Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Ondrej Mikle-Barát

Strong Proof Systems

Department of Software Engineering
Supervisor: Prof. RNDr. Jan Krajíček, DrSc.
Study program: Computer Science, Software Systems

I would like to thank my advisor for his valuable comments and suggestions. His extensive experience with proof systems and logic helped me a lot.

Finally I want to thank my family for their support.

Contents

Název práce: Silné důkazové systémy
Autor: Ondrej Mikle-Barát
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. RNDr. Jan Krajíček, DrSc.
e-mail vedoucího: krajicek@math.cas.cz
Abstrakt:

R-OBDD je nový Cook-Reckhowův důkazový systém pro výrokovou logiku založen na kombinaci OBDD důkazového systému a rezolučního důkazového systému. R-OBDD má sílu OBDD důkazového systému – tautologie s exponenciálně velkými důkazy v rezoluci jako PHP_n nebo Tseitinovy kontradikce mají v R-OBDD systému polynomiální důkazy (R-OBDD p-simuluje jak OBDD důkazový systém, tak rezoluci). Na druhé straně, odvozovací pravidla R-OBDD systému byly navrhnuty, aby se podobaly odvozovacím pravidlům rezoluce. Tím pádem je možné vytvořit modifikaci DPLL algoritmu, která bude pracovat v R-OBDD systému a zároveň použít některé heuristiky známé z algoritmů založených na DPLL. Vzniká možnost vytvořit efektivnejší algoritmus na řešení problému splnitelnosti formule (SAT).

Ukážeme návrh algoritmu, který je adaptací DPLL algoritmu pro R-OBDD důkazový systém. Přikládáme důkaz jeho korektnosti a ukážeme, že jeho běh nad nesplnitelnou formulí je možné transformovat do stromového důkazu v R-OBDD systému.

Klíčová slova: důkazový systém, rezoluce, OBDD, výroková logika

Title: Strong Proof Systems
Author: Ondrej Mikle-Barát
Department: Department of Software Engineering
Supervisor: Prof. RNDr. Jan Krajíček, DrSc.
Supervisor's e-mail address: krajicek@math.cas.cz
Abstract:

R-OBDD is a new Cook-Reckhow propositional proof system based on combination of OBDD proof system and resolution proof system. R-OBDD has the strength of OBDD proof system – hard tautologies for resolution like PHP_n or Tseitin contradictions have polynomially sized proofs in R-OBDD (R-OBDD p-simulates OBDD proof system as well as resolution). On the other hand, inference rules of R-OBDD are designed to be similar to inference rules of resolution, thus allowing to create a modified version of DPLL algorithm and possibly using heuristics used in various DPLL-like algorithms. This gives a possibility for a SAT solver more efficient than SAT solvers based on resolution proof system.

We present design of a SAT solver, which is an adaptation of DPLL algorithm for the R-OBDD proof system. The algorithm is accompanied with proof of its correctness and we show that the run of the algorithm on an unsatisfiable formula can be transformed into tree-like refutation in the R-OBDD proof system.

Keywords: proof system, resolution, OBDD, propositional logic

Chapter 1

Preliminaries

1.1 Propositional logic

Propositional logic (also called boolean logic) is a formal system where propositions are expressed using formulas written in De Morgan Language. Propositions can be also thougt of as boolean-valued functions, i.e. with values in the boolean set $B = \{0, 1\}$ and domain of B^k for k-ary functions.

The De Morgan language used to construct and express the formulas consists of connectives, propositional variables and auxiliary symbols:

- connectives (operators): $\{\vee, \wedge, \neg\}$, meaning disjunction, conjunction and negation, respectively. Other commonly used logical connectives such as $\{\Rightarrow, \equiv\}$ (implication and equivalence) can be defined using the former three.
- brackets (,) to define operator scope
- constants 1, 0 and propositional variables: $x_1, x_2, \ldots$

We are only interested in well-formed formulas. They must be constructed recursively using the following rules:

- a propositional variable or a constant is a well-formed formula
- conjunction $(A \wedge B)$ or disjunction $(A \vee B)$ of two well-formed formulas A, B is a well-formed formula
- negation $(\neg A)$ of a well-formed formula A is a well-formed formula

In practice however, some brackets may be left out due to priority of operators: $\neg$ having the highest priority, followed by $\wedge$ and finally $\vee$ with the lowest priority.

Also, formulas are mostly expressed in one of the two *normal forms*: disjunctive normal form (DNF) or conjunctive normal form (CNF). Each of these forms is an expression built from *literals* – a variable x_i or its negation $\neg x_i$. A DNF formula is disjunction of conjunctions of the form

$$\bigvee_{j=1..m} \bigwedge_{k=1..n_j} l_{jk}$$

with l_{jk} being the literals, e.g.

$$(x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge \neg x_5 \wedge \neg x_7) \vee (x_2 \wedge \neg x_3)$$

The idea of CNF is very similar, it is conjunction of disjunctions:

$$\bigwedge_{j=1..m} \bigvee_{k=1..n_j} l_{jk}$$

with l_{jk} again being the literals. Each of the inner disjunctions of literals is also called a *clause*. An example of CNF formula would be

$$(x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_7)$$

There are also CNF variations, such as k-CNF, which is CNF with exactly k literals in each clause. Every propositional formula can be transformed into CNF or DNF, although some transformations might make the formula grow exponentially in number of literals (e.g. converting CNF to DNF). However, there exists a way to transform a formula A into a “short” CNF formula B (with some additional variables) while preserving the satisfiability. This method is called *limited extension* (see e.g. Buss [6]).

The notion of formula size and the relative growth after applying operations upon it bring us to the field of computational complexity.

1.2 Proof complexity

Definition 1.2.1 (p-time). A relation $R(x_1, x_2, \ldots, x_n)$ on $\{0,1\}^*$ is *p-time* decidable if there exists a deterministic Turing machine and a polynomial $p(x) \in \mathbb{N}[x]$ that decides R in at most $p(|x_1| + |x_2| + \cdots + |x_n|)$ steps.

Definition 1.2.2 (Cook-Reckhow [8]). A propositional *proof system* for language $L \subseteq \{0,1\}^*$ is a binary relation $P(x,y)$ satisfying the following conditions:

1. Completeness: $x \in L \Rightarrow \exists y : P(x,y)$
2. Soundness: $\exists y : P(x,y) \Rightarrow x \in L$
3. p-verifiability: $P(x,y)$ is a p-time decidable relation

Relation $P(x,y)$ can be interpreted as "y is *P-proof* of x". We will focus on the language $L = TAUT$, the set of all tautologies in De Morgan language. The original definition of Cook and Reckhow used a function $f : \{0,1\}^* \rightarrow TAUT$ as the proof system, the relational definition is equivalent.

Definition 1.2.3 (Cook-Reckhow [8]). Proof system $P(x,y)$ is *p-bounded* iff there exists polynomial $p(x) \in \mathbb{N}[x]$ such that $\forall x, y \in \{0,1\}^*$:

$$P(x,y) \Rightarrow \exists z(|z| \leq p(|x|)) : P(x,z)$$

Definition 1.2.4 (Cook-Reckhow [8]). Let P, Q be two propositional proof systems. Proof system P *p-simulates* Q if there exists a p-time computable function $g : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ such that $\forall x, y \in \{0,1\}^*$:

$$Q(x,y) \Rightarrow P(x, g(x,y))$$

The g function "translates" Q-proofs into P-proofs. Since g is p-time, the size P-proofs may grow at most polynomially.

There is a tight connection between proof complexity and general computational complexity as expressed by the following definition and theorem.

Definition 1.2.5 (Cook [7]). $\mathcal{NP}$ is a class of languages admitting a p-bounded proof system. *co*$\mathcal{NP}$ is the class of complements of $\mathcal{NP}$-languages.

Theorem 1.2.6 (Cook-Reckhow [8]). *$\mathcal{NP}$ = co$\mathcal{NP}$ iff the set of propositional tautologies TAUT admits a p-bounded proof system.*

Proof. The task of deciding whether a formula $\varphi \in TAUT$ is a problem belonging to *co*$\mathcal{NP}$ (the assignment of variables showing that $\varphi \notin TAUT$ is polynomially long). Let P be p-bounded propositional proof system with bound $p(x)$. Under such circumstances, the problem of belonging to $TAUT$ is also a $\mathcal{NP}$ problem, since the string witnessing the fact of belonging to $TAUT$ (the proof) is bounded by $p(x)$.

The other way round, if $\mathcal{NP} = co\mathcal{NP}$, the problem of belonging to $TAUT$ is in *co*$\mathcal{NP}$ as well as in $\mathcal{NP}$, so there exists a p-bounded proof system. $\square$

Length-of-proofs lower bounds for various proof systems suggest that $\mathcal{NP} \neq co\mathcal{NP}$ and if we believe so, no propositional proof system is p-bounded. The question $\mathcal{NP} \overset{?}{=} co\mathcal{NP}$ is unlikely to be solved by means of exhaustive search by proving lower bounds for various proof systems, however it may expose some yet unknown structure in the nature of proofs.

1.3 Resolution

Resolution is a proof system with a single simple rule. Nevertheless, it is currently the only system with a known algorithm better than brute-force backtracking. Resolution proves tautologies in DNF form. In fact, resolution does not prove tautologies directly, instead the formula is first negated and resolution is used to *refute* it. Note that negating a DNF formula "turns" it into a CNF formula (if we propagate the outside negation using De Morgan's rules). Also, formulas to be proved in different form than CNF can be first negated and then transformed into CNF using the mentioned limited extension while preserving (un)satisfiability.

Resolution proof system operates on clauses, since the formula to be refuted is in CNF. We will think of the clause $l_1 \vee l_2 \vee \cdots \vee l_n$ as a set $\{l_1, l_2, \ldots, l_n\}$. The only *inference rule* is resolving two clauses $C \cup \{\neg x\}$ and $D \cup \{x\}$:

$$\frac{C \cup \{\neg x\} \qquad D \cup \{x\}}{C \cup D}$$

We say that the variable x has been *resolved*, $C \cup D$ is called the *resolvent.* Only one variable can be resolved at a single step.

It is quite easy to see that the resolution rule is sound – regardless of the value of x, if both clauses $C \cup \{\neg x\}$ and $D \cup \{x\}$ were satisfied by the same assignment, then $C \cup D$ must be satisfied as well.

The goal of *refutation* is to derive an empty clause, which is not satisfiable.

Definition 1.3.1 (Resolution proof). Let A be a DNF formula, $B = \neg A$ in CNF form, B_t the clauses of B. A *resolution proof* of a DNF formula A is a sequence of clauses C_i, $i = 1, \ldots, m$ such that:

- last clause C_m is the empty clause $\emptyset$
- each intermediate clause C_i is either one of B_t or a clause derived using resolution rule from some $C_j, C_k,\ j, k < i$

Example 1.3.2. An example of resolution proof of formula A

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

$C_1 = \{a, b, \neg d\}$	B_1
$C_2 = \{a, b, c, d\}$	B_2
$C_3 = \{a, b, c\}$	resolved from C_1, C_2
$C_4 = \{\neg a\}$	B_4
$C_5 = \{b, c\}$	resolved from C_3, C_4
$C_6 = \{\neg b, c\}$	B_3
$C_7 = \{c\}$	resolved from C_5, C_6
$C_8 = \{\neg c\}$	B_5
$C_9 = \emptyset$	resolved from C_7, C_8

Theorem 1.3.3 (Soundness and completeness). *DNF formula A is provable in resolution proof system iff it is a tautology.*

Proof. Soundness: resolution rule is sound, if an assignment satisfies the hypotheses, then it also safisfies the resolvent. Thus the last clause is satisfied if the initial clauses can be satisfied by an assignment. Since the empty clause in refutation cannot be satisfied, there is no satisfying assignment for the initial clauses either.

Completeness: Suppose A is tautology, formula B and clauses B_t are defined as in definition 1.3.1. We will prove completeness by induction on number of distinct variables n in A. In case of a single variable l_1, there is only one possibility for B: $\{l_1\}, \{\neg l_1\}$. Single application of resolution rule derives $\emptyset$.

For $n > 1$ pick variable l_n. Split the clauses B_i into disjoint sets:

- $\mathcal{B}$ with clauses containing neither l_n, nor $\neg l_n$
- $\mathcal{B}_l$ with clauses containing literal l_n, not $\neg l_n$
- $\mathcal{B}_{\neg l}$ with clauses containing literal $\neg l_n$, not l_n

Then, resolve each clause from $\mathcal{B}_l$ and $\mathcal{B}_{\neg l}$ against each other, forming new set of clauses $\mathcal{C}$ without the variable l_n. Add elements of $\mathcal{B}$ to $\mathcal{C}$. The set $\mathcal{C}$ contains no clauses with l_n and has $n-1$ variables.

Set $\mathcal{C}$ is unsatisfiable – if it was satisfiable, then an assignment v to variables $\{l_1, \ldots l_{n-1}\}$ would satisfy either all $\{C_i | C_i \cup \{l_n\} \in \mathcal{B}_l\}$ or all $\{D_j | D_j \cup \{\neg l_n\} \in \mathcal{B}_{\neg l}\}$. Otherwise there would exist two clauses C_i, D_j not satisfied simultaneously. Thus we could add a suitable value for l_n to v that would also satisfy set of B_t and B – contradiction with unsatisfiability of B. Finally, the existence of refutation for $\mathcal{C}$ follows from induction hypothesis. □

1.3.1 Pigeonhole principle - PHP_n

Pigeonhole principle (PHP) states that, given two natural numbers n and m with $n > m$, if n items are put into m pigeonholes, then at least one pigeonhole must contain more than one item. More formally, it says that every function $f : N \to M, |N| > |M|$ is non-injective. PHP is a widely studied concept in propositional proof complexity. It is one of “hard” tautologies, a tautology that has exponential size proofs in certain proof systems.

Let us have language with relation $R(x, y)$ and constant 0 on universe $M, |M| = n$. We will define a property of a relation on $\{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$ by the first-order formula:

$$\begin{gathered}
\exists x \forall y : \neg R(x, y) \vee \\
[\exists x_1, x_2, y : x_1 \neq x_2 \wedge R(x_1, y) \wedge R(x_2, y)] \vee \\
[\exists x, y_1, y_2 : y_1 \neq y_2 \wedge R(x, y_1) \wedge R(x, y_2)] \vee \\
\exists x : R(x, 0)
\end{gathered}$$

Translated to words, the disjunctions above mean: either there is an x mapped to no y or at least two distinct x_1, x_2 are mapped to the same y or one x is mapped to at least two distinct y_1, y_2 or there exists mapping x to constant 0.

For every $n \geq 1$ we can translate the formula into propositional form. First, replace $\exists$ and $\forall$ with disjunction and conjunction respectively over all elements of $\{0, \ldots, n-1\}$, replace true statements with 1 and false statements with 0. Finally replace $R(i, j)$ with new atoms r_{ij} whose truth assignments reflect the relation $R(i, j)$. The formula in propositional form then has form:

$$\bigvee_i \bigwedge_j \neg r_{ij} \;\vee\; [\bigvee_{i_1<i_2} \bigvee_j r_{i_1 j} \wedge r_{i_2 j}] \vee$$
$$[\bigvee_i \bigvee_{j_1<j_2} r_{ij_1} \wedge r_{ij_2}] \;\vee\; \bigvee_i r_{i0}$$

By forbiding j to range over 0, we get the mapping from n element set to $n-1$ element set and we can delete the last disjunct (it evaluates to false), thus the propositional translation PHP_n would be:

$$\bigvee_i \bigwedge_{0<j} \neg r_{ij} \;\vee\; [\bigvee_{i_1<i_2} \bigvee_{0<j} r_{i_1 j} \wedge r_{i_2 j}] \vee$$
$$[\bigvee_i \bigvee_{0<j_1<j_2} r_{ij_1} \wedge r_{ij_2}]$$

It has been proven that PHP_n has exponentially long proof in resolution:

Theorem 1.3.4 (Haken [11]). *The lower bound for a proof of PHP_n in the resolution proof system is $exp(c \cdot n)$ for some constant $c > 0$.*

Chapter 2

OBDD proof system

The proof system introduced by Atserias et al. [1] views the satisfiability problem for boolean formulas in CNF as an instance of constraint satisfaction problem (CSP).

2.1 OBDD

The OBDD proof system is a case of CSP using *ordered binary decision diagrams* (OBDD) as the syntactic representation of boolean functions. OBDD is a rooted directed acyclic graph comprised of decision nodes and two terminal nodes – 0-terminal (false) and 1-terminal (true). Terminal nodes have no children. Each decision node is labeled by a variable name and has two children called high child and low child. The edge from a node to a low (high) child represents an assignment of the variable to 0 (1). The assignment to the variables thus determines an unique way to one of the terminal nodes (the result of the function). For an example of OBDD, see figure 2.1.

The ordering is an additional requirement that the variables on every path are consistent with some linear ordering of variables. Each OBDD can be transformed into *reduced* form that is the smallest OBDD computing the function as shown by Bryant [5]. Unless stated otherwise, we will only focus on reduces OBDDs.

Constructing an OBDD from a clause

Every clause has a short equivalent OBDD. It is straightforward to see that an OBDD constructed from disjunction of n literals will have the size $n + 2$: one node for each variable with one edge leading to terminal 1 and the other leading to the next variable (except for the last variable, where one edge leads to terminal 0). Note that the size of the OBDD is independent of the variable ordering. See figure 2.1 for an example.

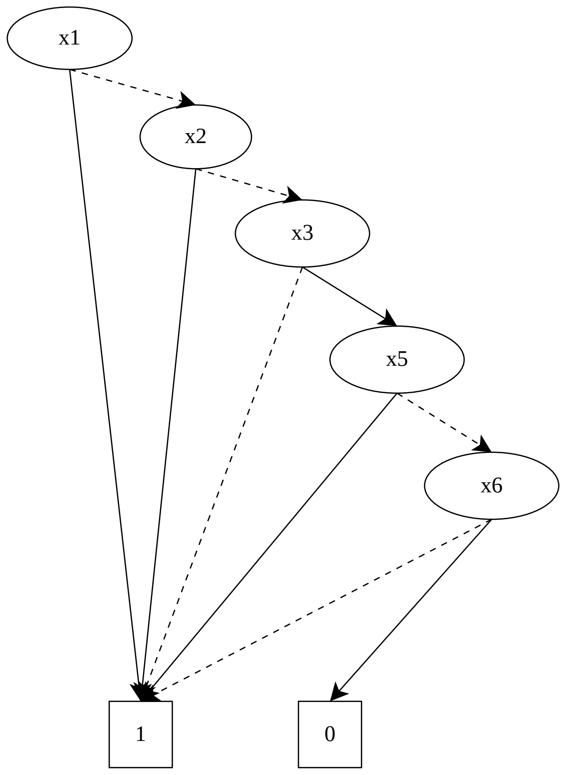

Figure 2.1: OBDD for clause $\{x_1, x_2, \neg x_3, x_5, \neg x_6\}$, dashed lines mean assignment of variable to 'false'

OBDD properties

Some properties of OBDDs that we shall use are included in the following list. Note that we assume some fixed ordering has been chosen.

- transforming OBDD into reduced form is p-time (similar to Moore reduction procedure in finite automata)
- every boolean function can be represented by a unique reduced OBDD, reduced form is canonical, thus equivalence check of two OBDDs is p-time

- for each OBDDs A, B the operations constructing new OBDDs equivalent to $A \vee B, A \wedge B, \neg A$ are p-time. Those operations can be viewed as union, intersection and complement of finite automata, first two having quadratic bound on number of new nodes and complement being just switching the 0-terminal and 1-terminal.
- decision whether OBDD A majorizes another OBDD B is p-time

The size of OBDD is determinded both by the function represented as well as the chosen variable ordering. Depending on the ordering the OBDD for the same function may vary in size from linear to exponential growth in respect to number of variables. Finding the optimal variable ordering is a NP-hard problem in general [4].

2.2 Inference rules

Let A, B, C be OBDDs and let $A[\mathbf{x}]$ express that A depends on set of variables $\mathbf{x}$. Denote $A(v)$ the result of A with an assignment of v to the variables $\mathbf{x}$, $v \cup \{x_i = e\}$ is extending the assignment v with an assignment of $e \in \{0,1\}$ to the variable x_i. The three inference rules are defined as follows:

$$\text{join} \qquad \frac{A[\mathbf{x}] \quad B[\mathbf{y}]}{C[\mathbf{x} \cup \mathbf{y}]} \quad \text{where } C = A \wedge B \tag{2.1}$$

$$\text{projection of } x_i \qquad \frac{A[\mathbf{x}]}{C[\mathbf{x} \backslash \{x_i\}]} \quad \text{where } C(v) \Leftrightarrow \exists e \in \{0,1\} : A(v \cup \{x_i = e\}) \tag{2.2}$$

$$\text{weakening} \qquad \frac{A[\mathbf{x}]}{C[\mathbf{x}]} \quad \text{where } A \Rightarrow C \tag{2.3}$$

The join rule is straightforward, weakening is relaxing the constraints on satisfiability of an OBDD. Projecting a variable x_i out of A means that the resulting OBDD C is satisfied with an assignment v whenever A is satisfied with assignment $v \cup \{x_i = e\}$ for some value $e \in \{0,1\}$. If we viewed the satisfying assignments for A as rows in a relational database with columns labeled by variables, the satisfying assignments for C would be the same after "cutting off" the column labeled x_i. In the same way the join rule can be viewed as natural join and weakening as adding additional rows to the relation table.

All the three inference rules are special case of “short-form” semantic rule:

$$\frac{A \quad B}{C} \quad \text{such that } A \wedge B \Rightarrow C$$

We can also say that C majorizes $A \wedge B$. Nonetheless the three explicit rules are probably better in practical applications, since the shorthand rule does not explicitly denote the resulting OBDD.

Lemma 2.2.1 (Atserias, Kolaitis, Vardi [1]). *OBDD proof system p-simulates resolution proof system.*

Lemma 2.2.2. *OBDD proof system is Cook-Reckhow proof system.*

Proof. Soundness and completeness follows from p-simulation of resolution, p-verifiability follows from OBDD properties as described before. □

2.3 Strength of OBDD proofs

OBDD proof system has shown itself to be very strong. Even without the use of weakening it can p-simulate resolution, using only join and projection [1]. OBDD proof system also p-simulates numerous other proof systems, such as Gaussian calculus proof system (described in [2]) and CP* (cutting planes with coefficients encoded in unary, introduced in [9]). However, it would be interesting to know if OBDD proof system can simulate cutting planes with binary coefficients, which is unknown to-date.

Moreover, there exists a particular system of equations known as *Tseitin contradictions* that is exponentially hard for resolution, but not for OBDD proof system [1]. Thus OBDD proof system is exponentially stronger than resolution.

Nevertheless, there still are hard tautologies for OBDD proof system as well. A lower bound has been proven using feasible interpolation by Krajíček [12].

Chapter 3

R-OBDD

3.1 Motivation

Resolution (its tree-like form) is one of the weakest of commonly used proof systems. However, it is the only system that can be reasonably used in computer-aided proof search. All stronger proof systems offer at most brute-force backtrack algorithms. While we know that resolution has exponentially longer proofs in some cases compared to those strong proof systems, the resolution proof-search algorithms are better in actual finding proofs with current state of knowledge.

Therefore we attempt to create a system somewhat syntactically and semantically similar to resolution. We hope that DPLL algorithm [10] and its heuristics could be used for this stronger proof system (with some modifications).

3.2 Definitions

Definition 3.2.1 (OBDD-clause). OBDD-clause is a disjunction of OBDDs. Size of an OBDD-clause is the sum of size of OBDDs in the OBDD-clause.

Constructing an OBDD-clause

Each clause can be transformed into multiple OBDD-clauses depending on how many literals do we "group" into each OBDD. It is possible to represent each literal by a separate OBDD as well as represent the entire clause by a single OBDD or anything in between.

3.3 Inference rules

R-OBDD defines following rules, with Γ and Δ being disjunction of OBDDs from a OBDD-clause.

$$\frac{\Gamma \vee P_1 \quad \Delta \vee P_2}{\Gamma \vee \Delta \vee P_3} \qquad \text{such that } P_3 \text{ is join of } P_1, P_2 \tag{3.1}$$

$$\frac{\Gamma \vee P}{\Gamma \vee Q} \qquad \text{such that } Q \text{ is projection of } P \tag{3.2}$$

$$\frac{\Gamma \vee P}{\Gamma \vee Q} \qquad \text{such that } Q \text{ is weakening of } P \tag{3.3}$$

$$\frac{\Gamma \vee P \vee P}{\Gamma \vee P} \tag{3.4}$$

$$\frac{\Gamma \vee P_1 \vee P_2}{\Gamma \vee P_3} \qquad \text{where } P_3 = P_1 \vee P_2 \tag{3.5}$$

$$\frac{}{1} \tag{3.6}$$

The first three rules are equivalents of the inference rules of OBDD proof system – join, projection and weakening. The rule 3.4 could be omitted if we treated the OBDD-clauses as sets.

Note that the rule 3.5 is not necessary, it is provided for convenience. The rule can be derived using following steps:

$$\frac{\Gamma \vee P_1 \vee P_2}{\Gamma \vee P_1 \vee P_3} \qquad \text{where } P_3 = P_1 \vee P_2 \text{ (weakening of } P_2 \text{ - rule 3.3)}$$

$$\frac{\Gamma \vee P_1 \vee P_3}{\Gamma \vee P_3 \vee P_3} \qquad \text{(weakening of } P_1 \text{ - rule 3.3)}$$

$$\frac{\Gamma \vee P_3 \vee P_3}{\Gamma \vee P_3} \qquad \text{(reducing duplicate } P_3 \text{ - rule 3.4)}$$

3.4 The proof system

Definition 3.4.1 (R-OBDD proof). Let A be a formula in DNF, $B = \neg A$ consisting of clauses $C_1, \ldots, C_k$. An R-OBDD proof of A is a sequence $\Gamma_1, \ldots, \Gamma_m$ of OBDD-clauses where each Γ_i is either an OBDD-clause constructed from C_j or derived from Γ_s, Γ_t for some $s, t < i$ such that $\Gamma_m \equiv 0$,

i.e. Γ_m is an unsatisfiable OBDD with a single node – the 0-terminal. This process is called *R-OBDD refutation.*
Size of R-OBDD proof is the sum of sizes of all Γ_i used in proof, length of the proof is m.

Lemma 3.4.2. *R-OBDD p-simulates OBDD proof system.*

Proof. OBDD proof system is a special case of R-OBDD proof system – every line in proof consists of just one OBDD. Use of the first three rules is sufficient to simulate all OBDD inference rules. □

Theorem 3.4.3 (Soundness and Completeness). *Let A be a DNF formula, $B = \neg A$ in CNF. Then A has R-OBDD proof if and only if it is tautology.*

Proof. Soundness: Suppose we have R-OBDD proof of A – refutation of B. In each inference rule, if an assignment satisfies the hypotheses, then it also safisfies the derived OBDD clause. Thus the last OBDD clause is satisfied if the initial OBDD clauses can be satisfied. The last OBDD clause in the proof is unsatisfiable OBDD with only a 0-terminal, therefore initial OBDD clauses cannot be satisfied.

Completeness: Suppose A is tautology and $B = \neg A$ is in CNF with clauses $\mathcal{C}$. To see that A has R-OBDD proof, it we can use the same technique as described in proof of theorem 1.3.3 with minor adjustments: instead of clauses with literals we use OBDD clauses constructed from $\mathcal{C}$ where each literal is represented by a separate OBDD and resolution rule is replaced by rule 3.1 which derives 0 from the complementary literals. The 0 in the derived OBDD clause may then be absorbed using rule 3.5. This is basically p-simulation of resolution. □

It is possible to prove completeness in a broader sense:

Theorem 3.4.4 (Generalized completeness). *Every unsatisfiable set of OBDD clauses can be refuted.*

Proof. Suppose we have such set $\{\Gamma_1, \ldots, \Gamma_m\}$. Create a new set set of OBDD clauses $\{\Delta_1, \ldots, \Delta_m\}$ by applying repeatedly the rule 3.5 to each Γ_i until it consists only of a single OBDD and let us call the result Δ_i. Suppose $\Gamma_i = \{P_1, \ldots, P_k\}$. The derivation of Δ_i is:

$$\frac{P_1 \vee P_2 \vee \cdots \vee P_k}{\Pi_2 \vee P_3 \vee P_4 \vee \cdots \vee P_k}$$

$$\frac{\Pi_2 \vee P_3 \vee P_4 \vee \cdots \vee P_k}{\Pi_3 \vee P_4 \vee \cdots \vee P_k}$$

$$\vdots$$

$$\frac{\Pi_{k-1} \vee P_k}{\Delta_i}$$

Now we can use the first three inference rules to p-simulate OBDD proof system on the set of $\{\Delta_1, \ldots, \Delta_m\}$ (actually, for purposes of this proof it would be sufficient to use the rule 3.1 repeatedly as join until we derive unsatisfiable OBDD). As a final note, the size of each Δ_i may be exponential in comparison to Γ_i it has been derived from. □

Lemma 3.4.5. *R-OBDD is a Cook-Reckhow proof system.*

Proof. Soundness and completeness has been proven in Theorem 3.4.3. The p-verifiability follows from p-computability of various operations on OBDDs mentioned in chapter 2. □

Chapter 4

Automated theorem proving in R-OBDD

The similarity between R-OBDD and resolution proof systems allows us to modify DPLL algoritm for theorem proving in R-OBDD. However, we will not go into implementation details, since the fundamental part is the new proof system, but we will describe the core of the algorithm.

The original DPLL algorithm can be expressed by the following pseudocode with `Phi` being the CNF formula and `Con` the partial assignment to propositional variables (empty at the beginning, "Con" meaning constraints):

```
function DPLL(Phi, Con)
    if (Phi=True)
       then return True;
    if (Phi=False)
       then return False;
    if (unit clause L occurs in Phi)
       then return DPLL(assign(L,Phi), append(Con,L));
    if (literal L occurs pure in Phi)
       then return DPLL(assign(L,Phi), append(Con,L));
    L := chooseLiteral(Phi);
    return DPLL(assign(L,Phi), append(Con,L)) OR
        DPLL(assign(negate(L),Phi), append(Con, negate(L)));
endfunction
```

The literal `L` is either in positive form v or negative form $\neg v$ for some variable v. Function `assign(L, Phi)` returns formula `Phi` where it replaces every positive occurence of `L` by `True` and every negative occurence with `False` and returns the simplified formula. Simplifying means removing clauses that evaluate to `True` and removing literals that evaluate to `False`. If a clause becomes empty, the whole formula evaluates to `False`.

An *unit clause* is a clause with only a single literal (thus there is only one way of assigning it) and *pure literal* is a literal which occurs only in one form (positive or negative) throughout the whole formula.

Function `negate(L)` negates the literal `L` and `append(Con,L)` means extending the assignment `Con` by assigning to literal `L`. Assignment to `L` is `True` for literal in positive form and `False` otherwise.

The only function left is `chooseLiteral` which chooses the next literal to be assigned and further branching depends on that literal. There are numerous ways and heuristics to decide the next branching literal. Note that the choice of branching literal strongly affects efficiency.

4.1 R-OBDD solver – DPLL modification for R-OBDD

The DPLL adaptation for R-OBDD proof system works in a similar way like classic DPLL. The input for the algorithm are the clauses of the CNF formula. The first step consists of partitioning the literals in the clauses into OBDDs. There are two special cases – if every OBDD contains just one literal, the algorithm becomes a classic DPLL, in the other case each clause is represented by a single OBDD, which in turn resembles the OBDD proof system. Otherwise any combination of literals in a OBDD is possible. We will discuss the efficiency impact of this partitioning later.

The fundamental change is in the branching part of the algorithm – instead of branching upon assigning a literal, branching is done upon "assigning" 0 or 1 to an OBDD P, we will call this "assignment" a *constraint* or *assumption.* The branching part of the algorithm then becomes:

1. choose an OBDD P from an OBDD clause for the next constraint
2. assume the value of OBDD P is 1 or 0
3. let $R := P$ if $P = 1$, and $R := \neg P$ otherwise, R is the constraint
4. take each clause containing an OBDD Q such that $R \Rightarrow Q$ and delete the clause (since R is satisfied, so is Q and the whole OBDD clause containing Q is satisfied)
5. every OBDD Q sharing at least one variable with R is replaced by join of R, Q

6. any unsatisfiable OBDD in any clause is deleted
7. if any clause contains no more OBDDs, it is unsatisfiable, thus we return false in this level of recursion

The set of assumptions is used later to output satisfying assignment in case of satisfiable formula. When the formula is found to be satisfiable, the constraints are partitioned into a number of disjunct subsets $K_1, \dots, K_l$ so that no OBDD from set K_i shares any variable with any OBDD from set K_j for $i \neq j$. To find the assignment for variables used by OBDDs in set K_i, all of the OBDDs in the set are joined into a single OBDD S_i and any path from the 1-terminal up to the root can be used as the assignment to the variables. Of course, it is best to partition the set of constraints into as many subsets K_i as possible, since it affects the size of each S_i.

To sum it up, the new `ROBDDSolver` algorithm could be described with the following pseudocode. In this case, `Phi` is an array of OBDD clauses and `Con` a set of constraints made during the computation.

```
function ROBDDSolver(Phi, Con)
    if (Phi=True)
       then return True;
    if (Phi=False)
       then return False;
    if (unit OBDD P occurs in Phi)
       then return DPLL(assign(P,Phi), append(Con,P));
    P := chooseOBDD(Phi);
    return DPLL(assign(P,Phi), append(Con,P)) OR
        DPLL(assign(negate(P), Phi), append(Con, negate(P)));
endfunction

function assign(R, Phi)
    for each (OBDD Q in Phi)
        if (R implies Q)
            then delete clause containing Q;
        if (Q shares variable with R)
            then Q := OBDDJoin(Q, R);
        if (Q=False)
            then delete(Q);
    return Phi;
endfunction
```

The only new function is `OBDDJoin` which returns join of two OBDDs and `chooseOBDD` which is analogy of `chooseLiteral` in DPLL. The meaning of `assign` and `negate` functions is analogous to their meaning in the original DPLL algorithm, except they operate on OBDDs. Negating of an OBDD is simply switching 1 and 0-terminals, while `assign` is extended to accomodate new possible states in R-OBDD proof system. Note that the `ROBDDSolver` algorithm has no analogy for pure literals – in the R-OBDD case it is much less likely to find a "pure OBDD". Also, pure literal elimination is not used much in DPLL variants nowadays.

Theorem 4.1.1. *R-OBDD solver algorithm is correct and terminating – outputs satisfying assignment iff the input formula is satisfiable.*

Proof. Termination: The depth of the recursion tree is bounded by the number of all OBDDs in the OBDD clauses. At every node of the recursion tree there are at most two possible branches of computation, thus the number of nodes in computation tree is bounded from above by two raised to the number of all OBDDs in the OBDD clauses.

Correctness: At every branching step, an OBDD P is chosen and is assumed to be either 0 or 1. In the first case, $R := \neg P$ is added to the set of constraints, $R := P$ otherwise. Other OBDDs that might be affected by such constraint are exactly those sharing at least one variable with P. The newly created OBDDs will not cause missing any solution – the new OBDDs have exactly the same solution as before, constrained by the requirement of validity of R. This fact holds because join of two OBDDs is equivalent to boolean conjunction. The same way replacing an implied OBDD Q with 1 (and later deleting the clause containing it) is correct since any assignment that satisfies R also satisfies Q. Deleting OBDDs evaluated to 0 and deleting clauses containing OBDD evaluated to 1 will not change the set of solutions, since already evaluated OBDDs cannot change value deeper in the recursion tree.

Also, variable ordering only affects the size of created OBDDs, not the function they represent. The set of constraints may never contain contradicting OBDDs (i.e. OBDDs whose join would be always 0) since all such constraints already have been applied before choosing the next constraint.

The only operation that affects the set of solutions is making the assumptions about value of OBDD P, but one solution is sufficient. If the assumption is not correct, algorithm obtains an empty OBDD clause deeper in the recursion tree and is forced to backtrack, choosing the other variant of constraint. In case of unit OBDDs there is obviously only one constraint that might lead to satisfiable assignment. □

Theorem 4.1.2. *Given unsatisfiable input formula F in CNF, the computation of `ROBDDSolver` on on F can be transformed into R-OBDD refutation of F.*

Proof sketch. The idea of converting the run of `ROBDDSolver` into R-OBDD refutation is similar to converting run of `DPLL` into resolution refutation. Formally the proof would be based on induction on recursion depth showing which invariants are valid at a given depth, but is it easier to understand if we look at the invariants and how the R-OBDD proof would be constructed.

First let us assume that each clause and each OBDD has some unique identifier (e.g. a number) at the beginning. If an OBDD is joined with a constraint, its number does not change, and deleting a clause or an OBDD does not affect their number identifiers either.

Whenever the algorithm encounters an empty OBDD clause in the recursion tree, it remembers which OBDD clause became empty and the identifier of the last OBDD that evaluated to 0 at this step, then backtracks. This way every leaf in the recursion tree is labeled by an OBDD clause and an OBDD and every branch in the tree is labeled by the constraint chosen in the given step. See figure 4.1.

Now, let us make some observations: take any two leaves with common parent, e.g. $(K_1, O_1), (K_2, O_2)$ in our example figure 4.1. The constraint applied in the last algorithm branching was $C = 0$ for one leaf and $C = 1$ for the other leaf. Apply join (rule 3.1) on K_1, K_2 with $P_1 = O_1, P_2 = O_2$, name the result $J_{1,2}$ and remember which OBDD in $J_{1,2}$ is the result of join of O_1, O_2, name it $O_{1,2}$.

If we applied (using join) all the other constraints ($A = 0, B = 0$) leading to these leaves before the branching on C to $J_{1,2}$, it would evaluate to false independent of value of C. In the same manner we would join K_3, K_4 into $J_{3,4}$, then $J_{1,2}, J_{3,4}$ into $J_{1,4}$. Note that $J_{1,4}$ does not depend on value of B anymore.

Using these joins, we are moving "upwards" in the computation tree. Each of the $J_{x,y}$ is an invariant and does not depend on any of the constraints

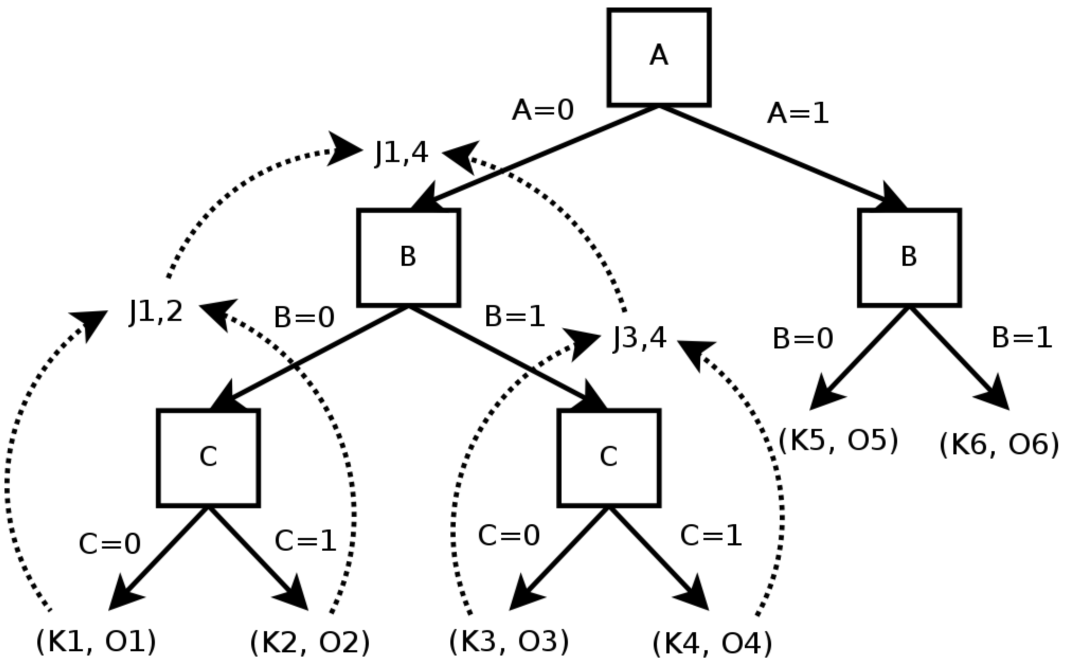

Figure 4.1: Example of `ROBDDSolver` computation tree – solid lines representing computation branches are labeled by the constraints made (for some OBDDs A, B, C), leaves are labeled using a pair of (K_x, O_y) denoting which clause K_x became false and which OBDD O_y was evaluated to false as the last one with respect to constraints used. Dotted lines represent joins of OBDD clauses while turning the computation into R-OBDD refutation (not all of them are drawn due to space limitation).

below in the computation tree. Each of the $J_{x,y}$ is false when joined with the contraints leading to the root of the tree.

If some unit OBDD U occured in an OBDD clause in the computation tree, the constraint used must have been $U = 1$ ($U = 0$ would have forced the whole OBDD clause containing U to false immediately, so it has no siblings). Nevertheless, independent of value U at least one OBDD clause K_z below in the tree evaluated to false, so K_z is invariant towards value of U and can be "moved upwards" in the tree without join with any other OBDD clause.

Following our observations, in order to turn the computation tree of `ROBDDSolver` on F into R-OBDD refutation:

1. let $\mathcal{J}_0$ contain all the OBDD clauses from F

2. let $i := 1$
3. Denote current height of computation tree h. Take each pair of leaves with common parent K_x, K_y at the depth h and apply join (rule 3.1) on them as described in the paragraph above. This yields a set $\mathcal{J}_i$ of new OBDD clauses.
4. Take each leaf with no sibling and add it to $\mathcal{J}_i$ (case of unit OBDDs).
5. Delete each pair of leaves processed in previous steps and replace the parent with the result of the join from $\mathcal{J}_i$. In case of leaf with no sibling, delete the leaf and move its label onto its parent.
6. $i := i + 1$
7. if the tree has any leaves, go to step 3

The sets $\mathcal{J}_i$ contain a tree-like R-OBDD refutation of F. Since each of the $J_{i,k} \in \mathcal{J}_i$ is false when joined with the constraints on the path to the root node, once we work the way up to the tree, there are no more constraints above. Thus it yields and empty OBDD clause and F is refuted.

□

4.2 Discussion

In order to devise an actual `ROBDDSolver` algorithm implementation, we would need to develop ways to decide variable ordering in the OBDDs and how to choose the OBDD for the next constraint (`chooseOBDD` function). Both variable ordering and constraint selection have tremendous impact on the algorithm speed and memory footprint.

Even most variations on classical DPLL algorithm focus on similar topic – how to best choose the literals during computation. One noteworthy DPLL-like algorithm is the Chaff method [13]. Aside from literal-choosing heuristics it features fast unit propagation, clause learning and "soft restarts", making it possible to find non-treelike resolution proofs. One of its implementations, zChaff, ranked well in competitions in the past [3].

At the moment, the `ROBDDSolver` does not make use of weakening and projection in the R-OBDD proof system. First of all, it is somewhat tricky, because in contrast to join there are many possibilities of a single weakening or projection. However, it could yield better performance and shorter proofs.

Also, there are some cases when the memory usage while printing the satisfying assignment using `ROBDDSolver` may grow exponentially. One such case would occur if each clause from the original formula would be represented by a single OBDD. In such case, join of all the OBDD clauses would have to be computed, potentially leading to exponential memory consumption. This is another reason why working implementation of `ROBDDSolver` would require good OBDD-partitioning heuristics.

Bibliography

[1] Albert Atserias, Phokion G. Kolaitis, and Moshe Y. Vardi. Constraint propagation as a proof system. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2004.

[2] Eli Ben-Sasson and Russell Impagliazzo. Random CNF's are hard for the polynomial calculus. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 415, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Daniel Le Berre and Laurent Simon. SAT solver competitions, 2007. Retrieved June 5, 2007 from http://www.satcompetition.org/.

[4] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] S. Buss. Weak formal systems and connections to computational complexity. Lecture Notes for a Topics Course, University of Califorina, Berkeley, 1988.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[8] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[9] W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Appl. Math.*, 18(1):25–38, 1987.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[11] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[12] Jan Krajíček. An exponential lower bound for a constraint propagation proof system based on ordered binary decision diagrams. Technical Report TR07-007, Electronic Colloquium on Computational Complexity, January 2007.

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.