A NEW ALGORITHM FOR THE QUANTIFIED SATISFIABILITY PROBLEM, BASED ON ZERO-SUPPRESSED BINARY DECISION DIAGRAMS AND MEMOIZATION.

by:

Mohammad Ghasemzadeh

A Dissertation Submitted to the COMPUTER SCIENCE DEPARTMENT
of the faculty of MATHEMATHICS AND SCIENCES
in Partial Fulfillment of the Requirements
for the DEGREE OF Ph.D. IN COMPUTER SCIENCE
in the GRADUATE DIVISION of the

UNIVERSITY OF POTSDAM, Germany

2 0 0 5

The dissertation of Mohammad Ghasemzadeh is approved.

_____

Chair Date

_____

Date

_____

Date

University of Potsdam, Germany.

2005

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Potsdam and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Mohammad Ghasemzadeh

ACKNOWLEDGEMENTS

My supervisors professor Christoph Meinel provided consistent and valuable support in all aspects of the work. My thanks to him for allowing me great freedom to pursue my own ideas while always giving very helpful suggestions. I would also like to thank individually: Volker Klotz for sharing with me his insights and helping me in evaluating and improving my algorithms; Volker Schillings for his advice in the early stages of the work and for helping me at my arrival to Germany in finding residence place, doing official regulations and so on; Dr. Fazlollah Adibnia, my colleague in computer department of the university of Yazd, for his guidance and help when I was providing to leave Iran toward Germany for my Ph.D. program. Acknowledgement is also due to the research environment and personnel of the Hasso-Plattner-Institute at the university of Potsdam for their hightech equipments and excellent service. I would also like to acknowledge the financial support provided by the university of Yazd in Iran and the ministry of science, research and technology in Iran which enabled me to carry out this research. If I did not list someone, it does not mean that I am not grateful to him or her.

DEDICATION

To my beloved mother:

She convoyed me when I was leaving my homeland for my Ph.D. program,

She passed away last year, I never saw and will never be able to see her again.

TABLE OF CONTENTS

TABLE OF CONTENTS — *Continued*

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Quantified Boolean formulas (QBFs) play an important role in theoretical computer science. QBF extends propositional logic in such a way that many advanced forms of reasoning can be easily formulated and evaluated. In this dissertation we present our ZQSAT, which is an algorithm for evaluating quantified Boolean formulas. ZQSAT is based on ZBDD: Zero-Suppressed Binary Decision Diagram , which is a variant of BDD, and an adopted version of the DPLL algorithm. It has been implemented in C using the CUDD: Colorado University Decision Diagram package.

The capability of ZBDDs in storing sets of subsets efficiently enabled us to store the clauses of a QBF very compactly and let us to embed the notion of memoization to the DPLL algorithm. These points led us to implement the search algorithm in such a way that we could store and reuse the results of all previously solved subformulas with a little overheads. ZQSAT can solve some sets of standard QBF benchmark problems (known to be hard for DPLL based algorithms) faster than the best existing solvers. In addition to prenex-CNF, ZQSAT accepts prenex-NNF formulas. We show and prove how this capability can be exponentially beneficial.

Keywords: DPLL, Zero-Suppressed Binary Decision Diagram (ZDD), Quantified Boolean Formula (QBF), Satisfiability, QSAT.

Ein neuer Algorithmus für die quantifizierte Aussagenlogik, basierend auf Zero-suppressed BDDs und Memoization.

In der Dissertation stellen wir einen neuen Algorithmus vor, welcher Formeln der quantifizierten Aussagenlogik (engl. Quantified Boolean formula, kurz QBF) löst. QBFs sind eine Erweiterung der klassischen Aussagenlogik um die Quantifizierung über aussagenlogische Variablen. Die quantifizierte Aussagenlogik ist dabei eine konservative Erweiterung der Aussagenlogik, d.h. es können nicht mehr Theoreme nachgewiesen werden als in der gewöhnlichen Aussagenlogik. Der Vorteil der Verwendung von QBFs ergibt sich durch die Möglichkeit, Sachverhalte kompakter zu repräsentieren.

SAT (die Frage nach der Erfüllbarkeit einer Formel der Aussagenlogik) und QSAT (die Frage nach der Erfüllbarkeit einer QBF) sind zentrale Probleme in der Informatik mit einer Fülle von Anwendungen, wie zum Beispiel in der Graphentheorie, bei Planungsproblemen, nichtmonotonen Logiken oder bei der Verifikation. Insbesondere die Verifikation von Hard- und Software ist ein sehr aktuelles und wichtiges Forschungsgebiet in der Informatik.

Unser Algorithmus zur Lösung von QBFs basiert auf sogenannten ZBDDs (engl. Zero-suppressed Binary decision Diagrams), welche eine Variante der BDDs (engl. Binary decision Diagrams) sind. BDDs sind eine kompakte Repräs- entation von Formeln der Aussagenlogik. Der Algorithmus kombiniert nun bekannte Techniken zum Lösen von QBFs mit der ZBDD-Darstellung unter Verwendung geeigneter Heuristiken und Memoization. Memoization ermöglicht dabei das einfache Wiederverwenden bereits gelöster Teilprobleme.

Der Algorithmus wurde unter Verwendung des CUDD-Paketes (Colorado Uni. Decision Diagram) implementiert und unter dem Namen ZQSAT veröffentlicht. In Tests konnten wir nachweisen, dass ZQSAT konkurrenzfähig zu existierenden QBF-Beweisern ist, in einigen Fällen sogar bessere Resultate liefern kann.

CHAPTER 1

Introduction

1.1 Overview

Propositional satisfiability (SAT) is a central problem in computer science with numerous applications. SAT is the first and prototypical problem for the class of NP-complete problems. Many computational problems such as constraint satisfaction problems, many problems in graph theory and forms of planning can be formulated easily as instances of SAT.

Theoretical analysis has shown that some forms of reasoning such as: non-monotonic reasoning, reasoning about knowledge and STRIPS-like planning have computational complexity higher than the complexity of the SAT problem. These forms can be formulated by quantified Boolean formulas then can be can solved as instances of the QSAT: Quantified Boolean formula satisfiability problem. QSAT is a generalization of the SAT problem, it is also the prototypical problem for the class of PSPACE-complete problems.

QBFs can represent many classes of formulas more concisely than conventional Boolean formulas. For each QBF there is an equivalent un-quantified formula. One can reduce a QBF to its equivalent propositional formula and test its truth by a conventional SAT-solver. In this approach, the size of the resulting formula can grow exponentially. This usually makes the method impractical for all but the simplest QBFs. However, the connection between the two problems is close, this is why most of the recent QBF solvers (48; 36; 26; 22) are extensions of the DPLL: Davis-Putnam-Logemann-Loveland procedure (17) presented for the SAT problem.

ZDDs are variants of BDDs. While BDDs are better suited for representing Boolean functions, ZDDs are better for representing sets of subsets. Considering all the variables appearing in a QBF as a set, the propositional part of the formula can be viewed as a set of subsets. This idea has already been used in some previous research works (15; 1) where the SAT problem was considered. They use ZDDs to store the CNF formula and the original DP (18) algorithm to search its satisfiability.

We also found ZDDs suitable for representing and solving the QSAT problem. We represent the clauses in the same way in a ZDD, but we employ an adopted version of the DPLL (17) algorithm to search the solution. In fact, our adopted version simulates the "Semantic tree method in evaluating QBFs". It benefits from an adopted unit-mono-resolution operation which is very fast thanks to the data structure holding the formula. In addition, it stores all already solved subformulas along their solutions to avoid resolving same subproblems. Sometimes the split operation generates two subproblems which are equal. It is very easy to compare and discover their kind of equality with ZDDs; therefore, our algorithm can easily prevent solving both cases when it is not needed.

There are some benchmark problems which are known to be hard for DPLL (semantic tree) based algorithms. ZQSAT is also a DPLL based algorithm, but it manages to solve those instances very fast. This shows how the idea of embedding memoization to the DPLL algorithm can be beneficial.

We improved ZQSAT to let it accept prenex-NNF (QBFs in the form of $Q_1 x_1 \dots Q_n x_n \Phi$, where Φ is a propositional formula in NNF: Negation Normal Form). We showed and proved that this possibility may be exponentially beneficial.

1.2 Thesis Plan

In chapter 2, we first give several definitions about Boolean formulas and their normal forms (which are important in this thesis). Then, we introduce the SAT and QSAT problems. Since the thesis is also concerned with the complexity of algorithms, we give some definitions and classification of complexity classes as well.

Chapter 3 introduces Binary Decision Diagrams (BDDs). Their properties, algorithms, variants and applications are discussed in this chapter. In chapter 4, Zero-Suppressed Binary Decision Diagrams (in short ZBDDs or ZDDs) and their properties are considered in detail. First, we give the definitions, then we compare BDDs with ZDDs. The main ZDD based operations on functions and sets are discussed next. Finally, we present a short introduction to the CUDD package (we used in implementing ZQSAT).

Chapter 5 is dedicated to introducing the QSAT problem in detail, introducing QDIMACS which is a standard input/output format for the QSAT problem, and a short review to the best existing QSAT solvers. Chapter 6 contains the main contributions of the thesis. The basic algorithms for evaluating QBFs and our ideas are given. Embedding memoization and its benefits are followed by the experimental results. The capability of ZQSAT in accepting prenex-NNF is discussed, experimental results and a formal proof is also given to show that it can be exponentially beneficial.

1.3 Acknowledgment

This thesis is mainly involved with the following subjects:

- Boolean formulas and complexity of algorithms.
- Binary Decision Diagrams (BDD) and its variant ZDD.
- The QSAT problem and related solutions.

In preparing the chapters describing backgrounds we have taken some parts from related materials, where we have referenced them immediately.

CHAPTER 2

Boolean Formulas and Complexity

2.1 Introduction

In this chapter first we present a number of definitions to which we will refer in the rest of this thesis. We also take a look at the idea of algorithm complexity. Some parts of this background chapter are taken from(34).

2.2 Boolean Formulas

A Boolean formula like $(x \vee (\neg y \rightarrow z))$ is a formula built up from Boolean variables and Boolean operators like *conjunction, disjunction, and negation.* A *literal* is a variable or the negation of a variable, so for example $\neg x_1$ is a literal and so is x_3. A *clause* is formed by taking one or more literals and connecting them with operator $\vee$ (disjunction also called OR operation), so for example $(x_2 \vee \neg x_4 \vee x_5)$ is a clause, and so is (x_3).

Definition 1 *A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals, that is,* $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_n$, *where* $c_i = (l_{i1} \vee \ldots \vee l_{im_i})$ *and* l_{ij} *is a negative or positive literal.*

For example $(x_3 \vee \neg x4) \wedge (x_1) \wedge (\neg x_3 \vee x_2)$ is a formula in conjunctive normal form (Henceforth, we will just say "CNF formula" or "formula"). Each Boolean formula can be transformed into a logically equivalent Boolean formula in CNF.

The Problem SAT: SAT that stands for CNF-satisfiability, is one of the most important problems in computer science with numerous applications . In this problem we are given Boolean variables $x_1, x_2, \ldots, x_n$ and a Boolean formula ϕ involving such variables; the formula is given in CNF. The question is whether there is a way to assign Boolean values to the variables so that the formula is satisfied. For example, the expression $\neg x \wedge (x \vee y)$ can be satisfied by setting x to `false` and y to `true`, while the expression $x \wedge (\neg x \vee y) \wedge \neg y$ is impossible to satisfy. SAT is the first and the prototypical problem for NP-Complete problems which is a vary important complexity class. (See the next section for explanations about the complexity classes).

Quantified Boolean Formulas: In quantified Boolean formulas (QBF for short), quantifiers may also appear in the formula, like in $\exists x(x \wedge \forall y(y \vee \neg z))$. The $\exists$ symbol is called existential quantifier and the $\forall$ symbol is called universal quantifier.

A number of normal forms are known for each of the above families. In this thesis, the *prenex normal form*, *conjunctive normal form* (*CNF*) and *the negation normal form* (*NNF*) are important.

Definition 2 *A QBF Φ is in prenex normal form, if it is in the form:*

$$\Phi = Q_1 V_1 Q_2 V_2 \ldots Q_n V_n \phi,$$

where $Q_i \in \{\forall, \exists\}$, V_i $(1 \leq i \leq n)$ are disjoint sets of propositional variables and ϕ is a propositional formula over the variables $x_1, \ldots, x_n$. The expression $Q_1 V_1 Q_2 V_2 \ldots Q_n V_n$ is called the prefix and ϕ the matrix of Φ. A QBF is in prenex-CNF if it is in prenex form and its matrix is in CNF.

Definition 3 *A QBF is called closed if all occurrences of variables are in the scope of a quantifier (there are no free variables). For closed QBF formulas the notions of truth, validity and satisfiability coincide.*

The Problem QSAT: QSAT that stands for "satisfiability of quantified Boolean formulas", is a general form of the SAT problem. In this problem we are given a closed prenex-CNF ϕ and similar to the SAT problem the question is whether there is a way to assign Boolean values to the variables so that the formula is satisfied. QSAT is known to be harder than SAT. The problem is the prototypical problem for the PSPACE complexity class. (See next section for explanations about the complexity classes).

As an example:

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

means: "for each truth assignment to x_1 there exists a truth assignment to x_2 such that $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is true". The above QBF is indeed true: if x_1=`true` then x_2 can be assigned to `false`; if x_1=`false` then x_2 can be assigned to `true`; in both cases $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is true.

Although each Boolean formula can be transformed into a logically equivalent Boolean formula which is in conjunctive normal form (CNF), generally this transformation cannot be done efficiently. Another normal form which can be obtained efficiently is NNF which stands for "negation normal form". Since we have considered this format in our research as well, here below we give some definitions.

Definition 4 *A Boolean formula is in negation normal form (NNF) if it is constructed over literals and the Boolean operators conjunction and disjunction. The negation operator can only appear before a variable. A QBF is in prenex-NNF if it is in prenex form and its matrix is in NNF.*

For example $f = (x \wedge \neg y) \vee (\neg z \wedge (x \vee y))$ is in NNF, but $g = \neg(x \wedge \neg y) \vee (\neg z \wedge (x \vee y))$ is not in NNF. Each Boolean formula can be transformed efficiently into a logically equivalent Boolean formula which is in negation normal form (NNF).

Definition 5 *The size of a Boolean formula ϕ in NNF, denoted by size(ϕ), is defined recursively as:*

$$size(v) = size(\neg v) = 1 \text{ for a propositional variable } v \text{ and}$$
$$size(\phi_1 \otimes \phi_2) = size(\phi_1) + size(\phi_2) \text{ for Boolean formulas } \phi_1, \phi_2 \text{ and } \otimes \in \{\wedge, \vee\}.$$

Definition 6 *If $\phi = \bigwedge_{i=1,\ldots,n}(\bigvee_{j=1,\ldots,m_i} l_{ij})$ is a Boolean formula in CNF, then $[\phi]$ denotes the set of subsets of literals $[\phi] = \{\{l_{11}, l_{12}, \ldots, l_{1m_1}\}, \ldots, \{l_{n1}, \ldots, l_{nm_n}\}\}$.*

2.3 Complexity of Algorithms

Which of the following problems is easier to solve: the addition or the multiplication of two n-bit numbers? In general, people feel that adding is easier to perform and indeed people as well as our computers perform additions faster than multiplications. Comparing two algorithms by examining the time consumed by the two procedures is obviously not the right way to compare two algorithms. We need fair criteria for the comparison of algorithms and problems.

Algorithms can typically be judged according to their running time and the memory space requirements. In complexity theory, these resource demands are measured in term of input size. In this way, different problems or different algorithms for the same problem can be compared to each other.

Complexity: The *time complexity* $t_A(n)$ of an algorithm A denotes the maximal number of steps A needs for solving a problem instance with input size n. Analogously, the space complexity $s_A(n)$ denotes the maximal number of memory cells used to solve a problem with input size n. In this regard we consider the main operation in an algorithm and try to give an expression for $t_A(n)$ (saying, according to the input size, the main operation must be executed how many times until the algorithm terminates) and $s_A(n)$ (saying, according to the input size, the algorithm would need at most how many memory cells during its execution).

Asymptotic complexity analysis: Often, it is impossible to determine the exact complexity of an algorithm A. However, one is at least interested in the rate of growth of the functions $t_A(n)$ and $s_A(n)$. Good estimations of this growth serve as criteria to judge the quality of the algorithm. When describing the growth, it is useful to neglect constant factors. This approach is justified as the constant factors depend quite strongly on such technical implementation details as the chosen programming language.

In the analysis, the influence of those technical details should not prevent one from recognizing the essentials. Moreover, it seems reasonable not to neglect only constant factors, but to concentrate in the complexity analysis solely on the dominant terms of the complexity functions. Here, one speaks of *asymptotic analysis*, a widespread analysis technique used in computer science. Indeed, without this technique many basic insights in computer science would not have been gained.

Big-O notation: This notation is used to express the asymptotic complexity of an algorithm. Informally, saying $f(n) = O(g(n))$ "*Read*, f of n is big oh of g of n", means the equation $f(n)$ is less than some constant multiple of $g(n)$. Formally, $f(n) = O(g(n))$ means there are positive constants c and k, such that $0 \leq f(n) \leq c.g(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

As an example, $n^2 + 3n + 4$ is $O(n^2)$, since $n^2 + 3n + 4 < 2n^2$ for all $n > 10$. Obviously $f(n) = O(1)$ means $f(n) \leq c$ for a constant c and $f(n) = n^{O(1)}$ expresses that f is bounded by a polynomial in n. The importance of this measure can be seen in trying to decide whether an algorithm is adequate, but may just need a better implementation, or the algorithm will always be too slow on a big enough input. For instance, quicksort, which is $O(nlogn)$ on average, running on a computer can beat bubble sort, which is $O(n^2)$, specially when number of items n is not too small.

The complexity of an algorithm usually falls in one of the following orders:

- $O(1)$ constant
- $O(log_a n)$ logarithmic
- $O(n)$ linear
- $O(nlog_a n)$
- $O(n^2)$ quadratic
- $O(n^r)$ polynomial
- $O(a^n)$ exponential
- $O(n!)$ factorial

It is not so hard to see that for $a > 1$ and $r > 2$:

$$O(1) < O(log_a n) < O(n) < O(nlog_a n) < O(n^2) < O(n^r) < O(a^n) < O(n!)$$

In many applications, only those algorithms are practical whose running time is bounded from above by a polynomial. An important maybe the most important task in theoretical computer science is to answer the question of when and for which problems such algorithms exist.

There is another notation Ω which is used when lower bounds are considered. We say $f(n) = \Omega(g(n))$ read "f is of order at least g", if there exist two constants $c, n_0 \in N$, such that for all $n > n_0$ $f(n) \geq c.g(n)$. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then we say $f(n) = \theta(g(n))$ and read "f is of order g"

The complexity class P: The class P (polynomial time) denotes the set of all decision problems that can be solved by means of polynomial time algorithms. Obviously the space complexity of a polynomial time algorithm is also polynomially bounded.

The complexity class NP: The subsequently formally defined class NP (nondeterministic polynomial) denotes the complexity class of decision problems for which the answers *can be checked* by an algorithm whose run time is polynomial in the size of the input. Note that this doesn't require or imply that an answer *can be found* quickly. Only can the claimed solution be verified quickly.

For instance, a "yes" answer to the question, "is there a Hamiltonian cycle, or tour, in a certain graph?" cannot be found efficiently (by polynomial algorithm), but if someone gives a tour as a solution, we can efficiently verify whether the claimed answer is valid or not. Therefore, a decision problem belongs to NP If there exists a polynomial algorithm which determines whether the certificate is valid for each input and each possible certificate.

Reduction: Let A and B be two decision problems. A is called *polynomial time reducible* to B, denoted by $A \leq_P B$, if there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$.

Immediate observations: if $A \leq_P B$ and $B \in P$, then also $A \in P$ (conversely, if $A \leq_P B$ and $A \notin P$ then also $B \notin P$); if $A \leq_P B$ and $B \leq_P C$, then also $A \leq_P C$.

NP-Hard: A decision problem A is NP-hard if, for every problem B in NP, $B \leq_P A$. An NP-Hard problem is not necessarily in NP.

NP-Complete: A decision problem A is NP-Complete if, A is in NP, and A is in NP-Hard.

There are thousands of problems which are NP-Complete (no polynomial time algorithm has been found for them yet and no one has ever proved that no such algorithm would exist). As mentioned before, only those algorithms are practical whose run time is bounded by a polynomial.

P = NP?: Obviously, the class P is contained in the class NP, that is, $P \subset NP$. The most important open problem in complexity theory is the question *"P = NP?"*. The importance of this question originates from the fact that there are numerous practically relevant tasks for which polynomial algorithms are not known, but membership in the class NP can be proven. Without a clarification for "$P = NP$? it is not possible to decide whether these tasks can be solved in polynomial time at all, or whether such algorithms have not been found so far.

Nearly all experts in the area of complexity theory conjecture that the classes P and NP are different. This conjecture is supported by a series of results in the investigation of the subsequently defined concepts of polynomial time reduction, and NP-complete problems. NP-Complete problems represent the "hardest" problems within the class NP. It is proven that if any NP-complete problem can he solved in polynomial time, then so can all other problems in NP, and it follows that P = NP.

Space Complexity: Some times algorithms are compared according to the memory space needed by the algorithm. In this regard the following classes are defined.

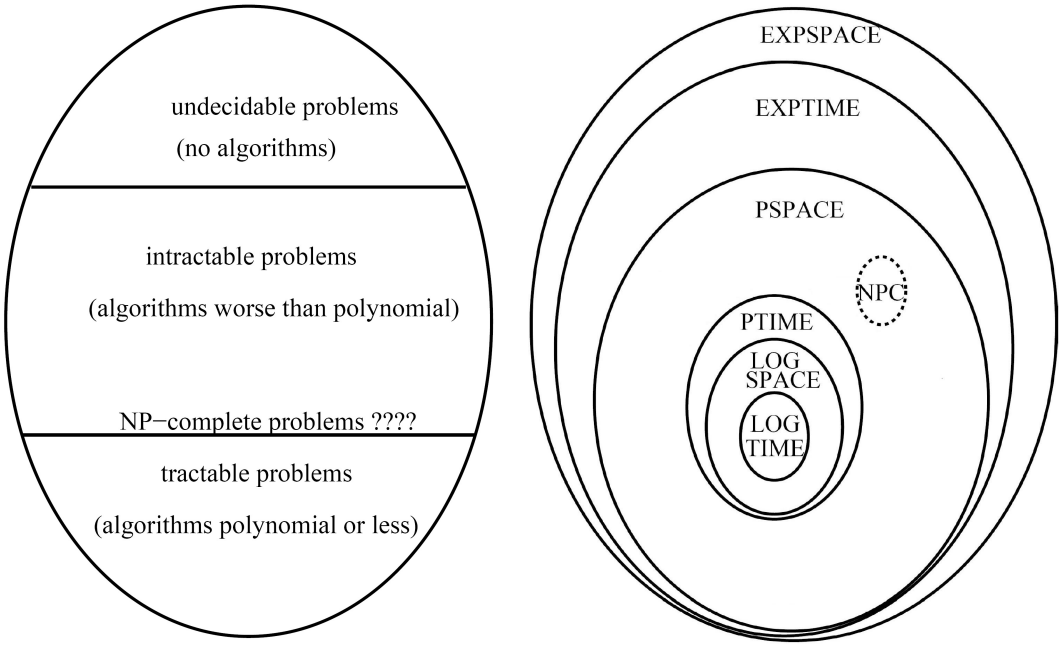

Figure 2.1: Complexity hierarchy.

- LOGSPACE (Logarithmic Space)
- PSPACE (Polynomial Space)
- EXPSPACE (Exponential Space)

Obviously an algorithm is PSPACE if its memory complexity is expressed be a polynomial function of the input size. LOGSPACE and EXPSPACE are deffind similarly.

Figure 2.1-left classifies the problems in three categories:

1. Undecidable problems (no algorithm is known for them).
2. Intractable problems (there are algorithms but worse than polynomial.
3. Tractable problems (polynomial algorithms or even faster are known for them).

Since the answer for "N = NP ?" is not known yet, the "NP-Complete problems" have been considered in the second category, but next to the border with tractable problems. Figure 2.1-right shows that taking time complexity classes into account then:

$$\text{LOGTIME} \subset \text{LOGSPACE} \subset \text{P} \overset{?}{\subseteq} \text{NP} \overset{?}{\subseteq} \text{PSPACE} \subset \text{EXPTIME} \subset \text{EXPSPACE}.$$

CHAPTER 3

Binary Decision Diagrams

Binary Decision diagrams (BDDs) are the state-of-the-art data structure in Boolean formulas representation and manipulation. They have been successfully used in VLSI CAD and widely integrated in commercial tools. In this chapter we review the basic definitions of binary decision diagrams (BDDs) and study their theoretical and practical aspects. Extensions of BDDs are also investigated briefly. We give a detailed discussion about ZDDs in next chapter because they are specially important in this thesis. We will also outline several applications of BDDs and their extensions. Finally we suggest a number of articles and books for those who wish to pursue the topic in more depth. Content of this chapter is based on several articles, but mostly taken from (21).

3.1 Introduction

Binary decision diagrams (BDDs) as a data structure for representation of Boolean functions were first introduced by Lee (35) and further popularized by Akers and Moret (46). Bryant (8) introduced its restricted form OBDD (Reduced Ordered BDD), which is a canonical representation. He also proved that OBDDs allow efficient manipulations of Boolean formulas.

OBDDs have also been used successfully in other areas such as SAT-solving. This data structure and its variants can be implemented efficiently in modern

computers using a programming language such as C. CUDD (Colorado University Decision Diagram) Package, provided at the university of Colorado by Fabio Somenzi (51), is an open source package written in C. This package is known to be the most useful package for construction and manipulation of BDDs and their variants.

3.2 Ordered binary decision diagrams

3.2.1 Definition and examples of OBDDs

An OBDD is a graphic description of an algorithm for the computation of a Boolean function. The following definition describes the syntax of OBDDs, i.e., the properties of the underlying graph. The semantics of OBDDs, i.e., the functions represented by OBDDs, is specified in definition 8.

Definition 7 *An OBDD G representing the Boolean functions $f^1, \ldots, f^m$ over the variables $x_1, \ldots, x_n$ is a directed acyclic graph with the following properties:*

1. For each function f^i there is a pointer to a node in G.
2. The nodes without outgoing edges, which are called *sinks* or *terminal nodes*, are labeled by 0 or 1.
3. All non-sink nodes of G, which are also called internal nodes, are labeled by a variable and have two outgoing edges, a *0-edge* and a *1-edge*.
4. On each directed path in the OBDD each variable occurs at most once as the *label of a node*.
5. There is a variable ordering π, i.e., a permutation of $x_1, \ldots, x_n$, and on each directed path the variables occur according to this ordering. This means, if x_i

is arranged before x_j in the variable ordering, then it must not happen that on some path there is a node labeled by x_j before a node labeled by x_i.

In the figures we draw sink nodes as squares and internal nodes as circles. We always assume that edges are directed downwards. 0-edges are drawn as dashed lines while 1-edges are drawn as solid lines. Figure 3.1 shows an OBDD G_f with the variable ordering x_1, x_3, x_2 and an OBDD G_g with the variable ordering x_1, y_1, x_0, y_0.

Definition 8 *Let G be an OBDD for the functions $f^1, \ldots, f^m$ over the variables $x_1, \ldots, x_n$, and let $a = (a_1, \ldots, a_n)$ be an input. The* computation path *for the node v of G and the input a is the path starting at v which is obtained by choosing at each internal node labeled by x_i the outgoing a_i-edge.*

Each node v represents a function f_v, where $f_v(a)$ is defined as the value of the sink at the end of the computation path starting at v for the input a. Finally, f^j is defined as the function represented at the head of the pointer for f^j. Definition 8 can be seen as the description of an algorithm to obtain for each function f^j and each input a the computation path and therefore the value of $f^j(a)$.

In the OBDD G_f in Figure 3.1, the computation path for the input $(x_1, x_2, x_3) = (1, 1, 0)$ passes from x_1 in the root, then from the right x_3, then from the right x_2 and finally goes to the 0-Sink. Furthermore, for each node v of G_f the function f_v represented at v is given. By definition 8, it is easy to verify that the OBDD G_f in Figure 3.1, represents the function $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$ and the OBDD G_g represents the function $g(x_1, y_1, x_0, y_0) = (s_2, s_1, s_0)$, where (s_2, s_1, s_0) is the sum of the two 2-bit numbers (y_1, y_0) and (x_1, x_0).

The function represented at the sink labeled by $c \in \{0, 1\}$ is the constant function c. Now let v be an internal node which is labeled by x_i. Let v_0 be the 0-successor of v, i.e., the node reached via the 0-edge leaving v, and let v_1 be the 1-successor of v. We consider the computation of f_v for some input. If in the input

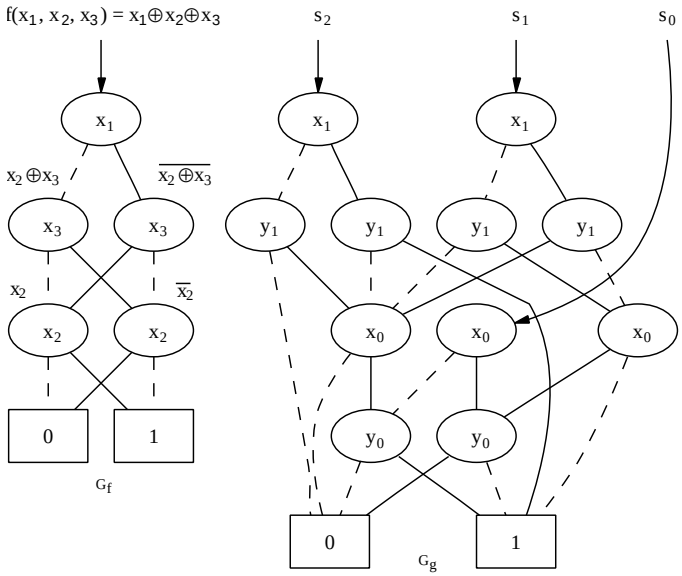

Figure 3.1: Examples of OBDDs.

the value of x_i is 0, then by definition 8 we may obtain f_v by evaluating f_{v_0} and, if the value of x_i is 1, by evaluating f_{v_1}. This can be expressed by the equation:

$$f_v = \bar{x}_i . f_{v_0} \vee x_i . f_{v_1} \tag{3.1}$$

Using equation 3.1 we may compute the functions represented at the nodes of an OBDD in a bottom-up fashion. However, the opposite is also true. If a node v labeled by x_i represents the function f_v, then the 0-successor of v represents the subfunction (sometimes called cofactor) $f_{v|x_i=0}$ and the 1-successor the subfunction $f_{v|x_i=1}$. In other words, at v the function f_v is decomposed using *Shannons decomposition rule*:

$$f_v = \bar{x}_i . f_{v|x_i=0} \vee x_i . f_{v|x_i=1} \tag{3.2}$$

We point out that in section 3.3 we shall consider variants of OBDDs where Shannons decomposition rule is replaced by a different decomposition rule. Equation 3.2 shows that we can decompose the function f_v in different ways by choosing

different variables x_i for decomposition. Hence, we may get different OBDDs for the same function if we use different variable orderings. Later on, we shall see that the size of an OBDDs usually depends strongly on the chosen variable ordering.

3.2.2 The canonicity of reduced OBDDs

A representation method is canonical if gives a unique representation for a Boolean function. We can compute the reduced OBDD G representing $f^1, \ldots, f^m$ and the variable ordering π from some OBDD for $f^1, \ldots, f^m$ by applying two reduction rules called S-deletion rule and the merging rule. The result G is called reduced OBDD or OBDD It can be shown that OBDD representation is unique and of minimum size. An obvious advantage of canonical representations is that two functions are identical if and only if their reduced OBDDs are identical.

The main idea of the reduction rules is to remove redundancies from the OBDD, namely, unnecessary tests of variables and tests that are represented more than once. Both reduction rules are sketched in Figure 3.2. The S-deletion rule can be applied to nodes v for which both outgoing edges lead to the same node w. It is obvious that we can redirect all edges leading to v to the node w and that we can delete v afterwards without changing the function represented by the OBDD. We use the term S-deletion rule (for Shannon deletion rule) instead of simply deletion rule in order to distinguish this rule from a different deletion rule for the variants of OBDDs which are not based on Shannons decomposition rule. The merging rule is applicable if there are nodes v and w with the same label, the same 0-successor, and the same 1-successor. Again, it is obvious that we can redirect all OBDD edges leading to v to the node w and that we can delete v afterwards without changing the represented function. Furthermore, the same operation can be performed on sinks with the same label. Bryant (8) proved that these reduction rules suffice to obtain the canonical representation for each function and each variable ordering.

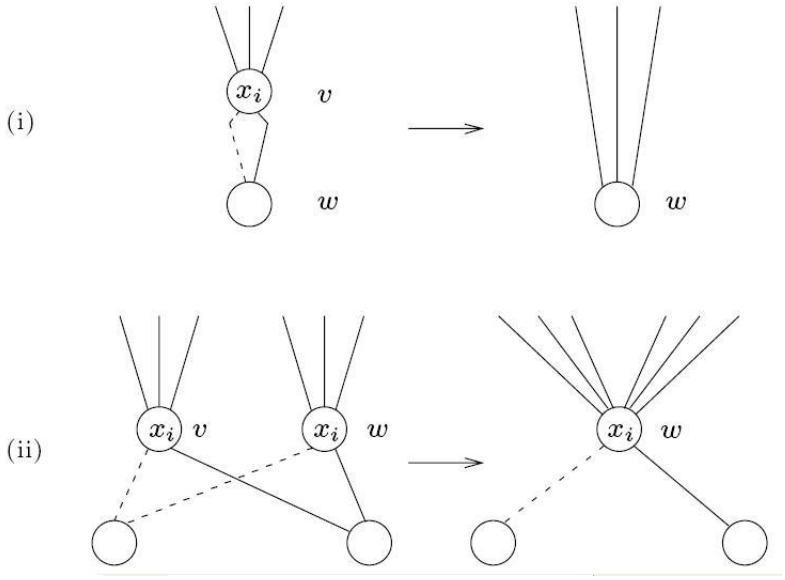

Figure 3.2: The (i) S-deletion rule, (ii) merging rule.

Theorem 1 *For all functions $f^1, \ldots, f^m$ and for each variable ordering π there is a unique OBDD G which can be obtained from each OBDD for $f^1, \ldots, f^m$ and π by applying the S-deletion rule and the merging rule until neither of the rules is applicable.*

In particular, the reduction of an OBDD yields the same result even if the reduction rules are applied in a different order. Theorem 1 also implies that the functions f^i and f^j represented in the same reduced OBDD are identical if and only if the pointers for f^i and f^j lead to the same node. An example about application of mentioned reduction rules is shown in Figure 3.3. Bryant (8) constructed an efficient algorithm for the reduction of OBDDs.

In order to make an efficient reduction algorithm it is essential to apply the reduction rules level-wise bottom-up. In fact, a reduction rule is applicable to a node v if we have done possible reductions to its successors. If during the bottom-up traversal on some node neither of the reduction rule is applicable, it cannot happen

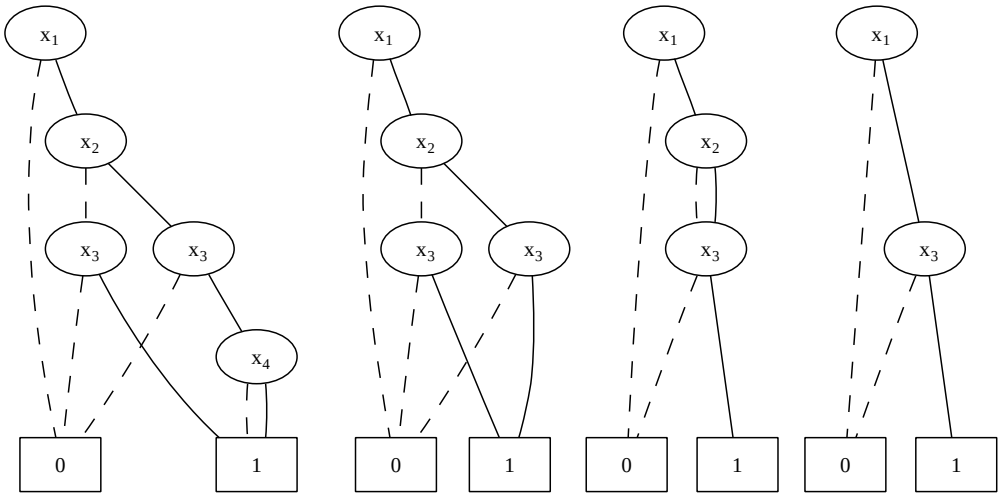

Figure 3.3: An example about reduction of an OBDD.

that later on some reduction rule is applicable. The algorithm of Bryant has on input G, a run time of $O(|G|.log|G|)$ where $|G|$ denotes the number of nodes of G. As can be seen in Figure 3.3, by S-deletion rule only nodes v labeled by x_i are removed where represent functions not essentially depending on x_i. Furthermore, it is easy to see that v and its successor w represent the same function. In addition, for the merging rule it is obvious that only nodes representing the same function are merged. Thus, the reduction rules make sure that for each function there is only one node representing that function.

3.2.3 Algorithms on OBDDs

Many operations can be performed efficiently on Boolean functions presented by OBDD. The most important of them are:

Evaluation: For an OBDD representing f and input a compute the value $f(a)$.

Reduction: For an OBDD compute the equivalent reduced OBDD.

Equivalence test: Test whether two functions represented by OBDDs are equal.

Satisfiability: Including:

- SAT-Test: For an OBDD G representing f find an input a for which $f(a) = 1$ or output that no such input exists.
- SAT-Count: For an OBDD G representing f compute the number of inputs a for which $f(a) = 1$.

Synthesis: For functions f and g represented by an OBDD G, include into G a representation for $f \otimes g$ where $\otimes$ is a binary Boolean operation (e.g., $\wedge$).

Replacements: There are two replacement operations:

- Replacement by constants: For a function f represented by an OBDD, for a variable x_i and a constant $c \in \{0, 1\}$ compute the OBDD for $f|_{x_i=c}$.
- Replacement by functions: For functions f and g represented by an OBDD and for a variable x_i compute the OBDD for $f|_{x_i=g}$.

Quantification: For a function f represented by an OBDD and for a variable x_i compute an OBDD for $(\forall x_i\, f) = f|_{x_i=0} \wedge f|_{x_i=1}$ or $(\exists x_i\, f) = f|_{x_i=0} \vee f|_{x_i=1}$.

We have already described the evaluation and reduction operations. In OBDD packages reduction is usually integrated into other operations such as synthesis. In this way only reduced OBDDs would be constructed.

Many applications consider functions that are given as circuits. Hence, an important operation is computation of an OBDD for such functions. In these cases, first we construct OBDDs for each variable appearing in the function. This is easy to do, because the OBDD for a variable x_i consists of a node labeled by x_i with the 0-sink as its 0-successor and the 1-sink as its 1-successor. Afterwards we run

through the circuit in some topological order (each gate is considered after all its predecessors have been considered) and compute for each gate a representation of the function at its output by combining the OBDDs representing the functions at its input by the synthesis operation. Another possibility is to construct OBDDs for some large blocks of the given circuit, then build the final OBDD from the OBDDs obtained for the blocks.

Equivalence

BDD-packages usually represent many functions in a single OBDD. Therefore the equivalence of Boolean functions f and g can be checked easily. In fact in order to check whether $f = g$, it suffices to check whether the pointers for f and g lead to the same node.

Satisfiability

If we just want to know whether f is satisfiable, it suffices to check whether f is pointing to the 0-sink or not. This can be done by just one pointer comparison. In order to find an input a for which $f(a) = 1$, we start at the pointer for f and search for a path to the 1-sink. For this, a simple *depth-first* search approach is sufficient. If there is no such path, we conclude that $f = 0$. If the OBDD is reduced, the run time of this algorithm is $O(n)$.

Synthesis

Synthesis is probably the most important operation since it is needed in almost all applications. The usual way of generating new BDDs is to combine existing BDDs with connectives like AND, OR, EX-OR. If we want to make an OBDD for a given Boolean function, first we make OBDDs for each variable of the Boolean function,

and then we parse the Boolean function and combine the existing OBDDs to make OBDDs for the needed sub functions and finally the OBDD representing the whole given Boolean function.

As suggested by Brace, Rudell, and Bryant (6), in OBDD packages usually the synthesis algorithm is called *ITE* ("if-then-else") where:

$$ite(f, g, h) = f.g \vee \bar{f}.h$$

The *ITE()* procedure receives OBDDs for two Boolean functions f and g, builds the OBDD for $f < op > g$. In fact it receives three arguments: *I,T,E* which are OBDDs and returns the OBDD representing: $(I \wedge T) \vee (I \wedge E)$. All binary Boolean operations can be simulated by the ite-operator, e.g.: $f \vee g = ite(f, 1, g)$, $f \wedge g = ite(f, g, 0)$ or $f \oplus g = ite(f, g, g)$.

ITE() is a combination of depth-first traversal and dynamic programming. (A recursive, Bottom-up procedure with tabulation). The basic idea of *ITE()* comes from the expansion theorem:

$$F < op > G = v(F_v < op > G_v) + v'(F_{v'} < op > G_{v'})$$

ITE() maintains a table called *Computed Table* to avoid computing the same combination repeatedly. It also maintains another table called *Unique Table* to avoid producing subgraphs representing the same sub-function. The benefit of this technique is the important result that *ITE()* becomes polynomial rather than exponential. Figure 3.4 displays the pseudocode for the *ITE* operator.

If f and g are given by OBDDs with different variable orderings, the *ITE()* procedure would not work since for the simultaneous traversal, the variables have to be encountered in the same ordering in both OBDDs. In this situation the synthesis problem is much harder.

In order to construct an OBDD for $f|_{x_i} = c$ from the OBDD for f, we redirect all edges leading to node labeled by x_i to the c-successor of x_i. The obtained

```
ITE(f, g, h)
 {
  if(f == 1) return g;
  if(f == 0) return h;
  if(g == h) return g;
  if((p = HASH_LOOKUP_COMPUTED_TABLE(f,g,h)) return p;
  v  = TOP_VARIABLE(f, g, h );  // top variable from f,g,h
  fn = ITE(fv0,gv0,hv0);        // recursive calls
  gn = ITE(fv1,gv1,hv1);
  if(fn == gn) return gn;       // reduction
  if(!(p = HASH_LOOKUP_UNIQUE_TABLE(v,fn,gn))
     p = CREATE_NODE(v,fn,gn);  // and insert into UNIQUE_TABLE
  INSERT_COMPUTED_TABLE(p,HASH_KEY{f,g,h});
  return p;
 }
```

Figure 3.4: The ITE algorithm for ROBDDs.

OBDD would not be necessarily reduced. Therefore we may apply a reduction algorithm to obtain the final result. If we consider an OBDD representing several functions this algorithm destroys the representation of other functions. Thus, we may need to create a copy of the representation for f before performing above operations. Replacement by a function can be solved by combining the above idea with the synthesis algorithm to find: $f|_{x_i=g} = ite(g, f|_{x_i=1}, f|_{x_i=0})$. Similarly it follows from the definition of the quantification operations that they can be implemented by combining replacement by constants and synthesis.

3.2.4 The variable ordering problem for OBDDs

OBDDs share a fatal property with all kinds of representations of switching functions: the representation of almost all functions needs exponential space. Bryant (8) discovered that OBDD size strongly depends on the chosen variable ordering. Figure 3.5, shows the effect of variable ordering for a switching function. Notice that

both OBDDs represent the same Boolean function: $F = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$.

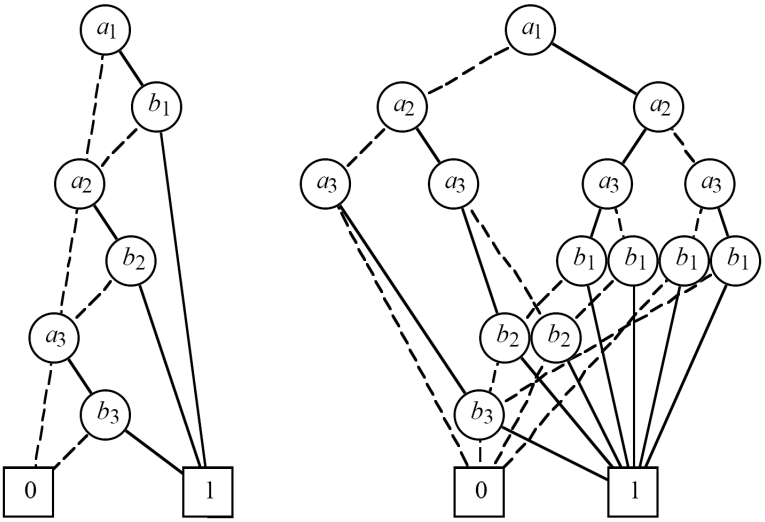

Figure 3.5: Effect of variable ordering on OBDD size (Bryant 1986).

Different functions have different ordering sensitivities. Some functions have a high and others have a low variable order sensitivity. The practicability of OBDDs strongly depends on the existence of suitable algorithms and tools for minimizing the graphs in the relevant applications. There are many improvements, optimization algorithms, and additions to the basic OBDD model. It is known by experience that:

- Many tasks have reasonable OBDD representations.
- Algorithms remain practical for up to 100,000 OBDDs nodes.
- most proposed heuristic ordering methods are generally satisfactory.

However, because of the practical applicability of this data structure, investigation and development of new optimization techniques for OBDDs is still a rewarding research topic.

3.3 Generalizations of OBDDs

There are many functions which their OBDD representation are too large to be stored in a computer memory. For example, Bryant (8; 9) proved that OBDDs for multiplication of two n-bit numbers have at least $2^{n/8}$ nodes. In other words OBDDs for multipliers are exponential size for all variable orderings. This is why quite a large number of extensions of the basic OBDD have been suggested. Each of these extensions allow a more compact representation than OBDD representation. On the other hand, the extensions have the disadvantage that for some of the basic operations listed in section 3.2 only less efficient algorithms are known. In the following list we give a rough classification of the extensions:

1. Extensions obtained by replacing Shannon's decomposition rule by a different rule. Examples for such extensions are *ordered functional decision diagrams* (OFDDs), *ordered Kronecker functional decision diagrams* (OKFDDs) and *parity OBDDs* (39).

2. Extensions obtained by relaxing the variable ordering; (condition 5 of definition 7). Free BDDs are examples for such extension.

3. Extensions obtained by relaxing the read-once; (condition 4 in definition 7). Examples are *kOBDDs and IBDDs* (see Meinel's book "Algorithms and data structures in VLSI design: OBDD - foundations and applications" (43).)

4. Extensions obtained by introducing a transformation τ and representing a function g such that $f = g \circ \tau$. This variant is called *transformed BDD* (TBDD). Transformed BDDs have been suggested by Bern, Meinel, and Slobodova (38; 41). The results of Bern, Meinel, and Slobodova (38) imply that the synthesis algorithm for OBDDs can be applied for OBDDs with linear tests as well, and that OBDDs with linear transformations are canonical representations.

5. Extensions that represent functions $f : \{0,1\}^n \to Z$ instead of Boolean functions. In order to represent such functions, we may extend OBDDs by allowing sinks labeled by numbers from Z. Examples of this variant are *multi-terminal BDDs* (MTBDDs), which are also called *algebraic decision diagrams* (ADDs).

We remark that another extension of OBDDs, namely Zero-suppressed BDDs (ZBDDs or ZDDs), are considered in detail in next chapter because they play special role in this thesis.

3.4 Applications

In this section, we outline some applications of BDDs and briefly mention fields where the extensions introduced above have been used. The list is not complete in the sense that "all" applications are covered and not all applications have equal importance. Here only the main ideas of the applications are proposed, while the given references present more details.

Verification: Nowadays modern circuit design can contain several million gates. Verification of such a large designs is becoming more and more difficult. Pure simulation cannot guarantee correct behavior and exhaustive simulation is too time consuming. Verification (in its "simplest" form) consists of checking equivalence of two functions representing two circuits. If an OBDD can be constructed for the specified circuit and for the designed circuit then verification would be very easy. After OBDDs were introduced in the mid 1980s they were used in such tools by Bryant et al. (10). OBDDs are nowadays the state-of-the-art data structure in verification and have been integrated in many commercial tools.

In the field of sequential circuits OBDDs have also shown to work very well. Using OBDDs, reachability analysis and image computation for finite state machines with more than several million states could be verified. These techniques

have been applied in the field of symbolic model checking.

Logic synthesis: The classic 2-level minimizers, such as ESPRESSO (7), use enumeration of the terms in an array structure for representing the prime implicants. The construction of circuits resulting from a direct mapping of OBDDs has gained great interest, since the graph directly corresponds to a multiplexor network. One argument is that the mapping process can be simplified due to the simple structure. Additionally, the resulting circuits have good testability properties. In this area, more general types than OBDDs, such as OKFDDs, have several advantages, since in the area of circuit design a small reduction in the graph size may also have a large influence on the size of the resulting circuit.

Testing: One possibility of using DDs in the area of testing has just been mentioned, i.e., in the form of design for testability. Alternatively, the symbolic representation of a circuit by OBDDs can be used for many problems in testing, such as test pattern generation and fault simulation By using OBDDs, it was possible for the first time to compute exact signal probabilities and fault detection probabilities These probabilities are used in different areas in testing, such as weight optimization in built-in self-testing.

Combinatorial optimization: DDs have been used in many other combinatorial optimization problems. For example, in many tasks during logic synthesis, sets of elements have to be represented. ZBDDs have been introduced by Minato (29) especially for this problem. Even though from the theoretical point of view they only differ by a linear factor from OBDDs with respect to size, they are frequently used in applications such as 2-level minimization.

Binary decision diagrams in complexity theory: In complexity theory, BDDs are called branching programs and are used as a model for describing the space complexity of computations. First BDDs without the read-once property and without variable orderings were considered. It can be proved that polynomial size BDDs

represent exactly those functions that can be computed by any reasonable model of sequential computation with a logarithmic amount of memory. Thus the problem to prove super-logarithmic space bounds for explicitly defined functions reduces to proving super-polynomial size bounds for BDDs. However, obtaining such a proof is considered to be as difficult as obtaining a solution of the $P = NP$? question. After recognizing the importance of BDD-like representations as a data structure for Boolean functions, researchers in complexity theory also investigated the complexity of operations on BDDs. The most important classification of problems in complexity theory is the distinction between problems solvable in polynomial time and NP-complete problems The advantage of this classification is that it is independent of the model of computation. However, in some applications exponential time algorithms may be useful for small instances, while in other applications even polynomial time algorithms may be too slow. Nevertheless, the classification helps to determine whether it makes sense to search for an efficient algorithm for solving the considered problem or whether we have to be satisfied with approximate solutions or heuristic solutions.

We conclude conclude this chapter with a list of the books that have been published on OBDDs and related topics.The interested reader can find some more information about OBDDs and their generalizations in these books. The books in alphabetic order of the first authors name are: Drechsler and Becker (20), Hachtel and Somenzi (28), Meinel and Theobald (43), Minato (31), and Wegener (52).

CHAPTER 4

Zero-Suppressed Binary Decision Diagrams

4.1 Introduction

Zero-suppressed Binary Decision Diagrams (ZDDs) (29) provide an efficient way of solving problems expressed in terms of set theory. Content of this background chapter is mostly taken from a tutorial paper presented by Alan Mishchenko (44).

Several years ago, BDD (8) and its variations entered the scene of computer science. Since that time they have been used in research software and industrial CAD tools. They are used as a memory-efficient storage and convenient processing media for Boolean functions and as a representation facilitating the analysis of data leading to new implicit algorithms, which tend to be more efficient.

One of the reasons why decision diagrams, and in particular BDDs, became useful for the CAD tool developers, is that they provide canonical representation of discrete objects. Canonical means that under certain condition for every object there is only one representation of this kind. This property is extremely important for verification because in order to prove the identity of two objects it is sufficient to build their canonical representations and show that these representations are identical. The experience of using BDDs in numerous applications shows that they are not a panacea for all types of problems. In some cases, due to the specific properties of the discrete data arising in particular settings, the BDDs grow large making processing inefficient or impossible. In particular, this situation occurs when the applications work with sparse sets represented by characteristic functions.

A set is sparse if number of elements in it is much smaller than the total number of elements that may appear in the set. Cube covers are an example of sparse sets. A typical cube contains only a few literals out of all possible literals that may appear in the cube. The maximum number of literals is reached when a cube is a minterm containing each variable as either the negative or the positive literal. In the case, the sparseness of the set is $\frac{1}{2}$, because each minterm contains exactly one half of all possible literals that may appear in the cubes. The problem of prohibitively large BDD size of the sparse set representation can be remedied by introducing a different brand of decision diagrams, called Zero-suppressed binary Decision Diagrams (ZDDs) (29). These diagrams are similar to BDDs with one of the underlying reduction principles modified.

While BDDs are better for the representation of functions, ZDDs are better for the representation of covers. Additionally, there are efficient procedures to perform conversions between them. Taken together, BDDs and ZDDs provide a powerful framework to solve problems in logic synthesis, such as two-level sum-of-product (SOP) minimization, three-level minimization, factorization, and decomposition. The use of ZDDs is not limited to logic synthesis. They have been used, independently of BDDs, in a number of applications, ranging from the graph-theory problems to handling polynomials and regular expressions.

This section is designed to be an introduction to ZDDs for a reader with background in Boolean algebra and an understanding of basic principles of BDDs. The goal is to present the applications of ZDDs for sets and cube covers. We first discuss the basic principles and uses of ZDDs. In particular, Section 4.2 focuses on the main differences between BDDs and ZDDs when it comes to representing Boolean functions, sets, and cube covers. In Section 4.3, we classify and discuss the elementary ZDD operators provided by the DD (Decision Diagram) packages.

The following terminology is accepted in the BDD research (8). The BDD represents the function as a rooted directed acyclic graph. Each non-terminal node N is labeled by a variable v and has edges directed towards two successor (children) nodes, else(N) and then(N), representing the cofactors of N w.r.t. v. Each terminal node is labeled with 0 or 1. For a given assignment of the variables, the value of the function is found by tracing a path from the root to a terminal vertex following the branches indicated by the values assigned to the variables. The function value is given by the terminal vertex label. For example, Figure 4.1(left) shows the BDD of the Boolean function $F = ab + cd$. The edges are directed downwards. The dashed edges (solid) edges correspond to $v = 0(v = 1)$. In the following, we use the terms "procedure", "functions", "routine" and "operator" interchangeably to denote a fragment of functionality implemented with DDs.

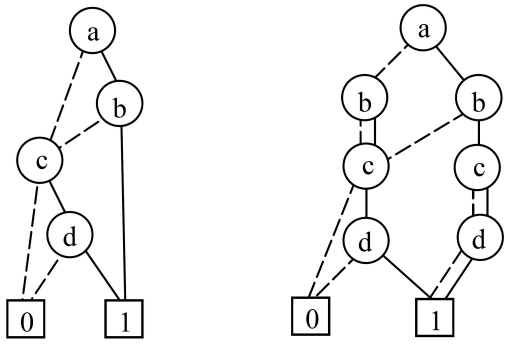

Figure 4.1: BDD and ZDD for $F = ab + cd$.

4.2 Comparing BDDs and ZDDs

Both BDDs and ZDDs can be seen as decision trees, simplified using two reduction rules that guarantee the canonicity of the resulting representation. The second reduction rule (merging of isomorphic subgraphs) holds for both BDDs and ZDDs; however, they differ in the first reduction rule (node elimination).

For BDDs, the node is removed from the decision tree if both its edges point to the same node. For ZDDs, the node is removed if its positive edge (then-edge) points to the terminal node 0. This variation in the rule, as mentioned before, explains the improved efficiency of ZDDs when handling sparse sets and the semantic differences between the two types of diagrams. One way of understanding the principles of ZDDs is to compare them with BDDs for simple illustrative functions while keeping in mind their main difference.

4.2.1 Boolean functions

It can be shown that, in a BDD, all paths from the root to terminal node 1 can be seen as cubes constituting a disjoint cover of the function. A variable is present in the *positive (negative) polarity* in the corresponding cube if the path contains the 1-edge (0-edge) of a node labeled by this variable; the variable is *absent* in the cube if the path does not go through a node labeled by this variable.

In a ZDD for the same function, all paths from the root to terminal 1 also represent a disjoint cover of the function. (This cover is the same if the variable ordering is the same in both diagrams.) A variable is present in the *positive polarity* in the corresponding cube if the path goes through the 1-edge of a node labeled with this variable. A variable is present in the *negative polarity* in the cube if the path goes through the 0-edge *or* if the path does not go through a node labeled by this variable. A variable is *absent* in a cube, if the path goes through a node labeled by this variable and both edges of the node point to the same node.

Consider a BDD and a ZDD of the function $F = ab+cd$ shown in Figure 4.1. Both diagrams can be used to trace the disjoint cover of the function: $\{ab, \bar{a}cd, a\bar{b}cd\}$. As can be seen from Figure 4.1, the size of the ZDD (the number of nodes in the diagram) is almost two times larger than that of the BDD. This is because ZDDs are not as efficient as BDDs when they represent typical Boolean functions.

4.2.2 Sets of subsets

The classical BDDs represent completely specified Boolean functions. In order to represent sets (and sets of subsets), the characteristic functions have been introduced (13) in such a way that each set is put in one-to-one correspondence with its characteristic function.

Informally, the characteristic function of a set of subsets is a CSF that depends on as many input variables as there are elements that can potentially appear in a subset. For each subset in the set, one minterm is added to the on-set of the characteristic function. In this minterm, variables appear in positive (negative) polarities if they are present (absent) in the set. For example, given three elements (a, b, c), consider the set of subsets $\{\{a, b\}, \{a, c\}, \{c\}\}$. If we associate each element with a binary variable having the same name, the characteristic function of the set of subsets would be $F = ab\bar{c} + a\bar{b}c + \bar{a}\bar{b}c$. The first minterm corresponds to the subset $\{a, b\}$, and so on.

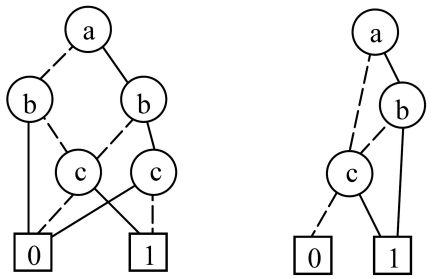

Figure 4.2: The BDD and the ZDD for the set of subsets $\{\{a, b\}, \{a, c\}, \{c\}\}$.

Important for our discussion are the following observations: The empty subset is represented by the minterm $F = \bar{a}\bar{b}\bar{c}$, while the subset containing all elements is represented by $F = abc$. The empty set is represented by the characteristic function $F = 0$, while the set composed of all possible subsets is represented by the characteristic function $F = 1$. The latter is obvious if we observe that the

constant-0 function has no on-set minterms, while the constant-1 function has 2^n on-set minterms, corresponding to the complete Boolean space.

Notice that there is a difference between the empty set and the set of subsets composed of the empty set only. The former has the characteristic function equal to constant 0, while the latter has the characteristic function $F = \bar{a}\bar{b}\bar{c}$. Once the characteristic function is constructed, it can be represented using a BDD or a ZDD. The two representations of the set of subsets $\{\{a, b\}, \{a, c\}, \{c\}\}$ are given in Figure 4.2. In both diagrams, there are three paths from the root node (on top) to the terminal node 1, which correspond to the subsets $\{a, b\}$, $\{a, c\}$, and $\{c\}$. The encoding of the variables in the paths is discussed in section 4.2.1.

Notice that the size of the ZDD in Figure 4.2 is smaller than that of the BDD. It can be proved that the upper bound on the size of the ZDD is the total number of elements appearing in all subsets of a set. Meanwhile, the upper bound on the size of the BDD is given by the number of subsets multiplied by the number of all elements that can appear in them. This observation shows that ZDDs should be much more compact when representing sets of subsets. The above theoretical upper bound on the ZDD size is rarely reached; in practice ZDDs tend to be even more compact.

4.2.3 Cube covers

Let us now consider the ZDD representation of cube covers. First, it is necessary to introduce additional variables, because BDDs and ZDDs depending on the primary input variables represent only one type of *disjoint* covers. To represent arbitrary covers, two variables are used for each primary input: one of them stands for the positive literal and another for the negative literal. These variables are always kept adjacent in the variable order. Similarly to the set of subsets, a cube cover is represented by its characteristic function introduced as follows:

- The characteristic function depends on the $2n$ variables, where n is the number of primary inputs.
- For each cube of the cover, one minterms is added to the on-set of the characteristic function.
- The minterm has those variables in the positive polarity that correspond to literals present in the cube and those variables in the negative polarity that correspond to literals missing in the cube.

For example, to represent arbitrary covers of the four variable function $F = ab + cd$, eight variables are used: $(a_1, a_0, b_1, b_0, c_1, c_0, d_1, d_0)$. These variables correspond to the positive and negative literals of each input variable. The characteristic functions of the cover $\{ab, cd\}$ is: $X = a_1\bar{a}_0b_1\bar{b}_0\bar{c}_1\bar{c}_0\bar{d}_1\bar{d}_0 + \bar{a}_1\bar{a}_0\bar{b}_1\bar{b}_0c_1\bar{c}_0d_1\bar{d}_0$

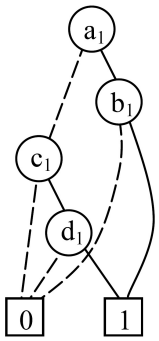

Figure 4.3: ZDD for the cover $\{ab, cd\}$.

Figure 4.3 displays the ZDD for the characteristic function. Unlike the BDD for function X, which depends on all eight variables, the ZDD depends on four variables only. These are the variables that appear in the characteristic function in the positive polarity and correspond to literals actually present in the cover. All other variables are missing in the ZDD, because according to the ZDD reduction rules a variable missing on the path is interpreted as a variable taking the value 0

in the minterm of the characteristic function. This property makes ZDD ideal for representing and manipulating large cube covers.

The terminal node 0 in a ZDD representing covers stands for the empty cover. In this case, there are no assignments for which the characteristic function evaluates to 1 and therefore there are no cubes in the cover. The terminal node 1 stands for the cover containing only the tautology cube, that is the cube in which all the literals are missing. Indeed, there is only one path for which the characteristic function evaluates to 1, and this path does not go through any nodes. According to the ZDD reduction rules, it means that all the variables on the path are equal to zero, which in turn means that all the literals are missing in the cube.

The decomposition of a cover w.r.t. the primary input variable is a triple of covers that contain cubes: (1) with the variable as the positive literal, (2) with the variable as the negative literal, (3) without the variable. The inverse operation is the composition of the three covers into one cover. The composition is performed using a primary input variable that is not currently used in the covers.

For example, consider the cover $C = ab\bar{c} + a\bar{b}d + ac + d$. Decomposing C w.r.t. the primary input variable b yields: $C_0 = ad$, $C_1 = a\bar{c}$, $C_2 = ac + d$. The reverse operation, the composition of the covers C_0, C_1, and C_2 w.r.t. b, which does not appear in them, produces the initial cover C. If the cover is represented by its ZDD, the decomposition and composition operations are performed by functions DecomposeCover() and ComposeCover() of the ZDD package. DecomposeCover() takes the cover and the primary input variable and returns three subcovers. ComposeCover() takes three subcovers and the primary input variable and returns the composed cover. For a detailed analysis of ZDD in the representation of cube covers, it is recommended for the reader to review references (30; 32) where some basic ZDD-based recursive operators are introduced and explained.

4.3 Basic ZDD procedures

The basic functions dealing with ZDDs can be classified as follows:

4.3.1 Procedures working with functions

These procedures are similar to those developed to manipulate Boolean functions using BDDs.

1. Procedures returning elementary functions:

   - Constant-0 ZDD (constant zero function, $F = 0$)
   - Universal ZDD (constant one function, $F = 1$)
   - Single-variable ZDD (the function for the elementary variable, $F = v$)

2. Procedures performing boolean operations:

   - If-Then-Else (ITE) operator. This function returns the result of applying ITE to A, B, and C: $ITE(A, B, C) = AB + \bar{A}C$.

Notice that the complement of a ZDD cannot be computed by complementing a pointer, as it is done using BDDs with complement edges. Instead, we should apply the ITE operator to the function and constants and compute the complement as follows: $F = ITE(F, 0, 1)$.

4.3.2 Procedures working with sets

1. Procedures returning elementary sets:

   - Constant-0 ZDD (the empty set, {})

- Constant-1 ZDD (the set of subsets consisting of the empty set, $\{\{\}\}$)
- Single-variable ZDD (the set of subsets with a subset containing element v, $\{\{v\}\}$)

2. Procedures performing operations on the set of subsets w.r.t. to a single element (variable):

   - $Subset0(S, v)$ returns the set of subsets of S not containing element v.
   - $Subset1(S, v)$ returns the set of subsets of S containing element v.
   - $Change(S, v)$ returns the set of subsets derived from S by adding element v to those subsets that did not contain it and removing element v from those subsets that contain it.

3. Procedures performing standard set operations for two sets of subsets:

   - $Union(X, Y)$ returns the set of subsets belonging to X or Y.
   - $Intersection(X, Y)$ returns the set of subsets belonging to X and Y.
   - $Difference(X, Y)$ returns the set of subsets of X not belonging to Y.

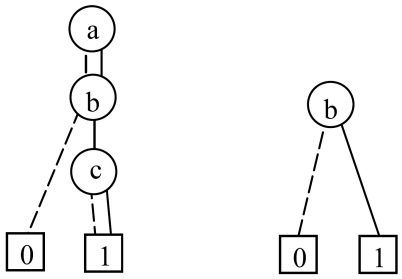

Figure 4.4: ZDDs for variable 'b', (right) as a function, (left) in set manipulation.

It is important to distinguish single-variable ZDDs as used in the manipulation of functions and from those used in the manipulation of sets. Figure 4.4 shows the ZDDs for $F = b$, assuming that there are three variables (a, b, c). In the case of functions, this ZDD represents the function $F = b$. In the case of sets, the ZDD represents the characteristic function of the set containing a single element b. In this case, the characteristic function is $F = \bar{a}b\bar{c}$.

In addition to above elementary operators defined for sets, the two pairs of set product and weak division operators have been implemented using ZDDs. These two pairs of operators correspond to unate and binate algebras (30; 32). Speaking informally, in unate algebra, every literal of a cube is either positive polarity or missing, while in binate algebra every literal may be non-complemented, complemented, or missing. Correspondingly, to manipulate sets in unate algebra every literal is encoded with one ZDD variable, while to manipulate sets in binate algebra, two literals are used, one represents the variable in positive polarity, the other presents it in negative polarity. In terms of the definitions introduced above, ZDDs used to manipulate sets of subsets implement the unate algebra, while ZDD used to manipulate the cube covers implement the binate algebra. In terms of procedures implemented in the main distribution of the CUDD package, unate product and division are *Cudd_zddUnateProduct()* and *Cudd_zddDivide()*, while binate product and division are *Cudd_zddProduct()* and *CUDD_zddWeakDiv()*.

4.4 The CUDD Package

In this section we introduce the CUDD package, which is known to be the best open source package for manipulating BDDs and their variants. Content of this subsection is prepared very briefly and mostly taken from its help manual (51).

The CUDD package provides functions to manipulate Binary Decision Diagrams (BDDs) (43), Algebraic Decision Diagrams (ADDs), and Zero-suppressed

Binary Decision Diagrams (ZDDs) (29). BDDs are used to represent switching functions; ADDs are used to represent function from $\{0,1\}^n$ to an arbitrary set. ZDDs represent switching functions like BDDs; however, they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-Set of the function to be represented is very sparse. They are inferior to BDDs in other cases.

The CUDD package can be used in three ways:

- As a black box. In this case, the application program that needs to manipulate decision diagrams only uses the exported functions of the package. The rich set of functions included in the CUDD package allows many applications to be written in this way. An application written in terms of the exported functions of the package needs not concern itself with the details of variable reordering, which may take place behind the scenes.

- As a clear box. When writing a sophisticated application based on decision diagrams, efficiency often dictates that some functions be implemented as direct recursive manipulation of the diagrams, instead of being written in terms of existing primitive functions.

- Through an interface. Object oriented languages like C++ can free the programmer from the burden of memory management. A C++ interface is included in the distribution of CUDD. It automatically frees decision diagrams that are no longer used by the application. Almost all the functionality provided by the CUDD exported functions is available through the C++ interface, which is especially recommended for fast prototyping.

How to Get CUDD: The package is available via vlsi.Colorado.EDU. A compressed file named cudd2.3.1.tar.gz can be found there. Once you have this file,

```
gzip dc cudd2.3.1.tar.gz | tar xvf -
```

will create directory cudd2.3.1 and its subdirectories. These directories contain the decision diagram package, a few support libraries, and a toy application based on the decision diagram package.

Compiling and Linking: To build an application that uses the CUDD package, you should add the following files:

```
#include "util.h"
#include "cudd.h"
```

to your source, and should link: `libcudd.a, libmtr.a, libst.a, and libutil.a` to your executable. (All these libraries are part of the distribution.) :

Nodes: BDDs and ZDDs are made of DdNode's. A DdNode (node for short) is a structure with several fields. Those that are of interest to the application that uses the CUDD package as a black box are the variable index, the reference count, and the value. The remaining fields are pointers that connect nodes among themselves and that are used to implement the unique table.

The index field holds the name of the variable that labels the node. The index of a variable is a permanent attribute that reflects the order of creation. Index 0 corresponds to the variable created first. On a machine with 32bit pointers, the maximum number of variables is the largest value that can be stored in an unsigned short integer minus 1. The largest indices are reserved for the constant nodes. When 64bit pointers are used, the maximum number of variables is the largest value that can be stored in an unsigned integer minus 1. When variables are reordered to reduce the size of the diagrams, the variables may shift in the order, but they retain their indices. The package keeps track of the variable permutation (and its inverse).

The CUDD package relies on garbage collection to reclaim the memory used by diagrams that are no longer in use. The scheme employed for garbage collection is based on keeping a reference count for each node. The references that are

counted are both the internal references (references from other nodes) and external references (typically references from the calling environment). When an application creates a new BDD, ADD, or ZDD, it must increase its reference count explicitly, through a call to $Cudd\_Ref$. Similarly, when a diagram is no longer needed, the application must call $Cudd\_RecursiveDeref$ for BDDs or $Cudd\_RecursiveDerefZdd$ (for ZDDs) to "recycle" the nodes of the diagram. Terminal nodes carry a value. By default, the value is a double.

The Manager: All nodes used in BDDs and ZDDs are kept in special hash tables called the unique tables. As the name implies, the main purpose of the unique table is to guarantee that each node is unique; that is, there is no other node labeled by the same variable and with the same children. This uniqueness property makes decision diagrams canonical. The unique tables and some auxiliary data structures make up the DdManager (manager for short). Though the application that uses only the exported functions needs not be concerned with most details of the manager, it has to deal with the manager in the following sense. The application must initialize the manager by calling an appropriate function. Subsequently, it must pass a pointer to the manager to all the functions that operate on decision diagrams. With the exception of a few statistical counters, there are no global variables in CUDD.

Cache: Efficient recursive manipulation of decision diagrams requires the use of a table to store computed results. This table is called here the cache because it is effectively handled like a cache of variable but limited capacity. The CUDD package starts by default with a small cache, and increases its size until either no further benefit is achieved, or a limit size is reached. The user can influence this policy by choosing initial and limit values for the cache size. The default parameters work reasonably well for a large spectrum of applications. The cache of the CUDD package is used by most recursive functions of the package, and can be used by user supplied functions as well.

Initializing and Shutting Down a DdManager: To use the functions in the CUDD package, one has first to initialize the package itself by calling *Cudd_Init*. This function takes four parameters:

- numVars: It is the initial number of variables for BDDs (and ADDs). If the number is unknown, it can be set to 0, or to any other lower bound on the number of variables.
- numVarsZ: It is the initial number of variables for ZDDs.
- numSlots: Determines the initial size of each subtable of the unique table. There is a subtable for each variable. The size of each subtable is dynamically adjusted to reflect the number of nodes. It is normally O.K. to use the default value for this parameter, which is $CUDD\_UNIQUE\_SLOTS$.
- cacheSize: It is the initial size (number of entries) of the cache. Its default value is $CUDD\_CACHE\_SLOTS$.
- maxMemory: It is the target value for the maximum memory occupation (in bytes). The package uses this value to decide two parameters: 1. The maximum size to which the cache will grow, regardless of the hit rate or the size of the unique table, 2. The maximum size to which growth of the unique table will be preferred to garbage collection. If maxMemory is set to 0, CUDD tries to guess a good value based on the available memory.

A typical call to *Cudd_Init* may look like below.To reclaim all the memory associated with a manager, an application must call *Cudd_Quit* (normally before exiting).

```
manager = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
```

Constant Functions: The CUDD Package defines several constant functions. These functions are created when the manager is initialized, and are accessible through the manager itself.

The constant 1 (returned by *Cudd_ReadOne*) is common to BDDs and ZDDs. However, its meaning is different for BDDs, on one hand, and ZDDs, on the other hand. The diagram consisting of the constant 1 node only represents the constant 1 function for BDDs. For ZDDs, its meaning depends on the number of variables: It is the conjunction of the complements of all variables. Conversely, the representation of the constant 1 function depends on the number of variables. The constant 1 function of n variables is returned by *Cudd_ReadZddOne*. The constant 0 apply to ZDDs, but not to BDDs. The BDD logic 0 is not associated with the constant 0 function: It is obtained by complementation (*Cudd_Not*) of the constant 1. (It is also returned by *Cudd_ReadLogicZero*.)

Creating Variables: Decision diagrams are typically created by combining simpler decision diagrams. The simplest decision diagrams, of course, cannot be created in that way. The ZDD of the i-th projection function is returned by *Cudd_zddIthVar*. Simple variable functions are also known as projection functions. The projection functions of ZDDs have diagrams with $n + 1$ nodes, where n is the number of variables.

Basic ZDD Manipulation

It is possible to build ZDDs by applying Boolean operators to other ZDDs, starting from constants and projection functions. The following fragment of code illustrates how to build the ZDD for the function $f = x_0' + x_1' + x_2' + x_3'$. We assume that the four variables already exist in the manager when the ZDD for f is built. Note the use of DeMorgan's law.

CUDD provides functions for the manipulation of covers represented by ZDDs. For instance, *Cudd_zddWeakDiv* performs the weak division of two covers given as ZDDs. These functions expect the two ZDD variables corresponding to the two literals of the function variable to be adjacent.

```
DdManager *manager;           /* pointer to DD manager */
DdNode *f , *var, *tmp;       /* pointers to nodes */
int  i;
......
manager = Cudd_Init(0,4,CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS,0);
tmp = Cudd_ReadZddOne(manager, 0); CuddRef(tmp);
for (i=3 ; i>=0 ; i-- )
 {
  var = Cudd_zddIthVar( manager, i ); CuddRef(var);
  f = Cudd_zddIntersect(manager,var,tmp); Cudd_Ref(f);
  Cudd_RecursiveDerefZdd(manager,tmp);
  Cudd_RecursiveDerefZdd(manager,var);
  tmp = f;
 }
f = Cudd_zddDiff(manager,Cudd_ReadZddOne(manager,0),tmp);
Cudd_Ref(f);
Cudd_RecursiveDerefZdd(manager,tmp);
```

The above introduction to the CUDD package is succinct, provided for fast overview. We refer the interested researchers (specially who want to use it) to its manuals available on the web (51).

CHAPTER 5

The QSAT Problem and Existing Solutions

5.1 Introduction

QSAT that stands for "satisfiability of quantified Boolean formulas", is a general form of the SAT (SATisfiability of propositional formula) problem. In this problem we are given a closed *prenex-CNF* ϕ and the question is whether there is a way to assign Boolean values to the variables so that the formula evaluates to `true` .

As an example: $\forall x_1 \exists x_2((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$ means: "for each truth assignment to x_1 there exists an assignment to x_2 such that $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is true". The above QBF is satisfiable: if x_1=`true` then x_2 can be assigned to `false`; if x_1=`false` then x_2 can be assigned to `true`; in both cases $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is `true`.

In this chapter, first a more formal definition of the QBF problem is given, then a suggested input standard for representing QBF instances is described. In the last section we take a look to the best existing QBF solvers. The main strategies and heuristics used by each of them are also explained briefly.

5.2 The QSAT Problem

QBF in CNF-format or prenex-CNF: Consider a set P of propositional letters. An *atom* is an element of P. A *literal* is an atom or the negation of an atom. A

clause C is a *p-ary* ($p \geq 0$) disjunction of literals. A propositional formula is an *m-ary* ($m \geq 0$) conjunction of clauses. As customary, we represent a clause as a set of literals, and a propositional formula as a set of clauses. A QBF is an expression of the form:

$$Q_1 x_1 \dots Q_n x_n \Phi \quad (n \geq 0) \tag{5.1}$$

where every $Q_i (1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$; The literals $x_1 \dots x_n$ are pairwise distinct atoms in P; and Φ is a propositional formula in the atoms $x_1 \dots x_n$. In 5.1, the expression $Q_1 x_1 \dots Q_n x_n$ is called *prefix* and Φ is called *matrix*. The QBF problem is: Given a QBF formula is it `true`?

Example 1 *Here below we can see a QBF in prenex-CNF, later we show how this formula can be represented in a suggested input format.*

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

5.3 Input and output format

This section outlines a suggested format for the QSAT problem. The format is known as Q-DIMACS (25). It is an extension of the DIMACS standard for the SAT problem. In this format, the input file is an ASCII file consisting of the following three sections:

Preamble : The preamble contains information about the instance. This information is contained in a few lines. Each line begins with a single character that determines the type of the line. These types are as follows:

- **Comments:** comment lines give human-readable information about the file and are ignored by programs. Comment lines appear at the beginning of the preamble. Each comment line begins with a lowercase character c. Example :

```
c This is an example of a comment line
```

- **Problem line:** there is only one problem line per input file. The problem line must appear before the prefix and after any comment line. The problem line has the following format:

```
p cnf VARIABLES CLAUSES
```

The lower case string "`p cnf`" shows that the instance is a "cnf formula". The "VARIABLES" field contains a positive integer value specifying the total number of variables in the instance. The "CLAUSES" field contains an integer value specifying the number of clauses in the instance. This line must be the last line of the preamble.

Prefix: The prefix encodes the information about the quantifiers and the way they are applied to propositions. Each line in the prefix begins with a letter followed by a set of positive integers. The letters are:

- a if the line represents a universal quantifier list,
- e if the line represents an existential quantifier list.

Each line ends with a 0. For example referring to Example 1, the prefix $\forall x_1 \exists x_2 \forall x_3 \exists x_4 x_5$ would be represented as:

```
a 1 0
e 2 0
a 3 0
e 4 5 0
```

Matrix: The matrix represents clauses of the QBF. It appears immediately after the prefix line. The variables are assumed to be numbered from 1 up to VARIABLES. It is not necessary that every variable appear in an instance. Each clause will be represented by a sequence of integers, which are separated by either a space, a tab, or a new line character. The non negated version of a variable is represented by i; the negated version is represented by $-i$. Each clause is terminated by a value 0. This format allows clauses to be presented on multiple lines. Considering example 1, $\forall x_1 \exists x_2 \forall x_3 \exists x_4 x_5 (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee x_5) \wedge (x_1 \vee \neg x_3 \vee x_4 \vee \neg x_5) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_5) \wedge (x_1 \vee \neg x_4) \wedge (x_3 \vee \neg x_2 \vee x_1)$, a possible input file would be:

```
c Example CNF format file
p cnf  5  9
a  1  0
e  2  0
a  3  0
e  4  5  0
 1  3  4  0
-1  3  4  0
 1 -4 -5  0
-1  2  5  0
 1 -3  4 -5  0
-1  3 -4  0
-1 -2 -3 -5  0
 1 -4  0
 3 -2  1  0
```

Every variable that does not show up in the prefix and occurs in the matrix is intended to be existentially quantified and in the outermost position of the prefix.

There is another format which is very similar to Q-DIMACS. It is known as "Rintanens format". Instead of 'a' for $\forall$ and 'e' for $\exists$, it uses the character 'q' alternatively. The first 'q' is for 'e', the second for 'a' and so on. For example the QBF: $\forall x \exists y \exists z \exists u((x \vee y) \wedge (\neg x \vee \neg z) \wedge (\neg u \vee z))$ is written as:

```
p cnf 4 3          # header: 4 variables, 3 clauses
q  0               # alternating quantifier prefix: must always
q  1  0            # start with a (possibly empty) sequence of
q  2  3  4  0      # existential variables; 0 is separator
 1  2  0           # clauses; 0 is separator
-1 -3  0           #
-4  3  0           #
```

5.4 The best existing solvers

In this section we introduce the best existing QSAT solvers. We know them from related published papers and the "Comparative evaluation of solvers for quantified Boolean formulas (QBF 2004)" (47) which has been held as a joint program with the "International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)". They comprise sixteen solvers from eleven authors. Here below we give short descriptions about each of them. Useful references are also given for who are interested in details.

CLearn and CSBJ: These solvers are presented by Andrew G D Rowley from "Department of Computer Science, university of St Andrews, Scotland". They are search-based solvers written in C++, using conflict learning and conflict back-jumping as introduced in (54; 27). These solvers are based on a local QBF solving library which implements watched literals and watched clause data structures for

QBF. The library performs the QBF simplification, propagation and backtracking. The considered heuristic is an ordering based on prefix level (outermost to innermost) and variable identifier (smallest first).

GRL is another solver from Andrew G.D. Rowley. It improves the solution learning methods introduced in (54; 27) to identify all of the information that is possible to learn. This algorithm is described in detail in (23).

OpenQBF by Gilles Audemard, Daniel Le Berre and Olivier Roussel, is a search-based solver written in Java. It is based on the OPENSAT framework, featuring basic unit propagation and pure literal lookahead, plus a conflict back-jumping look-back engine. Its main heuristic is derived from Böhm and Speckenmeyer's heuristic for SAT. This solver is described in detail in (2).

ORSAT by Olivier Roussel, is a search-based solver written in C++, featuring an algorithm based on relaxations to SAT, plus special purpose techniques to deal with universal quantifiers. Its creators remark that the solver is still in its early stage of development, therefore most of the features found in mature solvers are missing. One can visit (49) for more information.

QBFL The QBF solver Qbfl is built by Florian Letombe from the university of Artois in France. Qbfl is a QBF solver which is based on the SAT-solver "Limmat". It is a DPLL based solver using branching method in splitting on universal quantifiers. It also relays on trivial falsity and trivial truth. Qbfl has been presented in two versions: Qbfl-BS and Qbfl-JW. The first one chooses variables following Böhm and Speckenmeyer's lexicographic heuristics (5) and the second one follows Jeroslow and Wang's order (33). Unit and pure literal propagation for QBF (12) are also implemented. Qbfl makes Horn and reverse-Horn formulas detection and resolution as proposed in Kleine-Büning et al.'s algorithm (11).

QSAT by Jussi Rintanen, is a search-based solver written in C, featuring a

lookahead with failed literal rule, sampling, partial unfolding and quantifier inversion (48).

QMRes is developed by Guoqiang Pan and Moshe Y. Vardi from Department of Computer Science in Rice University. It is written in C, based on a symbolic implementation of the original DP algorithm, achieved using ZBDDs. The algorithm features multi-resolution, a simple form of unit propagation, and heuristics to choose the variables to eliminate. Multi-resolution (14) takes a symbolic approach to directional resolution (19), where clause sets are represented by ZDDs (31).

Quantor: The QBF solver QUANTOR is built by Armin Biere from Computer System Institute in Swiss Federal Institute of Technology in Zurich. It uses resolution to eliminate existentially quantified variables and expansion to eliminate universally quantified variables until a propositional formula in CNF is obtained, which then can be solved by a standard SAT solver. In has also a number of features such as equivalence reasoning, subsumption checking, pure literal detection, unit propagation, and also a scheduler for the elimination step. More details on QUANTOR can be found in (4).

QuBE: The QBF solver QuBE is built by Enrico Giunchiglia, Massimo Narizzano and Armando Tacchella from university of Genova in Italy. It is a search-based solver written in C++ featuring lazy data structures for unit and pure literal propagation; QuBE comes in two QuBE-bj, featuring conflict- and solution-directed backjumping, and QuBE-Lrn, featuring conflict and solution learning; the heuristic is an extension to QBF of zCHAFF heuristic for SAT.

QuBE explores an implicit AND-OR search tree alternating three phases: simplification of the formula (lookahead), choice of a branching literal (heuristic), and backtracking when a contradiction or a satisfying assignment is found. Look-back is based on learning as introduced in (27), with improvements used in GRASP. The heuristic is designed to leverage the information extracted during look-back.

ssolve by Rainer Feldmann and Stefan Schamberger, is a search-based algorithm written in C, featuring trivial truth and a modified version of Rintanen's method of inverting quantifiers. The data structures used are extensions of the data structures of Max Böhm's SAT-solver.

Semprop by Reinhold Letz, is based on the method of semantic trees, which is implemented on Davis, Putnam, Logemann, and Loveland (DPLL). The system incorporates the standard techniques like unit propagation and the purity rule, which are more powerful in the quantified Boolean case. Furthermore, a dependency-directed backtracking mechanism and learning features are integrated in it. It is written in Bigloo (a dialect of Scheme), featuring dependency directed backtracking and lemma/model caching for false/true subproblems. Details on Semprop can be found in (36).

WalkQSAT: The WalkQSAT QSAT solver is built by Andrew G D Rowley and Kevin Smyth from Department of Computer Science, at the university of St. Andrews, Scotland. It is the first incomplete solver for QBF. This solver performs standard conflict learning and solution directed back-jumping but also uses WaIkSAT to guide the search. If the solver returns true or false, the QBF is guaranteed to be true or false respectively. WalkQSAT can also return unknown, meaning that the QBF could not be solved. It is implemented using the same library as CLearn and CSBJ. For a more detailed description of WalkQSAT see (24).

yQuaffle: The QBF solver Quaffle is built by Yinlei Yu and Sharad Malik at Princeton University in Princeton,USA. It is a search-based solver written in C++, featuring multiple conflict driven learning, solution based backtracking, and inversion of quantifiers. It is designed to be fast and efficient for both real world problems and randomly generated problems. It is a DPLL based solver originated from Quaffle by Lintao Zhang (54). It also incorporates a few new techniques aiming at speeding up the evaluation of QBF formula.

CHAPTER 6

ZQSAT: A New Algorithm for the QSAT problem

6.1 Introduction

Almost all QBF solvers are based on extensions of the "DPLL algorithm for QBFs", which is also known as "search based method" or "the semantic tree approach for deciding QBFs". Most of them are based on some SAT solvers. There is a close connection between SAT and QSAT. Also there are some important differences which we mention later.

In preliminary studies we observed the capability of ZDDs in storing sets compactly. Furthermore, we realized that many operations needed in searching the solution can be implemented more efficiently when the QBF formula is represented in a ZDD. These observations led us to implement a ZDD-based QBF solver. At first it was a little slow, but after further observations, reasonings and investigations we managed to construct an efficient algorithm which we called it ZQSAT. It is a ZDD-based QBF solver that benefits from the idea of memoization and removing the subsumed clauses. Although it is also an extension to the DPLL algorithm, it can solve a set of QBF benchmark problems, which is known to be hard for DPLL based algorithms, much faster than almost all existing QSAT solvers. In addition to prenex-CNF, ZQSAT accepts prenex-NNF formulas. We will show it by experimental results and by giving a formal proof that accepting prenex-NNF formulas can be exponentially beneficial.

6.1.1 The DPLL Algorithm for the SAT and QBFs

Most former and recent SAT/QSAT solvers are in some way extensions of the DPLL (17) algorithm. DPLL tries to find a satisfying assignment for a CNF formula by making an exhaustive search. Each variable is assigned a truth-value (`true` or `false`) which leads to some simplifications of the formula. Since the function is in CNF, the assignment can be done efficiently. If an assignment forces the formula to be reduced to `false` then a backtrack will take place to make another possible assignment. If non of the possible assignments satisfy the formula then it is unsatisfiable. The DPLL algorithm is described in Figure 6.1.

```
Boolean DPLL( CNF_Formula  F )
{
1  F = Simplify F using simplification rules;  // Unit Resolution,..
2  if (F == empty clause) return TRUE;         // SATISFIABLE
3  if (F includes empty clause) return FALSE;  // UNSATISFIABLE
4  x = choose one of the remaining variables using Heuristic A;
5  v = choose a truth value using Heuristic B;
6  Fa = F[x=v];  Fb = F[x=not(v)];
7  if (DPLL(Fa)==TRUE) return TRUE;
8  return DPLL(Fb);
}
```

Figure 6.1: The DPLL Algorithm for SAT Problems.

Line 1 makes all possible simplifications. In order to prune the search space, variables which can get only one value must be assigned and be removed from the formula. A *unit clause* is a clause with exactly one literal. For example in $f = (a \vee \neg b \vee \neg c) \wedge (a \vee \neg c \vee d) \wedge (b) \wedge (a \vee b \vee c)$, the third clause is a unit clause. Since the algorithm tries to satisfy the formula, the variable appearing in a unit clause can essentially get one of the truth-values. In the above example, b can only receive the value `true`, which lets f be simplified to: $f_1 = (a \vee \neg c) \wedge (a \vee \neg c \vee d)$. In f_1 the first clause *subsumes* the second clause. In other words, if an assignment satisfies

the first clause then it will surely satisfy the second clause too. In such cases the subsumed clause can be removed. If all literals in a clause simplify without satisfying the clause then the condition in line 3 happens and returns the UNSATISFIABLE result, but if all the clauses satisfy and are removed then the condition in line 2 would happen and return the SATISFIABLE result.

In line 4 the procedure chooses one variable for splitting. Line 5 selects a truth value (`true` or `false`). In the next instruction, this value determines which branch should be investigated first. The splitting step (line 6) removes one variable and consequently a number of clauses are removed. Two smaller CNF formulas would be generated, out of which at least one must be satisfiable to make the original formula satisfiable. We see that there are two important decision points in the algorithm. The first one is choosing the next variable to assign a value (line 4), second one is the next split to investigate (which is determined in line 5 by choosing the truth value for the chosen variable).

The basic DPLL algorithm, in the worst case, is in the order of 2^n, where n is the number of variables. There are improved versions, for example, the deterministic local search method of (16), that can solve random 3-SAT instances in the expected runtime of $O(1.481)^n$. This is a significant improvement over $O(2^n)$; for example, instances with 60 variables can be solved in approximately 10^{10} steps, instead of 10^{18} (which may not be tractable). The study of improved algorithms is a practical way to work around the difficulty of solving NP-hard problems by solving small to medium-sized instances provably quickly. The main efforts to improve the DPLL algorithm in recent solvers are:

- Clever selection of the branching/splitting variable.
- Clever selection of the next branch to examine.
- Utilisation of efficient data structures.
- Profit-making from local search and learning.

6.1.2 The DPLL Algorithm for QBF:

This method (also known as "Semantic Tree Approach for deciding QBF") is very similar to the above mentioned DPLL algorithm. It splits the problem of deciding a QBF of the form $Qx\,\Phi$ into two subproblems $\Phi[x = 1]$ and $\Phi[x = 0]$ (assignment of x respectively with `true` or `false`), and the following rules:

- $\exists x\,\Phi$ is valid iff $\Phi[x = 1]$ is valid **or** $\Phi[x = 0]$ is valid.
- $\forall x\,\Phi$ is valid iff $\Phi[x = 1]$ is valid **and** $\Phi[x = 0]$ is valid.

Figure 6.2 displays the pseudo code of this algorithm, which we call it QDPLL.

```
Boolean QDPLL( Prenex-CNF  F )
 {
1  F = Simplify F using simplification rules; // Unit_Res., ...
2  if (F == empty_clause) return TRUE;        // SATISFIABLE
3  if (F includes empty_clause) return FALSE; // UNSATISFIABLE
4  x = choose one of the variables from
       the outermost quantifier-block using Heuristic A;
       // Very few freedom in choosing variables.
5  v = choose a truth value using Heuristic B;
6  Fa = F[x=v];  Fb = F[x=not(v)];
7  Solution=QDPLL(Fa);
8  if (Solution==TRUE  and  Existential_Literal(x) ) return TRUE;
9  if (Solution==FALSE and  Universal_Literal(x)   ) return FALSE;
10 return QDPLL(Fb);
 }
```

Figure 6.2: The sematic tree approach for QBFs.

The differences between QDPLL (for QBFs) and the DPLL algorithm (for propositional satisfiability) can be enumerated as follows:

1. In Unit-Resolution step (a simplification in line 1), if a universally quantified variable is found to be a unit clause then the procedure can immediately conclude the UNSAT result and terminate.

2. A literal is called *monotone or mono literal* if its complementary literal does not appear in the formula. In the Mono-Reduction step (another simplification of line 1), if any universally quantified variable is found to be a mono-literal, then it can be removed from all the clauses (rather than removing the clauses, as it applies to existentially quantified mono literals). The Mono-Reduction step can result in new unit clauses. Therefore the procedure must continue line 1 as long as new simplifications are possible.

3. In choosing the splitting variable (line 4), there is much less freedom than that of DPLL for propositional satisfiability. In fact, the sequence of the quantifier-blocks must be respected exactly in the same order as they appear in the prefix. The procedure is only allowed to choose any order for the variables which appear in a quantifier-block. As an example in the QBF: $\forall x_1 \forall x_2 \forall x_3 \exists y_1 \exists y_2 \exists y_3 \forall z_1 \forall z_2 \forall z_3 \phi$, all x_i must be assigned before any assignment for any y_j takes place, similarly, all y_j must be assigned before any assignment for any z_k takes place. On the other hand, we are allowed to consider any order when we are processing the variables of for example the x block.

4. After solving one of the branches, even if the result is `true` (line 7-10), it could be necessary to solve the other branch as well. Since universal variables are allowed to appear in QBFs, the `false` result (line 9) for one of the branches can signify the UNSAT result and terminate the procedure without checking the other branch.

From another point of view, this method searches the solution in a tree of variable assignments. Figure 6.3 displays a semantic tree for:

$$\Phi = \exists y_1 \forall x \exists y_2 \exists y_3 (C_1 \wedge C_2 \wedge C_3 \wedge C_4), \textit{where} :$$

$C_1 = (\neg y_1 \vee x \vee \neg y_2)$, $C_2 = (y_2 \vee \neg y_3)$, $C_3 = (y_2 \vee y_3)$, and $C_4 = (y_1 \vee x \vee \neg y_2)$.

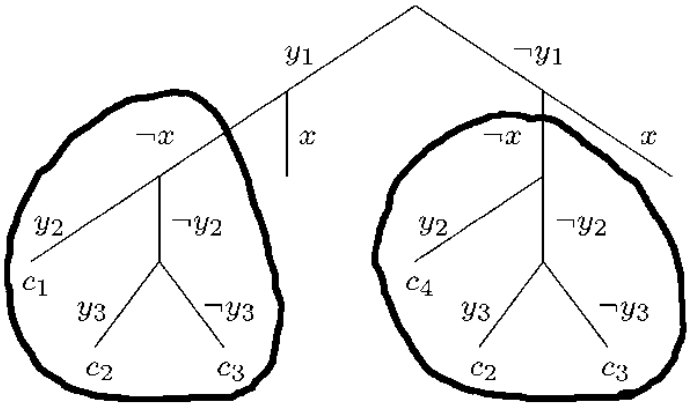

Figure 6.3: A semantic tree proof for the falsity of the above example.

We can follow the tree and realize that Φ is invalid. A very interesting point can be easily seen in the tree. It is the duplication problem in the semantic tree method, namely, the same subproblem can appear two or more times during the search procedure. In a big QBF this situation can happen frequently and in different levels. The superiority of ZQSAT, is its possibility to detect and avoid solving such duplications separately.

Some other differences between SAT and QSAT

We enumerated some important similarities and differences between SAT and QSAT problems through displaying and comparing the DPLL and the QDPLL procedures. Here we enumerate some other differences (53) which could not be seen in the pseudo codes.

1. **Lack of local search:** The technique of "local search" is used in some successful SAT solvers. Considering QBF as a two-player game, where competitors take turn in setting values to variables, it appears difficult or even impossible to model a greedy local search method.

2. **Lack of autarkies:** The concept of autarky has been used in a number of SAT algorithms. An assignment to some variables $V = \{v_{i1}, ..., v_{ik}\}$ is autarky if every clause containing a v_{ij} is satisfied by the assignment. This is nice because if an assignment of V is autarky, then while preserving satisfiability, we may remove all of the clauses containing one or more v_{ij}. While this property has been beneficial for SAT, it does not seem to be applicable with QBFs.

3. **Deficient Q-resolution:** Q-resolution (11) which is an extension of the well-known resolution proof method, is a sound and complete proof system for QBFs. Several improved algorithms for SAT crucially depend on resolution. Q-resolution is not as useful as the resolution for propositional satisfiability.

6.2 ZQSAT: A ZDD-based algorithm for deciding QBFs

ZQSAT is the name we used for our QSAT solver. There are three major points which are specific to our algorithm:

1. Embedding memoization to overcome the mentioned duplication problem (in order to avoid solving the same subproblem repeatedly).

2. Using ZDDs to represent the QBF matrix (the formula clauses). (We adopted this idea form (14; 1) then established the specific rules suitable for QBF evaluation).

3. Accepting Prenex-NNF in addition to Prenex-CNF formulas. We will show how this possibility can be considerably useful.

6.2.1 Embedding Memoization to QDPLL

We embedded memoization to the mentioned QDPLL algorithm to overcome the duplication problem. Figure 6.4 displays the pseudocode for MQDPLL (which stands

for our 'DPLL for QBF, strengthened with Memoization') procedure.

```
Boolean MQDPLL( Prenex-CNF  F )
  {
 1  if ( F is Primitive or AlreadySolved ) return Solution;
 2  S = Simplify F by repeated Unit-Resolution, Removal-of-
        Subsumed-Clauses and possible MonoLiteral-Reductions;
 3  if ( S is Primitive or AlreadySolved )
      {Add F along with the Solution to the SolvedTable;
       return Solution; }
 4  x = choose one of the variables from
        the outermost quantifier-set using Heuristic A;
 5  v = choose a truth value using Heuristic B;
 6  Fa = S[x=v];  Fb = S[x=not(v)];
 7  Solution=MQDPLL(Fa);
 8  if (Fa==Fb) or
        (Solution==TRUE  and  Existential-Literal(x) ) or
        (Solution==FALSE and  Univarsal-Literal(x)   )
          {Add F along with the Solution to SolvedTable;
           return Solution; }
 9  Solution=MQDPLL(Fb);
 10 Add Fb along with the Solution to SolvedTable;
    return Solution;
  }
```

Figure 6.4: MQDPLL: Our 'DPLL with Memoization' procedure.

MQDPLL is different from QDPLL in the following aspects: Firstly, it benefits from a memoization strategy (dynamic programming tabulation method) to store and reuse the results of already solved subproblems (lines 1, 3, 8, 10 in above pseudocode). Secondly, the situation where both subfunctions f_a and f_b are equal can be detected and the subproblem would be solved only once (line 7,8).

Storing all already solved subproblems and detecting the equality of two subproblems (functions) is usually very expensive. We managed to overcome these difficulties thanks to ZDD, holding the formula. This data structure let us to store the QBF's matrix very efficiently and allowed us to store every subfunction created

in the splitting step or obtained after the simplification operations, with no or very few overheads (see Figure 6.6). Since ZDD is a canonical representation of a function, the equality of two functions can be linearly decided, in many implementations with only one pointer comparison.

6.2.2 Using ZDD to represent a CNF formula

A ZDD can be used to represent a set of subsets. Since each propositional CNF formula ϕ can be represented as a set of subsets of literals $[\phi]$ we can represent a CNF formula by means of a ZDD (see Figure 6.5).

- ZDDs are very good in representing sets of subsets or a CNF Boolean formula.
- Collection of subsets:
  $S = \{\{a, c\}, \{b, c\}, \{c\}\}$
- or the CNF formula:
  $F = (a \vee c) \wedge (b \vee c) \wedge (c)$

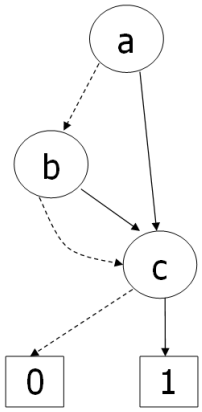

Figure 6.5: A ZDD representing a set of sets and a boolean formula in CNF.

In order to represent a set of clauses, we assign two successive ZDD indices to each variable, one index for positive and the next for its complemented form (14). Figure 6.6 shows how this idea works for a small CNF formula (1). In such a ZDD, each propositional literal l is represented by a ZDD variable v_l, and a set of clauses is represented by the following rules:

- The empty clause is represented by the terminal node 1.
- The empty set is represented by the terminal node 0.

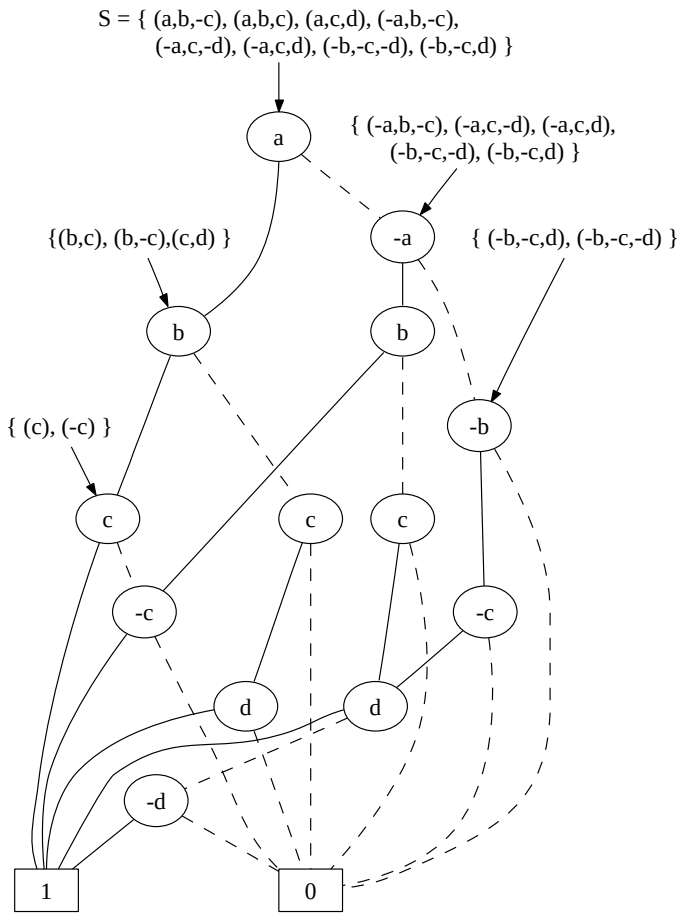

Figure 6.6: ZDD encoding of a CNF formula.

- Given a set C of clauses and a literal l whose ZDD variable v_l is lowest in a given variable order, we can split C into two subsets: $C_l = \{c : c \in C, l \in c\}$ and $C' = C - C_l$. Given ZDDs representing $C'' = \{c : c \vee l \in C_l\}$ and C', a ZDD representing C would be rooted at v_l and have ZDDs for C'' and C' as its left and right children.

In ZDDs (like BDDs), the variable order can considerably affect the shape and size of the resulting graph. As we pointed out earlier, in evaluating QBFs, the variable selection is strongly restricted. In general the order of the quantifier-blocks

of the prefix must be respected. In representing a QBF $\Phi = \exists x_1 \ldots \forall x_n \phi$ using ZDDs, we consider the extended literal order $x_1 \leq \neg x_1 \leq \ldots \leq x_n \leq \neg x_n$. The following theorem gives a good estimate for the size of such a ZDD.

Theorem 2 *Let f be a formula in conjunctive normal form. The number of nodes of the ZDD $\Gamma_{[f]}$ encoding the set of clauses of f is always at most equal to the total number of literals of f, i.e. $|\Gamma_{[f]}| \leq size(f)$.*

Proof 1 *Let f be in the form $\bigwedge_{i=1,\ldots,n}(\bigvee_{j=1,\ldots,m_i} l_{ij})$, where l_{ij} is a literal. In ZDD Γ_f each path from the root to the 1-terminal represents a clause of f. Therefore, $\Gamma_{[f]}$ contains n paths to the 1-terminal (if the clauses of f are pairwise disjunct). The path, representing the i-th clause of f includes m_i then-arcs, for each literal one then-arc. Therefore the ZDD $\Gamma_{[f]}$ has at most $\Sigma_{i=1}^{n} m_i$ then-arcs (some arcs could be shared by clauses) therefore at most $\Sigma_{i=1}^{n} m_i = size(f)$ nodes.*

When ZDDs are used to represent a set of *clauses* in the mentioned form, we can benefit from some additional reduction rules which normally cannot be used with ZDDs. Considering $P(x, \Gamma_1, \Gamma_2)$ as the ZDD having x in the root and Γ_1, Γ_2 as its "then" and "else" children, these reduction rules are:

1. **Elimination of subsumed clauses:** Any node of the form $P(x, \Gamma, \Gamma)$ can be eliminated. In other words, if both outgoing arcs of a node v point to the same node w, then v can be eliminated. In addition, any node of the form $P(x, \Gamma_1, \Gamma_2)$ can be replaced by $P(x, \Gamma_1, \Gamma_2 \setminus \Gamma_1)$, where $\Gamma_2 \setminus \Gamma_1$ stands for the set of clauses in Γ_2 which are not subsumed by clauses appearing in Γ_1.

2. **Elimination of tautologies:** A ZDD of the form $P(x, P(\neg x, \Gamma_1, \Gamma_2), \Gamma_3)$ can be replaced by $P(x, \Gamma_2, \Gamma_3)$.

6.2.3 Benefits of using ZDDs along with our MQDPLL-Algorithm

In Figure 6.6, we can also see another interesting characteristic of the ZDD, that is, their possibility of sharing nodes and subgraphs. In fact each node in a ZDD stands for a unique function. In many situations, this property lets ZDDs to hold new (sub)functions by producing a few or no additional nodes.

In our search procedure, after the simplification operations and after the splitting step, new functions arises. We noticed that many of these functions are the same, therefore we let ZQSAT to retain all already produced functions along with their solutions, to prevent solving the same functions repeatedly(Memoization). This idea also helped ZQSAT to generate exponentially fewer function calls. We mentioned earlier, this idea is embedding dynamic programming (memoization with tabulation) into the DPLL algorithm. In fact, after inserting this possibility, ZQSAT managed to solve the instances known to be hard for DPLL-based methods very fast (see Table 6.1).

Considering ZDDs as the data structure holding the formula affects the search algorithm and its complexity considerably. Operations like detecting unit clauses, detecting mono variables, performing unit/mono resolution and detecting SAT/UNSAT conditions strongly depend on the data structure holding the formula. Here we give some rules concerning these operations. These rules can be concluded from the basic properties known for QBFs, form some lemmas which are presented in (12) and from the properties of representing CNF clauses in a ZDD. Performing these operations with other data structures is often much slower. Reminding that Minato (29) has presented efficient algorithms for set operations on ZDDs. His algorithms are mostly based on dynamic programming and efficient caching techniques. We used them (through the CUDD package) in implementation of ZQSAT.

In the following rules we suppose that we have read the clauses and represented them in a ZDD Γ. The rules are applicable when we are examining satisfiability of Γ:

Rule 1 (Finding all unit clauses): A unit clause is a clause with exactly one literal. If the literal is universally quantified, then that clause and subsequently the QBF is unsatisfiable. If the literal is existentially quantified, then the truth value of the literal can be determined uniquely. Let $\Gamma = P(l_1, \Gamma_1, \Gamma_2)$ be a ZDD where l_1 is the topmost literal in the order, then the literal l_2 is a unit clause in Γ iff:

$l_2 = l_1$ and Γ_1 contains the empty set. In other words, the literal appearing in the root of the ZDD is a unit clause if moving to its Then-child followed by moving always toward the Else-child leads us to the 1-terminal.

$l_2 \in var(\Gamma_2)$ and l_2 is a unit clause in Γ_2. Note: if $l_2 \in var(\Gamma_1)$ then it can not be a unit clause.

Finding all unit clauses can be accomplished in at most $\frac{(2n-1)}{2}$ steps, where n is number of variables in the set of clauses represented by Γ.

Rule 2 (Trivial UNSAT): If x is a unit-clause and it is universally quantified, then the QBF formula is unsatisfiable. This operation needs only one comparison instruction and can be done during the step of finding the unit clauses.

Rule 3 (Trivial UNSAT): If x is an existentially quantified unit-clause and its complementary literal is also a unit clause, then the QBF formula is unsatisfiable. This operation can be performed during the identification of unit clauses.

Rule 4 (Variable assignment/ Splitting operation): In a ZDD $\Gamma = (x, \Gamma_1, \Gamma_2)$, considering x to be `true`, simplifies Γ to $Union(Then(\Gamma_2), Else(\Gamma_2))$. Similarly considering x to be `false`, simplifies Γ to $Union(\Gamma_1, Else(\Gamma_2))$. This operation is quadratic in the size of the ZDD.

Rule 5 (Propagation of a unit clause): If x is a unit clause and located in the root node then Γ it can be simplified to Γ_2. If Γ_2 has complement of x at its root then the result will be: $Union(Then(\Gamma_2), Else(\Gamma_2))$. On the other hand, if x is a unit clause but not located in the root node then, first we must remove all the clauses including x from Γ to obtain $\Gamma' = Subset0(\Gamma, x)$. Afterwards we must remove the complementary literal of x, denoted by $\overline{x}$ from Γ'. This operation can be done by: $\Gamma'' = Union(Subset1(\Gamma', \overline{x}), Subset0(\Gamma', \overline{x}))$.

Rule 6 (Mono Variables): If l is a mono-literal and existentially quantified, then we can replace it by `true`, which simplifies Γ to Γ_2, but if l is universally quantified we must replace it by `false`, which simplifies Γ to $Union(\Gamma_1, \Gamma_2)$.

```
UnitMonoReduction( FormulaZDD  F )
 {
  do{
     do{
       1.Find all Unit-Clauses in F, but if any Universally
         Quantified Literal found to be Unit return UNSAT,
         also if a literal is Unit and its complement is also
         Unit return UNSAT;
       2.//Reduce F for all Literals found to be Unit;
         for all Indices found to be Unit
           if ( Index is in the root of F ) F=Else(F);
           else{ F=RemoveClauses(F,Index);
                 F=RemoveLiteral(F,Not(Index));}
       }
      while(more iterations needed);

   3. while( Index of the root node is a mono Literal)
        if (the Index is universally quantified) F=Else(F);
        else F=Union(Else(F),Then(F));
    }
  while(more simplifications are possible);
}
```

Figure 6.7: Unit resolution and mono literal reductions.

Rule 7 (Detecting SAT/UNSAT): If the ZDD reduces to the 1-terminal then the QBF is SAT. Similarly, if the ZDD reduces to 0-terminal then the QBF is UNSAT. This operation needs only one comparison instruction.

These rules are the basic stones in implementing the operations needed in ZQSAT, specially the unit resolution and mono literal reduction in MQDPLL procedure. Figure 6.7 displays the pseudocode of the respective procedure. Line 1 is responsible for detecting all existing unit clauses, line 2 performs the unit resolutions, where line 3 stands for finding and performing mono literal reductions. We believe, using ZDDs along with benefiting from the above rules is one of the reasons for the success of ZQSAT.

6.2.4 Representing a Boolean formula in NNF by a ZDD

In representing a set of subsets of literals given by a CNF formula like:

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

First we make ZDDs representing the first two clauses. Then we use "zddUnion" operation to combine them. The result of this operation, as shown in Figure 6.8, is the ZDD holding: $[f_3] = [f_1] \wedge [f_2] = \{\{a, \neg b, \neg c\}, \{a, \neg c, d\}\}$.

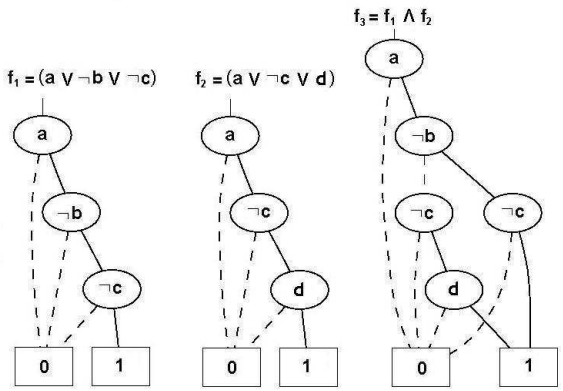

Figure 6.8: ZDD representations of two clauses and their conjunction.

After that we make the ZDD representation of the third clause and combine it with the result of the previous step. We do the same operations for all clauses to obtain the ZDD representing the whole function. In this approach the ZDDs representing individual clauses are in the same shape (also can be seen in Figure 6.8).

In representing an NNF formula in a ZDD we cannot follow the above procedure, but since in NNF, negations appear only in front of variables and only $\wedge$ and $\vee$ operations can appear, parsing and making the ZDD representing the formulas can be performed efficiently.

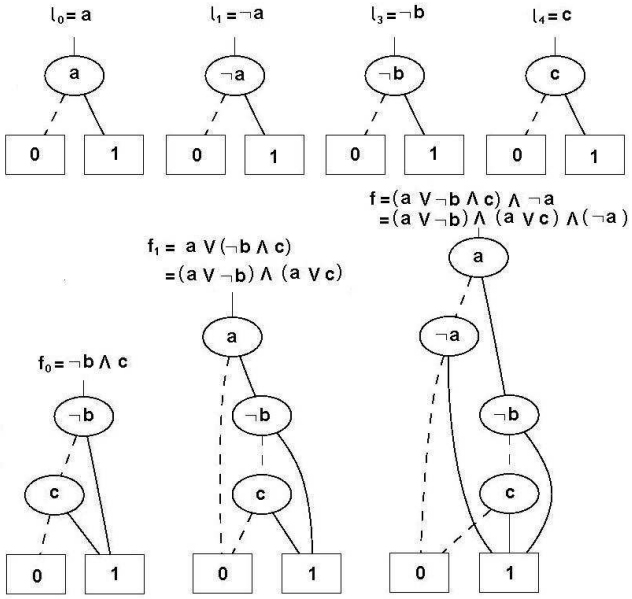

Figure 6.9: Making the ZDD representing a simple NNF formula.

Initially, according to the number of variables, we make individual ZDDs for each of the literals (positive or complemented form of a variable) which may probably appear in the formula. After that we parse the formula, which can be done in different ways. The result would be something like a parse tree holding literals in leaf nodes as well as $\wedge$ and $\vee$ operators in internal nodes. The parse tree shows how the ZDDs representing literals or set of clauses of sub-formulas must

be combined to obtain the ZDD representing the whole formula. Since a function is interpreted as a set of clauses, the operator $\wedge$ is emulated by "zddUnion" and "zddProduct" emulates $\vee$. Figure 6.9 shows how this idea works for the small NNF formula, $f = (a \vee (\neg b \wedge c)) \wedge \neg a$.

We can see another interesting point in Figure 6.9. It is an implicit conversion property which comes inherently with the procedure. In other words, from the ZDD representing $f_1 = (a \vee (\neg b \wedge c))$ we can easily give the CNF equivalent of f_1. The ZDD representation of the set of clauses of a NNF formula f is denoted by $\Gamma_{[f]}$.

6.2.5 Accepting NNF formulas can be Exponentially Beneficial

Transforming a QBF into prenex-CNF can be done by application of distributivity laws which can increase the size of the formula exponentially, but transforming any QBF into prenex-NNF can be done efficiently. Here we show how representation of a formula in NNF by a ZDD is beneficial compared to restricting ourselves to CNF formulas. Since the prenex-CNF and prenex-NNF differ only in their propositional part, we focus our attention on these parts. We define the size of an NNF-formula g, denoted by $size(g)$, to be the number of literals occurring in g. In our definition, if a literal occurs more than once, then each occurrence will be counted.

Theorem 3 *1) There are Boolean functions $f : \{0,1\}^n \rightarrow \{0,1\}$ in NNF with linear $size(f) = n$ whose logically equivalent minimal CNF representation f' are of exponential size $size(f') = 2^{\frac{n}{2}} \cdot \frac{n}{2}$.*

2) f can be represented by ZDDs of linear size n and there exists a synthesis algorithm for these ZDDs such that each ZDD occurring during the synthesis process of f would never be larger than n.

Proof: Consider $f = (l_1 \wedge l_2 \wedge \ldots \wedge l_k) \vee (l_{k+1} \wedge \ldots \wedge l_{2k}) \vee \ldots \vee (l_{m(k-1)+1} \wedge \ldots \wedge l_{mk})$ be a Boolean formula in disjunction normal form (disjunction of conjunctions of literals - therefore NNF), we will suppose f to be defined over n Boolean variables, holding m terms and each term include k literals. Also we will suppose that any variable appears only once (positive or negated) in the formula. Therefore, the size $size(f)$ of function f would be equal to $n = m \times k$. Below, Lemma 1 gives the function $f' \equiv f$ in CNF. If each term of f consists of exactly 2 literals, namely $k = 2$, then $size(f') = 2^{\frac{n}{2}} \cdot \frac{n}{2}$. On the other hand from Lemma 2 we get a ZDD representation of f, $\Gamma_{[f]}$ with $|\Gamma_{[f]}| = n$. The proof of Lemma 2 describes a synthesis algorithm in which all the ZDDs occurring during the synthesis of $\Gamma_{[f]}$ would never be larger than n.

lemma 1: Let $f : \{0,1\}^n \rightarrow \{0,1\}$ be a Boolean function in disjunction normal form (DNF). Suppose that f has n variables, where each variable - positive or negated - appears exactly one time in the function f, and that f consists of m terms each with k literals, i.e. $size(f) = m \cdot k$. Then the logically equivalent minimal CNF-representation $f' : \{0,1\}^n \rightarrow \{0,1\}$ of f has k^m clauses where each clause includes m literals. Therefore the size of f' is $k^m \cdot m$.

Proof: Proof by induction on the number m of clauses.

- $m = 1$: Then $f = (l_1 \wedge l_2 \wedge \ldots \wedge l_k)$. In this case $f' = (l_1) \wedge (l_2) \wedge \ldots \wedge (l_k)$ would be the CNF equivalent of f. In this case the size of f' is k, where we have k clauses with exactly one literal.

- $m \mapsto m + 1$: Let $f = g \vee (l_{(m+1)1} \wedge \ldots \wedge l_{(m+1)k})$, where $g = (l_{11} \wedge \ldots \wedge l_{1k}) \vee \ldots \vee (l_{m1} \wedge \ldots \wedge l_{mk})$. By induction hypothesis there exists a function $g' \equiv g$ in CNF where $size(g') = k^m m$ and g' has k^m clauses where each clause includes m literals. Considering $f_1 = g' \vee (l_{(m+1)1} \wedge \ldots \wedge l_{(m+1)k})$, to obtain f' from f_1, we must distribute $(l_{(m+1)1} \wedge \ldots \wedge l_{(m+1)k})$ over g', through which the number of clauses is increased by the factor k, i.e. g' includes k^{m+1} clauses

(all literals $l_{(m+1)j}$, $1 \leq j \leq k$, are pairwise disjoint). The size of each clause in f' is increased by one additional literal and would be $m+1$. Since each literal appears uniquely in f, none of the clause in f' can be removed by subsumption, also no literal can be removed from any clause. Therefore the size of f' is $k^{m+1} \cdot (m+1)$.

lemma 2: Let f be a Boolean function with conditions of Lemma 1. Let the variable order π of the ZDD be the same as the order in which literals appear in f, then the size of the ZDD representing f, as well as all ZDDs occurring during the synthesis of the ZDD representing f will never be larger than n.

Proof: Let $t = (l_1 \wedge l_2 \wedge \ldots \wedge l_k)$ be a term, where $l_i \neq l_j$ for $i, j \in \{1, \ldots, k\}$ and $i \neq j$. The ZDD $\Gamma_{[t]}$ representing t is of the form of the ZDD for f_{789} in Figure 6.10. The size of $\Gamma_{[t]}$ is k. For each literal l_i there is a node in $\Gamma_{[t]}$ labeled by l_i whose then-arc points to the 1-terminal and whose else-arc points to the node labeled by l_j. We suppose that l_i appears before l_j in the variable order and they are adjacent. The else-arc of the last node points to the 0-terminal. Now if $\Gamma_{[g]}$ is the ZDD representation of the first $m' < m$ terms, and $\Gamma_{[t]}$ is the representation of the next term, then the disjunction of $\Gamma_{[g]}$ and $\Gamma_{[t]}$ can be obtained as follows: we point each then-arc of $\Gamma_{[g]}$ leading to the 1-terminal to the root node of $\Gamma_{[t]}$, then consider the root node of $\Gamma_{[g]}$ to be the root node of the disjunction. The size of the ZDD-representation of the disjunction is given by $|\Gamma_{[g]}| + |\Gamma_{[f]}|$. Figure 6.10 illustrates this idea.

We need to mention that a QBF Φ can also be transformed into prenex-CNF by "structure preserving normal form transformation", where time and space complexity of the transformation is quadratic, but it produces a lot of new variables. Therefore the evaluation of the obtained formula could be increased substantially.

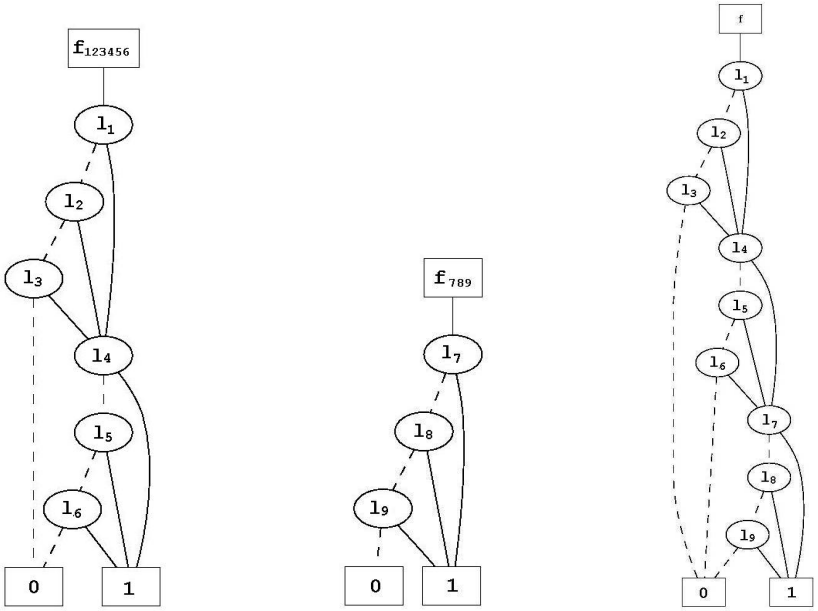

Figure 6.10: The ZDD representation of the function $f_{123456} = (l_1 \wedge l_2 \wedge l_3) \vee (l_4 \wedge l_5 \wedge l_6)$, the term $f_{789} = (l_7 \wedge l_8 \wedge l_9)$, and the function $f = f_{123456} \vee f_{789}$.

6.2.6 ZQSAT: Our MQDPLL, ZDD based QSAT solver

ZQSAT is nothing more than putting the mentioned ideas together. Figure 6.11 displays its pseudocode. It is based on our MQDPLL algorithm (the DPLL algorithm strengthened by memoization and ...), representing the QBF matrix in a ZDD and applying mentioned rules to perform related operations. Q-DIMACS is a suggested format for prenex-CNF. Almost all QBF benchmarks are presented in this format, therefore we also considered this input format in our implementation. In order to accept prenex-NNF QBFs, we introduced and used a similar format.

First ZQSAT reads and stores the prefix of the QBF in an array. This prefix holds the quantifiers of the QBF variables, then according to the input format (prenex-CNF or prenex-NNF), it reads the CNF formula clauses or the NNF formula. After that, it makes a ZDD representing the propositional part of the QBF

formula. The *variable order* of this ZDD is the same as the variable order in the prefix. We assign two successive indices to each variable, the first for its positive and the second for its complimented form. This is exactly the same as what we saw in Figure 6.6.

```
main() /*ZQSAT*/
{
 Get QBF Preamble to find input format, Nr. of Vars,...;
 Get QBF Prefix into an array;

 if ( Input is in prenex-CNF )
   Get clauses and represent them in a ZDD;
 else
   Get NNf formula and represent it in a ZDD;

 Result=MQDPLL(ZDD);
 Output the Result;
}
```

Figure 6.11: Pseudocode of ZQSAT.

6.3 Experimental Results

We evaluated our algorithm in two directions. It was evaluated firstly over different known benchmarks presented in QBFLIB (QBF satisfiability LIBrary) (25) and secondly over instances of the 'sequential depth of circuits' problem represented in NNF. We ran ZQSAT along with some of the best existing (47) QBF-Solvers such as QuBE (26), Decide (48), Semprop (36) and QSolve (22). The platform was a Linux system on a 3000-Mhz, 2G-RAM desktop computer. We also considered 1G-RAM limit which was never totally used by any of the above programs, and a 900 second timeout which was enough for most solvers to solve many of benchmark problems. The results we obtained can be summarized as follows:

1. Over QBFLIB benchmarks (Prenex-CNF)
   ZQSAT is very efficient and in many cases better than the state-of-the-art QSAT solvers. It solves many instances which are known hard for DPLL (semantic-tree) method, in a small fraction of a second. Like almost all other QSAT solvers it is inefficient in solving random QBFs.

2. Over sequential depth of circuits (Prenex-NNF)
   We considered the 'sequential depth of circuits' problem and made instances in Prenex-NNF. ZQSAT was able to solve them much faster than corresponding prenex-CNF instances.

The following subsections give detailed information on the above findings.

6.3.1 Evaluating ZQSAT over standard prenex-CNF benchmarks

Structured formulas: Most structured formulas come form real word problems. We used the benchmarks of Letz (25) and Rintanen (48). The benchmarks of Letz include instances known to be hard for DPLL (tree-based) QBF solvers. In a real problem there are always some relations between its components. These relations remain in some form in their corresponding QBF representation. This feature causes similar subproblems to be generated (at the beginning and during the search step). Also assignment of values to variables causes sharp simplification on generated subformulas. Therefore our Memoization idea helps very much in these circumstances. Table 6.1 shows how ZQSAT is faster than other recent QBF solvers in evaluating these benchmark problems.

Next, we considered the benchmarks of Rintanen, where some problems from AI planning and other structured formulas are included. They include some instances form blocks world problem, Towers of Hanoi, long chains of implications, as well as the bw-large.a and bw-large.b blocks world problems. The experimental

problem		QuBE				
tree-exa-	ZQSAT	BJ	Rel	Decide	Semprop	QSolve
10-10	< .01	< .01	< .01	< .01	< .01	< .01
10-15	< .01	< .01	< .01	0.06	0.01	< .01
10-20	< .01	< .01	0.01	1.89	0.27	< .01
10-25	< .01	0.01	0.07	63.95	8.51	< .01
10-30	< .01	0.11	0.75	(?)	273.28	0.03
2-10	< .01	< .01	< .01	< .01	< .01	< .01
2-15	< .01	< .01	< .01	0.01	< .01	< .01
2-20	< .01	0.01	< .01	0.1	0.01	< .01
2-25	< .01	0.12	< .01	1.16	0.1	0.04
2-30	< .01	1.29	< .01	12.9	1.06	0.53
2-35	< .01	14.42	< .01	144.16	11.98	5.85
2-40	< .01	158.41	< .01	(?)	130.19	65.73
2-45	< .01	(?)	< .01	(?)	(?)	729.7
2-50	< .01	(?)	< .01	(?)	(?)	(?)
(?): Not solved in 900 seconds						

Table 6.1: Runtimes of different QBF solvers over a number of QBFs. The instances are known to be hard for tree-based QBF solvers.

results for these benchmarks are presented in Table 6.2 and Table 6.3. These tables shows that ZQSAT works well on most instances. We are comparable and in many cases better than other solvers. Here we mention that 'Decide' is specially designed to work efficiently for planning instances. In Table 6.2 we see that our implementation could not solve any instance of blocks-world. We observed that MQDPLL benefited just a few times from the already solved subformulas. In other words, our pruning method was not successful for this problem. In fact this is a matter of pruning strategy, different pruning strategies behave differently facing various sets of QBF benchmarks.

Random formulas: For random formulas we used the benchmarks of Massimo Narizzano (25). ZQSAT is inefficient in big unstructured instances. ZDDs are very good in representing sets of subsets, but they are less useful, if the information is unstructured. ZDDs explore and use the relation between the set of subsets. Therefore if there is no relation between the subsets (clauses) then it could not play its role truly. Fortunately in real word problems there are always some connections between the problem components. In our effort to investigate why ZQSAT is slow on the given instances, we found that in these cases the already solved subformulas

problem	ZQSAT	QuBE BJ	QuBE Rel	Decide	Semprop	QSolve
B*3i.4.4	(?)	(?)	0.02	0.02	(?)	(?)
B*3i.5.3	(?)	(?)	516.67	10.51	(?)	(?)
B*3i.5.4	(?)	(?)	(?)	1.84	(?)	(?)
B*3ii.4.3	(?)	0.81	0.01	0.01	2.25	(?)
B*3ii.5.2	(?)	23.25	0.41	0.02	65.76	(?)
B*3ii.5.3	(?)	(?)	33.12	0.36	160.93	(?)
B*3iii.4	(?)	0.25	0.01	< .01	12	(?)
B*3iii.5	(?)	(?)	0.48	0.1	0.53	(?)
B*4i.6.4	(?)	(?)	264.76	1.28	(?)	(?)
B*4ii.6.3	(?)	(?)	27.64	1.1	(?)	(?)
B*4ii.7.2	(?)	(?)	(?)	2.28	(?)	(?)
B*4iii.6	(?)	(?)	13.62	0.59	(?)	(?)
B*4iii.7	(?)	(?)	(?)	67.28	(?)	(?)
C*12v.13	2.66	0.12	1.41	0.19	0.06	1.96
C*13v.14	3.76	0.26	3.44	0.38	0.13	6.52
C*4v.15	5.27	0.55	9.17	0.77	0.27	21.98
C*15v.16	7.08	1.22	24.21	1.62	0.54	62.53
C*16v.17	9.43	3.09	60.68	3.31	1.14	205.72
C*17v.18	12.49	5.86	148.58	6.9	2.43	633.44
C*18v.19	16.2	12.87	352.21	14.4	5.12	(?)
C*19v.20	21.01	31.93	840.26	30.29	10.59	(?)
C*20v.21	26.69	91.23	(?)	61.93	22.24	(?)
C*21v.22	33.17	195.12	(?)	129.24	46.61	(?)
C*22v.23	40.8	494.26	(?)	272.24	98.53	(?)
C*23v.24	50.24	(?)	(?)	571.12	202.3	(?)
i*02	< .01	< .01	< .01	< .01	< .01	< .01
(?): Not solved in 900 seconds						

Table 6.2: Experimental results for some instances in benchmarks of Rintanen. Here ZQSAT is often slower.

were never or rarely used again, also the mono and unit resolution functions could not reduce the size of the (sub)formula noticeably.

The variable order: Size of a ZDD depends on the chosen variable order. As mentioned in Section 6.2, we used the variable order given in the prefix of the QBF as the variable order for the ZDD. In QBF evaluation we are not allowed to choose any variable order, but at the same time we are allowed to change the variable order in any quantifier-block. This fact encouraged us to test the allowed variable re-orderings in our method. We implemented the sifting algorithm of Rudell (50; 42) to be suitable for QBFs. We had to consider some restrictions because:

1. each variable of the QBF is in accordance with two ZDD indices which are adjacent (these couples should not be separated).

problem	ZQSAT	QuBE BJ	QuBE Rel	Decide	Semprop	QSolve
i*02	< .01	< .01	< .01	< .01	< .01	< .01
i*04	< .01	< .01	< .01	< .01	< .01	< .01
i*06	< .01	< .01	< .01	0.01	< .01	< .01
i*08	< .01	< .01	< .01	0.14	0.02	0.01
i*10	< .01	0.01	< .01	1.12	0.14	0.07
i*12	< .01	0.04	< .01	8.69	1.04	0.5
i*14	< .01	0.18	< .01	65.27	7.74	3.69
i*16	< .01	0.74	< .01	482.97	56.88	27.04
i*18	< .01	3.12	< .01	(?)	423.41	200.82
i*20	< .01	13.06	< .01	(?)	(?)	(?)
l*A0	0.06	< .01	< .01	< .01	(?)	0.01
l*A1	(?)	50.28	5.79	0.67	3.68	(?)
l*B0	0.2	< .01	< .01	0.01	(?)	0.01
l*B1	(?)	407.65	14.62	3.25	13.91	(?)
T*10.1.iv.20	2.65	(?)	(?)	0.58	(?)	(?)
T*16.1.iv.32	26.08	(?)	(?)	7.38	(?)	(?)
T*2.1.iv.3	< .01	< .01	< .01	< .01	< .01	< .01
T*2.1.iv.4	< .01	< .01	< .01	< .01	< .01	< .01
T*6.1.iv.11	(?)	2.1	205.75	4.78	2.25	3.66
T*6.1.iv.12	0.24	0.79	29.44	0.04	0.4	2.65
T*7.1.iv.13	(?)	37.45	(?)	63.87	39.7	134.02
T*7.1.iv.14	0.5	12.25	521.59	0.09	5.22	64.17
(?): Not solved in 900 seconds						

Table 6.3: Experimental results for some instances in benchmarks of Rintanen. Here ZQSAT is often faster.

2. the algorithm can swap two variables if and only if they are in the same quantifier-block;

3. the sequence of quantifier-blocks should not be changed.

benchm.-sets	Nodes [avg]	Sifting [avg]	improve [%]
2QBF-5CNF-100	5571	5513	1
3QBF-5CNF-50	2951	2920	0.7
4QBF-5CNF-50	3283	3263	0.6
Letz	272.8	272.8	0
Scholl, Becker	12004.7	11307.3	5.8
Pan	68563.5	68178.8	0.6

Table 6.4: Number of ZDD nodes before and after applying the sifting algorithm.

Table 6.4 gives a summary of our experimental results concerning variable reordering. We considered instances from Narizzano, Letz, Scholl-Becker, and Pan from the QBFLIB (25) site. In the table we have only included the average number

of nodes needed to represent any instance before and after applying the sifting algorithm. Also, we applied the sifting algorithm only to the representation of the initial formula. So far we did not apply it over all ZDDs obtaining during the search step.

6.3.2 Evaluating ZQSAT over instances in Prenex-NNF

In order to examine ZQSAT for prenex-NNF formulas, we needed a set of benchmarks given in prenex-CNF as well as in prenex-NNF, concerning the same problem(s). For this purpose we considered the "Sequential depth of circuits", which is an important problem in formal hardware verification.

Bounded model checking (BMC) (3) is a branch of model checking which verifies safety and liveness properties of systems by SAT/QSAT solvers. In BMC, in order to examine the safety or liveness property of a system, the property is transformed into a formula, then one checks satisfiability of the obtained formula. The formula is satisfiable if and only if there exists a path sinking in a state that violates the considered property.

Mneimneh and Sakallah (45) have addressed "sequential depth of a sequential machine" problem by QBFs. In their work, description of the sequential machine is as usual given by a finite-state-machine(FSM). They show how the question, "Does a path of the length i in the machine exist" can be expressed by a QBF. At first the obtained QBF isn't in any normal form. Most resent QBF-solvers accept only prenex-CNF, therefore they require the QBF to be transformed into prenex-CNF. We designed and implemented ZQSAT in such a way that accepts both prenex-CNF and prenex-NNF. ZQSAT represents a prenex-NNF directly into a ZDD then starts its common way to search the solution. Transformation of an arbitrary QBF into its equivalent in prenex-NNF can be efficiently done and gives a much more compact result than transforming an arbitrary QBF into prenex-CNF.

QBFLIB (25) provides a lot of instances of the "sequential depth problem" for some circuits of ISAC89. The QBFs are presented in prenex-CNF. In order to have the same instances in prenex-NNF, Klotz (40) implemented a procedure (based on Mneimneh and Sakallah (45)) that transforms each instance of the problem into a QBF. The procedure gets an FSM M given in kiss-format and the depth i, then it outputs a QBF in prenex-NNF. The obtained QBF is satisfiable iff the sequential depth of M is at least i. We produced QBFs in prenex-NNF for a number of LGSynth93 (37) benchmarks and solved them with ZQSAT. Let's bear in mind, that the instances from QBFLIB and the instances we generated, encode the same problems for the same circuits, but there are some differences. Firstly, our encoding results in a QBF in NNF instead of prenex-CNF. Secondly, there are differences in the bases of the encoding method. Therefore one must be careful when comparing the experimental results.

We investigated effectiveness of ZQSAT in accepting prenex-NNF, by running it along with some of the already mentioned QSAT solvers over the above benchmarks. Again the platform was a Linux system on a 3000-Mhz, 2G-RAM desktop computer. We also considered 1G-RAM limit and a 300 second timeout. The results are reported in Table 6.5 and Table 6.6.

Circuit	depth	ZQSAT	decide	QuBE-BJ	semprop
s27	2	0.02	0.07	< 0.01	< 0.01
s27	3	0.71	13.93	30.95	6.37
s27	4	18.11	171.12	(?)	(?)
s27	5	63.38	(?)	(?)	(?)
s386	2	(?)	(?)	(?)	(?)
s510	2	(?)	(?)	(?)	(?)
s820	2	(?)	(?)	(?)	(?)
(?): Not solved in 300 seconds					

Table 6.5: Runtime of depth formulas in prenex-CNF on various QBF solvers.

In table 6.5 we have considered only prenex-CNF benchmarks. column 1 lists the name of the circuit, where other columns present the runtimes of different solvers. For the big circuits all of the solvers timed out after 300 seconds and were not able to solve the instance.

Next we run ZQSAT over prenex-NNF, while other solvers over equivalent benchmarks in prenex-CNF. The results are reported in Table 6.6.

Circuit	depth	SAT/ UNSAT	ZQSAT	QBFLIB
s27	2	SAT	< 0.01	< 0.01
s27	3	UNSAT	0.02	0.71
s27	4	UNSAT	0.04	18.11
s27	5	UNSAT	0.06	63.38
s386	2	SAT	0.14	(?)
s386	6	SAT	0.99	(?)
s386	8	UNSAT	2.04	(?)
s386	10	UNSAT	6.71	(?)
s510	2	SAT	1.19	(?)
s510	6	SAT	7.14	(?)
s510	12	SAT	22.3	(?)
s820	2	SAT	3.32	(?)
s820	6	SAT	20.78	(?)
s820	10	SAT	37.32	(?)
s820	12	UNSAT	70.4	(?)
s1488	2	SAT	3.58	(?)
s1488	6	SAT	22.08	(?)
s1488	10	SAT	70.79	(?)
(?): Not solved in 300 seconds				

Table 6.6: Runtime of depth formulas in prenex-NNF on ZQSAT.

Here column 1 lists the name of the circuit, while column 2 shows the considered depth i. The next column shows, if the circuit depth is at least i or not. Column 4 gives the runtime in seconds for ZQSAT to solve the prenex-NNF instance while the last column displays the runtime of the best solver on the same instance from the QBFLIB (prenec-CNF). We can observe that ZQSAT can solve much higher depth and much faster than other solvers.

CHAPTER 7

Summary and Future Work

7.1 Introduction

SAT (SATisfiability of propositional Boolean formulas) is the first and prototypical problem for the class of NP-complete problems. Many computational problems such as the constraint satisfaction problem and many problems in graph theory as well as forms of planning can be easily formulated as instances of SAT. Theoretical analysis has shown that some forms of reasoning such as: non-monotonic reasoning, reasoning about knowledge and STRIPS-like planning have a computational complexity higher than the complexity of the SAT problem. These forms can be formulated by quantified Boolean formulas (QBF) and can be solved as instances of the QSAT (SATisfiability of Quantified Boolean Formulas) problem. QSAT is a generalization of the SAT problem. QSAT is the prototypical problem for the class of PSPACE-complete problems.

Zero-suppressed Binary decision Diagrams (for short ZBDDs or ZDDs) are variants of BDDs (Binary decision Diagrams). While BDDs are better suited for representing Boolean functions, ZDDs are better for representing sets of subsets. Considering all the variables appearing in a QBF as a set, the propositional part of the formula can be viewed as a set of subsets. This observation was the motivation to use this data structure for implementing a new QSAT solver.

7.2 What did we do?

We learned about BDD and its variants, specially ZDD. These are efficient data structures in representation and manipulation of Boolean formulas and sets of subsets. We also learned that QSAT is one of the important problems in computer science with numerous applications.

After learning about the basic algorithms for the problem and the techniques which are already used by the best existing solutions (QSAT solvers), we came to a point to adopt ideas from them and invent conjectures in creating a new algorithm for the problem. Finally we invented an algorithm which is comparable and in some cases (for some known benchmarks) much faster than the best existing QSAT solvers.

7.3 Why does it matter?

A QBF is an expression of the form: $Q_1 x_1 \dots Q_n x_n \Phi$, where every $Q_i (1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$; The literals $x_1 \dots x_n$ are pairwise distinct atoms; and Φ is a propositional formula in CNF on atoms $x_1 \dots x_n$. The QSAT problem is: Given a QBF formula, is it True?

It can be shown that many scientific problems with numerous applications can be reduced or formulated in the form of the above problem. Therefore any progress in solving the QSAT problem, is a progress in solving a lot of important problems. In fact, this is why there is a one million dollar prize for the famous "P=NP?" problem, which has a strong connection to the SAT problem. We need to mention that although there are many solutions for the SAT and QSAT problems, almost all of them in the worst case are still exponential. Researchers are trying to find faster solutions, or perhaps a polynomial solution for subclasses of the problem. Finding a polynomial solution for the general form seems to be impossible.

7.4 How did we do it?

Almost all existing QSAT solvers are based on an extended version of the well known DPLL algorithm. They just use different data structures and heuristics to find faster solutions. The main efforts to improve the DPLL algorithm consist of:

- Clever selection of the branching/splitting variable.
- Clever selection of the next branch to examine;
- Utilization of efficient data structures.
- Profit-making from local search and learning.

We realized and learned from some materials that ZDDs can represent boolean formulas given in CNF (Conjunctive Normal Form) efficiently. We also learned that one important drawback of the main algorithm is its duplication. These observations led us to use ZDD as the data structure holding the QBF formula and to embed the idea of memoization to the named algorithm. We implemented our algorithm using CUDD (Colorado University Decision Diagram) package. We called it ZQSAT.

First, the experimental results showed that ZQSAT was comparable with other existing solutions. We tried a number of other heuristics to improve our ZQSAT or at least investigate their effect. However we discovered and showed how ZQSAT was comparable and sometimes much faster than all other existing solutions in solving some sets of known benchmark problems.

We improved ZQSAT to let it accept prenex-NNF (QBFs in the form of $Q_1 x_1 \dots Q_n x_n \Phi$, where Φ is a propositional formula in NNF (Negation Normal Form). We showed and proved that this possibility can be exponentially beneficial. We may summarize our contribution as:

- Using ZDDs to represent the QBF matrix. We adopted this idea and then established the specific rules suitable for QBF evaluation.

- Embedding memoization to overcome the duplication problem of the "DPLL for QBF algorithm" (to avoid solving the same subproblem repeatedly).
- Accepting Prenex-NNF in addition to Prenex-CNF formulas.

7.5 What did we learn?

What we learned from ZQSAT can summarize as follows:

- The representation of Boolean formulas by ZDDs is beneficial, since this data structure allows the compact representation of the formula and lets us store new (sub)formulas with no or very few additional nodes.
- Embedding memoization to the 'DPLL for QBF' algorithm lets us avoid solving the same subproblems repeatedly which was sometimes exponentially beneficial.
- Removing subsumed clauses was very useful in the simplification of the formula leading to an efficiency in the speed of the search procedure.
- The transformation of a general Boolean formula into its equivalent in NNF can be made efficiently. Nevertheless, the transformation of a general Boolean formula into its equivalent in CNF can be made in exponential time. Therefore accepting NNF formulas can be exponentially beneficial.

7.6 Future Work

For future work in the direction of our research we suggenst:

- Carrying out an in-depth investigation of variable ordering.
- Integrating ZQSAT with the techniques used in other solvers.
- Looking for more examples where accepting NNF formulas is beneficial.
-

REFERENCES

[1] Fadi A. Aloul, Maher N. Mneimneh, and Karem A. Sakallah. ZBDD-Based Backtrack Search SAT Solver. In *International Workshop on Logic and Synthesis (IWLS)*, pages 131–136, New Orleans, Louisiana, 2002.

[2] Gilles Audemard, Bertrand Mazure, and Lakdar Sais. Dealing with symmetries in quantified Boolean formulas. In *Proceedings of SAT*, 2004.

[3] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *Proceedings of the 36th Design Automation Conference*, 1999.

[4] Armin Biere. Resolve and expand. In *International Conference on Theory and Applications of Satisfiability Testing -SAT 04*, BC, Canada, 2004.

[5] Max Böhm and Ewald Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence.*, 17:381–400, 1996.

[6] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *DAC: Design Automation Conf.*, pages 40–45, 1990.

[7] Robert King Brayton. *Logic minimization algorithms for VLSI synthesis* . Boston : Kluwer Academic Publishers, 1984.

[8] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[9] Randal E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[10] Randal E. Bryant, Derek L. Beatty, Karl S. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In *DAC: Design Automation Conference*, pages 9–16, 1987.

[11] Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12–18, 1995.

[12] Marco Cadoli, Marco Schaerf, Andrea Giovanardi, and Massimo Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[13] Eduard Cerny and Miguel A. Marin. An Approach to Unified Methodology of Combinational Switching Circuits. *IEEE Transactions on Computers*, 26(8):745–756, 1977.

[14] Philippe Chatalic and Laurent Simon. Multi-resolution on compressed sets of clauses. In *International Conference on Tools with Artificial Intelligence*, pages 2–10, 2000.

[15] Philippe Chatalic and Laurent Simon. ZRES: The Old Davis-Putman Procedure Meets ZBDD. In *International Conference on Automated Deduction (CADE)*, pages 449–454, 2000.

[16] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-\frac{2}{k+1})^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 2001.

[17] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5(7):394–397, 1962.

[18] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[19] Rina Dechter and Irina Rish. Directional Resolution: The Davis-Putnam Procedure, Revisited. In *Principles of Knowledge Representation and Reasoning*, pages 134–145, 1994.

[20] Rolf Drechsler and Bernd Becker. *Binary decision diagrams : theory and implementation.* Kluwer Academic Publishers, Boston.

[21] Rolf Drechsler and Detlef Sieling. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):112–136, 2001.

[22] Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 2000.

[23] Ian P Gent and Andrew G D Rowley. Solution Learning and Solution Directed Backjumping, Revisited. citeseer.ist.psu.edu/gent04solution.html, 2004.

[24] I.P. Gent, H.H. Hoos, A.G.D. Rowley, and K. Smyth. Using Stochastic Local Search to Solve Quantified Boolean Formulae. In *Constraint Programming (CP'2003)*, volume 2833 of *LNCS*. Springer-Verlag, 2003.

[25] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB). www.qbflib.org, 2001.

[26] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In *International Joint Conference on Automated Reasoning(IJCAR)*, pages 364–369, 2001.

[27] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *AAAI/IAAI*, pages 649–654, 2002.

[28] Gary D Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms.* Kluwer Academic Publishers, Boston, 1996.

[29] Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, Dallas, TX, 1993.

[30] Shin ichi Minato. Calculation of Unate Cube Set Algebra Using Zero-Suppressed BDDs. In *Design Automation Conference (DAC)*, pages 420–424, 1994.

[31] Shin ichi Minato. *Binary decision diagrams and applications for VLSI CAD.* Boston : Kluwer Academic Publishers, 1996.

[32] Shin ichi Minato. Fast Factorization Method for Implicit Cube Cover Representation. *IEEE Transactions on CAD*, 15(4):377–384, 1996.

[33] Robert G. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence.*, 1:167–187, 1990.

[34] Geoff Leach. Lecture Handouts for the Computational Science course. http://goanna.cs.rmit.edu.au/ gl/teaching.

[35] C.Y. Lee. Representation of switching circuits by binary decision diagrams. *Bell System Technical Journal*, 38:985–999, 1959.

[36] Reinhold Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proceedings of TABLEAUX*, pages 160–175, 2002.

[37] LGSynth93 Benchmarks - Circuits from LGSynth91, ISCAS89 and ISCAS85. . http://www.cbl.ncsu.edu/CBL_Docs/lgs93.html.

[38] Christoph Meinel, Jochen Bern, and Anna Slobodová. Efficient OBDD-Based Boolean Manipulation in CAD beyond Current Limits. In *Design Automation Conference (DAC)*, pages 408–413, San Francisco, CA, 1995.

[39] Christoph Meinel and Jordan Gergov. Mod-2-OBDDs - A Data Structure that Generalizes EXOR-Sum-of-Products and Ordered Binary Decision Diagrams. *Formal Methods in System Design*, 8(3):273–282, 1996.

[40] Christoph Meinel, Mohammad GhasemZadeh, and Volker Klotz. A QSAT Solver for QBFs in Prenex NNF. In *International Workshop on Logic and Synthesis (IWLS)*, pages 135–142. California, USA, June 2004.

[41] Christoph Meinel and Anna Slobodová. A Unifying Theoretical Background for Some Bdd-based Data Structures. *Formal Methods in System Design*, 11(3):223–237, 1997.

[42] Christoph Meinel, Fabio Somenzi, and Thorsten Theobald. Linear sifting of decision diagrams. In *DAC*, pages 202–207, 1997.

[43] Christoph Meinel and Thorsten Theobald. *Algorithms and data structures in VLSI design: OBDD - foundations and applications.* Berlin, Heidelberg, New York: Springer-Verlag, 1998.

[44] Alan Mishchenko. An Introduction to Zero-Suppressed Binary Decision Diagrams. http://www.ee.pdx.edu/ alanmi/research/dd/zddtut.pdf, June 2001.

[45] Maher Mneimneh and Karem Sakallah. Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[46] B. Moret. Decision trees and diagrams. *Computer Survey*, (14):593–623, 1982.

[47] Massimo Narizzano and Armando Tacchella. Quantified Boolean Formula Solvers Evaluation. http://www.qbflib.org/qbfeval/.

[48] Jussi Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1192–1197, 1999.

[49] Olivier Roussel. The ORSAT web page. http://www.cril.univartois.fr.

[50] R. Rudell. Dynamic Variable reordering for Ordered Binary Diagrams. In *Proceedings IEEE International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, 1993.

[51] Fabio Somenzi. CUDD Package. ftp://vlsi.colorado.edu/pub/.

[52] Ingo Wegener. *Branching programs and binary decision diagrams : theory and applications* . Philadelphia : Society for Industrial and Applied Mathematics(SIAM), 2000.

[53] Ryan Williams. Algorithms for Quantified Boolean Formulas. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[54] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *CP*, pages 200–215, 2002.