly from the unsatisfiable clauses, as exemplified in 4.7, they can also be viewed as the result of applying selective resolution, as exemplified in 4.8.

Example 4.7 *Consider the formula φ having the following clauses: $\omega_1 = (x_1 \vee x_2)$, $\omega_2 = (\neg x_2 \vee x_3 \vee x_4)$, $\omega_3 = (\neg x_2 \vee x_5)$ and $\omega_4 = (\neg x_4 \vee \neg x_5 \vee x_6)$.*

Assume the current assignment is $A = \{(x_3, 0), (x_6, 0)\}$. Assigning $x_1 = 0$ other assignments are implied $(x_1 = 0) \Rightarrow (x_2 = 1) \Rightarrow (x_4 = 1) \Rightarrow (x_5 = 1)$. Therefore, since $\omega_4 = (\neg x_4 \vee \neg x_5 \vee x_6)$ becomes an unsatisfied clause, a conflict is reached. From the unsatisfied clause ω_4, and considering both decision and implied assignments, we can conclude either $(x_1 = 0) \wedge (x_3 = 0) \wedge (x_6 = 0) \Rightarrow (\varphi = 0)$ or $(\varphi = 1) \Rightarrow (x_1 = 1) \vee (x_3 = 1) \vee (x_6 = 1)$. As a result, applying clause recording one creates a new clause $\omega_5 = (x_1 \vee x_3 \vee x_6)$.

Example 4.8 *Consider the formula φ having the following clauses: $\omega_1 = (x_1 \vee x_2)$, $\omega_2 = (\neg x_2 \vee x_3 \vee x_4)$, $\omega_3 = (\neg x_2 \vee x_5)$ and $\omega_4 = (\neg x_4 \vee \neg x_5 \vee x_6)$.*

Applying resolution to ω_1 and ω_2 and to ω_1 ω_3 we obtain, respectively, $\omega_5 = (x_1 \vee x_3 \vee x_4)$ and $\omega_6 = (x_1 \vee x_5)$. Applying resolution once more to ω_4 and ω_5 we obtain $\omega_7 = (x_1 \vee x_3 \vee \neg x_5 \vee x_6)$. Applying resolution to ω_6 and ω_7, the conflicting clause $\omega_5 = (x_1 \vee x_3 \vee x_6)$ is created.

Unrestricted clause recording is in most cases impractical. Recorded clauses consume memory and repeated recording of clauses eventually leads to the exhaustion of available memory. Furthermore, large recorded clauses are known for not being particularly useful for search pruning proposes.

As a result, there are two main solutions for clause recording, in order to select the clauses to add to the clause database:

1. Clauses can be temporary recorded while they either imply variable assignments or are unit clauses, being discarded as soon as the number of unassigned literals is greater than one.

2. Clauses with a size less than a threshold k are kept during the subsequent search, whereas larger clauses are discarded when the number of unassigned literals becomes greater than one. We refer to this technique as *k-bounded learning*.

Consequently, using k-bounded learning, clause recording can be made polynomial, having growth in the number of clauses polynomial in k.

Relevance-Based Learning

Relevance-based learning is a simple and effective improvement to clause recording. It was originally introduced by Bayardo and Schrag in [6].

Remember that, in the previous section, we assumed that unrestricted clause recording is in most cases impractical, and therefore larger clauses are discarded when the number of unassigned literals becomes greater than one. Relevance-based learning basically consists of allowing recorded clauses to be deleted only when a larger number m of literals becomes unassigned, i.e., when the number of free literals is larger than m. As a result, recorded clauses may get used again, either for yielding conflicts or for implying variable assignments.

From the current and previous sections one can envision integrating k-bounded learning with relevance-based learning. The search algorithm is organized so that all recorded clauses of size no greater than k are kept, whereas larger clauses are deleted only after m literals have become unassigned.

4.2.2 Non-Chronological Backtracking

Recorded clauses can be used to perform non-chronological backtracking, that takes place when all the literals in the newly created conflicting clause correspond to variables that were assigned at decision levels that are lower than the current decision level. In this case, the backtracking decision level is defined as the highest decision level of all variable assignments of the literals in the recorded clauses.

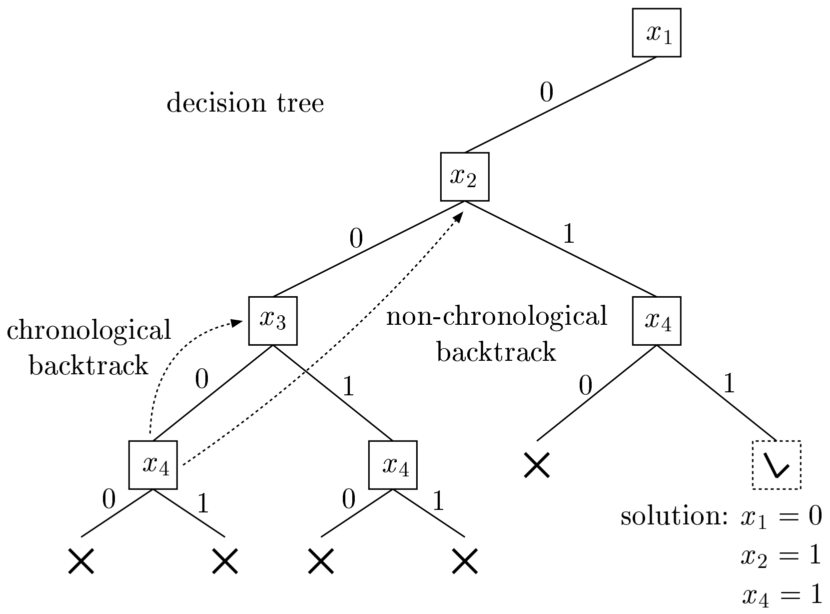


Figure 4-3: Non-Chronological Backtracking

Figure 4-3 illustrates non-chronological backtracking, in opposition to chronological backtracking. In this example, chronological backtracking would waste a potentially significant amount of time exploring a useless region of the search space, only to discover after much effort that the region does not contain any satisfying assignments. In contrast, using non-chronological backtracking, the search backtracks in the decision tree, considering the conflict resulted from an earlier conflicting assignment.

4.2.3 An Example: GRASP

GRASP (Generic seaRch Algorithm for the Satisfiability Problem) [58] is an intelligent backtrack search algorithm which incorporates several search-pruning techniques. GRASP is premised on the inevitability of conflicts during the search and its most distinguishing feature is the augmentation of basic backtracking search with a powerful conflict analysis procedure.

As a whole, the GRASP SAT algorithm can be described specifying the configuration of each engine of the generic backtrack search algorithm described in section 4.1.2. Hence, GRASP is organized by having the three different engines performing the following operations:

1. The decision engine uses one of the heuristics described in section 4.1.3, depending on the user-specified configuration.

2. The deduction engine implements Boolean Constraint Propagation and (implicitly) maintains the resulting *implication graph*.

3. The diagnosis engine identifies the causes of conflicts, augments the CNF formula with additional clauses (possibly using k-bounded learning and relevance-based learning) and backtracks non-chronologically.

A final note to say that the conflict analysis procedure in GRASP differs from other used in AI. First, conflict analysis in GRASP is tightly coupled with BCP and the causes of conflicts need not necessarily correspond to decision assignments. Second, clauses can be added to the original set of clauses, and the number and size of added clauses is user-controlled. Third, GRASP employs techniques to prune the search by analyzing the implication structure generated by BCP, which has no equivalent in other areas.

5

Integrating Algebraic Simplification in SAT Algorithms

One of the several effective approaches proposed for solving instances of the Satisfiability problem (SAT) is algebraic manipulation. Most if not all of the simplification techniques described earlier in chapter 3, were never applied within SAT algorithms with intelligent backtracking. In this chapter we address this issue.

Of the simplification techniques described in chapter 3, we implemented the pure literal rule[1], two-variable equivalence and inference of binary clauses (using clause patterns 2B/1T and 0B/4T [50]). Additional simplifications were also implemented, to prevent tautologous clauses or repeated literals within a clause, as a consequence of replacing variables.

We start from an existing SAT solver — GRASP [58], whose algorithm was already explained in chapter 4. The integration of these techniques into GRASP imply adding specific series of data structures for implementing two-variable equivalence and the reasoning rules 2B/1T and 0B/4T (that can be used to deduce more binary clauses). Some of them, used to detect the simplification patterns, are the same during the whole search. On the contrary, those that recorded the simplifications that have been applied thus far, have a proportional growth with the deep of the search tree.

We should note that equivalency reasoning in EQSATZ [41] already incorporates a few

[1]Implementation of the pure literal rule and obtained results are reported in Appendix A.

related techniques. However, integrating them in a non-chronological algorithm brings additional technical challenges.

5.1 Two-variable Equivalence

Those who ever implemented this technique have faced several difficulties. Warners and Van Maaren [85] simply say that careful implementation is needed to keep the cost of reasoning low. Some years before, Van Gelder and Tsuji [25] alert to the fact that achieving acceptable performance depends strongly on efficient data structures. They found that memory was often more problematic than time for a large number of variables. On the other hand, space-efficient data structures often require some overhead in time. Keeping this overhead low is a key implementation challenge.

We also believe there is room for significant additional improvement to the proposed algorithm to identify two-variable equivalences, even without changing the basic structure.

The implementation of two-variable equivalence in GRASP will be explained in two phases: during preprocessing and during the search. The first one is simpler, since those equivalences are valid for all partial assignments during the whole search process. Hence, under no conditions they will have to be undone. During the search, one has to ensure the formula is the same before and after backtracking, keeping equivalences registered with the necessary data structures. In detail, if an equivalence $(x \equiv y)$ takes place at a decision level d, thus replacing all occurrences of y by x, the replacement is valid as long as the search is performed at deeper levels. If the search backtracks to a previous level, occurrences of x that have replaced y are now replaced by y again.

5.1.1 Two-variable Equivalence during Preprocessing

The first approach to the implementation of two-variable equivalence considers only preprocessing. Remember that if x and y are said to be equivalent, one can replace y by x on all occurrences of y, thus eliminating variable y from the CNF formula.

However, GRASP data structures used for variables do not contemplate the elimination of variables. Nevertheless, if a variable equivalence $(x \equiv y)$ is detected during preprocessing, thus replacing y by x on all occurrences of y, variable y will be useless to the search process. This is even more notorious if the number of equivalences is significant, considering that each two-variable equivalence originates a useless variable. The fact is whenever a new variable equivalence takes place, one of the variables ends up with no clauses. If this variable is left in the data structures, it will continue to be traversed as other variables, despite the existence of no associated clauses.

Let us consider now the implemented algorithm to identify two-variable equivalence during preprocessing. The primary goal was to choose space-efficient data structures, but also taking into account the computational overhead.

The generic algorithm, applied to each variable v_i from the CNF formula, can be organized as follows:

1. From the clauses of v_i, identify those that are binary, which means that can be responsible for new equivalences. A selected binary clause should be registered in a specific data structure *eq_clauses*, indexed by the other variable v_j, to be easily accessed later on.

2. After grouping each binary clause that contains variable v_i by the second variable v_j (whose entry is marked in our data structure *eq_clauses*), process each marked entry to identify existing equivalences.

3. Clean all used structures, which will be used in the next iteration.

Next we clarify how each entry *eq_clauses[v_j]* is organized, in order to optimize equivalence processing. Since we have binary clauses with v_i and v_j, we have to consider $2^2 = 4$ different clauses, regarding literal signs, since literal order is not relevant within the same clause. These clauses are $\omega_1 = l_i \vee l_j$, $\omega_2 = l_i \vee \neg l_j$, $\omega_3 = \neg l_i \vee l_j$ and $\omega_4 = \neg l_i \vee \neg l_j$. Figure 5-1 illustrates the data structure used by equivalence processing, considering we are trying to identify existing equivalences between variables v_i and v_j.

Due to the identification of an equivalence ($x \equiv y$), variable x has now the clauses that previously belonged to y. We will use the word *inheritance* to express this migration of clauses. So, we say variable x *inherited* variable y clauses.

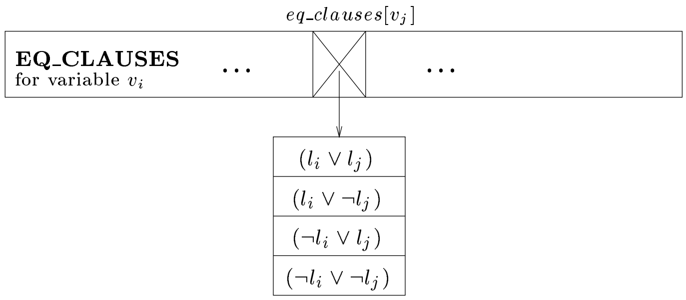


Figure 5-1: Data structure used by equivalence processing.

Clause Pattern	Equivalence
$(l_i \vee \neg l_j) \wedge (\neg l_i \vee l_j)$	$v_i \equiv v_j$
$(l_i \vee l_j) \wedge (\neg l_i \vee \neg l_j)$	$\neg(v_i \equiv v_j)$

Table 5.1: Clauses for Two-variable Equivalence

Whenever a variable inherits a binary clause, it is added to *eq_clauses*. Possible equivalences obtained from the inherited clause will only be analyzed in the next iteration. Figure 5-2 gives an example of successive iterations, due to the identification of two-variable equivalence and the consequent inheritance of clauses. We should note that a third and even more iterations could take place, generating a *snow ball* effect, depending on both the identification of more equivalences and the inheritance of binary clauses.

Using the same data structure and while looking for equivalences (based in clause pattern from table 5.1), we can also apply selective resolution, as shown in table 5.2, combining clauses in the remaining possibilities. Figure 5-3 shows how equivalence and resolution processing is done, whenever binary clauses were previously identified. Observe that the same data structure allows performing both operations.

As a consequence of applying resolution, we add new unit clauses to the clause database. Unit clauses are useful for boolean constraint propagation. BCP algorithm assigns the variable contained in a unit clause. Therefore, such assignment possibly originates more unit clauses, implying more variable assignments. However, BCP is performed *before* the two-variable equivalence algorithm. Henceforth, BCP must be applied after resolution/equivalence processing. Again this aspect was not considered in GRASP, since the

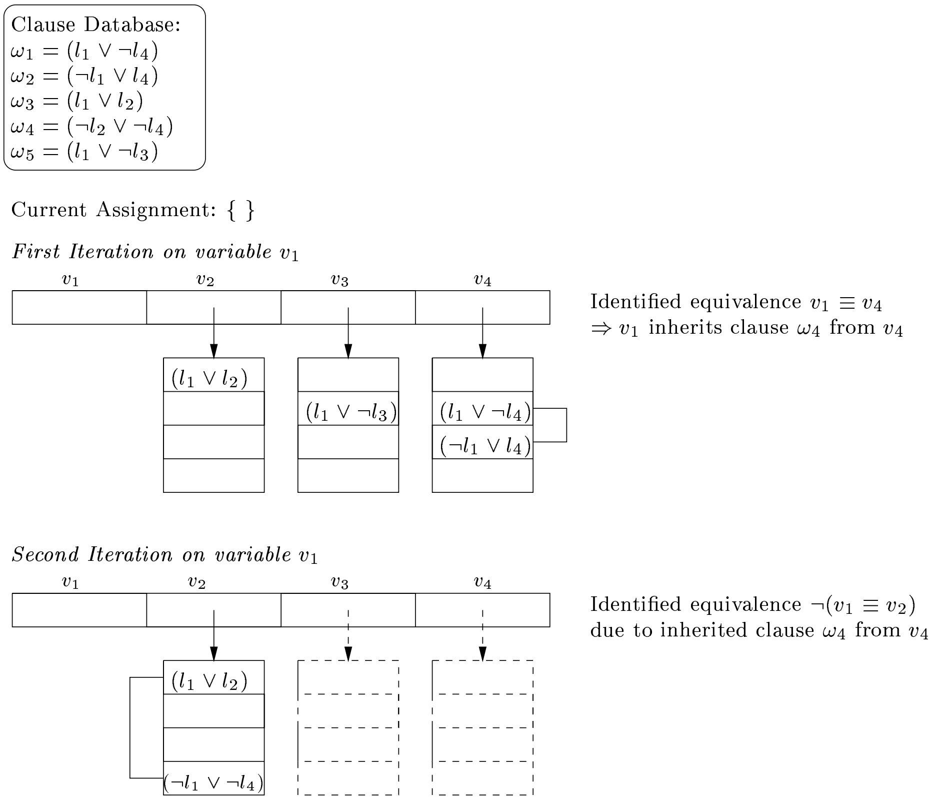


Figure 5-2: Data structure used by equivalence processing.

deduction engine was reduced to BCP and eventually to recursive learning. This means BCP takes place after equivalence identification provided unit clauses have been inferred. Moreover, there is a loop BCP/equivalences until there are no more unit clauses and no more equivalences.

Finally, notice that the binary clauses associated with v_i and referred to as ω_1, ω_2, ω_3 and ω_4, become tautologous clauses after equivalences. Hence, after the *first round* of preprocessing they are deleted. By first round we mean after BCP is applied for the first

Clause Pattern	Inferred Clause
$(l_i \vee l_j) \wedge (l_i \vee \neg l_j)$	(l_i)
$(\neg l_i \vee l_j) \wedge (\neg l_i \vee \neg l_j)$	$(\neg l_i)$
$(l_i \vee l_j) \wedge (\neg l_i \vee l_j)$	(l_j)
$(l_i \vee \neg l_j) \wedge (\neg l_i \vee \neg l_j)$	$(\neg l_i)$

Table 5.2: Clauses for Resolution

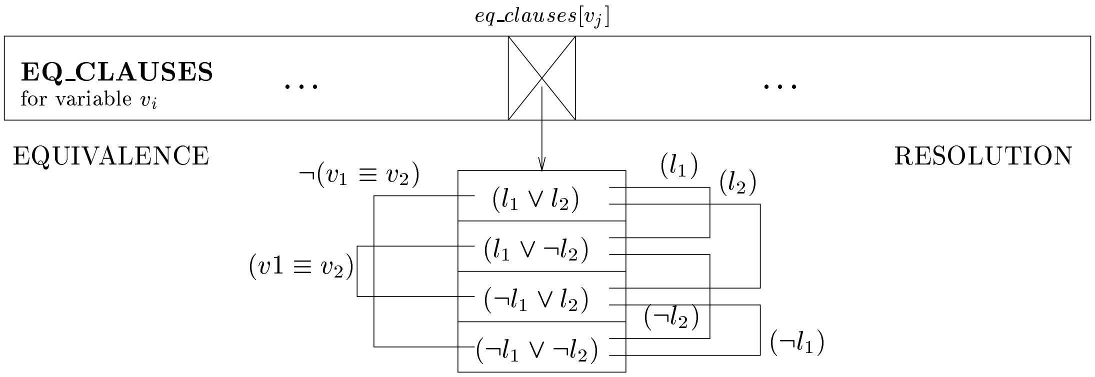


Figure 5-3: Equivalence and resolution processing.

time, and consequently when equivalences are identified for the first time. Under other circumstances, and similarly to what is done during the search, they will be temporary removed from the clause database. This additional simplification will be explained in section 5.3.2.

5.1.2 Two-variable Equivalence during the Search

The identification of two-variable equivalences during the search requires that we replace variables at each step of the search process, assuming that during the search a pair of variables can become dynamically equivalent. Integrating two-variable equivalence in GRASP presents two additional difficulties:

1. For a backtrack search algorithm, it is necessary to know when to replace or undo previous replacements of variables.

2. For an intelligent backtrack search algorithm, one has to consider special cases of clause recording, whenever conflicts involve clauses with variables replaced by equivalences.

Let us start by discussing the integration of two-variable equivalence in backtrack search algorithms. Figure 5-4 gives an example and shows the consequences of identifying an equivalence between variables x and y during the search process. With assignment $z = 0@d$, variables x and y become dynamically equivalent due to binary clauses ω_1 and ω_2. Considering the equivalence $(x \equiv y)$ takes place at decision level d, thus replacing

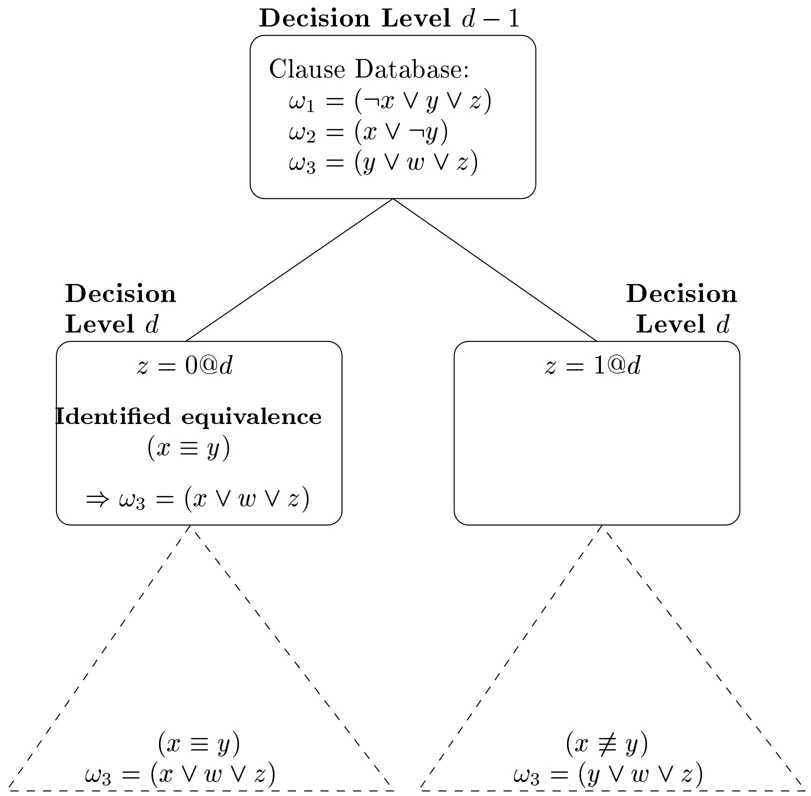


Figure 5-4: Two-variable Equivalence during the Search.

occurrence of y in ω_3 by x, the replacement is valid as long as the search is performed at deeper levels. If the search backtracks, the occurrences of x in ω_3 is now replaced by y again.

In parallel with identifying two-variable equivalence, and in order to undo those transformations when the search backtracks, all the needed information is recorded. Of course, additional data structures are required, as well as re-formulation of some concepts.

First, consider that identifying two-variable equivalence during the search implies replacing variables at different decision levels of the search process. This means a literal from a specific clause can be associated with different variables during the search, due to equivalences. For this reason, each literal now must have a set of variables and respective signs, which constitute a kind of literal *history*. Reporting again to figure 5-4, once the equivalence $(x \equiv y)$ is identified, thus replacing y from clause ω_3 by x, implies putting y in the literal history.

On the other hand, variable equivalences are identified at specific decision levels, that determine which equivalences have to be undone when the search backtracks. Consequently, within the clause database context, literals affected by equivalences have to be registered by decision level. Hence, in the clause database from figure 5-4 equivalence $(x \equiv y)$ has been registered at decision level d.

The next challenge is to guarantee the correctness of the creation of conflict clauses, involving clauses with literals changed by equivalences.

Let us translate the problem to the following questions: "Why does an equivalence between a pair of variables take place during the search?" "Why do clauses ω_i and ω_j originate a pair of equivalent variables at the current decision level, and do not at another decision level?" The answer is in the set of literals assigned 0 from both clauses ω_i and ω_j. Clearly, clauses ω_i and ω_j would not have become binary if it was not for those literals.

In conclusion, whenever we identify two equivalent variables x and y, due to the existence of binary clauses ω_i and ω_j, we select their literals assigned value 0. We will use the name *imply_equivalence_literals* to refer to this set of literals. These literals will be added to all clauses where y is replaced by x, assuming variable y is the eliminated one.

Example 5.1 *Consider the following clauses* $\omega_1 = (x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4)$, $\omega_2 = (\neg x_1 \vee x_3 \vee \neg x_4 \vee x_5)$ *and* $\omega_3 = (\neg x_1 \vee \neg x_3 \vee \neg x_6)$. *Moreover, assume that the current assignment is* $\{(x_2, 0), (x_4, 1), (x_5, 0)\}$. *Hence,* ω_1 *and* ω_2 *are binary clauses and* ω_3 *is a ternary clause.*

From clauses ω_1 *and* ω_2, *we derive equivalence* $(x_1 \equiv x_3)$ *and identify imply_equivalence_literals* $= \{x_2, \neg x_4, x_5\}$. *After replacing* x_3 *by* x_1, *in* ω_3 *we obtain a modified clause* $\omega_3 = (\neg x_1 \vee \neg x_1 \vee x_2 \vee \neg x_4 \vee x_5 \vee \neg x_6)$.

Notice that repetitions within *imply_equivalence_literals* set are avoided. On the contrary, nothing is done about repetitions in literals that occur in inherited clauses, after setting an equivalence. This aspect will be handled by additional simplifications; more concretely, by introducing the notion of *frozen literals*, that allows removing repeated literals.

Observe that clauses ω_1 and ω_2 are now tautologous clauses. By definition, a tautologous clauses is a clause that is always satisfiable. In this case, such clauses have become dynamically tautologous. During preprocessing, such clauses are deleted. During the search, they can be temporary *hidden*. We will refer to a clause as being a *hidden clause* if it was temporary removed from the clause database at a decision level d. This clause will remain hidden as long as the search is performed at a decision level $> d$.

5.2 Inference of Binary Clauses

One of the new simplification techniques, described in chapter 3, was the generalized inference of binary clauses. In the context of two-variable equivalence, the inference of binary clauses can originate more equivalences. From the several proposed rules for inferring binary clauses [50], we implemented 2B/1T and 0B/4T reasoning, since these two techniques alone supersede equivalency reasoning [41].

Inference of binary clauses using a clause pattern has never before been contemplated in GRASP. In the clause database we have either clauses that are part of the original problem specification or clauses that are derived from conflicts. With inference, we have to consider one additional type of clauses. For each decision level d, there is a list of inferred clauses created at that level. When performing the $erase(d)$ operation, all clauses that were inferred at the current decision level d are now deleted.

In what regards the implementation, the main goal was to match clause patterns using an efficient algorithm to minimize search cost whenever possible. Once more we pay efficiency with additional data structures. However, we have made an effort to always use, or whenever possible, the same data structures from two-variable equivalence.

5.2.1 2B/1T Reasoning

Clause pattern in the form of 2B/1T reasoning is represented by clauses $\omega_{B1} = (l_1 \vee \neg l_2)$, $\omega_{B2} = (l_1 \vee \neg l_3)$ and $\omega_T = (l_2 \vee l_3 \vee l_4)$. The inference of clause $(l_1 \vee l_4)$ can be explained by selective resolution.

Clause Pattern	Inferred Clause
$(l_1 \vee \neg l_2), (l_1 \vee \neg l_3), (l_2 \vee l_3 \vee l_4)$	$(l_1 \vee l_4)$
$(l_1 \vee \neg l_2), (l_1 \vee \neg l_3), (l_2 \vee l_3 \vee l_1)$	(l_1)

Table 5.3: 2B/1T Inference of Clauses.

Observe that binary clauses ω_{B1} and ω_{B2} have literal l_1 in common. The idea is to perform 2B/1T reasoning when trying to identify two-variable equivalence for a given variable v. This variable x is considered to have literal l_1, regarding the proposed clause pattern.

Due to the relation between algorithms for two-variable equivalence and 2B/1T reasoning, we will refer to the data structures explained earlier in sections 5.1.1 and 5.1.2.

Consider that we are applying the algorithm to identify two-variable equivalence for a given variable x. Hence, we have all binary clauses containing variable x in data structure *eq_clauses*. Remember each binary clause is indexed in *eq_clauses* by a second variable y. As a result, in order to match clause pattern for clauses ω_{B1} and ω_{B2}, we simply combine binary clauses having variable x and different variables y. However, we only try to match clause pattern starting from a binary clause containing variable y if equivalence $(x \equiv y)$ (or $\neg(x \equiv y)$) has not been previously identified.

Once clauses ω_{B1} and ω_{B2} are identified (and therefore literals l_1, l_2 and l_3), the next step consists in finding a ternary clause $\omega_T = (l_2 \vee l_3 \vee l_4)$, having both l_2 and l_3. If ω_T is identified, then add the inferred clause $(l_1 \vee l_4)$ to the clause database.

Finally, observe that from specific 2B/1T clause patterns we can also infer unit clauses. Different pattern possibilities are shown in table 5.3. Whenever a unit clause is inferred, it is added to the list of unit clauses. Afterwards, when equivalence and inference processing terminates, BCP operation is called again.

An example of the algorithm for applying 2B/1T reasoning is illustrated in figure 5-5. We have all binary clauses with l_1 available in the data structure *eq_clauses*. Such binary clauses are ω_1, ω_2 and ω_3. Two-variable equivalence was not identified. So, we combine the existing binary clauses, obtaining ω_{B1} and ω_{B2} for 2B/1T reasoning. We traverse the

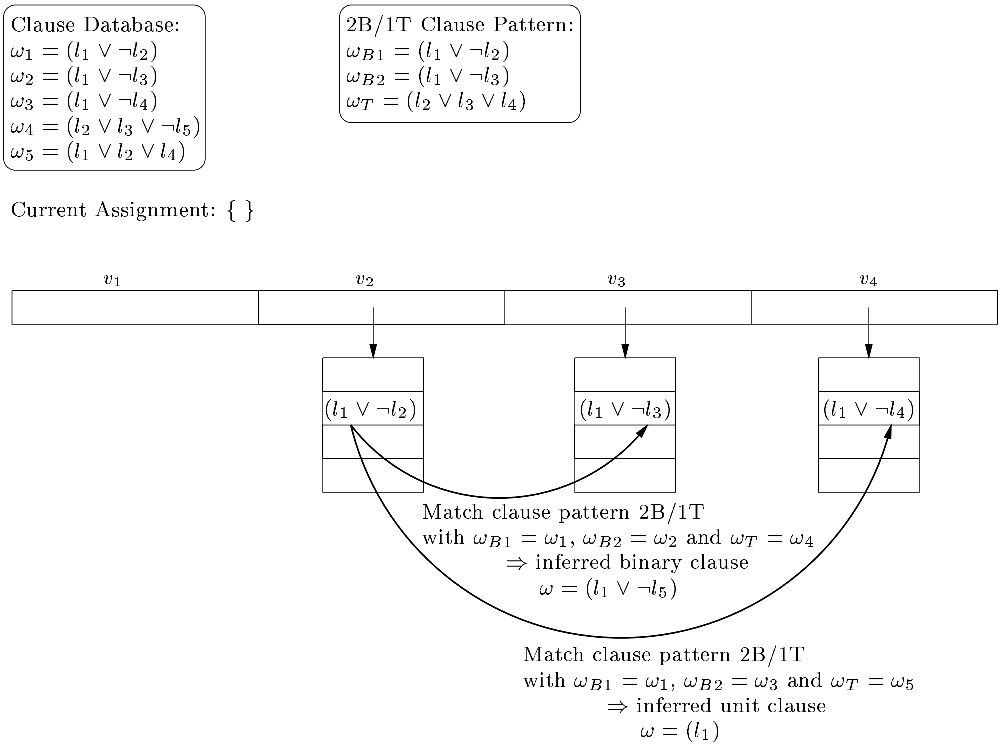


Figure 5-5: Example of 2B/1T reasoning.

Clause Pattern	Inferred Clause
$(l_1 \vee l_2 \vee l_3), (\neg l_1 \vee \neg l_2 \vee l_3), (l_1 \vee \neg l_2 \vee l_4), (\neg l_1 \vee l_2 \vee l_4)$	$(l_3 \vee l_4)$
$(l_1 \vee l_2 \vee l_3), (\neg l_1 \vee \neg l_2 \vee l_3), (l_1 \vee \neg l_2 \vee l_3), (\neg l_1 \vee l_2 \vee l_3)$	(l_3)

Table 5.4: 0B/4T Inference of Clauses.

remaining clauses to find ternary clause ω_T, therefore inferring a new clause.

5.2.2 0B/4T Reasoning

Clause pattern in the form of 0B/4T is represented by clauses $\omega_{T1} = (l_1 \vee l_2 \vee l_3)$, $\omega_{T2} = (\neg l_1 \vee \neg l_2 \vee l_3)$, $\omega_{T3} = (l_1 \vee \neg l_2 \vee l_4)$ and $\omega_{T4} = (\neg l_1 \vee l_2 \vee l_4)$. The inference of clause $(l_3 \vee l_4)$ can be explained by selective resolution. Notice that having $l_3 = l_4$ one therefore infers a unit clause. Table 5.4 summarizes clause pattern for 0B/4T reasoning, and also includes this possibility.

The approach followed to implement this simplification technique is similar to the algorithm proposed for 2B/1T reasoning. Once more, we start by identifying which literals 4T clauses have in common. We conclude that all ternary clauses have either literal l_1 or literal $\neg l_1$ and either literal l_2 or literal $\neg l_2$. Moreover, literals l_1 and l_2 are both contained in all 4T clauses, having literal signs combined in the four different possibilities.

Suppose the search is at decision level d. We are trying to match the clause pattern for 0B/4T reasoning beginning with clauses from variable v_1, since they have *changed* due to decision assignments made at decision level d. We use the word *change* applied to an unresolved clause meaning it has now (at the current decision level d) at least one more literal assigned value 0, compared with the previous decision level $d-1$.

The algorithm for 0B/4T reasoning can be described as follows:

1. Iterate the ternary clauses of variable v_1 and select the other literals to identify v_2 candidates, whose clauses will possible match clause pattern for 4T clauses.

2. For each selected variable v_2:

   (a) Iterate v_2 ternary clauses to identify common ternary clauses with variable v_1, and select the other literals to identify v_3 and v_4 candidates.

   (b) Iterate v_3 and v_4 specific ternary clauses (having literals from both variables v_1 and v_2) to match clause pattern.

To implement this algorithm, we created specific data structures to perform the iterations and optimize pattern matching. In concrete, and once ternary clauses common to v_1 and v_2 are selected, clauses are separated as having the patterns ω_{T1}, ω_{T2}, ω_{T3} or ω_{T4}. Finding any common literal l_3 to ω_{T1} and ω_{T2}, pattern matching consists in iterating l_3 with each common literal l_4 to ω_{T3} and ω_{T4}. Figure 5-6 shows a simple example of 0B/4T reasoning. From the given clause database, we match clause pattern 0B/4T with clauses ω_1,ω_3,ω_4 and ω_5, while clauses ω_2 and ω_6, even matching clause pattern 0B/4T for clauses ω_{T1} and ω_{T4}, would need other clauses (ω_{T2} and ω_{T3}) to match the target clause pattern 0B/4T as a whole.

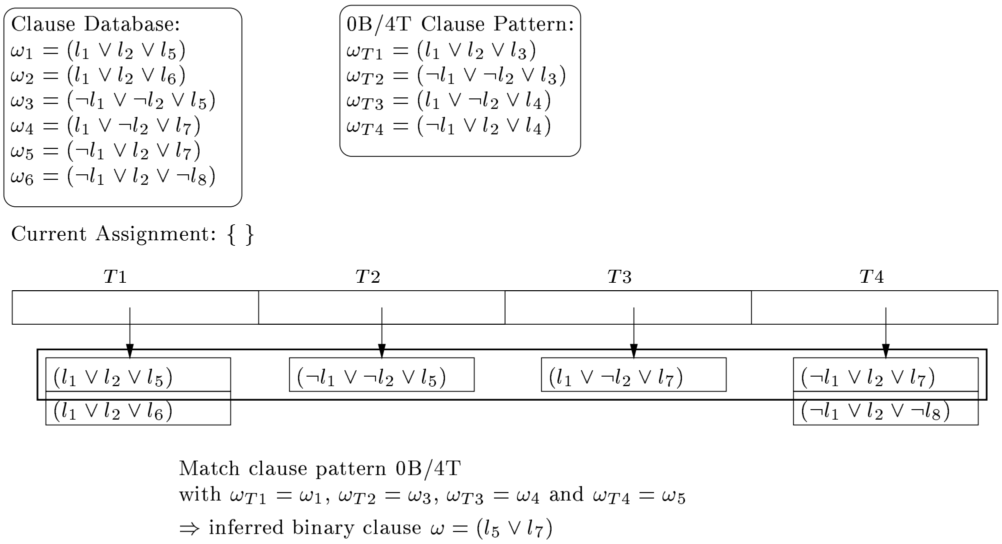


Figure 5-6: Example of 0B/4T reasoning.

5.3 Additional Simplifications

When explaining two-variable equivalence ($x \equiv y$), we alerted for the fact that tautologous clauses or repeated literals within a clause were obtained, as a consequence of replacing all occurrences of variable y by x. For these reasons, we established additional simplifications to be applied after setting equivalences. We will use the phrase *frozen literals* to refer to a repeated literal within a clause which is removed from the clause; hence, we suggest the name *freezing* to denote this action. Furthermore, we will apply the words *hidden clauses* to refer to tautologous clauses which are removed from the clause database, i.e., we *hide* these clauses since they are not considered to be part of the clause database.

The appearance of clauses with repeated literals or tautologous clauses is a consequence of identifying two-variable equivalence, considering that the original formula has neither clauses with repeated literals nor tautologous clauses. Henceforth, one has to keep in memory all equivalences obtained at the current decision level. Finally, additional simplifications are performed after identifying equivalences.

Considering again example 5.1 from section 5.1.2, remember we had identified the equivalence ($x_1 \equiv x_3$). After replacing occurrences of variable x_3 by x_1, we have the following clauses in the clause database:

$\omega_1 = (x_1 \vee x_2 \vee \neg x_1 \vee \neg x_4)$,

$\omega_2 = (\neg x_1 \vee x_1 \vee \neg x_4 \vee x_5)$ and

$\omega_3 = (\neg x_1 \vee \neg x_1 \vee x_2 \vee \neg x_4 \vee x_5 \vee \neg x_6)$.

When applying additional simplifications algorithm, we will traverse the clauses for variable x_1 in order to freeze literals (occurrence of repeated literals in the same clause) or hide clauses (having both literals x_1 and $\neg x_1$). Hence, clauses ω_1 and ω_2 will be hidden, and literal $\neg x_1$ from clause ω_3 will be frozen.

5.3.1 Frozen Literals

As already mentioned, clauses can have repeated literals due to equivalences. Although it does not affect search completeness and correction, it is certainly an overhead in the number of literals (and therefore an overhead to the entire search process) and also affects BCP. Moreover, identifying two-variable equivalence ($x \equiv y$) already implies an increasing in the number of literals of the clause database. Remember we have to add literals to every clause where we replace occurrence of variable y by x. These literals are all literals assigned value 0, obtained from binary clauses responsible for identifying two-variable equivalence (the named $imply\_equivalence\_literals$ in section 5.1.2).

On the other hand, having repeated literals in the same clause, some implicit binary clauses are not detected. In other words, a ternary clause with a repeated literal is not considered as a binary clause, even having only two different literals. Take for example clause $\omega = (x_1 \vee x_1 \vee x_2)$. Certainly, clause ω is a ternary clause for having three literals. Nonetheless, after freezing literal x_1 we obtain $\omega = (x_1 \vee x_2)$, revealing that clause ω is in fact a binary clause. (Observe that BCP is affected when considering unit clauses.)

The solution consists in temporary removing (*freezing*) a repeated literal, detected at a specific decision level d. Likewise two-variable equivalence, where an equivalence ($x \equiv y$) identified at a decision level d is valid as long as the search is performed at deeper levels, a frozen literal remains removed until the search backtracks to a decision level higher than the current one.

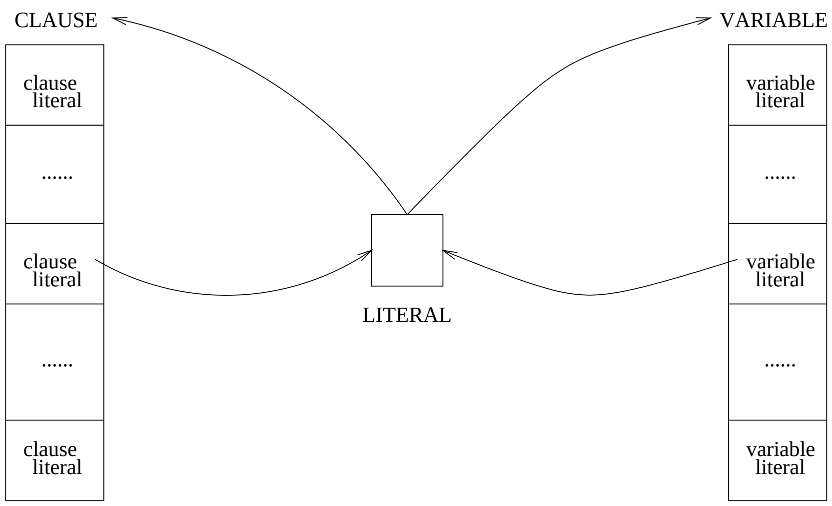


Figure 5-7: Clause database structure in GRASP.

In what regards the implementation details, we start by analyzing the GRASP clause database structure with figure 5-7. Observe that the presented structure establishes relations between a literal and the correspondent variable and clause. Therefore, a literal *knows* who are its variable and clause. There are also links between a clause and its literals, and a variable and its literals. Consequently, each clause and variable is able to access its literals.

The implementation of the freezing action is illustrated in figure 5-8. This action follows two-variable equivalence and for each decision level d has three distinct phases:

1. When a two-variable equivalence $(x \equiv y)$ is identified, record variable x, which has replaced the occurrences of another variable y.

2. For each recorded variable x, traverse all its literals to find out repetitions within a given clause.

3. Whenever a repeated literal l_f is detected, remove it from the respective clause and variable.

4. Append l_f to the frozen literals list at decision level d.

The opposite action, which takes place each time a decision level d is erased, consists in restoring each frozen literal to the original clause and variable.

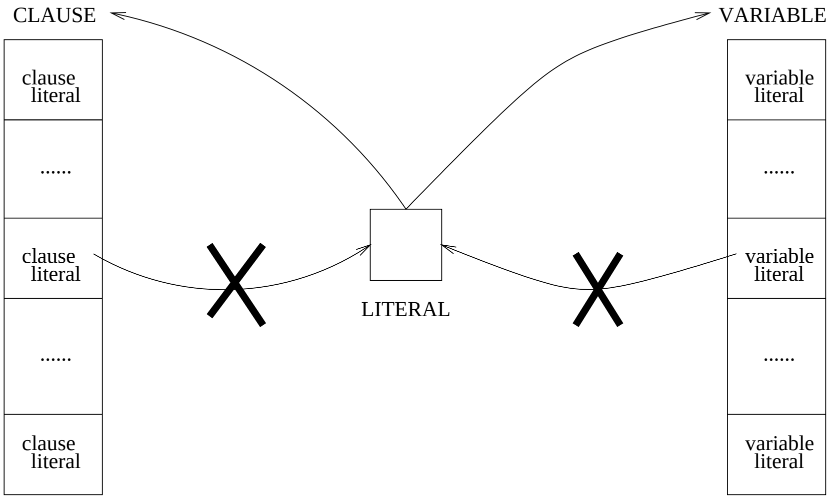


Figure 5-8: Freezing a literal.

5.3.2 Hidden Clauses

A tautologous clause is a clause that contains both a literal and its complement. Hence, such a clause is always satisfied. It does not affect search results but affects the decision engine, and therefore has a *polluting* effect in the search process.

Note that the tautology rule was already included in the original Davis-Putnam procedure. Nevertheless, as tautologous clauses can all be eliminated during preprocessing, this rule is often not applied.

However, with two-variable equivalence the situation changes radically. For each two-variable equivalence $(x \equiv y)$ (or $\neg(x \equiv y)$) we have at least two tautologous clauses. This number increases whenever the clause pattern defining an equivalence $(\neg x \vee y) \wedge (x \vee \neg y)$ (or $(x \vee y) \wedge (\neg x \vee \neg y)$) is matched by more than two clauses.

As usual we have different approaches in what regards hiding a clause during preprocessing and during the search. During preprocessing, tautologous clauses are simply deleted. On the other hand, during the search, such clauses can be *temporarily* removed from the clause database. They will have to be restored to the clause database as soon as the search backtracks to the decision level where these clauses were previously hidden.

Consider again the clause database structure from figure 5-7. Hence, if we want to remove a clause from the clause database, this implies removing links between its literals

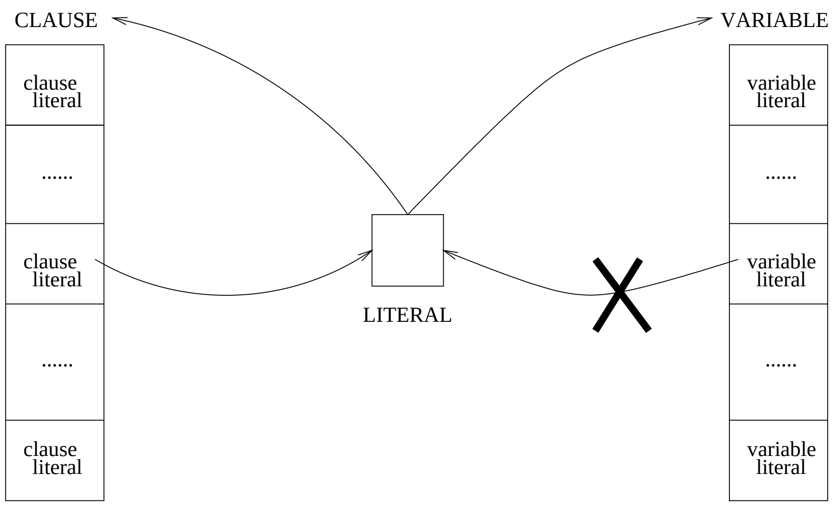


Figure 5-9: Hiding a clause.

and the respective variables (see figure 5-9). From now on, these literals can only be obtained from the respective clause.

Notice that breaking links between literals and variables is enough to hide a clause. In fact, when in GRASP we traverse the whole clause database, therefore traversing all literals, we start from variables to access literals. Once variables cannot reach literals from hidden clauses anymore, such clauses are no longer considered during the search. Of course clauses are only *temporarily* hidden, which means they will be added to the clause database if the search backtracks.

Similarly to freezing literals, clauses are hidden after equivalence processing for each decision level d. The proposed algorithm for hiding clauses is applied as follows:

1. When a two-variable equivalence $(x \equiv y)$ is identified due to existing binary clauses $\omega_1 = (\neg x \vee y)$ and $\omega_2 = (x \vee \neg y)$:

   (a) Record variable x, which has replaced the occurrences of another variable y.

   (b) Record clauses ω_1 and ω_2, which matched the two-variable equivalence clause pattern and therefore are now tautologous clauses.

2. Hide all recorded clauses ω_1 and ω_2.

3. For each recorded variable x, traverse all its literals, trying to find literal occurrences

with opposite signs within the same clause.

4. Whenever a clause ω is identified as being tautologous, hide all of its literals, removing each literal from the respective variable.

5. Append clause ω to the hidden clauses list at decision level d.

When erasing a decision level d, the opposite action is performed. Undoing the hide action to a clause means restoring links between each literal and the respective variable. Since literals *know* the variable they belong to, literals by themselves allow undoing hiding.

6

Experimental Results

This chapter evaluates the practical application of the algebraic simplification techniques described in chapter 3 and chapter 5. The main goal consisted in integrating such techniques as an extension to the usual *boolean constraint propagation* in backtrack search SAT algorithms. Moreover, we implemented such techniques in GRASP, an intelligent backtrack search SAT algorithm. As a result, algebraic simplification was integrated with non-chronological backtrack and clause recording.

We start by analyzing the possible ways of utilizing the proposed techniques, in concrete two-variable equivalence, 2B/1T reasoning and 0B/4T reasoning. Afterwards, these techniques are evaluated according to three main scenarios:

- The application of algebraic simplification techniques during preprocessing and during the search.
- The identification of two-variable equivalence, 2B/1T and 0B/4T clause patterns when using algebraic simplification techniques in intelligent backtrack search SAT algorithms.
- The effective contribution of 2B/1T and 0B/4T reasoning in identifying two-variable equivalence.

With the objective of evaluating the formula simplification ability of the proposed techniques, several representative problem instances from different real-world applica-

Table 6.1: Example Instances

Application Domain	# Instances
Pretolani (DIMACS) [36]	8
16-bit Parity (DIMACS) [36]	10
32-bit Parity (DIMACS) [36]	10
Combinational Equivalence Checking [39]	25
Bounded Model Checking [7]	34
Data Encryption Standard [60]	27
Superscalar Processor Verification [83]	40

tions were used. We selected these benchmarks since they are known to have a significant number of equivalency clauses [41]. As counterpart to having such characteristics, these instances are large, hard and structured, and most state-of-the-art SAT solvers are currently unable to solve them in a reasonable amount of time. The selected problem instances, distinguished by originating from different application domains, are shown in table 6.1, and can be obtained from the different authors [7, 36, 39, 60, 83].

Considering the main metrics we proposed to study, for each problem instance we present the following results:

- The number of variables (#var), clauses (#clause) and literals (#lit) before and after preprocessing, revealing the contribution of algebraic simplification techniques to simplify the original formula, as well as each instance real size when starting the search process.

- The number of identified two-variable equivalences (#eq), 2B/1T (#2B1T) and 0B/4T (#0B4T) inferred clauses, both the numbers obtained during preprocessing (#eq$_{pp}$, #2B1T$_{pp}$ and #0B4T$_{pp}$) and the total numbers for the whole search (#eq$_t$, #2B1T$_t$ and #0B4T$_t$). Our goal is to clearly separate results for preprocessing from results during the search.

- The number of two-variable equivalences identified during the search by decision node (#eq$_s$/node), either by applying two-variable equivalence or also 2B/1T and 0B/4T reasoning.

- The number of identified two-variable equivalence (#eq_vars), 2B/1T and 0B/4T inferred clauses (#2B1T and #0B/4T, respectively), as well as the number of de-

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
pret60_25	60	161	480	60	160	480
pret60_40	60	161	480	60	160	480
pret60_60	60	161	480	60	160	480
pret60_75	60	161	480	60	160	480
pret150_25	150	401	1200	150	400	1200
pret150_40	150	401	1200	150	400	1200
pret150_60	150	401	1200	150	400	1200
pret150_75	150	401	1200	150	400	1200

Table 6.2: Pretolani instances — formula simplification during preprocessing

cision nodes (Nodes), the number of non-chronological backtracks (NCB) and the number of recorded conflict clauses(CR). Observe that with these final results we obtain statistics for both algebraic simplification and intelligent search.

While GRASP has a large number of configuration options, for the experimental results given below, it was configured to use DLIS as the branching heuristic, to use relevance-based learning(4), to limit the size of clauses added to the clause database to 50 or fewer literals, and to limit the size of literals added to the inherited clauses (when identifying two-variable equivalence) to 2 or fewer literals. The maximum allowed CPU time was set to 500 seconds (on a PIII-733MHz Linux workstation)[1].

6.1 DIMACS Pretolani Instances

The DIMACS Pretolani instances [36] were contributed by Pretolani, inspired by Urquhart's construction [81]. They represent unsatisfiable instances of a graph 2-coloring with parity constraints. We have a total number of four instances of either 60 or 150 propositional variables.

From the experimental results, several conclusions can be drawn:

– Pretolani instances are not simplified during preprocessing for applying algebraic simplification. On the other hand, significant simplifications can be observed during the search.

[1]Instance entries with * indicate that the search terminated in the limit time

Instance	#eq$_{pp}$	#2B1T$_{pp}$	#0B4T$_{pp}$	#eq$_t$	#2B1T$_t$	#0B4T$_t$
pret60_25	0	0	0	863	3	3
pret60_40	0	0	0	709	0	3
pret60_60	0	0	0	645	18	8
pret60_75	0	0	0	755	0	10
pret150_25	0	0	0	3677	21	12
pret150_40	0	0	0	3266	8	18
pret150_60	0	0	0	3202	3	25
pret150_75	0	0	0	3097	2	11

Table 6.3: Pretolani instances — simplification during preprocessing and during the search

Instance	Equivalence			Equivalence & Inference		
	#eq$_{pp}$	#eq$_t$	#eq$_s$/node	#eq$_{pp}$	#eq$_t$	#eq$_s$/node
pret60_25	0	875	2	0	863	2
pret60_40	0	695	2	0	709	2
pret60_60	0	645	1	0	645	1
pret60_75	0	755	2	0	755	2
pret150_25	0	3721	11	0	3677	10
pret150_40	0	3392	10	0	3266	9
pret150_60	0	3335	9	0	3202	9
pret150_75	0	3079	9	0	3097	9

Table 6.4: Pretolani instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
pret60_25*	863	3	3	437	100	564
pret60_40*	709	0	3	370	85	484
pret60_60*	645	18	8	402	92	476
pret60_75*	755	0	10	338	74	443
pret150_25*	3677	21	12	2529	293	1252
pret150_40*	3266	8	18	2263	266	1141
pret150_60*	3202	3	25	1841	259	1125
pret150_75*	3097	2	11	2042	200	878

Table 6.5: Pretolani instances — final results

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
par16-1	1015	3310	8788	725	2761	7622
par16-1-c	317	1264	3670	317	1310	3762
par16-2	1015	3374	9044	732	2839	7906
par16-2-c	349	1392	4054	349	1443	4156
par16-3	1015	3344	8924	729	2803	7774
par16-3-c	334	1332	3874	334	1381	3972
par16-4	1015	3324	8844	720	2765	7658
par16-4-c	324	1292	3754	324	1343	3856
par16-5	1015	3358	8980	729	2817	7830
par16-5-c	341	1360	3958	341	1410	4058

Table 6.6: DIMACS 16-bit parity instances — formula simplification during preprocessing

Instance	$\#\mathrm{eq}_{pp}$	$\#\mathrm{2B1T}_{pp}$	$\#\mathrm{0B4T}_{pp}$	$\#\mathrm{eq}_{t}$	$\#\mathrm{2B1T}_{t}$	$\#\mathrm{0B4T}_{t}$
par16-1	290	31	0	55852	1395	0
par16-1-c	0	31	15	50852	1580	15
par16-2	283	31	0	80333	105	0
par16-2-c	0	31	20	64452	253	20
par16-3	286	31	0	66108	453	0
par16-3-c	0	31	18	38784	314	18
par16-4	295	31	0	58805	185	0
par16-4-c	0	31	20	33594	57	20
par16-5	286	31	0	56294	460	0
par16-5-c	0	31	19	56868	987	19

Table 6.7: 16-bit parity instances — simplification during preprocessing and during the search

– The number of identified two-variable equivalences is relevant, although the number of inferred clauses obtained from 2B/1T and 0B/4T reasoning is negligible. As a result, the contribution of clause inference to identification of two-variable equivalence is not conclusive.

6.2 DIMACS 16-bit Parity Instances

DIMACS 16-bit parity instances [36] were formulated by Selman et al. [3] at IJCAI'97. These instances arise from the problem of learning the parity function. The name parxx-y denotes a parity problem on xx bits, while y is simply the instance number. The name parxx-y-c denotes an instance identical to parxx-y, except from the fact that the instances have been simplified (to create an equivalent problem). All instances are satisfiable.

Instance	Equivalence			Equivalence & Inference		
	#eq$_{pp}$	#eq$_t$	#eq$_s$/node	#eq$_{pp}$	#eq$_t$	#eq$_s$/node
par16-1	290	61126	11	290	55852	12
par16-1-c	0	67036	12	0	50852	11
par16-2	283	94650	17	283	80333	17
par16-2-c	0	75024	14	0	64452	14
par16-3	286	53647	10	286	66108	14
par16-3-c	0	64208	12	0	38784	8
par16-4	295	62228	11	295	58805	12
par16-4-c	0	43920	8	0	33594	7
par16-5	286	73281	13	286	56294	12
par16-5-c	0	69214	13	0	56868	12

Table 6.8: 16-bit parity instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
par16-1	55852	1395	0	6955	20	20625
par16-1-c	50852	1580	15	6981	26	20696
par16-2	80333	105	0	6446	12	19042
par16-2-c	64452	253	20	6652	46	19648
par16-3	66108	453	0	6855	1	20520
par16-3-c*	38784	314	18	5606	4	16790
par16-4	58805	185	0	6723	73	19807
par16-4-c*	33594	57	20	4900	57	14417
par16-5	56294	460	0	6204	3	18488
par16-5-c	56868	987	19	7319	20	21751

Table 6.9: DIMACS 16-bit parity instances — final results

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
par32-1	3176	10277	27501	2073	8196	23205
par32-1-c	1315	5254	15390	1315	5427	15736
par32-2	3176	10253	27405	2087	8200	23165
par32-2-c	1303	5206	15246	1303	5377	15588
par32-3	3176	10297	27581	2106	8282	23417
par32-3-c	1325	5294	15510	1325	5467	15856
par32-4	3176	10313	27645	2124	8334	23553
par32-4-c	1333	5326	15606	1333	5495	15944
par32-5	3176	10325	27693	2130	8358	23625
par32-5-c	1339	5350	15678	1339	5519	16016

Table 6.10: 32-bit parity instances — formula simplification during preprocessing

From the experimental results, the following conclusions can be drawn:

– In DIMACS 16-bit parity instances the final number of variables after preprocessing is approximately two thirds of the original number for parxx-y instances. During preprocessing, there are no simplifications for the parxx-y-c instances, which can be understood since these instances have been simplified from parxx-y instances, in order to obtain an equivalent problem. Although a few clauses are inferred using 2B/1T reasoning during preprocessing, they do not affect the number of identified two-variable equivalence. 0B/4T reasoning only applies to parxx-y-c instances, even though allowing to infer a small number of clauses.

– Most part of the equivalences are identified during the search. 2B/1T reasoning allows inferring a significant number of binary clauses. However, the contribution of such clauses to the increasing of the number of equivalences is not conclusive. Moreover, 0B/4T reasoning does not apply during the search.

6.3 DIMACS 32-bit Parity Instances

DIMACS 32-bit parity instances [36] were also formulated by Selman et al. [3]. These instances arise from the problem of learning the parity function on 32 bits.

The experimental results lead to conclusions which are very similar to those that were drawn to 16-bit parity instances, namely:

Instance	$\#eq_{pp}$	$\#2B1T_{pp}$	$\#0B4T_{pp}$	$\#eq_t$	$\#2B1T_t$	$\#0B4T_t$
par32-1	1103	125	0	114115	125	0
par32-1-c	0	125	48	74466	125	48
par32-2	1089	125	0	122128	3624	0
par32-2-c	0	125	46	101029	459	46
par32-3	1070	125	0	137011	1493	0
par32-3-c	0	125	48	100812	1429	48
par32-4	1052	125	0	110572	562	0
par32-4-c	0	125	44	97870	163	44
par32-5	1046	125	0	133222	1207	0
par32-5-c	0	125	44	94769	210	44

Table 6.11: 32-bit parity instances — simplification during preprocessing and during the search

Instance	Equivalence			Equivalence & Inference		
	$\#eq_{pp}$	$\#eq_t$	$\#eq_s$/node	$\#eq_{pp}$	$\#eq_t$	$\#eq_s$/node
par32-1	1103	130306	24	1103	114115	24
par32-1-c	0	85552	16	0	74466	16
par32-2	1089	146365	27	1089	122128	26
par32-2-c	0	121072	23	0	101029	22
par32-3	1070	157690	29	1070	137011	29
par32-3-c	0	117676	22	0	100812	22
par32-4	1052	127477	24	1052	110572	23
par32-4-c	0	115142	21	0	97870	21
par32-5	1046	153110	28	1046	133222	28
par32-5-c	0	129882	24	0	94769	20

Table 6.12: 32-bit parity instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
par32-1	114115	125	0	5013	0	14968
par32-1-c	74466	125	48	5176	0	15450
par32-2	122128	3624	0	4566	0	13628
par32-2-c	101029	459	46	4889	0	14583
par32-3	137011	1493	0	5333	0	15914
par32-3-c	100812	1429	48	4756	0	14182
par32-4	110572	562	0	4673	0	13936
par32-4-c	97870	163	44	4820	0	14374
par32-5	133222	1207	0	4582	0	13660
par32-5-c	94769	210	44	4417	0	13168

Table 6.13: 32-bit parity instances — final results

– The final number of variables after preprocessing is approximately two thirds of the original number for parxx-y instances.

– Clauses inferred by 2B/1T reasoning are not likely to affect the number of equivalences, neither during preprocessing nor during the search and there are more inferred clauses during the search than during preprocessing.

– 0B/4T reasoning only applies to parxx-y-c instances during preprocessing, allowing to infer a small number of clauses, and do not apply during the search.

– Most part of the equivalences are identified during the search.

6.4 Combinational Equivalence Checking Instances

Combinational equivalence checking instances were contributed by [39]. The problem of combinational equivalence checking (CEC) forms one of the key components of the current verification methodology for digital systems.

Analysis of the experimental results yields the following conclusions:

– In general, the final number of variables after preprocessing is significantly reduced when applying algebraic simplification techniques. In some cases (e.g. *c1908* and *c2670_bug*) the final number of variables is near a half of the original number.

– Clauses inferred by 2B/1T reasoning do cause an increase in the number of identified two-variable equivalences, but only during the search. This is true even when are inferred more clauses during preprocessing than during the search (e.g. *c1908_bug*).

– 0B/4T reasoning only applies to the c499 and c499-s instances and during preprocessing.

– Most part of the variable equivalences are identified during the search.

6.5 Bounded Model Checking Instances

Bounded model checking (BMC) instances were contributed in [7]. Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
c1355	1294	3662	8478	1086	3950	9054
c1355-s	1294	3670	8502	1086	3958	9078
c1908	1917	5096	11518	986	3694	8714
c1908-s	1919	5108	11550	991	3716	8766
c1908_bug	1919	5100	11526	986	3690	8706
c2670	2703	6756	15326	1597	5018	11796
c2670-s	2940	7436	16892	1772	5731	13425
c2670_bug	2708	6696	15136	1524	4772	11234
c3540	3450	9326	21176	1965	7435	17394
c3540-s	3499	9526	21678	2029	7750	18126
c3540_bug	3446	9316	21154	1963	7431	17384
c432	389	1115	2709	294	1017	2513
c432-s	392	1128	2742	298	1032	2550
c499	606	1870	4894	462	1630	4414
c499-s	606	1878	4918	462	1638	4438
c5315	5399	15024	34628	3347	12067	28632
c5315-s	5408	15110	34868	3372	12205	28975
c5315_bug	5396	15004	34574	3344	12040	28564
c6288	5008	14813	34485	4846	17920	40667
c6288-s	5025	14882	34658	4865	18044	40918
c7552	7652	20423	46377	4576	16027	37448
c7552-s	7767	20812	47314	4722	16593	38737
c7552_bug	7559	20109	45621	4508	15761	36788
c880	957	2590	5974	727	2268	5330
c880-s	957	2590	5974	727	2268	5330

Table 6.14: CEC instances — formula simplification during preprocessing

Instance	#eq$_{pp}$	#2B1T$_{pp}$	#0B4T$_{pp}$	#eq$_t$	#2B1T$_t$	#0B4T$_t$
c1355	208	704	0	79066	1264	0
c1355-s	208	704	0	86593	835	0
c1908	931	460	0	139109	505	0
c1908-s	928	464	0	183863	548	0
c1908_bug	933	456	0	156131	457	0
c2670	1106	414	0	174342	671	0
c2670-s	1168	562	0	190674	760	0
c2670_bug	1184	384	0	2329	400	0
c3540	1485	1079	0	127712	1513	0
c3540-s	1470	1164	0	195697	3990	0
c3540_bug	1483	1081	0	145592	2102	0
c432	95	92	0	1637	169	0
c432-s	94	92	0	1647	165	0
c499	144	16	32	31177	32	32
c499-s	144	16	32	29814	32	32
c5315	2052	1063	0	186443	6839	0
c5315-s	2036	1080	0	205480	7629	0
c5315_bug	2052	1056	0	200089	9216	0
c6288	162	3399	0	232011	3405	0
c6288-s	160	3418	0	232010	3424	0
c7552	3076	1605	0	74643	1690	0
c7552-s	3045	1710	0	82665	1841	0
c7552_bug	3051	1603	0	8980	1619	0
c880	230	138	0	90804	1628	0
c880-s	230	138	0	90827	1628	0

Table 6.15: CEC instances — simplification during preprocessing and during the search

Instance	Equivalence			Equivalence & Inference		
	#eq$_{pp}$	#eq$_t$	#eq$_s$/node	#eq$_{pp}$	#eq$_t$	#eq$_s$/node
c1355	208	77493	5	208	79066	6
c1355-s	208	98402	6	208	86593	6
c1908	931	106024	7	931	139109	10
c1908-s	928	103716	6	928	183863	14
c1908_bug	933	98983	6	933	156131	12
c2670	1106	142692	9	1106	174342	13
c2670-s	1168	157034	10	1168	190674	14
c2670_bug	1184	2927	0	1184	2329	0
c3540	1485	142954	9	1485	127712	9
c3540-s	1470	147609	9	1470	195697	15
c3540_bug	1483	138137	9	1483	145592	11
c432	95	1646	0	95	1637	0
c432-s	94	2030	0	94	1647	0
c499	144	35281	2	144	31177	2
c499-s	144	36241	2	144	29814	2
c5315	2052	195948	13	2052	186443	14
c5315-s	2036	178246	11	2036	205480	15
c5315_bug	2052	196066	13	2052	200089	15
c6288	162	305306	20	162	232011	18
c6288-s	160	324283	21	160	232010	18
c7552	3071	90085	5	3076	74643	5
c7552-s	3038	45877	2	3045	82665	6
c7552_bug	3046	103895	6	3051	8980	0
c880	230	103965	6	230	90804	7
c880-s	230	104000	6	230	90827	7

Table 6.16: CEC instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
c1355	79066	1264	0	19759	589	56488
c1355-s	86593	835	0	19194	1054	52956
c1908	139109	505	0	16094	2006	37978
c1908-s	183863	548	0	22109	2678	53807
c1908_bug*	156131	457	0	16490	2173	41018
c2670	174342	671	0	85778	10578	46282
c2670-s	190674	760	0	69016	15429	69068
c2670_bug*	2329	400	0	662	94	635
c3540	127712	1513	0	12584	1439	30790
c3540-s	195697	3990	0	18316	2426	43407
c3540_bug	145592	2102	0	14163	1668	34932
c432*	1637	169	0	804	70	1813
c432-s*	1647	165	0	788	86	1593
c499	31177	32	32	12477	1863	25194
c499-s	29814	32	32	12632	1907	27277
c5315	186443	6839	0	29850	4087	21517
c5315-s	205480	7629	0	33666	4079	21117
c5315_bug	200089	9216	0	32027	4013	20040
c6288	232011	3405	0	2935	0	8726
c6288-s	232010	3424	0	2935	0	8726
c7552	74643	1690	0	50844	3550	21064
c7552-s	82665	1841	0	51452	4451	29955
c7552_bug*	8980	1619	0	3163	472	2450
c880	90804	1628	0	12754	1707	28724
c880-s	90827	1628	0	12756	1707	28728

Table 6.17: CEC instances — final results

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
barrel2	50	159	424	40	151	408
barrel3	275	942	2530	233	930	2506
barrel4	578	2035	5508	506	2051	5540
barrel5	1407	5383	14814	1242	5503	15054
barrel6	2306	8931	24664	2072	9219	25240
barrel7	3523	13765	38112	3208	14311	39204
barrel8	5106	20083	55716	4698	20995	57540
barrel9	8903	36606	102370	8219	35238	99634
longmult0	437	1206	2872	437	1206	2872
longmult1	791	2335	5617	475	1759	4283
longmult2	1164	3525	8507	758	2808	6876
longmult3	1555	4767	11517	1059	3915	9601
longmult4	1966	6069	14667	1380	5088	12478
longmult5	2397	7431	17957	1721	6327	15507
longmult6	2848	8853	21387	2082	7632	18688
longmult7	3319	10335	24957	2463	9003	22021
longmult8	3810	11877	28667	2864	10440	25506
longmult9	4231	13479	32517	3285	11943	29143
longmult10	4852	15141	36507	3726	13512	32932
longmult11	5403	16863	40637	4187	15147	36873
longmult12	5974	18645	44907	4668	16848	40966
longmult13	6565	20487	49317	5169	18615	45211
longmult14	7176	22389	53867	5690	20448	49608
longmult15	7807	24351	58557	6227	22339	54137
queueinvar2	116	399	998	102	429	1056
queueinvar4	256	955	2421	232	1080	2670
queueinvar6	463	2265	5723	432	3014	7221
queueinvar8	512	2273	5878	472	2545	6422
queueinvar10	886	5622	14355	840	7907	18925
queueinvar12	1112	7335	18671	1060	10642	25285
queueinvar14	1370	9313	23650	1312	13658	32340
queueinvar16	1168	6496	17172	1100	7232	18644
queueinvar18	2081	17368	44672	2008	25515	60966
queueinvar20	2435	20671	53074	2356	30776	73284

Table 6.18: BMC instances — formula simplification during preprocessing

used to traverse the model defined by the system and check if the specification holds or not. All instances are unsatisfiable.

From the experimental results, several conclusions can be drawn:

– In general, the final number of variables after preprocessing is not significantly reduced when applying algebraic simplification techniques.

– On the other hand, for *queueinvar* instances, one can observe an increase in the final number of clauses and consequently literals, due to the application of clause inference.

– 2B/1T and 0B/4T reasoning applies mostly during preprocessing. On the other hand,

Instance	#eq$_{pp}$	#2B1T$_{pp}$	#0B4T$_{pp}$	#eq$_t$	#2B1T$_t$	#0B4T$_t$
barrel2	10	0	12	34	0	12
barrel3	42	0	72	472	0	72
barrel4	72	0	160	1588	0	160
barrel5	165	0	450	42783	1	450
barrel6	234	0	756	8758	0	756
barrel7	315	0	1176	13910	0	1176
barrel9	684	0	3240	9254	0	3240
longmult0	0	0	0	0	0	0
longmult1	316	53	0	377	53	0
longmult2	406	92	0	721	92	0
longmult3	496	137	0	1467	147	0
longmult4	586	188	0	4267	193	0
longmult5	676	245	0	17566	253	0
longmult6	766	308	0	76454	342	0
longmult7	856	377	0	119935	626	0
longmult8	946	452	0	208198	551	0
longmult9	1036	533	0	265870	578	0
longmult10	1126	620	0	333336	870	0
longmult11	1216	713	0	381247	927	0
longmult12	1306	812	0	300222	1295	0
longmult13	1396	917	0	436449	1124	0
longmult14	1486	1028	0	437211	1146	0
longmult15	1580	1145	0	508790	1333	0
queueinvar2	14	52	4	94	60	4
queueinvar4	24	165	7	376	187	9
queueinvar6	31	802	9	1100	893	10
queueinvar8	40	343	9	1760	362	9
queueinvar10	46	2366	11	3507	2435	13
queueinvar12	52	3400	11	5236	3468	11
queueinvar14	58	4450	11	6426	4513	11
queueinvar16	68	861	11	7704	889	11
queueinvar18	73	8280	13	7327	8338	13
queueinvar20	79	10250	13	5622	10300	13

Table 6.19: BMC instances — simplification during preprocessing and during the search

Instance	Equivalence			Equivalence & Inference		
	$\#\text{eq}_{pp}$	$\#\text{eq}_t$	$\#\text{eq}_s$/node	$\#\text{eq}_{pp}$	$\#\text{eq}_t$	$\#\text{eq}_s$/node
barrel2	10	34	0	10	34	0
barrel3	42	472	1	42	472	1
barrel4	72	1588	3	72	1588	3
barrel5	165	72421	170	165	42783	110
barrel6	234	74913	175	234	8758	22
barrel7	315	14066	32	315	13910	3
barrel8	408	17029	39	—	—	—
barrel9	684	13170	29	684	9254	2
longmult0	0	0	0	0	0	0
longmult1	316	377	0	316	377	0
longmult2	406	726	0	406	721	0
longmult3	496	1467	2	496	1467	2
longmult4	586	4353	8	586	4267	9
longmult5	676	16833	38	676	17566	43
longmult6	766	79689	185	766	76454	197
longmult7	856	148099	346	856	119935	310
longmult8	946	202070	473	946	208198	539
longmult9	1036	342467	803	1036	265870	689
longmult10	1126	400733	940	1126	333336	865
longmult11	1216	420237	985	1216	381247	989
longmult12	1306	432486	1014	1306	300222	778
longmult13	1396	467423	1096	1396	436449	1132
longmult14	1486	403009	944	1486	437211	1134
longmult15	1580	431080	1010	1580	508790	1320
queueinvar2	14	91	0	14	94	0
queueinvar4	24	369	0	24	376	0
queueinvar6	31	1378	3	31	1100	2
queueinvar8	40	1728	3	40	1760	4
queueinvar10	46	4383	10	46	3507	9
queueinvar12	52	6040	14	52	5236	13
queueinvar14	58	8329	19	58	6426	16
queueinvar16	68	7983	18	68	7704	19
queueinvar18	73	7312	17	73	7327	18
queueinvar20	79	8189	19	79	5622	14

Table 6.20: BMC instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
barrel2*	34	0	12	1	1	3
barrel3*	472	0	72	23	1	68
barrel4*	1588	0	160	60	1	180
barrel5	42783	1	450	2267	17	6647
barrel6	8758	0	756	2007	15	5907
barrel7	13910	0	1176	4002	36	11686
barrel9	9254	0	3240	3446	304	9048
longmult0*	0	0	0	0	0	0
longmult1*	377	53	0	2	0	6
longmult2*	721	92	0	27	2	32
longmult3*	1467	147	0	91	5	160
longmult4*	4267	193	0	292	11	699
longmult5*	17566	253	0	1212	56	2991
longmult6*	76454	342	0	4361	123	12010
longmult7	119935	626	0	6038	123	17212
longmult8	208198	551	0	7448	153	21514
longmult9	265870	578	0	8704	233	24984
longmult10	333336	870	0	10804	459	30519
longmult11	381247	927	0	10620	939	28395
longmult12	300222	1295	0	10441	1184	26705
longmult13	436449	1124	0	10190	326	28942
longmult14	437211	1146	0	10703	301	30832
longmult15	508790	1333	0	10995	197	32122
queueinvar2*	94	60	4	15	0	47
queueinvar4*	376	187	9	77	1	246
queueinvar6*	1100	893	10	230	10	684
queueinvar8*	1760	362	9	384	4	1198
queueinvar10*	3507	2435	13	1160	66	3018
queueinvar12*	5236	3468	11	1729	84	4503
queueinvar14*	6426	4513	11	2619	149	6678
queueinvar16*	7704	889	11	3512	208	8932
queueinvar18	7327	8338	13	2250	79	6187
queueinvar20	5622	10300	13	2159	81	5544

Table 6.21: BMC instances — final results

results clearly associate different forms of reasoning with the distinct class benchmarks. In concrete *barrel* instances have clauses inferred by 0B/4T reasoning, *longmult* instances have clauses inferred by 2B/1T reasoning and *queueinvar* instances have clauses inferred by 2B/1T and 0B/4T reasoning.

– All instances, but in particular *longmult* instances, have an enormous increasing of two-variable equivalences identified during the search.

6.6 Data Encryption Standard Instances

Data encryption standard (DES) instances were contributed by Massacci [60]. They are SAT-encoding of cryptographic key search problems. The basic idea of the problem is to encode into a propositional formula the input/output relation of a cryptographic algorithm. In this way the bit values of the plaintext, the (secret) key and the ciphertext are encoded into propositional variables and the formula is true if and only if the ciphertext corresponds to the plaintext encrypted with the key. The original instances include a huge number of variables with no clauses. All instances are satisfiable.

Let us draw some conclusions from the results obtained with the DES instances:

– There are no two-variable equivalence during preprocessing, as well as inference of clauses applying 0B/4T reasoning. Nonetheless, during preprocessing are inferred clauses by 2B/1T reasoning, thus increasing the final number of clauses in the formula.

– Many equivalences can be observed during the search, as well as inference of clauses applying 2B/1T reasoning. Instances *cnf-r1-b*-k1.** are an exception for matching clause pattern 0B/4T during the search.

– 2B/1T is likely to contribute to the identification of two-variable equivalence for *cnf-r3-b*-k1.** instances. On the contrary, 2B/1T reasoning does not contribute to the identification of more two-variable equivalence for *cnf-r4-b*-k1.** instances.

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
cnf-r1-b1-k1.1	4644	1731	5285	4644	1752	5327
cnf-r1-b1-k1.2	135732	2124	6252	135732	2132	6268
cnf-r1-b2-k1.1	135732	3832	11423	135732	3893	11545
cnf-r1-b2-k1.2	397893	3186	9797	397893	3277	9979
cnf-r1-b3-k1.1	266821	5143	15469	266821	5345	15873
cnf-r1-b3-k1.2	660004	5206	15689	660004	5399	16075
cnf-r1-b4-k1.1	397893	6892	20597	397893	7210	21233
cnf-r1-b4-k1.2	922148	6646	20042	922148	6913	20576
cnf-r3-b1-k1.1	21536	8966	24748	21536	9088	24992
cnf-r3-b1-k1.2	43072	8966	24748	43072	9088	24992
cnf-r3-b2-k1.1	43072	8966	24748	43072	9088	24992
cnf-r3-b2-k1.2	414752	17960	49592	414752	18216	50104
cnf-r3-b3-k1.1	283680	26778	73980	283680	27155	74734
cnf-r3-b3-k1.2	676896	27503	75760	676896	27901	76556
cnf-r3-b4-k1.1	414752	35817	98918	414752	36322	99928
cnf-r3-b4-k1.2	939040	35963	99293	939040	36484	100335
cnf-r4-b1-k1.1	29728	14812	39764	29728	14932	40004
cnf-r4-b1-k1.2	160800	14812	39764	160800	14932	40004
cnf-r4-b2-k1.1	160800	29624	79528	160800	29864	80008
cnf-r4-b2-k1.2	1995808	236992	636224	1995808	236992	636224
cnf-r4-b3-k1.1	291872	44436	119292	291872	44796	120012
cnf-r4-b3-k1.2	685088	44436	119292	685088	44796	120012
cnf-r4-b4-k1.1	422944	59248	159056	422944	59728	160016
cnf-r4-b4-k1.2	947232	59248	159056	947232	59728	160016
cnf-r4-b8-k2.1	947232	118496	318112	947232	119456	320032
cnf-r4-b16-k2.1	1192070	18472	49394	1192070	18592	49634
cnf-r4-b16-k2.2	1192070	18472	49394	1192070	18592	49634

Table 6.22: DES instances — formula simplification during preprocessing

Instance	#eq$_{pp}$	#2B1T$_{pp}$	#0B4T$_{pp}$	#eq$_t$	#2B1T$_t$	#0B4T$_t$
cnf-r1-b1-k1.1	0	21	0	9	27	2
cnf-r1-b1-k1.2	0	8	0	16	16	0
cnf-r1-b2-k1.1	0	61	0	36	160	0
cnf-r1-b2-k1.2	0	91	0	25	169	10
cnf-r1-b3-k1.1	0	202	0	32	341	5
cnf-r1-b3-k1.2	0	193	0	39	393	31
cnf-r1-b4-k1.1	0	318	0	39	469	10
cnf-r1-b4-k1.2	0	267	0	67	517	64
cnf-r3-b1-k1.1	0	122	0	12861	1011	1
cnf-r3-b1-k1.2	0	122	0	12605	992	1
cnf-r3-b2-k1.1	0	122	0	12640	994	1
cnf-r3-b2-k1.2	0	256	0	11695	917	0
cnf-r3-b3-k1.1	0	377	0	18288	2185	6
cnf-r3-b3-k1.2	0	398	0	9133	1537	0
cnf-r3-b4-k1.1	0	505	0	19626	3982	2
cnf-r3-b4-k1.2	0	521	0	5238	1285	0
cnf-r4-b1-k1.1	0	120	0	18187	356	0
cnf-r4-b1-k1.2	0	120	0	14256	147	0
cnf-r4-b2-k1.1	0	240	0	28990	598	0
cnf-r4-b2-k1.2	0	240	0	16613	805	0
cnf-r4-b3-k1.1	0	360	0	23378	588	0
cnf-r4-b3-k1.2	0	360	0	10872	530	0
cnf-r4-b4-k1.1	0	480	0	19314	797	0
cnf-r4-b4-k1.2	0	480	0	2433	596	0
cnf-r4-b8-k2.1	0	960	0	296	1015	0
cnf-r4-b16-k2.1	0	120	0	77	123	0

Table 6.23: DES instances — simplification during preprocessing and during the search

Instance	Equivalence			Equivalence & Inference		
	#eq$_{pp}$	#eq$_t$	#eq$_s$/node	#eq$_{pp}$	#eq$_t$	#eq$_s$/node
cnf-r1-b1-k1.1	0	9	0	0	9	0
cnf-r1-b1-k1.2	0	16	1	0	16	1
cnf-r1-b2-k1.1	0	43	3	0	36	3
cnf-r1-b2-k1.2	0	24	1	0	25	2
cnf-r1-b3-k1.1	0	31	2	0	32	2
cnf-r1-b3-k1.2	0	48	3	0	39	3
cnf-r1-b4-k1.1	0	62	4	0	39	3
cnf-r1-b4-k1.2	0	88	6	0	67	6
cnf-r3-b1-k1.1	0	15577	1112	0	12861	1169
cnf-r3-b1-k1.2	0	15576	1112	0	12605	1145
cnf-r3-b2-k1.1	0	15550	1110	0	12640	1149
cnf-r3-b2-k1.2	0	10851	775	0	11695	1063
cnf-r3-b3-k1.1	0	19929	1423	0	18288	1662
cnf-r3-b3-k1.2	0	11291	806	0	9133	830
cnf-r3-b4-k1.1	0	24275	1733	0	19626	1784
cnf-r3-b4-k1.2	0	12425	887	0	5238	476
cnf-r4-b1-k1.1	0	23781	1698	0	18187	1653
cnf-r4-b1-k1.2	0	18232	1302	0	14256	1296
cnf-r4-b2-k1.1	0	39020	2787	0	28990	2635
cnf-r4-b2-k1.2	0	28592	2042	0	16613	1510
cnf-r4-b3-k1.1	0	46556	3325	0	23378	2125
cnf-r4-b3-k1.2	0	22818	1629	0	10872	988
cnf-r4-b4-k1.1	0	37389	2670	0	19314	1755
cnf-r4-b4-k1.2	0	5703	407	0	2433	221
cnf-r4-b8-k2.1	0	364	26	0	296	26
cnf-r4-b16-k2.1	0	78	5	0	77	7
cnf-r4-b16-k2.2	0	326	23	—	—	—

Table 6.24: DES instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
cnf-r1-b1-k1.1*	9	27	2	55	3	21
cnf-r1-b1-k1.2*	16	16	0	96	9	38
cnf-r1-b2-k1.1*	36	160	0	161	20	73
cnf-r1-b2-k1.2*	25	169	10	110	13	51
cnf-r1-b3-k1.1*	32	341	5	115	14	52
cnf-r1-b3-k1.2*	39	393	31	84	10	37
cnf-r1-b4-k1.1*	39	469	10	87	8	36
cnf-r1-b4-k1.2*	67	517	64	54	5	21
cnf-r3-b1-k1.1	12861	1011	1	6050	1076	12485
cnf-r3-b1-k1.2	12605	992	1	5963	1066	12275
cnf-r3-b2-k1.1	12640	994	1	5980	1068	12302
cnf-r3-b2-k1.2	11695	917	0	4638	1079	6638
cnf-r3-b3-k1.1	18288	2185	6	4846	1151	6818
cnf-r3-b3-k1.2	9133	1537	0	2771	687	3695
cnf-r3-b4-k1.1	19626	3982	2	3402	781	4826
cnf-r3-b4-k1.2	5238	1285	0	926	196	1168
cnf-r4-b1-k1.1	18187	356	0	4803	599	11668
cnf-r4-b1-k1.2	14256	147	0	4200	543	9793
cnf-r4-b2-k1.1	28990	598	0	4306	789	8784
cnf-r4-b2-k1.2	16613	805	0	2805	552	5596
cnf-r4-b3-k1.1	23378	588	0	2473	442	4926
cnf-r4-b3-k1.2	10872	530	0	1384	212	2741
cnf-r4-b4-k1.1	19314	797	0	1689	287	3491
cnf-r4-b4-k1.2	2433	596	0	187	19	340
cnf-r4-b8-k2.1	296	1015	0	11	0	0
cnf-r4-b16-k2.1	77	123	0	28	0	0

Table 6.25: DES instances — final results

6.7 Superscalar Processor Verification Instances

Superscalar processor verification (SSS) instances were contributed in [83]. These instances are generated from the formal verification of buggy variants of a dual-issue superscalar microprocessor with exceptions, multicycle functional units, and branch prediction. All instances are satisfiable.

Considering the obtained experimental results, some conclusions can be established:

– Clearly there is no reduction in the number of variables of the initial formula due to two-variable equivalences during preprocessing. Also during preprocessing, clause pattern 2B/1T is matched for many clauses and therefore increasing the final formula number of clauses.

– In contrast to the observed statistics during preprocessing, there is a significant number of two-variable equivalences during the search. This number seems to be increased with inferred clauses from 2B/1T reasoning.

– There are no inferred clauses when applying 0B/4T reasoning.

6.8 Conclusions

This chapter evaluated the practical application of the algebraic simplification techniques described in chapter 3, when integrated in GRASP, an intelligent backtrack search SAT algorithm, described in chapter 4.

From the experimental results obtained from different instance problems, the following conclusions can be drawn:

– Usually it is possible to draw similar conclusions for the instances of the same class of problems, establishing a pattern behavior. Furthermore, for the instances of the same class of problems, the number of applied simplifications and the instance size are usually directly proportional.

– Not all the instances gain when algebraic simplifications is applied during preprocessing. For instance, *DIMACS pretolani* and *superscalar processor verification* instances are not affected by algebraic simplifications during preprocessing, while for *bounded model*

Instance	Initial Formula			Final Formula		
	# var	#clause	# lit	# var	#clause	# lit
dlx2_cc_bug01	1515	12808	36242	1514	14206	39038
dlx2_cc_bug02	1515	12808	36242	1514	14206	39038
dlx2_cc_bug03	1515	12808	36242	1514	14197	39020
dlx2_cc_bug04	1515	12808	36242	1514	14158	38942
dlx2_cc_bug05	1515	12808	36242	1514	14206	39038
dlx2_cc_bug06	1515	12808	36242	1514	14206	39038
dlx2_cc_bug07	1515	12808	36242	1514	14192	39010
dlx2_cc_bug08	1515	12808	36242	1514	14131	38888
dlx2_cc_bug09	1512	12804	36236	1511	14203	39034
dlx2_cc_bug10	1512	12804	36236	1511	14203	39034
dlx2_cc_bug11	1513	12800	36222	1512	14228	39078
dlx2_cc_bug12	1513	12800	36222	1512	14228	39078
dlx2_cc_bug13	1513	12800	36222	1512	14225	39072
dlx2_cc_bug14	1516	12811	36249	1515	14202	39031
dlx2_cc_bug15	1690	14634	41470	1689	16188	44578
dlx2_cc_bug16	1516	12811	36249	1515	14202	39031
dlx2_cc_bug17	2388	22101	62649	2387	24278	67003
dlx2_cc_bug18	2047	19868	56392	2046	22039	60734
dlx2_cc_bug19	1516	12823	36279	1515	14314	39261
dlx2_cc_bug20	1513	12701	35925	1512	14106	38735
dlx2_cc_bug21	1506	12763	36111	1505	14247	39079
dlx2_cc_bug22	1516	12848	36354	1515	14271	39200
dlx2_cc_bug23	1513	12830	36306	1512	14227	39100
dlx2_cc_bug24	1516	12848	36354	1515	14271	39200
dlx2_cc_bug25	1518	12834	36314	1517	14261	39168
dlx2_cc_bug26	1518	12836	36320	1517	14263	39174
dlx2_cc_bug27	1408	11244	31756	1408	12541	34350
dlx2_cc_bug28	1600	14036	39756	1599	15493	42670
dlx2_cc_bug29	1515	12794	36200	1514	14209	39030
dlx2_cc_bug30	1516	12719	35973	1515	14255	39045
dlx2_cc_bug31	1515	12808	36242	1514	14232	39090
dlx2_cc_bug32	1514	12807	36239	1513	14198	39021
dlx2_cc_bug33	1516	12798	36210	1515	14201	39016
dlx2_cc_bug34	1516	12718	35970	1515	14173	38880
dlx2_cc_bug35	1520	12815	36259	1519	14258	39145
dlx2_cc_bug36	1519	12814	36256	1518	14224	39076
dlx2_cc_bug37	1516	12812	36252	1515	14206	39040
dlx2_cc_bug38	1515	12783	36167	1514	14221	39043
dlx2_cc_bug39	1482	12112	34220	1481	13625	37246
dlx2_cc_bug40	1516	12812	36252	1515	14206	39040

Table 6.26: SSS instances — formula simplification during preprocessing

Instance	$\#\text{eq}_{pp}$	$\#\text{2B1T}_{pp}$	$\#\text{0B4T}_{pp}$	$\#\text{eq}_{t}$	$\#\text{2B1T}_{t}$	$\#\text{0B4T}_{t}$
dlx2_cc_bug01	1	1400	0	65822	2668	0
dlx2_cc_bug02	1	1400	0	69341	3167	0
dlx2_cc_bug03	1	1391	0	67493	1916	0
dlx2_cc_bug04	1	1352	0	72967	2908	0
dlx2_cc_bug05	1	1400	0	71339	2745	0
dlx2_cc_bug06	1	1400	0	61551	2138	0
dlx2_cc_bug07	1	1386	0	76746	2259	0
dlx2_cc_bug08	1	1325	0	3022	1357	0
dlx2_cc_bug09	1	1401	0	4068	1464	0
dlx2_cc_bug10	1	1401	0	6971	1517	0
dlx2_cc_bug11	1	1430	0	74304	2463	0
dlx2_cc_bug12	1	1430	0	67190	2618	0
dlx2_cc_bug13	1	1427	0	61140	2277	0
dlx2_cc_bug14	1	1393	0	60711	2137	0
dlx2_cc_bug15	1	1556	0	337	1560	0
dlx2_cc_bug16	1	1393	0	61129	2141	0
dlx2_cc_bug17	1	2179	0	795	2199	0
dlx2_cc_bug18	1	2173	0	246	2177	0
dlx2_cc_bug19	1	1493	0	468	1501	0
dlx2_cc_bug20	1	1407	0	246	1411	0
dlx2_cc_bug21	1	1486	0	247	1490	0
dlx2_cc_bug22	1	1425	0	255	1430	0
dlx2_cc_bug23	1	1399	0	834	1424	0
dlx2_cc_bug24	1	1425	0	273	1430	0
dlx2_cc_bug25	1	1429	0	7789	1533	0
dlx2_cc_bug26	1	1429	0	759	1437	0
dlx2_cc_bug27	0	1297	0	259	1301	0
dlx2_cc_bug28	1	1459	0	31277	1908	0
dlx2_cc_bug29	1	1417	0	82896	2354	0
dlx2_cc_bug30	1	1538	0	92694	2509	0
dlx2_cc_bug31	1	1426	0	62935	2057	0
dlx2_cc_bug32	1	1393	0	57685	2177	0
dlx2_cc_bug33	1	1405	0	61845	2037	0
dlx2_cc_bug34	1	1457	0	63353	2085	0
dlx2_cc_bug35	1	1445	0	63009	2076	0
dlx2_cc_bug36	1	1412	0	57507	2195	0
dlx2_cc_bug37	1	1396	0	382	1401	0
dlx2_cc_bug38	1	1440	0	52557	2336	0
dlx2_cc_bug39	1	1515	0	61368	2360	0
dlx2_cc_bug40	1	1396	0	61199	2145	0

Table 6.27: SSS instances — simplification during preprocessing and during the search

Instance	Equivalence			Equivalence & Inference		
	#eq$_{pp}$	#eq$_t$	#eq$_s$/node	#eq$_{pp}$	#eq$_t$	#eq$_s$/node
dlx2_cc_bug01	0	66666	3	1	65822	3
dlx2_cc_bug02	0	56746	2	1	69341	4
dlx2_cc_bug03	0	78582	3	1	67493	4
dlx2_cc_bug04	0	59738	2	1	72967	4
dlx2_cc_bug05	0	9132	0	1	71339	4
dlx2_cc_bug06	0	9135	0	1	61551	3
dlx2_cc_bug07	0	9850	0	1	76746	4
dlx2_cc_bug08	0	11126	0	1	3022	0
dlx2_cc_bug09	0	8092	0	1	4068	0
dlx2_cc_bug10	0	5780	0	1	6971	0
dlx2_cc_bug11	0	77352	3	1	74304	4
dlx2_cc_bug12	0	66537	3	1	67190	4
dlx2_cc_bug13	0	74147	3	1	61140	3
dlx2_cc_bug14	0	461	0	1	60711	3
dlx2_cc_bug15	0	793	0	1	337	0
dlx2_cc_bug16	0	460	0	1	61129	3
dlx2_cc_bug17	0	1209	0	1	795	0
dlx2_cc_bug18	0	626	0	1	246	0
dlx2_cc_bug19	0	76183	3	1	468	0
dlx2_cc_bug20	0	627	0	1	246	0
dlx2_cc_bug21	0	10548	0	1	247	0
dlx2_cc_bug22	0	8651	0	1	255	0
dlx2_cc_bug23	0	8546	0	1	834	0
dlx2_cc_bug24	0	8629	0	1	273	0
dlx2_cc_bug25	0	9707	0	1	7789	0
dlx2_cc_bug26	0	1834	0	1	759	0
dlx2_cc_bug27	0	1030	0	0	259	0
dlx2_cc_bug28	0	771	0	1	31277	1
dlx2_cc_bug29	0	556	0	1	82896	4
dlx2_cc_bug30	0	568	0	1	92694	5
dlx2_cc_bug31	0	573	0	1	62935	3
dlx2_cc_bug32	0	573	0	1	57685	3
dlx2_cc_bug33	0	775	0	1	61845	3
dlx2_cc_bug34	0	77293	3	1	63353	3
dlx2_cc_bug35	0	811	0	1	63009	3
dlx2_cc_bug36	0	810	0	1	57507	3
dlx2_cc_bug37	0	742	0	1	382	0
dlx2_cc_bug38	0	76515	3	1	52557	3
dlx2_cc_bug39	0	112670	5	1	61368	3
dlx2_cc_bug40	0	84994	3	1	61199	3

Table 6.28: SSS instances — contribution of clause inference to two-variable equivalence

Instance	#eq_vars	#2B1T	# 0B4T	Nodes	NCB	CR
dlx2_cc_bug01	65822	2668	0	17403	2670	38879
dlx2_cc_bug02	69341	3167	0	17856	2514	40957
dlx2_cc_bug03	67493	1916	0	18213	2762	41096
dlx2_cc_bug04	72967	2908	0	18204	2941	39356
dlx2_cc_bug05	71339	2745	0	18794	2857	41514
dlx2_cc_bug06	61551	2138	0	18441	2815	40448
dlx2_cc_bug07	76746	2259	0	19574	3181	43542
dlx2_cc_bug08*	3022	1357	0	1259	237	2083
dlx2_cc_bug09*	4068	1464	0	1654	273	2813
dlx2_cc_bug10*	6971	1517	0	2448	400	4478
dlx2_cc_bug11	74304	2463	0	19362	2847	43293
dlx2_cc_bug12	67190	2618	0	18903	2900	42622
dlx2_cc_bug13	61140	2277	0	20162	3328	43867
dlx2_cc_bug14	60711	2137	0	16551	2466	37342
dlx2_cc_bug15*	337	1560	0	206	17	153
dlx2_cc_bug16	61129	2141	0	16641	2486	37525
dlx2_cc_bug17*	795	2199	0	515	81	671
dlx2_cc_bug18*	246	2177	0	99	0	0
dlx2_cc_bug19*	468	1501	0	275	38	359
dlx2_cc_bug20*	246	1411	0	99	0	1
dlx2_cc_bug21*	247	1490	0	97	1	7
dlx2_cc_bug22*	255	1430	0	116	6	22
dlx2_cc_bug23*	834	1424	0	506	78	778
dlx2_cc_bug24*	273	1430	0	115	4	29
dlx2_cc_bug25*	7789	1533	0	2822	544	4948
dlx2_cc_bug26*	759	1437	0	479	67	751
dlx2_cc_bug27*	259	1301	0	106	3	17
dlx2_cc_bug28*	31277	1908	0	7770	971	18293
dlx2_cc_bug29	82896	2354	0	21252	2839	51412
dlx2_cc_bug30	92694	2509	0	22811	3299	51065
dlx2_cc_bug31	62935	2057	0	17801	2575	41899
dlx2_cc_bug32	57685	2177	0	16734	2316	38587
dlx2_cc_bug33	61845	2037	0	17905	2538	41202
dlx2_cc_bug34	63353	2085	0	16881	2659	38221
dlx2_cc_bug35	63009	2076	0	17824	2578	41947
dlx2_cc_bug36	57507	2195	0	16686	2311	38469
dlx2_cc_bug37*	382	1401	0	185	14	141
dlx2_cc_bug38	52557	2336	0	16812	2287	39012
dlx2_cc_bug39	61368	2360	0	20903	2785	48858
dlx2_cc_bug40	61199	2145	0	16662	2489	37576

Table 6.29: SSS instances — final results

checking and *combinatorial equivalence checking* instances large reductions can be achieved in the number of variables and clauses.

– Instances which obtain fewer gains from the described techniques during preprocessing, do not necessarily have the same behavior during the search. Moreover, instances that benefit less from algebraic simplification techniques during preprocessing, are sometimes the ones that have more reductions during the search (e.g.*DIMACS parity*, *data encription standard* and *superscalar processor verification* instances).

– Identification of two-variable equivalence is observed in all instances and matching clause pattern for 2B/1T reasoning is also frequent, whereas matching clause pattern for 0B/4T reasoning is more rare.

On the other hand, when trying to solve a SAT instance, CPU time is certainly an important result to be observed. Moreover, using simplification rules is possible to prune the search space, and therefore we are expected to save time. As a counterpart, there is also a computational effort made to obtain simplifications. Nonetheless, in the end we want to conclude we save more time in search than loose in simplifications.

Table 6.30 shows the number of aborted instances for each application domain, considering the maximum allowed CPU time was set to 500 seconds (on a PIII-733MHz Linux workstation). We recognize the obtained results are not as promising as would be desirable. However, the developed tool is mostly a prototype, which can be distinguished for integrating algebraic simplification techniques in an intelligent backtrack search SAT algorithm. Moreover, the integrated techniques supersede Li's work with EQSATZ [41] and equivalency reasoning. Observe that equivalency reasoning does not integrate 2B/1T reasoning, and from the obtained results we conclude that a significant number of binary clauses can be inferred using this new technique.

Another possible explanation for having so many instances aborted is that we did not use recursive learning for the obtained results. Nevertheless, many of the presented instances have already been solved by GRASP using this technique [54].

On the other hand, adding all dependencies to clauses (and therefore adding a set of literals) whenever we have to replace variables by equivalences, often causes an un-

Table 6.30: Aborted Instances

Application Domain	# Instances	# Aborted Instances
Pretolani (DIMACS)	8	0
16-bit Parity (DIMACS)	10	8
32-bit Parity (DIMACS)	10	10
Combinational Equivalence Checking	25	20
Bounded Model Checking	34	16
Data Encryption Standard	27	19
Superscalar Processor Verification	40	23

controlled growth of the clause database. Alternatives to the implemented approach are discussed in Appendix B and a more detailed comparative study should be considered as future work.

Finally, note that the efficiency of a SAT algorithm is always a main concern, and therefore the implementation details of the developed tool can certainly be reviewed and improved.

7

Conclusions and Further Work

Propositional Satisfiability (SAT) is fundamental in solving many application problems in Artificial Intelligence and also in other fields of Computer Science and Engineering. Recent years have seen the proposal of several effective approaches for solving SAT instances. These approaches include, among others, backtrack search improved with different search pruning techniques and algebraic manipulation. Algebraic manipulation consists in finding equivalences between variables, which can reduce the dimension of the problem and therefore the expected computational effort of solving a target problem instance. In the past, it has already been shown that large simplifications can be obtained using algebraic simplification techniques [41, 50]. On the other hand, intelligent backtrack search algorithms for SAT have empirically been shown to be highly effective in pruning the amount of search, by applying strategies for non-chronological backtracking and procedures for clause recording [6, 58, 87].

In this thesis we integrated algebraic simplification techniques within intelligent backtrack search algorithms. The proposed techniques had already been used in intelligent backtrack search algorithms as a preprocessing engine, but never during the search phase.

We started by studying the existing algebraic simplification techniques, from the most well-known techniques — resolution, the unit clause rule, the pure literal rule, subsumption, two-variable equivalence — to the ones recently proposed in the literature [41, 50], which apply different forms of equivalency reasoning.

Next, we presented an overview of complete search algorithms for propositional satisfiability. Starting from a generic backtrack search SAT algorithm, which captures the organization of several of the most competitive SAT solvers, we discussed variations to the generic algorithm based on different branching heuristics and procedures for identifying necessary assignments. Then, we described intelligent backtrack search algorithms, which represent an improvement to traditional backtrack search SAT algorithms, providing clause recording and non-chronological backtracking. Moreover, we emphasized the description of the GRASP SAT algorithm, which was subsequently used for integrating algebraic simplification techniques.

The integration of algebraic simplification techniques in an intelligent backtrack SAT algorithm poses two additional difficulties: knowing when to replace or undo previous replacements of variables and also considering special cases of clause recording, whenever conflicts involve clauses with variables replaced by others that are dynamically declared equivalent. Furthermore, efficiency of the implemented algorithms should be a main concern, since we want to conclude we save more time by reducing the search than we loose by applying simplifications.

With the obtained experimental results, we have shown the importance of the proposed techniques, demonstrating that they are applied both during preprocessing as well as during the search for a wide variety of real-world problem instances.

Nonetheless, we recognize that the developed tool is mostly a prototype, and a re-implementation of the tool should be considered. Moreover, considering the promising results obtained, several interesting topics can be suggested for future work. Apart from implementation details, which can be decisive for reducing the expected computational effort, and therefore decisive for solving an instance in a reasonable period of time, other aspects should be considered. In concrete, we expect to integrate more simplification techniques, based on some theoretical aspects already mentioned through this dissertation, as well as obtaining more information from the already implemented techniques.

Appendix A — Integrating the Pure Literal Rule in SAT Algorithms

Implementing the Pure Literal Rule

The pure literal rule (PLR) was already implemented in GRASP as an option. However, the algorithm was conceptually very slow. In the deduction engine, *all* variables were traversed to identify the pure literals. We emphasize *all* because is possible to do an equivalent traversal on a subset of the variables, on a given decision level, starting from the variables assigned at that level. Boolean Constraint Propagation (BCP) should be applied before applying the PLR, since pure literals can be identified after variable assignments, but no implied assignments can be identified as a consequence of assigning a pure literal. By assigning a pure literal, all clauses containing it become satisfied and no more unit clauses can appear.

Example A. 1 *Consider the following subset of a clause database and the current assignment:*

Clause database: $w_1 = (x_4 \vee \neg x_5 \vee x_6)$, $w_2 = (\neg x_1 \vee x_2 \vee \neg x_5 \vee x_6)$, $w_3 = (x_4 \vee \neg x_5)$, $w_4 = (x_3 \vee \neg x_4 \vee x_6)$, $w_5 = (x_1 \vee \neg x_3 \vee x_6)$.

Current assignment: $\{(x_2,1),\ (x_5,0)\}$

Traversing all variables, and counting their occurrences in unresolved clauses in both forms of literals, one would detect that x_1 *and* $\neg x_4$ *are pure literals. So, using the pure literal rule, we can assign* $x_2 = 1$ *and* $x_4 = 0$. *Consequently clauses* w_4 *and* w_5 *become* *satisfied.*

Notice that we assume that the clause database can have more clauses, which are not shown because they are not relevant for this example. The irrelevancy arises from those clauses being satisfied or not having the literals that are identified as pure.

Our solution to reduce the overhead associated with the PLR procedure is to use an incremental approach. In other words, traverse only a subset of variables.

During preprocessing all unassigned variables must be analyzed. At this stage the algorithm is the same as the previously implemented in GRASP.

During search, we can use information from the search tree, namely the assigned variables at that decision level. The question is: "Why does a variable become a pure literal from one level to another?"

Due to BCP, variable assignments affect clause states and therefore variable occurrences in unresolved clauses. Consider that at a given decision level we have a variable x in both forms. The idea is that at the next decision level — depending on those level assignments — all clauses containing the variable x in one form can become satisfied. Hence, we may have a new pure literal.

The optimization of the PLR procedure was done in three stages: applying the PLR in an incremental way, avoiding repeated clauses and also repeated variables.

The first optimization consisted of traversing the assigned variables at the current decision level and obtaining a set of unassigned variables from the newly satisfied clauses.

For each assigned variable at the current decision level, get the clauses that are satisfied. If the variable is assigned to TRUE, clauses will be obtained from the positive literals; otherwise from the negative literals. If there are any new pure literals, they will be among the selected variables. The procedure to identify the pure literals is invoked recursively because the identification of a pure literal and the consequent assignment satisfies more clauses, and therefore can originate other pure literals.

Example A. 2 *Consider again the clause database and the current assignment from example 1:*

Clause database:
$w_1 = (x_4 \vee \neg x_5 \vee x_6)$
$w_2 = (\neg x_1 \vee x_2 \vee \neg x_5 \vee x_6)$
$w_3 = (x_4 \vee \neg x_5)$
$w_4 = (x_3 \vee \neg x_4 \vee x_6)$
$w_5 = (x_1 \vee \neg x_3 \vee x_6)$

Current assignment: $\{(x_2,1),\ (x_5,0)\}$

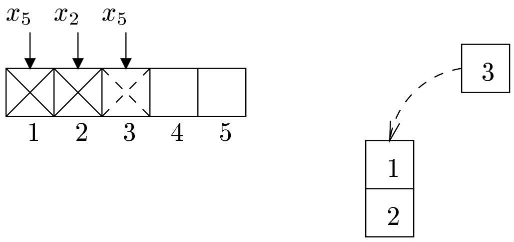


Figure 0-1: Marking clauses when applying the PLR.

Clause database: $w_1 = (x_4 \vee \neg x_5 \vee x_6)$, $w_2 = (\neg x_1 \vee x_2 \vee \neg x_5 \vee x_6)$, $w_3 = (x_4 \vee \neg x_5)$, $w_4 = (x_3 \vee \neg x_4 \vee x_6)$, $w_5 = (x_1 \vee \neg x_3 \vee x_6)$.

Current assignment: $\{(x_2,1),\ (x_5,0)\}$

Suppose that assignments to x_2 *and* x_5 *were made at the current decision level. Due to those assignments, clauses* w_1, w_2 *and* w_3 *became satisfied. From* w_1, *variables* x_4 *and* x_6 *are selected. Similarly, variables* x_1 *and* x_6 *are selected from* w_2, *and* x_4 *from* w_3. *Analyzing the unresolved clauses of variables* x_1, x_4 *and* x_6, *would be possible to find out if one of them is now a pure literal. Contrary to the initial original PLR procedure in GRASP, variable* x_3 *would not be considered as a potential pure literal.*

Comparing the proposed algorithm for applying the pure literal rule with the initial traversal of all the variables, clearly some efficiency has been introduced. Nevertheless, some problems remain. The second step aimed solving part of those efficiency problems.

Getting unassigned variables from newly satisfied clauses can bring redundancies if the clauses have more than one assigned variable at the considered level. By redundancies we mean we obtain unassigned variables from the same clause more than once. The solution is to mark *visited* clauses, avoiding them to be observed more than once.

The algorithm proposed will be frequently used during the implementation of the other simplification techniques. The needed data structures are an array and a stack. Figure 0-1 illustrates the algorithm described in the next paragraphs, using the same clause database and assignment from example 2.

Consider an array with the dimension of the number of clauses in the database. This structure will have to be updated since conflicting clauses require its growth. Whenever you get a clause with literals that are candidate to being pure, you mark the respective position (determined by the clause ID), using a Boolean value.

In addition, the stack accounts for every marked position, no more than once since you can detect if it was previously marked. Once you have the identification of all the interesting clauses, the next step is to get their unassigned variables and identify the pure literals. The marked positions will be cleaned at the end of the process.

In the exemplified case, we start by identifying clauses having potential pure literals from assignment $x_2 = 1$, thus marking the now satisfied clause $w_2 = (\neg x_1 \vee x_2 \vee \neg x_5 \vee x_6)$. Next, we apply the same reasoning to the assignment $x_5 = 0$. Clauses to consider are now w_1, w_2 and w_3. Nevertheless, w_2 has already been marked via assignment $x_2 = 1$ and therefore it is discarded.

The last optimization applied the previous registration algorithm also to variables, besides clauses. Even avoiding getting variables that can be pure from repeated clauses, those variables can be part of more than one clause.

Applying the same strategy, we mark the variables obtained from the marked clauses. Each variable is observed only once.

Example A. 3 *Consider again example 2 and the assignments made at the current decision level:*

Clause database: $w_1 = (x_4 \vee \neg x_5 \vee x_6)$, $w_2 = (\neg x_1 \vee x_2 \vee \neg x_5 \vee x_6)$, $w_3 = (x_4 \vee \neg x_5)$, $w_4 = (x_3 \vee \neg x_4 \vee x_6)$, $w_5 = (x_1 \vee \neg x_3 \vee x_6)$.

Current assignment: $\{(x_2,1),\ (x_5,0)\}$

Let us start by getting possible pure literals from assignment $x_2 = 1$*. From the now satisfied clause* $w_2 = (\neg x_1 \vee x_2 \vee \neg x_5 \vee x_6)$ *we get variables* x_1 *and* x_6*. Marks are associated with clause* w_2 *and variables* x_1 *and* x_6*.*

The next step is to identify possible pure literals from the assignment $x_5 = 0$*. Clauses to consider are now* w_1*,* w_2 *and* w_3*. However,* w_2 *has already been visited via assignment* $x_2 = 1$ *and therefore it is discarded. From* $w_1 = (x_4 \vee \neg x_5 \vee x_6)$ *one obtains variables* x_4

and x_6. Since x_6 was marked before, we only apply the algorithm to x_4. Nothing will be done regarding clause $w_3 = (x_4 \vee \neg x_5)$, since x_4 has already a mark associated.

Experimental Results

Experimental results refer to Superscalar Processor Verification (SSS) instances [83], with GRASP configured to use MOM's as the branching heuristic, to use relevance-based learning(4) and to limit the size of clauses added to the clause database to 50 or fewer literals. CPU times were obtained from a PIII-733MHz Linux workstation.

The statistics evaluated were the number of pure literals (#PLR), the number of decision nodes (#Nodes), the number of non-chronological backtracks (#NCB), the number of recorded conflict clauses (#CR) and the CPU Time (Time). Moreover, we distinguished between CPU Time either applying the pure literal rule — with the original (Time_{plr_orig}) and the new implementation (Time_{plr_new}) of the pure literal rule — or without applying the pure literal rule ($\mathrm{Time}_{w/plr}$).

From the experimental results presented in Table 1, one can draw the following conclusions:

– The new implementation of the pure literal rule is clearly more efficient than the original implementation.

– The integration of the pure literal rule does not imply a gain on CPU Time for solving the proposed problem instances. Observe that for some of them the time duplicates (e.g. *dlx2_cc_bug03* and *dlx2_cc_bug04*), an only one of them (*dlx2_cc_bug15*) benefits from applying the pure literal rule.

– The number of identified pure literals varies, and is premature to establish a relation between this number and other statistic numbers.

Instance	#PLR	#Nodes	#NCB	#CR	Time$_{plr_orig}$	Time$_{plr_new}$	Time$_{w/plr}$
dlx2_cc_bug01	492	1250	177	2387	10.54	9.37	7.76
dlx2_cc_bug02	506	1192	168	2230	9.37	8.38	8.35
dlx2_cc_bug03	728	2354	359	4374	21.15	19.32	8.79
dlx2_cc_bug04	809	2670	395	5281	25.33	23.8	10.73
dlx2_cc_bug05	400	1083	140	2095	8.58	7.78	4.82
dlx2_cc_bug06	360	908	120	1601	6.69	6	4.72
dlx2_cc_bug07	398	978	129	1996	8.1	7.33	4.76
dlx2_cc_bug08	398	980	131	1996	8.03	7.32	4.87
dlx2_cc_bug09	296	674	75	1202	4.61	4.19	3.46
dlx2_cc_bug10	392	801	94	1516	6.11	5.61	5.41
dlx2_cc_bug11	660	1312	155	2914	11.11	10.15	7.51
dlx2_cc_bug12	527	1088	150	2255	8.86	8.17	7.3
dlx2_cc_bug13	494	1391	209	2686	11.47	10.48	7.17
dlx2_cc_bug14	7	114	0	3	0.33	0.37	0.26
dlx2_cc_bug15	448	939	135	1846	8.18	7.73	18.5
dlx2_cc_bug16	7	115	0	3	0.34	0.38	0.26
dlx2_cc_bug17	435	1192	145	2146	13.45	12.55	11.12
dlx2_cc_bug18	66	145	6	37	0.98	1.03	0.69
dlx2_cc_bug19	72	177	10	183	1.04	1	0.72
dlx2_cc_bug20	153	301	53	362	2.12	1.96	0.75
dlx2_cc_bug21	66	188	12	186	1.02	0.99	0.7
dlx2_cc_bug22	83	191	13	189	1.15	1.08	0.76
dlx2_cc_bug23	90	187	12	172	1.09	1.06	0.77
dlx2_cc_bug24	83	191	13	189	1.14	1.09	0.76
dlx2_cc_bug25	392	1068	151	2075	8.96	8.32	5.03
dlx2_cc_bug26	430	1001	113	2133	8.26	7.65	6.31
dlx2_cc_bug27	537	1144	125	2487	8.13	8.21	6.49
dlx2_cc_bug28	721	1228	147	2650	10.72	10.01	6.47
dlx2_cc_bug29	59	169	4	105	0.78	0.8	0.55
dlx2_cc_bug30	76	190	7	185	1.21	1.15	0.84
dlx2_cc_bug31	79	189	8	123	0.94	0.92	0.64
dlx2_cc_bug32	79	188	8	123	0.94	0.93	0.64
dlx2_cc_bug33	73	185	6	120	0.9	0.88	0.63
dlx2_cc_bug34	76	194	8	194	1.25	1.19	0.87
dlx2_cc_bug35	78	189	8	123	0.95	0.93	0.65
dlx2_cc_bug36	78	190	8	123	0.94	0.94	0.64
dlx2_cc_bug37	425	939	109	1888	7.51	6.84	4.84
dlx2_cc_bug38	400	871	112	1659	6.52	5.96	4.92
dlx2_cc_bug39	390	968	126	1867	7.51	7.12	4.11
dlx2_cc_bug40	467	1162	142	2221	9.92	9.06	8.18

Table 1: Statistics for SSS instances applying the Pure Literal Rule

Appendix B — Clause Recording in Algebraic Simplification

One of the major challenges when integrating algebraic simplification in SAT solvers has been to guarantee the correctness of the creation of conflict clauses, involving clauses with literals replaced by identifying two-variable equivalence. Moreover, we followed different approaches during the implementation phase. Despite the fact that only one of those approaches has been described within the algorithm for identifying two-variable equivalence, we believe there is room for discussing the remaining possibilities.

Let us start by enumerating three different approaches for clause recording in algebraic simplification. All them have in common a careful account of clauses with replaced variables by equivalences. To such clause, one can do one of the following actions:

1. Add all dependencies to the clause, which implies adding a set of literals.
2. Add all dependencies to the clause in the form of an additional variable, which implies creating a clause with one new variable, that captures the dependency.
3. Consider the clause as a *tainted clause*, meaning that identifying equivalences implies replacing a literal in a clause, and henceforth the clause becomes tainted.

The first approach was the one which was implemented. Whenever we identify two equivalent variables x and y, due to the existence of binary clauses ω_i and ω_j, we select literals assigned value 0 from ω_i and ω_j. These literals will be added to all clauses where y is replaced by x, assuming variable y is the one that is eliminated. However, in some cases we pay a high price with the increasing of the number of literals in the clause database, depending on the number of literals to be added.

For these reasons, in our implementation is possible to specify an upper limit (`n`) on the number of literals to be added by each equivalence. Consequently, two-variable equivalence is only performed if the number of literals to be added is below a user-specified maximum allowed threshold size `n`.

Another direction would be the creation of special variables $v*$. Each variable $v*$ is associated with a set of literals obtained from clauses which permitted to identify an equivalence ($x \equiv y$). More precisely, each $v*$ represents all dependencies, which means literals are not implicitly added to clauses where variable y is replaced by x, but rather $v*$. Additionally, is created a new clause with $v*$ and all dependencies. Obviously, a dynamic creation of variables should be contemplated, therefore affecting other implementation details.

Finally, we are going to describe what we mean by considering clauses with variables replaced by equivalent variables as *tainted clauses*. Each tainted clause is associated with a decision level d_t, determined by the decision level where a two-variable equivalence was identified. If we create a conflict clause from an unsatisfied tainted clause, this clause will be deleted when the search backtracks to decision level d_t.

From the above description, one can envision using tainted clauses in another perspective. The idea is to merge the identification of tainted clauses with the action of adding all dependencies to clauses. As already mentioned, adding all dependencies to clauses causes a significant increasing of the number of literals in the clause database. On the other hand, only conflict clauses remain with all dependencies, even when the search backtracks. Hence, we suggest that tainted clauses with a size less than a threshold t are kept during the subsequent search, whereas larger clauses are discarded as soon as they become unresolved. Remember this technique is known as *k-bounded learning*, being applied to all conflict clauses, and was explained earlier in section 4.2.1. However, we are suggesting using the same algorithm and defining another value t, besides k. Afterwards, t value is expected to be lower than k, since conflict clauses being tainted are expected to be larger than other conflict clauses.

Bibliography

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

[2] L. Sais B. Mazure and E. Gregoire. Tabu search for SAT. In *Proceedings of the National Conference on Artificial Intelligence*, pages 281–285, 1997.

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

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

[5] Henry A. Kautz B. Selman and B. Cohen. Noise strategies for improving local search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 337–343, 1994.

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

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Proceedings of Tools and Algorithms for the*

Construction and Analysis of Systems, LNCS 1579. Springer-Verlag, March 1999. (BMC instances available from URL http://www.cs.cmu.edu/∼modelcheck.

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

[9] S. A. Cook. The complexity of theorem proving procedures. In *3rd ACM Simp. on Theory of Computing*, pages 151–158, Ohio, 1971.

[10] S.A. Cook and D.G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[12] Nuno Crato. O algoritmo do jantar de anos, October 7 2000. Jornal Expresso — Revista.

[13] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.

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

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

[16] J. de Kleer. Exploiting locality in a TMS. In *Proceedings of the National Conference on Artificial Intelligence*, pages 264–271, 1990.

[17] J. de Kleer. An improved incremental algorithm for computing prime implicants. In *Proceedings of the National Conference on Artificial Intelligence*, pages 780–785, 1992.

[18] J. P. A. Deutsch. A short cut for certain combinatorial problems. In *In British Joint Comput. Conference*, 1966.

[19] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[20] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.

[21] B. Dunham, R. Fridshal, and G. L. Sward. A non-heuristic program for proving elementary logical theorems. In *Proceedings of the International Conference on Information Processing*, pages 282–285, 1959.

[22] J. D. Findler. *Associative Networks: Representation and use of knowledge by computers*. Academic Press, New York, 1979.

[23] F. Frayman and S. Mittal. *Cossack: A constraint-based expert system for configuration tasks*. Computational Mechanics Publications, Nadel, 1987.

[24] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, May 1995.

[25] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[26] Ian Gent and Toby Wash. The search for satisfaction. Draft Paper, 1999. Available from http://dream.dai.ed.ac.uk/group/tw/sat/sat-survey2.ps.

[27] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures — the case study of modal k. In *Proceedings of the International Conference on Automated Deduction*. Springer-Verlag, 1996.

[28] F. Giunchiglia and R. Sebastiani. A new method for testing decision procedures in modal logics. In *Proceedings of the International Conference on Automated Deduction*. Springer-Verlag, 1997.

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

[30] Groote and Warners. The propositional formula checker HeerHugo. *Journal of Automated Reasoning*, 24(1-2):101–125, 2000.

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

[32] Hoos and Stutzle. SATLIB: An online resource for research on SAT. In H. van Maaren I.P.Gent and T. Walsh, editors, *SAT 2000: Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications series.* IOS Press, Amsterdam, The Netherlands, 2000.

[33] I.P.Gent and T. Walsh. Satisfiability in the year 2000. *Journal of Automated Reasoning*, 24(1-2):1–3, 2000.

[34] J. Franco J. Gu, P. W. Purdom and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–152. American Mathematical Society, 1997.

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

[36] D. S. Johnson and M. A. Trick, editors. *Second DIMACS Implementation Challenge.* American Mathematical Society, 1993. (DIMACS SAT instances available from URL ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/benchmarks/cnf).

[37] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence*, pages 359–363, 1992.

[38] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.

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

[40] G. A. Robinson L. Wos and D. Carson. Efficiency and completeness of the set of support strategy. *Journal of the Association for Computing Machinery*, 12(4):536–541, October 1965.

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

[42] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 366–371, 1997.

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

[44] R. J. Lipton. Usimg DNA to solve NP-complete problems. *Science*, 268:542–545, 1995.

[45] G. Stållmark. A proof theoretic concept of tautolocal hardness, 1994. Incomplete manuscript.

[46] D. W. Loveland. Automated theorem proving: A logical basis. North Holland, Amsterdam, New York, 1978.

[47] J. P. Marques-Silva. An Overview of Backtrack Search Satisfiability Algorithms. In *International Symposium on Artificial Intelligence and Mathematics*, January 1998.

[48] J. P. Marques-Silva. Improving satisfiability algorithms by using recursive learning. In *Proceedings of the International Workshop on Boolean Problems*, pages 47–58, September 1998.

[49] J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In P. Barahona and J. Alferes, editors, *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, LNAI 1695, pages 62–74. Springer-Verlag, September 1999.

[50] J. P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. Technical Report RT/01/2000, INESC, March 2000.

[51] J. P. Marques-Silva. Boolean satisfiability algorithms and applications in electronic design. Presented at the Conference on Computer-Aided Verification (CAV), July 2000. Tutorial.

[52] J. P. Marques-Silva. On applying algebraic simplification in boolean satisfiability. Presented at the SAT 2000 Workshop, May 2000.

[53] J. P. Marques-Silva. On selecting problem instances for evaluating satisfiability algorithms. In *ECAI Workshop on Empirical Methods in Artificial Intelligence*, August 2000.

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

[55] J. P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, pages 467–469, November 1996.

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

[57] J. P. Marques-Silva and K.A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, pages 152–161, June 1997.

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

[59] Kim Marriott and Peter J. Stuckey. *Programming With Constraints : An Introduction*. MIT Press, January 1999.

[60] Fabio Massacci. Using Walk-SAT and Rel-SAT for cryptographic key search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 290–295, 1999. (DES instances available from URL http://www.uni-koblenz.de/-∼massacci.

[61] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, 2000.

[62] D. Navinchandra. *Exploration and innovation in design.* Springer Verlag, Sadeh, 1990.

[63] D. Pretolani. Efficiency and stability of hypergraph sat algorithms. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.

[64] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993.

[65] Paul Walton Purdom, Jr. Solving satisfiability with less searching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–6(4):510–513, July 1984.

[66] Paul Walton Purdom, Jr. and Cynthia A. Brown. The pure literal rule and polynomial average time. *SIAM Journal on Computing*, 14(4):943–953, 1985.

[67] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 42(2):125–155, 1989.

[68] Irina Rish and Rina Dechter. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning*, 24(1-2):225–275, 2000.

[69] Craig K. Rushforth and Wei Wang. Local search for channel assignment in cellular mobile networks. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 689–709. American Mathematical Society, 1997.

[70] S. J. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[71] V. Agrawal S. Chakradhar and M. Bushnell. Neural net and boolean satisfiability model of logic circuits. *IEEE Design & Test of Computers*, pages 54–57, October 1990.

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

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

[74] Y. Shang and B. W. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–99, January 1998.

[75] W. M. Spears. A nn algorithm for boolean satisfiability problems. In *Proceedings of the International Conference on Neural Networks*, pages 1121–1126, 1996.

[76] W. M. Spears. Simulated annealing for hard satisfiability problems. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 533–558. American Mathematical Society, 1996.

[77] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, 1989. Swedish Patent 467 076 (Approved 1992), US Patent 5 276 897 (approved 1994), European Patent 0 403 454 (approved 1995).

[78] Robert Tarjan. Depth-first search and linear graph algoritms. *SIAM J. Comput.*, 1(2):146–160, June 1972.

[79] Robert Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, March 1974.

[80] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, 1982.

[81] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, January 87.

[82] Hans van Maaren. Highlights of satisfiability research in the year 2000. In H. van Maaren I.P.Gent and T. Walsh, editors, *SAT 2000: Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications series.* IOS Press, Amsterdam, The Netherlands, 2000.

[83] M. N. Velev and R. E. Bryant. 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, pages 37–53, September 1999.

[84] Jinchang Wang. Algorithms for the satisfiability (SAT) problem: A survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 351–364. American Mathematical Society, 1997.

[85] Joost P. Warners and Hans van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. In *90*, page 10. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, April 30 1998. SEN-R9802.

[86] Ramin Zabih and David McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, 1988.

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

[88] Hantao Zhang and Mark Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1-2):277–296, 2000.

[89] J. Zhang and H. Zang. SEM: a system for enumerating models. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 298–303, 1995.