Tel-Aviv University
Raymond and Beverly Sackler
Faculty of Exact Sciences
School of Computer Sciences

Intersection-based methods for boosting satisfiability testing using boolean rings

Thesis submitted as partial fulfillment of the requirements towards the M.Sc. degree

by

Daher Kaiss

The research work has been conducted under the supervision of
Prof. Nachum Dershowitz

JANUARY 2005

To my mother, who was illiterate...

Acknowledgements

I'm grateful to my supervisor prof. Nachum Dershowitz for the guidance, advice and for many fruitful discussions.
Special thanks to my master and teacher Ziyad Hanna, who made me love the domain of formal verification.
Many thanks to prof. Moshe Vardi, and Alex Nadel for their remarks.

Last but not least, to my wife for her support and lots of patience.

Abstract

We present a method for testing satisfiability of boolean formulas using intersection-based methods on boolean rings. Our method is different from standard DPLL methods is the fact that it performs efficient learning on the formulas before splitting, whereas other methods perform the learning generally after deep splitting. Our method takes the advantage of algebraic properties, which exist in boolean rings, in order to perform efficiently operations like unit propagation and intersection between two sets of formulas, while avoiding potential size increase associated with the use of the distributive law. We present the bin-lin approach, and show how we can use efficiently Gauss elimination on the linear part, and Horn algorithms in the binomial part, in order to increase the learning.

Contents

Acknowledgements **iii**

Abstract **iv**

1 Introduction **1**

1.1 Related work 2

1.2 Our results 3

1.3 The organization of the thesis 3

2 Overview of CSP **4**

2.1 Basic concepts in solving CSP 4

2.1.1 Motivation 4

2.1.2 Classification of CSP 6

2.1.3 Formulation 8

2.2 A survey of algorithmic aspects of CSP 11

2.2.1 The generate-and-test paradigm 11

2.2.2 Davis-Putnam procedure 13

2.2.3 Ordered binary decision diagrams 14

2.2.4 Integer linear programming 19

2.2.5 The boolean Ring method 20

2.3 Difficulties with CSP 21

2.3.1 Combinatorial explosion 21

2.3.2 NP-completeness 21

2.4 Heuristics 22

2.4.1 Complete methods 23

3 Boolean Rings **26**

3.1 Introduction 26

3.2 The formalism of boolean rings 27

4 Stålmarck's algorithm **32**

5 Horn Clauses **38**

5.1 Preliminaries 38

5.2 The class of propositional Horn 39

5.3 The Graph associated with a Horn propositions and pebblings 40

6 Stålmarck's algorithm on boolean rings **43**

6.1 Translating the SAT instance to boolean ring representation 43

6.2 A generalized algorithm for Satisfiability checking 45

6.3 Unit refutation (aka Simple Rules) 46

6.3.1 Unit refutation on the Linear equations 48

6.3.2 Unit refutation on the Binomial equations 48

6.4 The intersection operation 49

6.5 The Sat solver main steps 49

6.6 Examples 50

7 Numerical Experiments **53**

7.1 How the tests were carried out 53

7.2 Results 54

7.2.1 Explanation to the table contents 55

7.2.2 Analysis of the results 56

8 Conclusions and future work **57**

8.1 Conclusions 57

8.2 Future work 57

Bibliography **58**

Appendix 63

A Running BRSat 63

B The `.br` format 65

C Overview of the Source Code 66

List of Figures

2.1 Forward-checking and backtracking 12
2.2 Recursive learning 18

4.1 Splitting on one variable 34
4.2 Splitting on more variables 35
4.3 `Stalmarck-SAT` 36

6.1 `BRSat` 47

List of Tables

7.1 Representative runs with and without merging 55

Chapter 1

Introduction

Given a Boolean formula A, the *satisfiability problem* is concerned with finding an assignment of truth values (0 and 1) to the propositional variables in A which makes the formula equal to the constant 1 (true) or proving that the formula is equal to the constant 0 (false). The satisfiability decision problem arises frequently as a sub-problem in many applications, such as automated verification and automated theorem proving, Artificial Intelligence (AI), Electronic Design Automation (EDA), and many other fields of Computer Science and Engineering . Although Cook [8] showed the problem of testing satisfiability to be NP complete, however it is still open the possibility that some algorithms are acceptably efficient in a large number of important cases. Many standard search procedures yield a decision procedure for propositional logic. e.g. the Davis-Putnam procedure, resolution, model elimination and others. Beside the search procedures, graph based techniques like Binary Decision Diagrams (BDDs), were proven to give very promising results.

Whatever the above techniques strengths and weaknesses, all known methods take time exponential in the size of the input formula, in the worst case. Binary decision diagrams have been most successful in hardware verification, though other applications have been explored - Bryant [10] gives a survey. Stålmarck's algorithm has been applied to hardware verification, in a large number of real industrial situations, e.g. in railway interlocking systems, which generally require checking of tautology involving something like 10^5 variables. This is too much for traditional methods, including BDDs. Regardless of which of the above techniques performs

better, most of them put more focus on the algorithmic side and less on the representation of the formula. Most search procedures assume a conjunctive normal form (CNF) representation, while Stålmarck's algorithm assumes a triples representation.

In this thesis, we investigate a boolean ring representation of the formula. By exploiting algebraic properties of this representations, we show that Stålmarck's algorithm can perform more efficiently. Boolean ring is an algebraic structure which is equivalent to boolean algebra. The major representational differences are that boolean rings use *exclusive-or* $(+)$ instead of $(\vee)$ to represent boolean functions, and that there is no need for negation in boolean ring. We present also several fundamental simplifications that can be employed efficiently on boolean rings, and later on we show how such simplifications can be used in the Stålmarck's algorithm in order to boost it.

1.1 Related work

The isomorphic relationship between boolean algebras and boolean rings was found in 1963 by Stone [1] and probably date back to 1927 by Zhegalkin [3]. In 1983, Hsiang and Dershowitz [21, 22], presented a way to derive a boolean ring normal form from a given boolean function using a canonical set of rewrite rules. In 1997, Hsiang and Huang [24] presented some fundamental properties concerning boolean rings, and presented a simple method for deriving the boolean ring normal form directly from the truth table. They described a notion of normal form of boolean function with a don't care condition. Huang [2] showed that with an implementation of a naïve boolean ring based Davis-Putnam method, the number of splittings were reduced by 30%. With this saving, however, came the price of a time-consuming simplification process, to avoid potential size increase associated with the use of the distributive law.

1.2 Our results

Our experiments show that the combination between Gauss elimination and Horn method as simplifies in the bin-lin system can improve the stålmarck saturation pre-process step by finding more relations between variables. This kind of learning is proven to be very useful in reducing the number of splits in our DPL solver. Our experiments show that for more than 700 tests that we had, the above methods are showing approximately 13% improvement in reducing the number of the splits of the DPL solver. Notice also that for some of the tests, the saturation stage was enough to conclude the satisfiability checking result, thus eliminating the need for DPL splits at all.

1.3 The organization of the thesis

The sequel of this thesis is organized as follows: In section 2 we give an overview of the most known satisfiability testing methods. In section 3 we review boolean rings, their properties and the Bin-Lin representation. In section 4, we review the Stålmarck's algorithm and its main characteristics. Horn formulas are discussed in section 5. In section 6, we describe how boolean rings can be employed in the Stålmarck's algorithm. Numerical experiments as presented in section 7. We conclude in the last chapter and present future work.

Chapter 2

Overview of CSP

2.1 Basic concepts in solving CSP

2.1.1 Motivation

We begin this section by introducing a well-known constraint-satisfaction problem (CSP), the 8-queen problem. Suppose that there is an 8 by 8 chessboard, and there are eight queens. The goal of this game is to put these queens on the chessboard such that no queen can attack the other either vertically, horizontally, or diagonally. Although it is not difficult to find a solution that satisfies this requirement, we use it here to illustrate the basic ingredients in a constraint-satisfaction problem.

A CSP can be divided into two parts. The first part is the *problem domain*; it includes *domain variables* and the values that can be assigned to those variables. For the 8-queen problem, a natural way is to consider each queen as an individual variable and positions on chessboard as the domain of the variables. The second part in a CSP is the *constrains*, which specify the relationship between assignments on variables. In this problem, the goal is to place the queens so that the queens cannot attack one another. This, however, is only part of the constraints. There is additional constraint that no two queens can be placed on the same position.

The definition of a constraint-satisfaction problem (CSP) can be formalized as follows. There is a set of *variables*, say V, and for each variable x in V, there is a set D_x, which is the *domain* of x and defines the possible outcome of x. In many

cases, the domains for all variables are the same, and we can simply use notation D to refer to it and ignore the subscript x. There is also a set of *constraints* C, which specifies the relationship between variables. The formal definition of constraints will be given in Section 2.1.2. There a CSP is triple (V, D, C), which gives the set of variables, the domain of the variables and the constraints. The goal of a CSP is to find an assignment of variables such that all of the constraints are satisfied. Such an assignment is called a *feasible solution* (or for short, a *solution*).

Let us look at this definition more closely because the scope of it is very large. For example, solving the system of linear equations that we had learned in junior high school satisfies the definition: There is a set of variables, each variable is taken over the real numbers, and linear equations are the given constraints. We often use *Gaussian Elimination* or equation replacement to solve it. In this case, it finds a feasible solution by *solving constraints*. Linear equations are simple, since constraints are structured enough to be effectively manipulated and the process of gathering a feasible solution is straightforward. However, there are many relaxations that are still in the scope of a CSP but become much harder. For example, if we relax the linear equations to be multivariate polynomial equations, then finding a feasible solution over the real numbers (or complex numbers) has a much higher complexity. Buchberger's Gröbner bases construction [4, 5] is an algorithm that can be used to simplify a set of multivariate polynomial equations into univariate, so the problem becomes how to solve high order polynomial equation. The question that can raise then is what would happen if we want to solve a multivariate polynomial equation over integers? At first glance one would think that the problems with integer numbers are simpler than those with real or complex numbers, but this is not true. The problem to solve a multivariate polynomial equation over integers, called the Hilbert's tenth problem, is proven to be *undecidable* [16]. This means that there exists no algorithm that can report the answer if it exists and tell no if there is none.

Another approach to solving CSP is by *enumeration*. The idea in this approach is to enumerate all possible solutions and then to check whether any is feasible. There are two disadvantages with solving CSP's by enumeration. First, the problem

domain may be infinite, so exhaustive enumeration is impossible. Second, even if the problem domain is finite and there is a way to do enumeration, the space that needs to be explored may be too large to be enumerated in practice. However, in many real applications when the domain is finite, enumeration shows much better performance that constraint solving, especially when the feasible solutions scatter over the search space.

There is no clear answer to the question of which approach is better. If there is a well-developed mechanism to process the formulation of constraints, such as given a system of linear equations, then solving constraints should be better. However, in many hard applications, either the formulation of constraints is tool complicated to be efficiently manipulated or the overall complexity of applying it is too high. Enumeration seems to be the only alternative.

So far, we have defined what is CSP and shown the two general approaches to solve it. In the next section, we discuss the classification of CSP's according to the requirements on the goals and formulation of constraints.

2.1.2 Classification of CSP

We discuss the classification from two perspectives: goal and problem domain.

Perspective from goal. Even given the same problem specification, different requirements on the goal may lead to different levels of difficulty. In the 8-queen problem introduced in Section 2.1.1, it needs to find only *one* feasible solution. For problems of this nature, if the feasible solutions occupy a considerable proportion of all possible solutions or if there is a known deterministic procedure to the construction, then it is usually not difficult to obtain *one* satisfactory solution. However, if the 8-queen problem is made to ask for the *total number of feasible solutions*, then it becomes much harder, and it is not known how to find this number without exhaustive search. We discuss this issue in three levels: *the decision level, the construction level, and the optimization level.*

At the **decision level**, what we consider important is *whether there exists a feasible solution*; finding a solution is not the main concern, only its *existence* is.

For example, the answer for the 8-queen problem at the decision level is "yes". In fact, it can be shown that for any $n \geq 4$, the n-queen problem at the decision level is affirmative. However, not all decision problems are so simple. We shall discuss this later.

At the **construction level**, the requirement is to report a feasible solution if there is one, or to answer "no" if there is none. The 8-queen problem is such an instance. This level seems much important than the decision level, since it *actually* constructs an item that satisfies the required conditions. However, in many cases when the problem domain is finite, *constructing a solution is equivalent to deciding the existence*, where the equivalence is defined to be within polynomial time loss of efficiency. This fact is true as long as the formulation of a problem is general enough that *it is closed under divide-and-conquer*, a methodology used to break a large problem into smaller pieces. The linkage is as follows. We can choose a variable and break the problem into smaller ones by instantiate all possible values to this variable. If the decision procedure answers "yes" to one of the branches, then go to this branch and repeat this process again. Finally, a full solution will be constructed, and that is indeed what we need. But this strategy cannot be applied to the n-queen problem, since the problem is not an (n-1)-queen problem after we have placed one queen on the chessboard (because it is not closed). We should remark that at this level, if a solution is constructed, checking its correctness is usually easy (doable in polynomial time).

The third level is the **optimization level**. It associates a weight to each solution and asks for a solution of the minimum weight (or reversely, the maximum weight). The weight can come from different measures such as the number of constraints that have been satisfied or the sum of weights of the individual variables. This should be the hardest level, since for most cases it is difficult even to verify the answer. In discrete cases, it can actually be transformed into a decision problem. We can simply add a goal value, ask whether there is a solution with smaller weight (in the minimization version), then use bisection method to successively approximate the optimum. In some situation, only a *near optimal* solution is needed, in which case approximation algorithms can be used to find a solution within a certain bound.

Perspective from problem domain. The domain of the CSP can be discrete or continuous. In the 8-queen problem, the domain is the positions on a chessboard. Since there are 64 possibilities for each queen, it is discrete and finite. For solving a system of linear equations, the domain is usually the real numbers. In this case, it is continuous and infinite. Different kinds of domains lead to different strategies to process them. For example, in the 8-queen problem, exhaustive search is possible since there are only finitely many possibilities in the enumeration. Such a possibility does not exist for linear equations over real numbers.

2.1.3 Formulation

In this thesis we focus on finite domain problems, where enumeration is often more useful than solving constraints. We start this section by introducing general definitions and then refine them in several directions.

General definitions. Recall that a *constraint solving problem* consists of a set of variables, a set of domains over which the variables range, and a set of constraints that restrict the values that the variables can be assigned. We also consider *optimization problems* in which a cost function is used to evaluate different assignments. For simplicity we assume that the domains of all variables are the same. We call this definition *uniformity* and it can easily be extended to diverse domains. In the rest of the thesis, we use V to denote a finite set of variables, D to denote the (finite) domain of the variables in V and D^V to denote the set of all functions from V to D.

Definition 2.1.1 (CSP without cost function) A constraint solving problem (CSP) *is a triple* $(V, D, \mathcal{C})$ *where* V *and* D *are as described above, and* $\mathcal{C} = \{C_1, \ldots, C_k\}$ *is a set of subsets of* D^V*, called* `constraints`*. The goal of a CSP is to find an assignment* $I : V \to D$ *such that* $I \in C$ *for all* $C \in \mathcal{C}$*. Assignment* I *is called a* `feasible solution`*.*

Definition 2.1.2 (CSP with cost function) A CSP with cost function *is the same as above, plus a function* ρ *from* D^V *to* A*, where* $(A, <)$ *is a partially ordered set. The goal of a CSP with cost function is to find a feasible solution* $I : V \to$

D such that $\rho(I)$ is minimal for all feasible solutions. Assignment I is called an `optimal solution`*. When the ordering $(A, <)$ is linear, we can speak of the minimum solution, and we call this an* `optimum solution`*.*

The most common type of CSP has propositional variables. In other words, the variables have the truth values $\{0, 1\}$ as the domain D. It is also convenient to allow the variables to be undefined (and thus allow *partial assignments*). In this case we consider the domain $D^* = \{0, 1, *\}$, where $*$ means "undefined".

Definition 2.1.3 (Boolean constraint solving problem) A Boolean Constraint Solving Problem (BCSP) *is a CSP with cost function $(V, D^*, \mathcal{C}, \rho)$ in which $\mathcal{C}$ is a set of clauses and D^* is $\{0, 1, *\}$ where the elements stand for false, truth and undefined, respectively.*

The cost function is usually defined as a weighted sum of feasible solution. That is, a non-negative weight is attached to each possible outcome of a variable and the cost of a feasible solution is the sum of the costs of individual variables.

Definition 2.1.4 (The Satisfiability problem) Given a boolean formula A, the *satisfiability problem* is the task of determining whether there exists an assignment of truth values (0 and 1) to the propositional variables in A which makes the formula equal to the constant 1.

For verification, boolean formulæ are normally derived from two sources:

- boolean functions $z_i = f_i(x_1, \ldots, x_n)$ representing a circuit or program (registers, for example, can be expressed as a sequence of boolean bit variables).
- Specifications that the functions z_i and variables x_j should satisfy.

The validity and equivalence problems are special cases of satisfiability: validity of a formula A is the same as unsatisfiability of its negation A'; equivalence of formulæ A and B is the same as validity of the equality $A = B$ (biconditional $A \leftrightarrow B$). As is well-known, the satisfiability problem for formulæ given in conjunctive-normal-from (CNF) is NP-complete [8].

Logical formulation. Constraints in a CSP can be represented by a set of equations, inequalities, or logical formulas. We first show how to represent BCSP (Definition 2.1.3) by propositional logic, and then to the general case. A *literal* in propositional logic is a variable or the negation of the variable, and a *clause* is the disjunction of literals, i.e. it links literals only by $\vee$. A *sentence* is a conjunction of clauses, and such an expression is said to be in *conjunctive normal form* (CNF). The goal of this kind of CSP is to find a assignment that makes a sentence *true*. In this case, we often say that a sentence is *satisfied* by an assignment.

Constraints can also be expressed by more general expressions, the *boolean expressions*. It is defined inductively as (1) all variables are boolean expressions, (2) if f and g are boolean expressions, then$(\neg f)$, $(f \vee g)$, $(f \wedge g)$ are also boolean expressions. Any boolean expression can be translated into an equivalent expression in CNF (where the equivalence is defined on the satisfiability by assignments); however, the expressive power when consider the length of an expression as a resource is very different. For example, given the same number of propositional variables, say n, the parity function, which returns 1 if and only if there are odd numbers of inputs which are 1's, can be expressed as a boolean expression in length n^2, but it requires exponential length in CNF. Nevertheless, CNF is widely used instead for the general boolean expressions for three reasons. First, from a practical point of view, a CSP is usually specified by a set of *simple* constraints, which means that each individual constraint can be expressed by a *short* boolean formula and constraints are joined by the logical-and. This structure is very suitable for CNF because the logical-and that joins constraints can be easily replaced by $\wedge$, so the final CNF is simply to *and* the CNF's for each constraint. Second, it is a well-known fact that *any* boolean formula can be expanded into a CNF within a constant multiple factor expansion if extra variables are allowed. That is, satisfiability is kept under the transformation, but the length of it will not be too long. Third, CNF has many elegant properties. For example, it is closed under *resolution* and the expression is very simple and easily implemented in a computer program.

Of course, CNF and boolean expressions are not the only possibilities to express constraints (in fact, CNF is a special class of boolean expressions). Other ways to

express the binary domain, such as OBDD (Section 2.2.3), integer linear inequalities (Section 2.2.4), and boolean rings (Section 2.2.5), are also discussed.

When the domain of the CSP is finite but not binary, we can use an *encoding method* to transform it into binary. For example, in the 8-queen problem we can let variable $x_{i,j}$ signify "putting queen i at position j in row i". Since there are only eight queens, i ranges from 1 to 8. There are eight choices for each queen, so j also ranges from 1 to 8. There are totally 64 propositional variables and a constraint such as "each row must have one queen" can be expressed by

$x_{i,1} \vee x_{i,2} \vee x_{i,3} \vee x_{i,4} \vee x_{i,5} \vee x_{i,6} \vee x_{i,7} \vee x_{i,8}$, for $1 \leq i \leq 8$.

A constraint like "queen i cannot be placed at two places" can be expressed by

$\neg x_{i,j} \vee \neg x_{i,k}$, for $j < k$.

Other constraints can be expressed similarly.

2.2 A survey of algorithmic aspects of CSP

2.2.1 The generate-and-test paradigm

Recall that a CSP contains variables, a domain, and constraints. One way to solve a CSP is to generate all possible combinations of assignments to variables and then to check whether there exists a feasible solution. This approach is very brute-force and inapplicable. For example, in the 8-queen problem, if we place each queen freely on the chessboard, then each queen has 64 choices and totally there are 64^8, which equals to 2^{48}, ways to be generated and then verified. A better way to do this is by observing that queens are symmetric with each other and there is no difference if we interchanges the positions of two queens. Furthermore, if queen 1 has been put in row 1, then queen 2 cannot be put in the same row. Hence one has to put each queen in a different row. Then the domain of variable reduces from 64 to 8 and then overall combination of assignments is 8^8, which equals to 2^{24}, and the search space shrinks from 2^{48} to 2^{24}.

Forward-checking. One way to improve the performance is to formulate the enumeration as a tree search. Initially, we chose a variable as the root of a tree

```
Procedure Tree_search(M)
If no constraint is falsified by M then
begin
      If all variables have been assigned values, then
            Output M, which means a model is found.
      Otherwise, let x be a variable which has not been assigned;
            For each possible value v of x do
                  Tree_search (M ∪ {x ← v}).
end;
```

Figure 2.1: Forward-checking and backtracking

and split the root into branches according to the substitution of possible values to that variable. This process can be repeated until a *search tree* is obtained. But generating the tree alone does not save time since all of the instantiations happen at the leaf nodes. To reduce complexity, during the enumeration, a node should not spawn children if information about the node warrants that no solution for the constraints can be produced below that node. This scheme is usually realized by checking if there is a constraint that has been falsified. It relies on a fact that a constraint in a CSP is usually specified by some of the variables, so incomplete assignment can still determine the satisfiability of some constraints. This strategy is called *forward-checking*, and a pseudo code for it is shown in Figure 2.1; the initial call for the program is usually the empty set.

As in the 8-queen problem, suppose that the first queen is placed at position (1,1). Then forward-checking prohibits placing the other queens at (2,2), (1, i) and (i,1), for all $1 \leq i \leq 8$. This definitely reduces the required search space. Note that although forward-checking is only a sufficient condition for pruning a search tree, it can save time significantly.

Tree traversal. We did not specify how to traverse a search tree in our previous description. Actually, we use the *depth-first search* in the program in Figure 2.1. The most common methods are depth-first search (DFS) and *breadth-first search* (BFS).

Intuitively speaking, DFS traverses a tree in vertical way while BFS traverses it horizontally. The information required by both strategies are quite different: the data structure for DFS is a *stack*; the one for BFS is a *queue*. For practical reasons, DFS is used more often than BFS, especially in solving CSP's. First, the memory required by DFS is proportional to the number of variables, while in BFS, it is exponential. Second, for solving CSP, a feasible solution always occurs at the leaf nodes. Hence, DFS can examine complete assignments much earlier than BFS, which implied that a feasible solution has a better chance to be found by DFS than by BFS in the same time. We do not mean that DFS is intrinsically superior to BFS; the overwhelming use of DFS in solving CSP is simply because it is more suitable.

Divide-and-conquer. The tree-version of the generate-and-test is an application of the methodology *divide-and-conquer*. Divide-and-conquer breaks a large problem into small pieces, solves them, and combines the results together. The smaller problems are often simpler than the original one. If they are still hard to be solved, this methodology can be applied repeatedly. In the tree-version of generate-and-test, we choose a variable and split problem according to different values assigned to the selected variable.

There are many ways to reduce the search or to speed up search process. We discuss them in Section 2.4.

2.2.2 Davis-Putnam procedure

We introduce the most powerful procedure called the DPLL (Davis-Putnam procedure) [6, 7] in solving CSP in propositional logic. It was intended for solving the *Satisfiability Problem* (See defintion 2.1.4).

There are four inference rules in DPLL, but only the last one is necessary:

1. **Tautology Deletion:** Remove a clause that is always true since it doesn't affect the satisfiability.

2. **Pure-literal:** A variable is *pure* in a set of clauses if it occurs only positively or only negatively. If a variable is pure, setting its corresponding literal to

true gives more satisfiability than false. Hence, all clauses containing the literal can be removed.

3. **Unit-clause:** A clause is a *unit clause* if it contains only one literal. If there is a unit clause, say L, in the sentence, then L must be true. All occurrences of L in the set of clauses have to be true.

4. **Splitting:** Let S be the set of clauses and x be a variable in S. Let $S[x \mapsto \texttt{true}]$ and $S[x \mapsto \texttt{false}]$ be the sentences in which x is set to true and false, respectively. Then S is satisfiable if and only if $S[x \mapsto \texttt{true}]$ or $S[x \mapsto \texttt{false}]$ is satisfiable. So S can be split into two cases: $S[x \mapsto \texttt{true}]$ and $S[x \mapsto \texttt{false}]$.

It is easy to see that DPLL is a decision procedure for the SAT problem. We mention that Unit-literal rule should be applied as early as possible, since it only simplifies the set of clauses, and this simplification may propagate to another literal. The next rule, the Pure-literal rule, is often not implemented for two reasons. First, to identify the purity of a literal is time -consume (with respect to the other rules) and the effect is often not significant. Second, this rule is only useful for testing satisfiability. If one needs to enumerate all satisfiable assignments instead of just finding one, then applying the Pure-literal rule may loose some possible assignments. We mention again that only Splitting rule is necessary for completeness and that DPLL, in fact, lies in the generate-and-test paradigm.

2.2.3 Ordered binary decision diagrams

OBDD (Ordered Binary Decision Diagram) [9, 10] is designed for efficiently manipulating boolean functions. It was proposed by R. E. Bryant in 1986 and has many applications in digital-system design, finite-state system analysis, artificial intelligence, mathematical logic [10], model checking [11], and constraints-solving [12].

Let us imagine that we already have the mechanism of OBDD, and consider how to solve the SAT problem. Let S be a set of clauses. A clause is a boolean function, and hence, can be represented by an OBDD. Clauses are conjoined together in S, so it corresponds to combining all of the OBDD's by the logical-and operator. It follows that S is satisfiable if and only if the combined result is not the *null*

function (boolean function that always outputs 0). In OBDD, Bryant provides an $\mathcal{O}(m_f \cdot m_g)$-time algorithm to perform the *and*-operation, where m_f and m_g are the size of OBDD's for boolean functions f and g, respectively.

This approach incurs a natural question: How to represent a boolean function in OBDD? If the representation is not compact enough, then this approach is useless. A brute-force way is representing boolean functions by truth-tables; this is not amenable in practice since storing an n-variable boolean function requires 2^n bits. But for a usual SAT problem with about 100 propositional variables, it is easy to be solved by a DPLL prover. Bryant overcomes the truth-table representation problem by using the Shannon's expansion rule to expand a boolean function as a *discriminate tree*. We explain it briefly as follows. A boolean function f can be expanded into

$$x \cdot f_{x=1} \vee \bar{x} \cdot f_{x=0}, \tag{2.1}$$

where $f_{x=1}$ is the restriction function of f by $x{=}1$, and $f_{x=0}$ is by $x{=}0$; A total ordering on the variables is given so that the expansion rule can follow this arrangement. Then he defines two operations, the *merge and eliminate rules*, to compact a discriminate tree by an acyclic graph, and thus the number of nodes can be reduced significantly. The idea in the merge rule is combining duplicated structures by using hash table, while in the elimination rule, it removes nodes in the acyclic graph that are redundant in the OBDD representation. After applying these two rules, each boolean function can be represented by a *unique* OBDD, with the assumption that a total ordering on the variables is given. We summarize some features of OBDD:

1. The representation for boolean functions is unique with respect to the ordering on variables and it can be constructed quickly using a hash table.

2. Symmetrical boolean functions can be expressed by OBDD's with $\mathcal{O}(n^2)$ nodes, where n is the number of variables.

3. Given two OBDD's of sizes m_f and m_g, any binary operation on boolean functions can be performed in time $\mathcal{O}(m_f \cdot m_g)$, and identifying whether two OBDD's represent the same boolean function can be done in constant time.

4. Find an assignment that yields 1 or 0 for an OBDD can be done in $\mathcal{O}(n)$, and the number of assignments that satisfy the represented boolean function can be counted in time linear to the size of an OBDD (this is specially useful to CSP's that count the number of answers).

There are many variant of OBDD, including the ZBBD's (Zero suppressed BDD's) [13, 14], used for representing combinatorial sets, and the BRDD's (boolean Ring Decision Diagram's) [15], used for manipulating boolean rings.

Saturation. This traditional method to prove a theorem in logic uses inference rules to generate the consequences of a given formula. Logically, an unsatisfiable formula entails all formulæ. So, to establish satisfiability of formula A, one refutes its negation A', by inferring a contradiction from A'. The total number of consequences is in general exponential.

In general, a proof will require some form of case splitting. Let $A[x]$ denote the current set of formulæ, containing occurrences of the propositional variable x. From $A[x]$, one may infer the disjunction $A[0] + A[1]$, where $A[0]$ and $A[1]$ denote the formulæ after making the assignment $x = 0$ or $x = 1$, respectively. The original formulation of Davis and Putnam is such a method. Given a CNF formula $A[x]$, one splits on x by computing $A[0]$ and $A[1]$, which is followed by merging, which consists of converting (by distributing and simplifying) their disjunction $A[0]+A[1]$ to CNF. More generally, one can split on any formula. Let $A[B]$ be a formula containing a sub-formula B. From A one may infer anything that follows from the disjunction $A[1]B + A[0]B'$. In the clausal setting this is the ground resolution rule: $D+B, B'+C \vdash D+C$. In the sequent setting, this is the "consensus" rule, $B \rightarrow D$, $B' \rightarrow C \vdash D + C$.

One method which is based on the above inference rule is known as *Recursive Learning*, and it was presented in [41], as and efficient technique for a well known domain of automatic test pattern generation. This problem is indeed very useful in VLSI where, for a given circuit, one wants to check whether one of the signals in the design is stuck at zero or one (fault detection). This problem can be easily transformed to a satisfiability problem, (finding a feasible solution to the satisfiability

problem is indeed as generating the test vector). Traditional techniques are based on using a decision tree to systematically explore the search space when trying to generate a test vector. Using recursive learning with sufficient depth of recursion during the test generation process guarantees that implications are performed precisely; i.e. *all* necessary assignments for fault detection are identified at every stage of the algorithm so that no backtracks can occur. Knowledge about necessary assignments is crucial for limiting or eliminating altogether the number of backtracks that must be performed. Backtracks occur only after wrong decision have been made that violate necessary assignments. Hence, it is important to realize that if *all* necessary (mandatory) assignments are known at every stage of the test generation process, there can be *no* backtracks at all.

Learning is defined as *temporary* injection of logic values at certain signals in the circuit to examine their logical consequences. By applying simple logic rules, certain information about the current situation of value assignments can be learned. The learning routines can be called recursively and thus provide for completeness. The maximum recursion depth determines how much is learned about the circuit. The time complexity of this method is exponential in the maximum depth of recursion. It is clear that any method that identifies all necessary assignments must be exponential in time complexity because this problem is NP-complete.

The algorithm described in Figure 2.2 represents the recursive learning algorithm that was presented in [41]. G represents the gates of the circuits, while r is the current recursion level of the algorithm (initially equals 0), and r_{max} is the maximum recursion level. The algorithms starts with applying `MakeDirectImplications` on the gates of G. This function indeed computes implications according to the values on the pins of every gate. For example, assume an AND gate, g of two signals a and b. If it is given that the value of g is 1 then it can be implied immediately that the value of both a and b is 1. Next, the function function `UnJustifiedGates` is applied to compute all the gates which are not justified. A gate g is called *unjustified* if there are one or several unassigned input or output signals of g for which it is possible to find a combination of value assignments that yields a conflict at g. Otherwise, g is called justified.

```
Procedure RecursiveLearning(G, r+1, r_max)
    MakeDirectImplications(G)
    U^r := UnJustifiedGates(G)
    if (r < r_max) then
        foreach g_i ∈ U^r
            J := Justifications(g_i)
            foreach justification J_i ∈ J
                MakeAssignments(J_i)
                RecursiveLearning(G, r+1, r_max)
            If there is a signal f in the circuit G which has the
            same value V in all consistent justifications J_i ∈ J then,
                assume f = V in level r
                MakeDirectImplications(G)
            If all the justifications are inconsistent then,
                values in level r are inconsistent
```

Figure 2.2: Recursive learning

A *justification* of a gate g is a set of signal assignments of its unassigned input or output signals, that make the value of the gate g justified. As an example, assume an OR gate g between two signals a and b. Assume also that the signal a is already assigned with the value 0. The justifications of the gate g are $\{b = 0, g = 0\}$ and $\{b = 1, g = 1\}$. So the main part of the algorithm is to compute the assignments for all the justifications, and call recursively with the next recursive level. The learning part is whenever a signal f is the circuits get a concrete value V in all the consistent justifications, then $f = V$, and direct implications are computed out of this new fact. It is of high importance to notice that the consistent justification is made for concrete values V, only then a learning is achieved. In case the set of the assignments weren't consistent, then the current level of recursion is not consistent.

2.2.4 Integer linear programming

In *Linear Programming*, constraints are expressed by a system of linear inequalities

$$\sum_{1 \leq j \leq n} a_{i,j} \cdot x_j \geq b_i, for 1 \leq i \leq k, \tag{2.2}$$

where k is the number of constraints, and a's and b's are constants. It asks for the following question: Find and assignment that *minimizes* the following *objective function*

$$\sum_{1 \leq j \leq n} d_j \cdot x_j \tag{2.3}$$

for constants d's. *Linear Programming* is known to be solvable in polynomial time [17,18]. In practice, however, the potentially exponential *Simplex Method* [19] is widely used. When the variables are restricted to integers, it is called *Integer Linear Programming*, and is known to be much harder than the original case over the real numbers. We can easily reduce the SAT problem (cf. Section 2.2.2) to *Integer Linear Programming* by restricting the variables to be 0-1 values through inequalities and replacing the logical-or by summation and the negation of variable x by 1-x. For example, let S be $\{x \vee y, x \vee \neg y, \neg x \vee y, \neg x \vee \neg y\}$. Then S can be transformed into the following linear inequalities:

$$x + y \geq 1, x + (1 - y) \geq 1, (1 - x) + y \geq 1, (1 - x) + (1 - y) \geq 1, \tag{2.4}$$

which are equivalent to

$$x + y \geq 1, x - y \geq 0, -x + y \geq 0, -x - y \geq -1. \tag{2.5}$$

We also need to add $1 \geq x \geq 0$ and $1 \geq y \geq 0$ to make sure that x and y are binary. Hence, *Integer Linear Programming* is also NP-hard.

Many strategies are developed for *Integer Linear Programming* such as linear program relaxation, branch-and-bound, cutting-plane method, and interior-point methods. For extended reference, see [20].

2.2.5 The boolean Ring method

In boolean ring, two operators, the *logical-and* ($\cdot$) and *exclusive-or* ($\oplus$), are used instead of the three operators, *logical-and* ($\wedge$), *inclusive-or* ($\vee$), and *negation* ($\neg$), in boolean algebra. Boolean ring uses fewer operators, but the expressive power of boolean ring is much superior than boolean algebra in term of *formula complexity* [15]. Let x and y be propositional variables. It is not difficult to verify that the following rules define the correspondence between boolean algebra and boolean ring:

$$\begin{aligned} x \wedge y &\longleftrightarrow x \cdot y \\ x \vee y &\longleftrightarrow x \cdot y \oplus x \oplus y \\ \neg x &\longleftrightarrow x \oplus 1 \end{aligned}$$

The operator ($\cdot$) may be suppressed and operator $\oplus$ may be replaced by ($+$) since a boolean ring is indeed an integer polynomial over the filed $\mathbb{Z}_2$ with the idempotent rule ($x \cdot x = x$ for all x). Therefore the expressions in boolean ring are often called *boolean polynomials*. One of the advantages in boolean ring is that *equational replacement* can be applied naturally, which is difficult in boolean algebra. For example, if we have two equations

$$xy + z = 0 \tag{2.6}$$

$$xy + 1 = 0 \tag{2.7}$$

then it is easy to conclude that z should be 1. This fact can be seen by 'adding' Eq. 2.6 to Eq. 2.7; the sum should be $xy + xy + z + 1 = 0$, but since *exclusive-or* is nilpotent (i.e. $x + x = 0$ for all x), the two xy's disappear and it is equivalent to $z + 1 = 0$. Again by adding 1 on both sides and by the nilpotence, we have $z = 1$. This example illustrates that boolean ring is much more 'algebraic' than boolean algebra. Once constraints are expressed by boolean ring, several methods can be applied to it, which includes the term-rewriting method [21], the Gröbner bases construction [4, 5, 23], and the Davis-Putnam method in boolean ring [24].

2.3 Difficulties with CSP

Exhaustive search is helpful for solving combinatorial problems. In this section, we show the difficulties of this approach.

2.3.1 Combinatorial explosion

When the number of variables in a CSP increases, the search space usually grows exponentially. This fact implies that exhaustive search is not scalable.

This barrier is often called the *combinatorial explosion*, and can be explained by the following reasons. The first is that the possible combination in a problem is always *multiplicative* to the number of variables, which is often quadratic or cubic to the problem size. This kind of growth rate is extremely large. For example, if the nodes in the search space is 2^{n^2} for a problem, then when n (the size of input) goes from 5 to 10, the number of search spaces goes from 2^{25} to 2^{100}, which is not manageable by any modern computer. The second is that the running time that we would like to pay is only *additive*. Therefore, the time that we would like to pay is relatively smaller than the required running time for exhaustive search.

The above argument may seem naive since it assumes that the search is brute-force. However, no known improvement can completely overcome this problem. In the next section, we summarize *negative results* on solving CSP's.

2.3.2 NP-completeness

The first *negative result* in solving CSP's is that SAT is NP-complete. An NP-complete problem cannot have any polynomial-time algorithm unless P=NP. The complexity class P contains all decision problems that are solvable in polynomial time and the complexity class NP contains those that can be verified in polynomial time. However, no one knows whether P equals NP, and NP-completeness captures the idea that they are problems most unlikely in P. Furthermore, once an NP-complete problem is proven to be in P, then all NP-problems are polynomial-time solvable.

Some special cases of SAT remain NP-complete. The k-SAT problem is the

restriction that the number of literals in each clause is at most k. An instance of this, called the 3-SAT, is NP-complete [8]. It is the first known NP-complete problem and logical formulation makes it much easier to express other problems. Hence, 3-SAT is widely used to prove the NP-completeness for the other problems. In fact, 3-SAT is still NP-complete even if we restrict the occurrence of each variable (either positive or negative) at most three times. 2-SAT, however, is linear time solvable [25] (by using a slight modification of the Davis-Putnam procedure discussed in Section 2.2.2 by paralleling the Splitting rule). Another direction to impose the restriction is on the number of positive literals in each clauses. 3-SAT is still NP-complete even if the number of positive literals in each clause is at most two (with or without the restriction that the number of occurrences of each variable is three). However, Horn-SAT where each clause contains at most one positive literal is linear-time solvable. The other formulation, such as the integer linear programming, also faces the same condition. For comprehensive discussion on computational complexity, we suggest [26].

Being in NP-complete does not imply that it cannot be solved. It only says that it seems impossible to find an exact and deterministic way to collect a solution. There are several approaches to this. First, some problems do not require optimal answers and a near-optimal solution is still acceptable, so *approximation algorithm* may be useful. The second approach resorts randomness, and for example, randomized algorithms can do primality testing in polynomial time, and no deterministic polynomial-time algorithm is known. However, this approach is inadequate for solving mathematical problems, since a mathematical conclusion should not involve uncertainty in its proof. Therefore, exhaustive search is possibly the only candidate for solving hard CSP's.

2.4 Heuristics

Suppose that exhaustive search is the only way that we know how to solve a problem. This means that no *efficient* method is available, but among all ways to exhaustive search, some are more *effective* than the others. Strategies that improve exhaustive

search are called *heuristics*. We remark, however, that no method is appropriate in all instances. We separate the discussion according to *complete search* and *incomplete search*. The basic difference between them is that in complete search, when it reports no answer in a decision problem, there is really no answer. Incomplete search cannot guarantee that. Incomplete search is especially useful when there is an easy way to verify a solution and when the solutions are quite dense in the whole space. It is often more effective than complete search when it is applicable. But in many applications, incomplete search is either useless or provides only upper bounds or lower bounds. Problems that we consider here are all of this kind, so complete search is our main interest. Extended references and discussion can be found in [20, 27].

2.4.1 Complete methods

A straightforward way to achieve complete search is to explore the entire search space, like what we have described in Section 2.2.1. Recall that in a search process, a search tree is incrementally built and internal nodes on the tree represent *partial assignment* that can be extended by further instantiating variables. Not all search space defined in this way is necessary. There are two kinds of nodes in a search tree in a decision problem: Those that lead to a visible solution, and those that do not. If we can give up nodes that will not lead to a visible solution as early as possible, then the search space can be reduced greatly. Forward-checking (cf. Section 2.2.1) is such an example. On the other hand, the search space varies with different variable-splitting orderings. A better splitting strategy can often keep the whole search space smaller. We discuss these strategies as follows.

Branch-and-bound. Suppose we want to solve a minimization problem. As an example, consider the MBCSP (Monotone Boolean Constraint-Satisfaction Problem), which asks the following question: Given a set of clauses that contain no negative literals, find and assignment that uses as few 1's as possible. Informally speaking, when the search is proceeding, if we have already found an assignment that uses only five 1's, then we can prune all branches that have *already* used five 1's.

Branch-and-bound is useful in many optimization problems, such as the traveling-salesman problem. It is one of the most useful, general and elegant heuristic techniques.

Variable-splitting strategies. A search tree is determined by the ordering of split variables. Different orderings may lead to different sizes of search trees. For example, the Unit-clause rule in the Davis-Putnam procedure in section 2.2.2 can be seen as "lifting" a literal in a unit-clause to be split. This literal is set `true` and the effect can be propagated to the other clauses. This explains why the Davis-Putnam procedure is effective. For general CSP's, heuristics of variable splitting such as to split a variable with the least number of branches or a variable that satisfies the most number of constraints, is also useful. There is a limit to how much variable-splitting strategies can help. There exists a search tree with exponential nodes regardless of the splitting ordering when only forward-checking is implemented. We remark that although variable splitting strategies cannot make search polynomial-time solvable, they are usually quite useful in solving practical problems.

Prune-and-search. Prune-and-search always generates linear-time algorithms to solve a problem. It lies in different spectrum to the generate-and-test paradigm (cf. Section 2.2.1). Its general description is as follows. Suppose the input of a problem is of size m. If each time we can *prune* a constant fraction of the input in linear time, then the whole problem is still solvable in linear time. A typical example to this strategy is finding the k-th largest element is an array. If at each run, a constant fraction of the input can be identified not to contain the k-th largest element in linear time then they can be pruned, and the process can be applied recursively. The overall time is still linear.

Incomplete methods. Incomplete methods are useful for many applications and they gave different levels of satisfaction under different assumptions. If there exists an 'easy' way to verify the correctness of a solution found by an incomplete method, then its major problem is how to find such an answer. Several techniques can be used to this, however, they are not beyond two factors: local improvement and

randomness.

Chapter 3

Boolean Rings

3.1 Introduction

Definition 3.1.1 A *ring*, in the mathematical sense, is a set $\mathcal{R}$ together with two binary operators $+$ and $*$ (commonly interpreted as addition and multiplication, respectively) satisfying the following conditions:

1. Additive associativity: For all $a, b, c \in \mathcal{R}$, $(a + b) + c = a + (b + c)$,

2. Additive commutativity: For all $a, b \in \mathcal{R}, a + b = b + a$,

3. Additive identity: There exists an element $0 \in \mathcal{R}$, such that for all $a \in \mathcal{R}$, $0 + a = a + 0 = a$,

4. Additive inverse: For every $a \in \mathcal{R}$, these exists $b \in \mathcal{R}$, such that $a + b = b + a = 0$

5. Multiplicative associativity: For all $a, b, c \in \mathcal{R}$, $(a * b) * c = a * (b * c)$,

6. Left and right distributivity: For all $a, b, c \in \mathcal{R}, a * (b + c) = (a * b) + (a * c)$ and $(b + c) * a = (b * a) + (c * a)$

Definition 3.1.2 We say that a ring $\mathcal{R}$ has a *multiplicative identity* if there exists an element $1 \in \mathcal{R}$ such that $a * 1 = 1 * a = a$ for all $a \in \mathcal{R}$.

Definition 3.1.3 A *boolean ring* is a ring $\mathcal{R}$ that has a multiplicative identity, and in which every element is idempotent, i.e. $a * a = a$ for every $a \in \mathcal{R}$.

Boolean ring is an algebraic structure which is equivalent to boolean algebra. The major representational differences are that boolean rings use *exclusive-or* (+) instead of ($\vee$) to represent boolean functions, and that there is no need for negation in boolean ring. Furthermore, there is a *unique* boolean ring normal form for every boolean function. It is curious, however, that in spite of its long history and elegant algebraic properties, the boolean ring representation had rarely been used in the computational context.

3.2 The formalism of boolean rings

A *Boolean ring* is a commutative ring (B,+,$\cdot$,0,1) in which $\cdot$ is *idempotent* (i.e. $x \cdot x = x$) and + is *nilpotent* (i.e. $x + x = 0$). The operator + is known in logic design as *exclusive-or* (xor). By introducing the relationship

$$\begin{aligned} x \wedge y &\longrightarrow x \cdot y \\ x \vee y &\longrightarrow x \cdot y + x + y \\ \neg x &\longrightarrow x + 1 \end{aligned}$$

one can show that the corresponding algebraic structure (B,+,$\cdot$,0,1) is a boolean algebra [36]. Expressions in boolean ring are called *boolean polynomials*, or for short polynomials, in order to distinguish them from boolean formulas represented in boolean algebra. A *monomial* in boolean ring is either the expression 0, 1, or a 'product' of distinct variables. On other words, a polynomial that uses only '$\cdot$'. A boolean ring is in *Boolean Ring Normal Form* (BRNF) if it is a sum of distinct monomials. The axioms of boolean rings are:

$$\begin{aligned}
xx &= x \\
x0 &= 0 \\
x1 &= x \\
x + x &= 0 \\
x + 0 &= x \\
x(y + z) &= xy + xz \\
xy &= yx \\
x + y &= y + x \\
(xy)z &= x(yz) \\
(x + y) + z &= x + (y + z)
\end{aligned}$$

Distributivity (sixth axiom) is the potentially expensive step, even when the subterm x is shared in some directed-acyclic-graph. Propositional formulæ can be converted to boolean ring (exclusive-or) normal form (BRNF): tautologies reduce to 1; contradictions (unsatisfiable formulæ) to 0; contingent formulæ to neither. (See [34].)

The boolean ring formalism differs from boolean algebra in that it defines a unique normal form (up to associative and commutativity of the two operators) for each boolean formula, called a *boolean polynomial*, that can be directly by applying the first six axioms from left to right to any formula. Using boolean rings, however, is not without drawbacks. The main problem is that the distributivity law causes the length of the boolean polynomial to be exponential in the worst case.

Though, considering the above simplification rules as rewrite rules from left to right, we can easily recognize the distributive rule $x(y + z) = xy + xz$ as the reason for such exponential explosion. In our work we will eliminate the distributive rule from the boolean ring simplification rules, and instead other techniques will be used (See next section).

Definition 3.2.1 (Binomial Equation) A boolean polynomial is called binomial is it is either of the form $m_1 + m_2$ where m_1 and m_2 are distinct monomials. A boolean ring equation is called binomial if it is either $m_1 + m_2 = 0$ or of $m_1 = 0$

Definition 3.2.2 (Linear Equation) A boolean polynomial is called linear if it is

expressed as a sum of distinct variables or 1. A boolean equation is called linear if it is an equation of a linear polynomial.

In the following we define several parameters in a boolean polynomial. The *degree* of a monomial is the number of distinct variables in the monomial. The degree of the monomial 1 is zero. The degree of a boolean polynomial is defined to be the largest degree among all monomials in it. The *length* of a polynomial is the number of its distinct monomials. The number of different variables in a polynomial is called its *variety*. We use $\mathcal{B}$ to denote a set of binomial equations and use $\mathcal{L}$ to denote a set of linear equations. Therefore, by definition, the length of each equation in $\mathcal{B}$ is at most two and the degree of each equation in $\mathcal{L}$ is at most one.

Our algorithms are on $\mathcal{B}$ and $\mathcal{L}$. Let us express it by a pair $(\mathcal{B}, \mathcal{L})$. Instead of solving a set of boolean ring equations, we try only to decide the satisfiability of $\mathcal{B} \cup \mathcal{L}$. This splitting is useful since by this way we can get benefit from the characteristics of everyone of these representations.

Inference rules Previous studies of the above representation of boolean rings did not allow simplification across $\mathcal{B}$ and $\mathcal{L}$ in order to ensure the closeness of the formalism. Hence, equations in $\mathcal{B}$ and $\mathcal{L}$ were processed independently except for the *unit rules*. An equation is called a *unit rule* if it is of the form $x = 0$ or $x = 1$, where x is a variable. The notion $\mathcal{U}$ is generally used to store the set of unit rules currently discovered. Let *reduced*$(\mathcal{L})$ be the resulting set of equations that no more simplifications can be applied within $\mathcal{L} \cup \mathcal{U}$. (Since $\mathcal{L}$ is linear, the process is indeed the Gaussian elimination). The definition of *reduced*$(\mathcal{B})$ is similar, but in addition to this, a new rule is added: Split equation $x_1 \cdot x_2 \cdots x_n = 1$ into $x_i = 1$ for $1 \leq x \leq n$. Unit rules can be applied across $\mathcal{B}$ and $\mathcal{L}$, so the effect of the unit rules may propagate between $\mathcal{B}$ and $\mathcal{L}$. If contradiction happens during simplification, then the input $(\mathcal{B}, \mathcal{L})$ is unsatisfiable.

In our work, we extend the learning between the $\mathcal{B}$ and $\mathcal{L}$ to process also polynomials of the form $x + y = 0$. If this equation was inferred in the binomial system $\mathcal{B}$, then it is transferred automatically to the linear system $(\mathcal{L})$. Further details to follow.

The Bin-Lin Approach. We are studying the advantages of the following novel representation: Let $\mathcal{B}$ be a set of *binomial* equations (equations between conjunctions of propositional variables or constants), and $\mathcal{L}$ be a set of linear equations over the propositional variables. Instead of solving a general set of boolean-ring equations, we try only to decide the satisfiability of $\mathcal{B} \cup \mathcal{L}$. It is built on the combination of $\mathcal{B}$ and $\mathcal{L}$, with the restriction that simplification can only be applied within $\mathcal{B}$ or within $\mathcal{L}$. This model is powerful enough to handle all cases.

The input is a pair $\mathcal{B} \cup \mathcal{L}$ and an ordering $>$ on monomials is given. The inferences rules are:

1. Termination test: If $1 = 0$ has been inferred, the system is unsatisfiable.
2. Tautology Deletion: Remove all trivial equations $A = A$.
3. Decomposition: Decompose any equation $x_1 x_2 \cdots x_k = 1$ in $\mathcal{B}$ into $x_1 = 1, \ldots, x_k = 1$.
4. Propagate Rule: Use all equations of the form $x = 0$, $x = 1$, or $x = y$ in $\mathcal{B} \cup \mathcal{L}$ to simplify equations in $\mathcal{B} \cup \mathcal{L}$.
5. Simple Rules: Use additional optional inference rules to infer unit, binary, or monomial equations that can be added to $\mathcal{B}$ and/or $\mathcal{L}$.
6. Simplification: Reduce one equation by another within $\mathcal{B}$ or $\mathcal{L}$, but not across the two.
7. Splitting Rule: Split the system of equations by considering $\mathcal{B} \cup \{x = 1\}$ and $\mathcal{B} \cup \{x = 0\}$, individually.

Equations in $\mathcal{B}$ and $\mathcal{L}$ are for the most part processed independently. Relatively fast methods exist for processing each of the two components. The simplification step is not needed for completeness, but rather to improve search efficiency.

Simple inference rules are used to discover regularity between variables and generate new equations that can help reduce efforts in exhaustive search. For example, one can find all binary equations $x = y$ that follow from only *one* of $\mathcal{B}$ or $\mathcal{L}$ in polynomial time. The idea of performing shallow inferences to derive essential relations

is central to the method of "recursive learning" [35] that we described in section 2.2.3. Another approach which we are adapting is the Stålmarck's algorithm which is described in the next section.

Chapter 4

Stålmarck's algorithm

One important saturation-based approach is Stålmarck's algorithm [32]. Stålmarck's algorithm is a tautology checker. It deals with boolean formulae, i.e. expressions formed with the two constants 1 (true), 0 (false), the unary symbol $\neg$ (negation), the binary symbols $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (logical equivalence) and a set of variables. This method is working on a special representation of formulas knows as triples. The transformation of the formulas to triples is done in two phases : The first phase is a translation from formulas in propositional logic to formulas built from only implication and 0 (false). The following transformations are repeatedly applied :

$A \vee B$	to	$\neg A \rightarrow B$
$A \wedge B$	to	$\neg(A \rightarrow \neg B)$
$\neg\neg A$	to	A
$\neg A$	to	$A \rightarrow \bot$

The second phase of the transformation is to translate those clauses to triple representation, where a triple (x, y, z) is an abbreviation for $x \leftrightarrow (y \rightarrow z)$. 0 and 1 are treated as special cases of a propositional variable. 0 (False) is written as 0 in triplets, while 1 (True) is written as 1. This process involves defining new *intermediate* variables that will represent sub formulas.

Example: Assume the formula $p \rightarrow (q \rightarrow p)$. We introduce a new variable $b1$ that will be equivalent to the sub-cluase $(q \rightarrow p)$. We introduce another new variable $b2$

which will be equivalent to the whole formula. Thus we get the following :

$$b1 \leftrightarrow (q \rightarrow p)$$
$$b2 \leftrightarrow (p \rightarrow b1)$$

The next step is to translate those clauses to triple representation, where a triple (x, y, z) is an abbreviation for $x \leftrightarrow (y \rightarrow z)$. $\bot$ and $\top$ are treated as special cases of a propositional variable. $\bot$ (False) is written as 0 in triplets, while $\top$ (True) is written as 1. For the above example, we get:

$$(b1, q, p)$$
$$(b2, p, b1)$$

The algorithm works by using these triplets to make logical inferences. All the facts used and deduced by the algorithm can be considered as equations between literals. The starting point is the single assignment $v^* = 0$ where v^* represents the whole formula. The objective is to reach contradictory equation of the form $v = \neg v$.

The algorithm uses a set of inference rules, known as *simple rules* to derive new information about the variables. It uses the triplet together with the existing equations to deduce new equations via some obvious deductions. For example, we know that if $(y \rightarrow z)$ is false, the y must be true and z false. We write rule as :

$$(r1) \quad \frac{(0, y, z)}{y/1 \quad z/0}$$

A *terminal* triplet is one that is contradictory. For example, the triple $(1, 1, 0)$ is terminal because $1 \rightarrow 0$ cannot be true. The only other terminal triplets are $(0, x, 1)$ and $(0, 0, x)$. Applying a rule to an element of a set of triplets gives a new set of triplets into which we substitute the newly calculated variables instantiations. The remaining 6 simple rules are :

$$(r2) \quad \frac{(x, y, 1)}{x/1} \qquad (r3) \quad \frac{(x, 0, z)}{x/1}$$

$$(r4) \quad \frac{(x, 1, z)}{x/z} \qquad (r5) \quad \frac{(x, y, 0)}{x/\neg y}$$

$$(r6) \quad \frac{(x, x, z)}{x/1 \quad z/1} \qquad (r7) \quad \frac{(x, y, y)}{x/1}$$

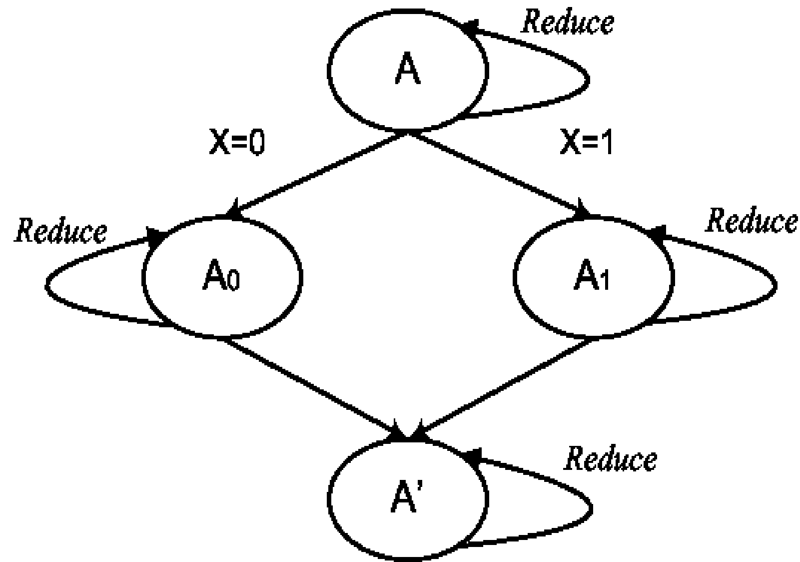

Figure 4.1: Splitting on one variable

In general, simple rules alone are not sufficient to prove formulas. If after applying simple rules, no contradictory assignment has been reached, the next step is to use the so-called *dilemma rule*. This involved case-split over a variable (real or intermediate).

Suppose that applying simple rules has yielded a set of equations (E), and that we choose v as a variable to split over. Then we applying simple rules on the sets $E \cup \{v = 1\}$ and $E \cup \{v = 0\}$ to produce new sets of equations E_1 and E_0 respectively. Even if we have not gained a contradictory assignment in both E_1 and E_0, the case split may still yield new information. Set $E = E_1 \cap E_0$. If a set of equations contains a contradictory assignment, then E equals to the other set. Notice that to distinguish the intersection operation with the recursive learning, the intersection in Stålmarck's method contains equations between variables, e.g. the fact that $v_i = v_j$, while in the recursive learning, only concrete values which where found to be justified are used in the learning process.

We presented in this thesis a similar algorithm based on boolean rings representation. Our procedure has three inference rules. The first one is *reduce*, which reduces a set of boolean ring equations A to an inter-reduced set *reduced(A)*, us-

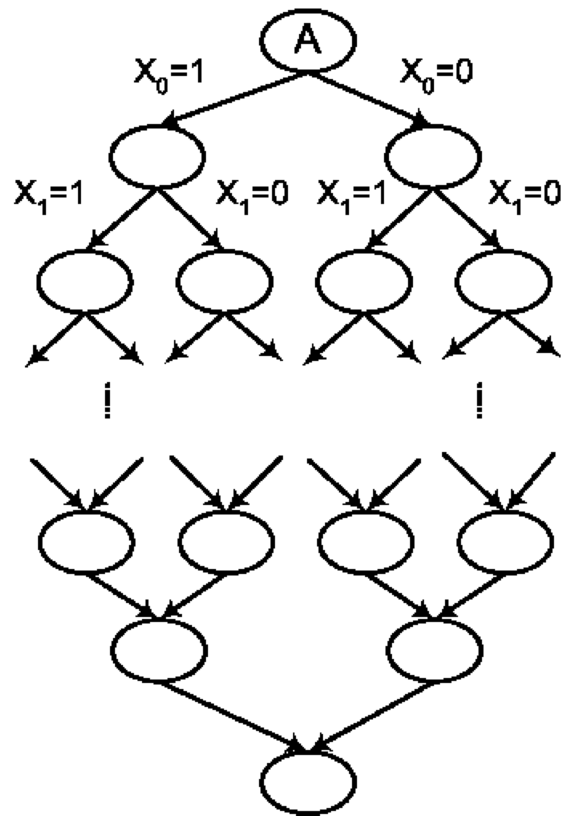

Figure 4.2: Splitting on more variables

ing a set of *Simple Rules*. The second inference rule is *split(A,x)*, which splits A into two sets, $A_0 = A \cup \{x = 0\}$ and $A_1 = A \cup \{x = 1\}$. The third inference rule is *intersect*(A_0,A_1), which computes the intersection between two boolean ring equations sets, A_0 and A_1. The procedure works are follows: Given the inputs set of equations A, we first *reduce* it using a set of simplification rules. If the resulting set contains contradiction (1=0), then A is inconsistent. Otherwise choose a boolean variable x and *split(reduced(A),x)* accordingly to generate A_0 and A_1. Finally *intersect*(A_0,A_1) to generate A'. Reduce A' to get *reduced(A')*. If the resulting set contains contradiction (1=0), then A is inconsistent. Otherwise, split on another variable. If all the variables were split, then we reiterate the splitting process but this time we split for more variables before computing the intersection. Figure 4.1 illustrates splitting on on variable, while Figure 4.2 illustrates splitting on more variables.

```
Procedure Stalmarck (A, V')
    if (V' = ∅) return A
    Pick x ∈ V'
        A_0 := Stalmarck (A ∪ x̄, V' \ x)
        A_1 := Stalmarck (A ∪ x, V' \ x)
        A' := Intersect(A_0, A_1)
        A := Simple Rules(A')
        if Sat(A) return TRUE
        if UnSat(A) return FALSE

Procedure Stalmarck-SAT(A, V, n) :
    A' := Simple Rules(A)
    if Sat(A') return TRUE
    if UnSat(A') return FALSE
    foreach n variables {x_1 ··· x_n} ∈ V
        A'' := Stalmarck (A', {x_1 ··· x_n})
        if Sat(A'') return TRUE
        if UnSat(A'') return FALSE
    return (A'')
```

Figure 4.3: `Stalmarck-SAT`

The above algorithm is illustrated in Figure 4.3. The main function is called `Stalmarck-SAT`. It accepts as arguments the problem A, the set of the variables V participating in the problem and the maximal splitting depth n. The algorithm works as follows: First it performs a call for `Simple Rules` which indeed performs a set of polynomial time inference rules (e.g. unit propagation). If upon completion the problem is found to be satisfiable or unsatisfiable, the result is returned. Otherwise, we iterate on every subset of variables in V of length n. These sets will determine the splittings which are going to be performed. For every one of these

sets, we call a subfunction `Stalmarck` which performs a splitting on every one of the variables in V', but the essence of the algorithm is that it performs an intersection between the two split sets A_0 and A_1, so that new facts which are found in the intersect are stored, and hopefully they can be useful for succeeding splits. In this way, the Stålmarck's method is a learning method.

Notice that some intersections might not be useful, especially if no new facts where found. However, in the framework we are presenting, the Stålmarck's method is performed as a pre-process to a brute-force search algorithm which performs exhaustive search. The main point here is that the Stålmarck's method is a polynomial time cost since we are limiting n to 3 for practical reasons. Further discussion is available in chapter 6 which deals with applying Stålmarck's method on boolean rings.

Chapter 5

Horn Clauses

It is well known that the satisfiability problem is NP-complete for the class of all propositions. Therefore, if one is looking for a polynomial-time satisfiability test, one is led to consider subclasses of propositions. One such class is the class of propositional Horn formulae, which enjoys nice properties [37–39]. We discuss these properties in our work and show how Horn clauses can be integrated into our Sat solver.

5.1 Preliminaries

Definition 5.1.1 A *literal* is either a propositional letter P(a positive literal) or the negation $\neg P$ of a propositional letter P (a negative literal). A *basic Horn formula* is a disjunction of literals, with at most one positive literal. A basic Horn formula will also be called a Horn clause, or simply a clause. A *Horn formula* is a conjunction of basic Horn formulae.

First, observe that every Horn formula A is equivalent to a conjunction of distinct basic Horn formulae by associativity , commutativity and idempotence of "$\wedge$". Since "$\vee$" also has these properties, each basic Horn formula is equivalent to a clause of one of three types:

1. Q, a propositional letter, or
2. $\neg P_1 \vee \ldots \vee \neg P_q$ where $q \geq 1$ and $P_1, \ldots P_q$ are distinct propositional letters; or

3. $\neg P_1 \vee \ldots \vee \neg P_q \vee Q$ where $q \geq 1$, $P_1, \ldots P_q$ are distinct propositional letters, and Q is a propositional letter.

For example, $(\neg P_1 \vee \neg P_2) \wedge (P_3) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_2) \wedge (\neg P_4 \vee P_5)$ is equivalent to $(\neg P_1 \vee \neg P_2) \wedge (P_3) \wedge (\neg P_4 \vee P_5)$. In the rest of the thesis, it will be assumed that Horn formulae are in this "reduced" form, i.e. that there are no duplicate clauses and no duplicate literals within clauses.

5.2 The class of propositional Horn

The class of propositional Horn formulae is obtained by restricting the form of the conjuncts in the conjunctive normal form of a proposition. If a proposition A has conjunctive normal form $C_1 \wedge \ldots C_m$, where each C_i is a disjunction of propositional letters (positive literal) or negations of propositional letters (negative literal), A is a Horn formula if and only if each C_i contains at most one positive literal.

From results of Jones and Laaser [39] ,it can be shown that testing satisfiability of propositional Horn formulae is complete class P of problems solvable in polynomial time (In the size of the input). By observing that the satisfiability problem of Horn propositions reduces to the problem of determining whether the empty string belongs to the language generated by a context-free grammar $G = (N, T, P, S)$, a very simple algorithm running in time $O(N^2)$ can be also obtained.

In [40], the authors present two linear algorithms for deciding whether a propositional Horn formula is satisfiable. The essence of these methods is to test whether sets of paths of a certain kind called *peddlings*, exist in a graph associated with the Horns formula. In brief, the methods differ in the strategy used to find a peddling.

The graph associated with a Horn proposition A describes the logical implications defined by the basic Horn propositions in it. The nodes of this graph are the distinct propositional symbols occurring in A plus two special nodes, one for TRUE and one for FALSE. The edges are labeled with basic Horn formulae. The fundamental property of the graph associated with the proposition A is that A is unsatisfiable if and only if there is a *peddling* from TRUE to FALSE.

5.3 The Graph associated with a Horn propositions and pebblings

Assume that the Horn formula A is a conjunction of M basic Horn formulae, that the number of occurrences of literals in A is N, and the number of distinct propositional letters occurring in A is K. A graph G_A is associated with a Horn proposition A. This graph implicitly represents all possible ways of checking the satisfiability of A, and is a powerful tool. Indeed, the satisfiability problem is expressible as a pebbling problem on G_A, and this provides intuition to the various strategies used by satisfiability testing algorithms. The graph associated with a Horn proposition can be used to determine which propositional letters must be TRUE in all truth assignments satisfying A, if any. A propositional letter Q is forced to be TRUE iff either Q is a basic Horn formula in A, or there is some basic Horn formula $C_i = \neg P_1 \vee \ldots \vee \neg P_q \vee Q$ and it is already been established that $P_1 \ldots \vee P_q$ must all be TRUE. If the above situation occurs and Q must also have the value FALSE (which is the case if $\neg Q$ is a basic Horn formula in A), there is an inconsistency and A is not satisfiable.

The number of nodes of the network corresponding to A is only $K + 2$, and its total size including edges is approximately the size of A. Since it can be processed in linear time this is a fast and novel approach to the Horn formula satisfiability problem.

The Horn clauses is used easily to check in linear time whether the Horn system is unsatisfiable or not by simply checking whether there is a pebbling (i.e. path) from the node marked with `true` to the node marked with `false`. If such path is found, then the system is not satisfiable. However, the system can be used also to infer equality between variables. If a pebbling is found from the node of variable x to a node of variable y and vise-versa, then we can conclude the equality between the variables x and y. This technique was proven to be highly useful in the inference rules system that we built on top of the boolean rings system in the Bin-Lin representation.

The Horn clauses is used in our system since the binomial part on the Bin-Lin

system that we described previously can be easily transformed to a Horn system. In the below, we list down transformation rules from any possible equation in the Bin-Lin system to its equivalent Horn formula.

- $x_i = 1 \quad \longrightarrow \quad x_i$
- $x_i = 0 \quad \longrightarrow \quad \neg x_i$
- $x_1 \cdots x_n = 0 \quad \longrightarrow \quad \neg x_1 \vee \cdots \vee \neg x_n$
- $x_1 \cdots x_n = x_j \quad \longrightarrow \quad \neg x_1 \vee \cdots \vee \neg x_n \vee x_j, \ \neg x_j \vee x_1, \cdots, \neg x_j \vee x_n$

Notice that the transformation in the last rule stems from that fact that the equivalence between $x_1 \cdots x_n$ and x_j can be represented by two implications:

- $x_1 \cdots x_n \Rightarrow x_j$
- $x_j \Rightarrow x_1 \cdots x_n$

while the first implication is by itself a Horn formula, the second one can be split into the following formulas, $x_j \Rightarrow x_1, \cdots, x_j \Rightarrow x_n$ and every one of them is Horn formula.

For the general binomial equation, the transformation is achievable in the same manner. Assume the Binomial equation $x_1x_2 \cdots x_n + y_1y_2 \cdots y_l = 0$. The equation is transformed into two implications:

$$\begin{aligned} x_1x_2 \cdots x_n &\rightarrow y_1y_2 \cdots y_l \\ y_1y_2 \cdots y_l &\rightarrow x_1x_2 \cdots x_n \end{aligned}$$

Every implication can be split into a set of implications such that the right hand side of the implication includes only on variable resulting the following implications.

$$\begin{aligned} x_1x_2 \cdots x_n &\rightarrow y_1 \\ x_1x_2 \cdots x_n &\rightarrow y_2 \\ &\vdots \\ x_1x_2 \cdots x_n &\rightarrow y_l \end{aligned}$$

and

$$
\begin{aligned}
y_1y_2\cdots y_l &\rightarrow x_1 \\
y_1y_2\cdots y_l &\rightarrow x_2 \\
&\vdots \\
y_1y_2\cdots y_l &\rightarrow x_n
\end{aligned}
$$

Notice that every implication is indeed a Horn formula, e.g. $x_1x_2\cdots x_n \rightarrow y_1 = \neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n \vee y_1$. This way, out of the equation $x_1x_2\cdots x_n = y_1y_2\cdots y_l$, the following Horn formulas are introduced:

$$
\begin{aligned}
&\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n \vee y_1 \\
&\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n \vee y_2 \\
&\vdots \\
&\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n \vee y_l
\end{aligned}
$$

$$
\begin{aligned}
&\neg y_1 \vee \neg y_2 \vee \cdots \vee \neg y_l \vee x_1 \\
&\neg y_1 \vee \neg y_2 \vee \cdots \vee \neg y_l \vee x_2 \\
&\vdots \\
&\neg y_1 \vee \neg y_2 \vee \cdots \vee \neg y_l \vee x_n
\end{aligned}
$$

Getting this system of Horn formulas, the associated Horn graph can be built. The main usage of the graph is to check satisfiability of the Horn formulas, by checking whether there is a pebbling from the node representing the TRUE value and the node representing the FALSE value. Though, the above graph can be used for unit refutation to detect equivalence between variables. Checking whether $x_i = x_j$, can be reduced to checking whether pebblings from x_i to x_j and from x_j to x_i exist. More on this in the next chapter.

Chapter 6

Stålmarck's algorithm on boolean rings

In this chapter, we describe how Stålmarck's algorithm can be applied on boolean rings using the Bin-Lin approach, by using efficient techniques like the Horn clauses and the Gaussian elimination.

6.1 Translating the SAT instance to boolean ring representation

An instance of the satisfiability (SAT) problem is a boolean formula that has the following components:

1. A set of n variables: $x_1, \ldots, x_n$
2. The three logical connectives: $logical{-}or(\vee)$, $logical{-}and(\cdot)$ and $negation(\neg)$
3. "(" and ")" for grouping subformulas

The boolean formula can be defined recursively by the following:

- Every variable is a boolean formula.
- If $\mathcal{F}$ is a boolean formula then $\neg\mathcal{F}$ is a boolean formula too.
- If $\mathcal{F}_1$ and $\mathcal{F}_2$ are boolean formulas, then $\mathcal{F}_1 \vee \mathcal{F}_2$ is a boolean formula too.

- If $\mathcal{F}_1$ and $\mathcal{F}_2$ are boolean formulas, then $\mathcal{F}_1 \cdot \mathcal{F}_2$ is a boolean formula too.
- If $\mathcal{F}$ is a boolean formula then $(\mathcal{F})$ is a boolean formula too.

Given a boolean formula $\mathcal{F}$, the *goal* of the satisfiability problem is to determine whether there exists a truth values to the variables of $\mathcal{F}$ that makes $\mathcal{F}$ satisfiable.

For every boolean formula $\mathcal{F}$, there exists a linear translation to boolean ring (BR) formula $\mathcal{F}_{BR}$ with the two logical connectives : $\cdot$ (*logical-and*) and $+$ (*exclusive-or*). The translation is done on the by induction on the subformulas of $\mathcal{F}$. For every subformula $\mathcal{F}_i$ of $\mathcal{F}$ we associate a variable. We use these variables to build a set of equations in BR. The translation is done by the following:

- If $\mathcal{F}_i$ is a variable, then $\mathcal{F}_i$ is also a variable in BR.
- If $\mathcal{F}_i$ is of the form $\neg\mathcal{F}_j$, and the equivalent variable of $\mathcal{F}_j$ is v_j, then a new variable v_i is defined and a new equation $v_i = v_j + 1$ is added to BR.
- If $\mathcal{F}_i$ is of the form $\mathcal{F}_j \vee \mathcal{F}_k$, and the equivalent variable of $\mathcal{F}_j$ ($\mathcal{F}_k$) is v_j (v_k respectively), then a new variable v_i is defined and a new equation $v_i = v_j + v_k + v_j \cdot v_k$ is added to BR.
- If $\mathcal{F}_i$ is of the form $\mathcal{F}_j \wedge \mathcal{F}_k$, and the equivalent variable of $\mathcal{F}_j$ ($\mathcal{F}_k$) is v_j (v_k respectively), then a new variable v_i is defined and a new equation $v_i = v_j \cdot v_k$ is added to BR.

For example, given $\mathcal{F} = x_1 \wedge (x_2 \vee x_3)$, it can be transformed into BR as follows:

$$v_1 = (x_2 \vee x_3) = x_2 + x_3 + x_2 \cdot x_3 \tag{6.1}$$

$$v_2 = x_1 \wedge v_1 = x_1 \cdot v_1 \tag{6.2}$$

The above translation process results a set of equations between variables and BR formulas, expressed using $\cdot$ and $+$ only. The next step is to transform this system to a Bin-Lin representation. Equation 6.2 is already a binomial one while equation 6.1 is not. In order to transform equation 6.1 to Bin-Lin, we introduce a new temporary variable $v3$ and the equation 6.1 is now written as:

$$v_3 = x_2 \cdot x_3 \tag{6.3}$$

$$v_1 = x_2 + x_3 + v_3 \tag{6.4}$$

In this case, equation 6.3 is binomial and equation 6.4 is linear. The number of the new variables defined in BR is equal to the number of the logical connectives in (F) and the number of the auxiliary variables which are defined in the above process the transformation to Bin-Lin representation. Assuming this number is m, then the variable v_m is equivalent to the formula (F). Testing the satisfiablity on this system, is equivalent to the task of testing whether there is an assignment to the variables $x_1, x_2 \cdots x_n$ that makes the variable v_m equals to one 1. This is done indeed by adding an extra linear equation $v_m = 1$ to the Bin-Lin system and checking for satisfiability.

6.2 A generalized algorithm for Satisfiability checking

The approach we are presenting in our work is different from the naïve DPL sat solvers. While DPL solvers are based on splitting on variables till a contradiction is found, and only then a process of conflict learning and backtracking is performed, in our approach, we perform a polynomial cost process of learning before the splitting is performed. This way, the learning can hopefully reduce the number of the splits and thus improve the overall run time.

Figure 6.1 illustrates our boolean ring Sat solver. First, the `Stalmarck` procedure, is as described in chapter 4,where `Simple Rules` are sound polynomial inferences, and `Intersect` is a polynomial under-approximation of the intersection of the theories of the two sets of formulæ. The output of the `Stalmarck` procedure is a set of formulas on which two tests can be applied.; `Unsat` and `Sat` are incomplete polynomial tests for unsatisfiability and satisfiability, respectively. The function `Stalmarck` is being used as a sub-routine in our main `BR-SAT` procedure which acts as a manager to the way the saturation is called. In the following, we are getting

into details of every one of the algorithm's components.

6.3 Unit refutation (aka Simple Rules)

The Bin-Lin representation of the equations is essential in our algorithm since it provides efficient unit refutation operations on the sets of equations. Splitting the set of boolean ring equations into Binomial and Linear enables using algebraic properties of polynomial complexity. Hence new information retrieved from the unit refutation and the intersection operation between two sets of equations, can be retrieved easily.

The first set of the inference rules is defined on formulas, and are derived from the boolean rings axioms. The used axioms are:

$$(1)\ \frac{x \cdot x}{x} \qquad (2)\ \frac{x + x}{0}$$
$$(3)\ \frac{x \cdot 0}{0} \qquad (4)\ \frac{x \cdot 1}{x}$$
$$(5)\ \frac{x + 0}{x}$$

The second set of the inference rules are defined on equations, and they are as follows:

$$(1)\ \frac{\mathcal{F} + 0 = 0}{\mathcal{F} = 0} \qquad (2)\ \frac{\mathcal{F} + 1 = 1}{\mathcal{F} = 0}$$
$$(3)\ \frac{\mathcal{F} + 1 = 0}{\mathcal{F} = 1} \qquad (4)\ \frac{\mathcal{F} + 0 = 1}{\mathcal{F} = 1}$$
$$(5)\ \frac{x_1 x_2 \cdots x_n = 1}{x_1 = 1, \cdots x_n = 1}$$
$$(6)\ \frac{x_1 x_2 + x_1 + x_2 = 0}{x_1 = 0, x_2 = 0}$$
$$(7)\ \frac{x_1 x_2 + x_1 = 1}{x_1 = 1, x_2 = 0}$$

The above rules are used in the `Simple Rules` function, which simplifies the set of the equations.

```
Procedure Stalmarck (A, V')
    if (V' = ∅) return A
    Pick x ∈ V'
        A_0 := Stalmarck (A ∪ x̄, V' \ x)
        A_1 := Stalmarck (A ∪ x, V' \ x)
        A' := Intersect(A_0, A_1)
        A := Simple Rules(A')
        if Sat(A) return TRUE

Procedure BR-SAT(A, V, MaxSaturationDepth) :
    A' := Simple Rules(A)
    if Sat(A') return TRUE
    if UnSat(A') return FALSE
    CurrentSaturationDepth := 1
    While (CurrentSaturationDepth ≤ MaxSaturationDepth)
    Begin
        repeat
            LearningWasFound := FALSE
            foreach CurrentSaturationDepth variables
                    {x_1 ··· x_CurrentSaturationDepth} ∈ V
                A'' := Stalmarck (A', {x_1 ··· x_n})
                if Sat(A'') return TRUE
                if (A'' ≠ A')
                    LearningWasFound := TRUE
        until LearningWasFound = FALSE
        CurrentSaturationDepth++
    End
    DPL (A'')
```

Figure 6.1: BRSat

6.3.1 Unit refutation on the Linear equations

Unit refutation on the *Linear* equations can be performed easily using Gaussian elimination. All the linear equations are boolean ring eqautions in which each monomial is either a single variable or a constant; it takes the form $x_1 + x_2 + \cdots + x_2 = 1$ or $x_1 + x_2 + \cdots + x_2 = 0$, where the x_i are distinct propositional variables. Having this type of equations, Gaussian elimination can be easily applied in order to compute:

- Variables which are equal to 1 (True)
- Variables which are equal to 0 (False)
- New linear equations. e.g. from $x_1 + x_2 = 1$ and $x_1 + x_3 = 1$, one can infer $x_2 + x_3 = 0$.

Notice that the above new facts can now be passed to the Binomial system and thus contribute to the unit refutation which is performed on the Binomial system.

6.3.2 Unit refutation on the Binomial equations

A *binomial equation*, as defined previously, is a boolean ring equation with at most two monomials, that is, an equation of one of the three forms: $m + m' = 0$, $m = 0$ or $m = 1$, where m and m' are products of distinct propositional variables. Unit refutation on the binomial equations is performed using Horn graphs (See chapter 5). The graph associated with the Horn formulas (which are being extracted from the Binomial system) supports very useful functionality. Besides the polynomial time cost for checking whether $(true \rightarrow false)$, it can be used as well for checking equivalency between variables. For example, checking whether the Horn system implies and equivalence between two variables x_i and x_j can be done by checking whether there are peddlings from x_i to x_j and vise versa. These peddlings imply that $x_i \rightarrow x_j$ and vise versa, and thus the equivalence between x_i and x_j.

Unit refutation can be employed by enumerating all the possible equivalence relations between all the variables in the binomial equations system. The equivalence check is performed on the Horn graph in polynomial time. Every equivalence relation

between two variables is added automatically to both the Binomial and the Linear systems.

6.4 The intersection operation

The intersection is performed between two sets of equations A_0 and A_1 which where split on some variable x_i. The purpose of this intersection is to compute relations between variables which may save the number of the needed splitting. In this way, if a new equivalence relation between two variables x_i and x_j belongs to the intersection, then the splitting is performed on one variable. In this way, the number of the splitting can be reduced.

Given two Bin-Lin systems ($\mathcal{B}_0$, $\mathcal{L}_0$) and ($\mathcal{B}_1$, $\mathcal{L}_1$), the purpose is to compute the intersection system ($\mathcal{B}'$, $\mathcal{L}'$) which include the maximum set of relations between variables that exist in both ($\mathcal{B}_0$, $\mathcal{L}_0$) and ($\mathcal{B}_1$, $\mathcal{L}_1$). The intersection is performed on the binomial and the linear systems separately, i.e. $\mathcal{B}' = \mathcal{B}_0 \cap \mathcal{B}_1$, and $\mathcal{L}' = \mathcal{L}_0 \cap \mathcal{L}_1$. Every equation which exists in both $\mathcal{B}_0$ and $\mathcal{B}_1$ (respectively $\mathcal{L}_0$ and $\mathcal{L}_1$) exists in $\mathcal{B}'$ (respectively $\mathcal{L}'$). The unit refutation which enumerates all the possible equivalence relation between all the variables, is the operation that enriches the intersection and causes the reduction in the splitting.

6.5 The Sat solver main steps

Notice that our sat solver consists of two main parts: The first one is the saturation phase, where learning new equations and relations between variables is performed. This is a limited procedure in the maximal allowed splitting depth. Once this depth is reached, and no results of the satisfiability of the problem are found, we skip to the DPL based solver. In our flow, we limit the number of the maximal saturation depth to 3. Notice also that the algorithm for which the results where performed, has a more reduced number of allowed splits in the saturation phase.

The saturation phase works as follows: we first perform simple rules and check whether the problem is satisfiable or unsatisfiable. If the answer is one of the above,

then we exist and the problem was solved in polynomial time. Otherwise, we iterate the saturation depth starting with 1 till we reach `MaxSaturationDepth`. For every iteration, we call `Stlamarck` procedure using any set of variables of length equation to current saturation depth (`CurrentSaturationDepth`). This process continues till no learning is found and in this case we increase `CurrentSaturationDepth`. The process ends till we reach `MaxSaturationDepth`.

Notice that the pre-process stage is assumed to find as much learning as it can in order to help the DPL in the splits. We tested this method and it was proven to be a useful one. More results in the next chapter.

6.6 Examples

Example 1: Consider the following Bin-Lin system:

$$y + z + t_1 = 1 \tag{6.5}$$

$$x + t_4 = 1 \tag{6.6}$$

$$x + t_3 + t_5 = 0 \tag{6.7}$$

$$x + t_3 + t_6 = 1 \tag{6.8}$$

$$t_1 \cdot t_2 + t_3 = 0 \tag{6.9}$$

$$t_3 \cdot t_4 + t_5 = 0 \tag{6.10}$$

$$x \cdot t_3 + t_6 = 0 \tag{6.11}$$

Notice that the first 4 equations are the linear part and the last 3 equations are the binomial. The inputs of the system are x, y, z while the other t_i variables are temporary variables . Based on the simple rules where were presented in section 6.3, no learning can be performed in this case, and thus a splitting is needed. Assume that the variable we need to split is x.

In case of $x = 0$, from equation 6.6 we get $t_4 = 1$ and from 6.11 we get $t_6 = 0$. From 6.8 we learn that $t_3 = 1$ and thus from equation 6.9 that $t_1 = 1$. From 6.5 that $y + z = 1$. Similarity, in case of $x = 1$, equation 6.6 implies that $t_4 = 0$.

Equation 6.10 implies $t_5 = 0$ and thus from equation 6.7 we learn that $t_3 = 1$. From equation 6.9 we learn that $t_1 = 1$. From 6.5 that $y + z = 1$.

This example demonstrates how inequality between the variables y and z can be easily concluded since it appears in both splitting branches. In the next example, we'll see how the combination of Gauss elimination and Horn clauses can be useful in a way that unsatisfiability of formulas can be detected after splitting on variable x while without these techniques further splitting is needed.

Example 2: Consider the following Bin-Lin system, as a complementary set to the one presented in example 1.

$$y + t_{10} = 0 \tag{6.12}$$

$$t_{10} + t_{11} = 0 \tag{6.13}$$

$$t_{13} + t_{14} = 0 \tag{6.14}$$

$$t_{12} \cdot t_{14} + t_{11} = 0 \tag{6.15}$$

$$z + t_{12} \cdot t_{13} = 0 \tag{6.16}$$

The system of example 1 is :

$$y + z + t_1 = 1 \tag{6.17}$$

$$x + t_4 = 1 \tag{6.18}$$

$$x + t_3 + t_5 = 0 \tag{6.19}$$

$$x + t_3 + t_6 = 1 \tag{6.20}$$

$$t_1 \cdot t_2 + t_3 = 0 \tag{6.21}$$

$$t_3 \cdot t_4 + t_5 = 0 \tag{6.22}$$

$$x \cdot t_3 + t_6 = 0 \tag{6.23}$$

Notice that the above first 5 equations can be transformed to a Horn system (based on the description in section 5.3). Applying the learning algorithm on the Horn graph (built on top of equations 6.12 ... 6.16) , we can learn that $y \rightarrow z$ and

also $z \to y$ which indeed an equivalence between the variables z and y, i.e. the above system implies $z + y = 0$. This fact accompanied with the above conclusion from example 1 ($z + y = 1$) immediately implies that the system is not satisfiable. Without the Horn learning we should need to split more on one of the variables y or z.

This example illustrates a case were extra learning would save redundant splitting. In this case, the critical learning about the equality between the variable z and y was done by performing Horn based learning. The same above example, if passed to a Stålmarck's based representation, further splitting would be needed since it doesn't have the above enriched learning techniques.

Chapter 7

Numerical Experiments

7.1 How the tests were carried out

The test base that we had included more than 700 real life tests which were taken from circuit design. The problems inputs are ranging from 5 to 12 inputs. The reason for not choosing larger number of inputs is due to the fact that for problems with more than 12 inputs, the Sat solver was exceeding the timeout limit which was specified (300 CPU seconds). It is important to note that the implementation is not performance tuned, and it is not designed to compete with any commercial or academic SAT solver. It was designed to show a Proof of Concept (POC) for the ideas which were presented in previous chapters. Thus, the criterion for the success of our proposed method was based on the number of the splits that our method was saving in the DPL stage, i.e. comparing the number of splits in the DPL stage, with and without the learning capabilities.

The implementation includes two parts: the first one is doing saturation using Stålmarck's algorithm in order to perform as much learning as it can, while the second part is doing simple DPL. We didn't invest effort in optimizing the DPL solver, however we had integrated a simple conflicts reasoning. In any case, the DPL solver was not taking more that $2^n - 1$ splits (where n is the number of the inputs).

The tests were performed on a 32 bit Linux machine (RedHat 7.1) with 2G memory.

7.2 Results

We had performed 5 experiments for every test. For every one of them we show the number of splits in the saturation stage (Stålmarck's algorithm) working as a pre-process, the number of splits in the DPL stage, and a percentage of the improvement in the number of the splits in the DPL stage vs. the initial version of DPL without the saturation.
Here is the description of the experiments which were performed.

1. **DPL without intersection** In this experiment, we perform the basic DPL Sat solver with simple conflict reasoning. No reasoning was performed in this stage. The number of splits for this stage is going to be our reference when we compare the other strategies which were tested.

2. **DPL with basic saturation** Here we apply the Stålmarck algorithm with the basic simple rules retrieved from the Boolean ring axioms as our saturation (learning rules) . The intersection between the two split branches is a syntactic one. We observed around 1.5% reduction in the DPL splits.

3. **DPL with Gauss based saturation** Here we enrich the saturation stage by applying a Gaussian elimination process on the linear part of the two Bin-Lin systems. The reduction average of the splits in the DPL stage raised to 2.2%.

4. **DPL with Horn based saturation** Here we enrich the saturation stage by applying a Horn clauses learning process on the binomial part of the two Bin-Lin systems. The reduction average raised to 6.7%.

5. **DPL with Gauss and Horn based saturation** Here we enrich the saturation stage by applying a Horn clauses learning process on the binomial part of the two Bin-Lin systems, and a Gaussian elimination process on the linear part. The reduction average raised to $\tilde{1}3\%$.

Some experimental results are displayed in Table 7.1.

Inps.	Gts.	I	II		III		IV		V		Avg.
		DPL	DPL	%	DPL	%	DPL	%	DPL	%	Sat.
5	18	15	14	6.7%	14	6.7%	14	6.7%	10	33.3%	13
5	38	31	31	0.0%	31	0.0%	13	58.1%	13	58.1%	15
6	37	63	63	0.0%	56	11.1%	63	0.0%	56	11.1%	18
6	10	20	19	5.0%	19	5.0%	17	15.0%	17	15.0%	16
7	33	38	37	2.6%	37	2.6%	37	2.6%	19	50.0%	19
7	36	95	95	0.0%	95	0.0%	95	0.0%	45	52.6%	21
8	30	116	115	0.9%	115	0.9%	115	0.9%	0	100%	22
8	34	191	191	0.0%	191	0.0%	191	0.0%	6	96.9%	24
9	39	507	507	0.0%	507	0.0%	504	0.6%	127	75.0%	27
9	45	17	0	100%	0	100%	0	100%	0	100%	9
10	63	511	511	0.0%	432	15.5%	511	0.0%	214	58.1%	30
10	54	258	257	0.4%	257	0.4%	257	0.4%	65	74.8%	28
11	41	1794	1794	0.0%	1794	0.0%	896	50.1%	896	50.1%	33
11	52	1187	1187	0.0%	593	50.0%	1187	0.0%	593	50.0%	33
11	46	1983	1983	0.0%	1983	0.0%	1983	0.0%	991	50.0%	33

Table 7.1: Representative runs with and without merging

7.2.1 Explanation to the table contents

The first column marked with "Inps." is indeed the number of the inputs of the problem, while the second column marked with "Gts." presents the number of the gates.

The third column is the results of the first experiment which was described above. The numbers represents the number of the splits in the DPL sat solver. Subsequently, we have 4 main columns which represent the results of the next 4 experiments (marked with I, II, III, IV). Every one of the main columns includes two sub-columns. the first one represents the number of the splits in the the corresponding excrement, while the second sub-columns represents the improvement which is measured by the percentage of the reduction of the number of the DPL splits in the same test compared to the origional learning free DPL run (represented

in the third columns marked with I). The last column, represents the number of the splits which were invested in the learning phase (Stålmarck's method), in average. Notice that in our method we limit the maximal depth in the saturation stage to be 3, which means that we are doing saturation on tree variables at most. This indeed puts polynomial time limits to the number of the splits in the saturation stage.

7.2.2 Analysis of the results

Notice that for the 700+ results that we had, the combination of Gauss and Horn simplification methods on the Bin-Lin system was showing approximately 13% improvement in average. However, we observed app. 10% improvement in average on 14% of the tests, app. 23% improvement in average on 25% of the tests and app. 50% improvement in average on app. 14% of the tests, which means that for 14% of the problems, we are saving 50% of the splits which is pretty remarkable.

Notice also that for some of the tests, the saturation stage was enough to conclude the satisfiability checking result. For some, the basic saturation was enough while for others, only the combination between Gauss elimination and Horn could lead to the result.

Chapter 8

Conclusions and future work

8.1 Conclusions

The experiments which were presented in the previous chapter show that the combination between Gauss elimination and Horn clauses on the Bin-Lin system can improve the Stålmarck saturation pre-process step by finding more relations between variables in the Bin-Lin system and later on, affect the number of the DPL splits. Notice that for some examples, Gauss elimination alone was responsible for the reduction, and in others, Horn algorithms was the main factor. The interesting cases where when combined both of them and then we noticed the extra improvement. We believe that integrating such polynomial time cost methods in existing fast DPL sat solvers can contribute a lot to the performance.

8.2 Future work

As previously mentioned in this thesis, our implementation was not tuned in terms of performance and thus it cannot compete with industrial or academic sat solvers. It was a proof of concept for the ideas that we presented.

Hence, next steps would be to integrate the learning methods that we presented in one of the industrial sat solvers which are performance tuned, and measure the improvement in terms of performance and not only the reduction of the number of the splits.

Bibliography

[1] M. Stone, *The theory of representation of boolean algebra*, Trans. Amer. Math. Soc., 40 (1963), 37-111.

[2] R. L. Jan, *Experimental rsults on propositional theorem proving with boolean rings*, Master's thesis, National Taiwan University, 1997.

[3] I. I. Zhegalkin, *On a technique of evaluation of propositions in symbolic logic*, Mat. Sb. 34(1927),9-27.

[4] B. Buchberger, *Ein algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-dimentional Polynomial Ideal)*, PhD thesis, Math. Inst. University of Innsbruck, Austria, 1965.

[5] Gröbner bases, *An algorithmic method in polynomial idea theory*, in Recent Trends in Multidimentional Systems Theory, N. Bose, ed., D. Reidel Publ. Comp., 1983, ch. 6.

[6] M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem proving*, Comm. ACM, 5(1962), pp. 394-397.

[7] M. Davis and H. Putnam, *A computing procedure for quantification theory*, I. Assoc. Comput. Mach., 7 (1960), pp. 201-215.

[8] S. A. Cook, *The complexity of theorem-proving procedures*, In proc. of the 3rd ACM Symposium on the Theory of Computing. Pages 151-158, 1971.

[9] R. E. Bryant, *Graph based algorithms for Boolean function manipulation*, IEEE Trans. Comput. 35(1986),pp. 677-691.

[10] R. E. Bryant, *Symbolic boolean manipualtion with ordered binary-decision diagrams*, In ACM computing surveys, 24(1992), pp. 293-318.

[11] E. Clarke, O. Grumberg, and D. Long, *Verification tools for finite-state concurrent systems*, in A Decade of Concurrency: Reflections and Perspectives, vol. 803 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 124-175.

[12] T. E. Uribe and M. E. Stickel, *Ordered binary decision diagrams and the Davis-Putnam procedure*, in First International Conference on Constraints in Computational Logics, vol. 845 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 34-49.

[13] S. Minato, *Zero-suppressed BDDs for set manipulation in combinatorial problems*, in Proceedings of the 30th ACM/IEEE Design Automation Conference, 1993, pp.272-277.

[14] H. G. Okuno, S. Minato, and H. Isozaki, *On the properties of combination set operations*, Inf. Process. Lett., 66 (1998), pp. 195-199.

[15] J. Hsiang and G. S. Huang, *Compact representation of Boolean formulas*, Chinees Journal of Advanced Software Research, 1999.

[16] Y. Matiyasevich, *Enumerable sets are Diophantine*, Dokl. Akad. Nauk. SSSR, 191 (1970), pp. 279-282. English translation in: Soviet Math. Doklady, 11 (1970), pp. 354-357.

[17] N. Karmarkar, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373-395.

[18] L. G. Khachian, *A polynomial algorithm in linear programming*, Dokl. Akad. Nauk. SSSR, 244 (1979), pp. 1093-1096. English translation in: Soviet Math. Doklady, 20 (1979), pp. 191-194.

[19] G. B. Dantzig, *Linear Programming and Extensions*, Prinston university Press, Prinston, N.J., 1963.

[20] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, *Algorithms for satisfiability (SAT) problem: A survey*, in Du et al. [22], pp. 19-152.

[21] J. Hsiang, *Refutational theorem proving using term-rewriting systems*, Artificial Intellegence, (1985) pp. 255-300.

[22] N. Dershowitz, *Orderings for term-rewriting systems*, Theoritical Computer Science, 17, (1982) pp. 279-301.

[23] M. Clegg, I. Edmonds, and R. Impagliazzo, *Using the grobner basis algorithm to find proofs of unsatisfiability.*, in Proceedings of the Twenty-Eighth Annual ACM Symposium on the theory of Computing, Philadelphia, Pennsylvania, 22-24 May 1996, pp. 174-183.

[24] J. Hsiang and G. S. Huang, *Some fundamental properties of Boolean ring normal forms*, DIMACS Series in Discrete Mathematics and Teoritical Computer Science Vol. 35, pp. 587-602, 1997.

[25] 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 (1979), pp. 121-132.

[26] C. H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.

[27] S. A. Cook and D. G. Mitchell, *Finding hard instances of the satisfability problem: A survey*, in Du et al. [22], pp 1-18.

[28] A. Haken, *The interactability of resolution*, Theoritical Computer Science, 30 (1985), pp. 297-308.

[29] K. L. M. J. R. Burch, E. M. Clarke and D. L. Dill (1990), *Sequential circuit verification using symbolic model checking*, In proc. 27th ACM/IEEE Design Automation Conference. IEEE Computer Society Press.

[30] M. Davis, G. Logemann and D. Loveland (1962), *A machine program for theorem-proving*, In Comm. ACM 5: 394–397, 1962.

[31] N. Dershowitz and N. Lindenstrauss (1989), *Average time analyses related to logic programming*, In G. Levi and M. Martelli, eds., Proceedings of the Sixth International Conference on Logic Programming (Lisbon, Portugal), pp. 369–381, Cambridge, MA. MIT Press.

[32] M. Sheeran and G. Stålmarck (1998), *A tutorial on Stålmarck's proof procedure for propositional logic*, In Proc. of the 2d Intl. Conference on Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, pp. 82–99. Springer-Verlag.

[33] G. Stålmarck, *A proof theoric concept of tautological hardness*, Unpublished manuscript, 1994.

[34] J. Hsiang and N. Dershowitz (1983), *Rewrite methods for clausal and non-clausal theorem proving*, In Proc. of the Tenth International Colloquium on Automata, Languages and Programming (Barcelona, Spain), volume 154 of *L*ecture Notes in Computer Science, pp. 331–346, Berlin.

[35] W. Kunz (1994), *Recursive learning: A new implication technique for efficient solutions to CAD problems — Test, verification, and optimization*, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 13(9): 1143–1158.

[36] M. Stone, *The throry of representation for boolean algebra*, In Trans. Amer. Math. Soc., 40 (1936), 37-111.

[37] Chang, C. and Lee, R., *Symbolic Logic and Mechanical Theorem-Proving*, Academic Press. New York, 1973.

[38] Henschen, L. and Wos, L., *Unit Refutations and Horn Sets*, J. ACM 21:590-605 (1974).

[39] Jones, N. D. and Lassser, W. Y., *Complete Problems for Deterministic Polynomial Time*, Theor, Comp, Sci. 3:107-117(1977).

[40] 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.

[41] Knux W., Pradhan D.K., *Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems - Test, Verification and Optimization*, IEEE Transactions of Computer-Aided Design (Sept. 1994)

Appendix A

Running BRSat

The way `BRSAT` is called is as follows:

`brsat -f` $< \mathit{InputFile} > < \mathit{Options} >$

The $< \mathit{InputFile} >$ is of the `.br` format (which is described in appendix B).
The $< \mathit{options} >$ are:

`-verbose [default, debug]`

`-initial_split_depth` $< \mathit{number} >$

`-max_split_depth` $< \mathit{number} >$

`-time` $< \mathit{number} >$

`-intersect [none , basic , gauss , horn , advanced]`

`-help`

where $< \mathit{number} >$ denotes an integer number $\geqslant 1$, and `[ val1, val2, ...]` denotes a list of choices, of which one should be selected.

`-verbose`: Determines the verbosity level of the run, where the `default` prints the result of the Sat solver run and some statistics about the number of the splits. Normal output can look like :

```
The system is NOT Satisfiable.
User input variables count = 8
User internal variables count = 30
```

```
Temporary variables count = 40
Total variables count = 78
Saturation split count = 8
Total split count = 8

Total run time is 3.969 CPU second.
```

while the `debug` verbosity level gives more information about the stage in which the sat solver is performing.

`-initial_split_depth`: The default value is 1. $< number >$ determines the starting saturation depth.

`-max_split_depth`: The default value is 3. $< number >$ determines the maximal saturation depth.

`-time`: $< number >$ determines the global timeout of the run in CPU seconds.

`-intersect`: The default value is `basic`. The switch determines the intersection operation which is going to be performed in the saturation level. `none` means that no saturation is going to be performed. `basic` means that simple intersection on the Bin-Lin system is going to be performed, and `gauss` means that Gauss elimination is going to be applied on the linear system before the intersection while `Horn` means that Horn clauses based learning is going to be applied on the Binomial system before the intersection. `advanced` means that a combination of Gauss elimination and Horn based learning is going to be applied on the Bin-Lin system before the intersection.

`-help`: Simply shows the switches and a brief description about everyone of them.

Appendix B

The .br format

$$
\begin{array}{lll}
Formula & \longrightarrow & \textbf{TRUE} \\
 & | & \textbf{FALSE} \\
 & | & \mathsf{variable} \\
 & | & \sim Formula \\
 & | & (\ Formula\ ) \\
 & | & Formula + Formula \\
 & | & Formula\ \&\ Formula \\
 & | & Formula\ \#\ Formula
\end{array}
$$

$$
\begin{array}{lll}
DefintionLine & \longrightarrow & \mathsf{variable}\ \mathbf{:=}\ Formula \\
DefintionSection & \longrightarrow & DefintionLine \\
 & | & DefintionSection
\end{array}
$$

$$
\begin{array}{lll}
SatProblem & \longrightarrow & \textbf{defintions:} \\
 & & DefintionSection \\
 & & \textbf{satisfy:}\ \mathsf{variable}
\end{array}
$$

Where variable := `[a-zA-Z][a-zA-Z0-9[].-]*`

Appendix C

Overview of the Source Code

BRSAT consists of the following source files:

buffer.h/c

Auxiliary functions for implementing a buffer data structure.

table.h/c

Auxiliary functions for implementing a table data structure, including hash functions manipulations.

utils.h/c

Global auxiliary functions.

timer.h/c

Auxiliary functions for controlling the timeout mechanism, including defining, starting and stopping a timer.

IR.h/c

Functions responsible for converting the input file to an Internal Format (IR) which is going to be transformed to a Bin-Lin representation.

binlin.h/c

Functions for manipulating the Bin-Lin representation of the formula.

variable.h/c

Definitions of variables which are mainly used during loading the input file.

monom.h/c

Monom definition and manipulation functions.

fomrula.h/c

Formula definition and manipulation functions.

equation.h/c

Equation definition and manipulation functions.

hornclauses.h/c

The Horn graph data structure and the learning functions.

brsat.h/c

Main sat solver algorithms and verification manager.

main.c

Main of the program. Includes initialization and call to BRSat main functions.