Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 90
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 90
Espoo 2004 HUT-TCS-A90

PROOF COMPLEXITY OF CUT–BASED TABLEAUX FOR BOOLEAN CIRCUIT SATISFIABILITY CHECKING

Matti Järvisalo

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 90
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 90
Espoo 2004 HUT-TCS-A90

PROOF COMPLEXITY OF CUT–BASED TABLEAUX FOR BOOLEAN CIRCUIT SATISFIABILITY CHECKING

Matti Järvisalo

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-7020-X
ISSN 1457-7615

Multiprint Oy
Helsinki 2004

ABSTRACT: This report deals with propositional satisfiability checking. Most successful satisfiability checkers are based on the Davis–Putnam method and assume that the input formulae are in conjunctive normal form (CNF). In this work an alternative approach is considered. A tableaux–based method for a more general formula representation called Boolean circuits is introduced. The method can be seen as a generalisation of the Davis–Putnam method to Boolean circuits.

The effectiveness of the tableau method is investigated. In particular, the role of the important splitting / cut rule is considered. The effect that restrictions on the use of the cut rule have on proof complexity, i.e., on the size of proofs producible, is studied.

It is shown that restricting the application of the cut rule in any of the natural locality based ways considered causes an exponential increase in the proof complexity. Moreover, there are exponential differences between the proof complexity of all the restricted methods. The results rely on the resolution–boundedness of the methods and on properties of certain circuit families such as a Boolean circuit representation of the well–known pigeon–hole principle.

The results apply to the Davis–Putnam method for formulae in CNF obtained from Boolean circuits using Tseitin's translation. Thus it is shown that locality based cut restrictions, such as splitting on the input gates only, increase the size of proofs exponentially in the worst–case in Davis–Putnam based satisfiability checkers.

KEYWORDS: propositional satisfiability, satisfiability checking, Boolean circuits, cut rule, proof complexity, polynomial simulation, resolution, Davis–Putnam method

Contents

List of Figures

1 INTRODUCTION

In this chapter general background is given and the scope of this work is defined.

1.1 Background

The problem of finding out whether a *propositional formula* is *satisfiable*, i.e., evaluates to true with some *truth assignment*, is called the *propositional satisfiability problem* (SAT) [28]. It is an archetypical **NP**-complete problem [6], and thus hard to solve. Because of its universal nature, a variety of problems, e.g., in the areas of planning [21, 22], model checking of finite state systems [5, 4], testing [23], and verification [3], can be seen as SAT problem instances. Due to this, there is a high demand for more feasible ways of solving SAT instances, ranging from industrial applications to pure research. Various methods for checking the satisfiability of SAT instances have been developed (see [15] and [35] for surveys) and applied successfully to many interesting domains. The success builds on recent significant advances in the performance of SAT checkers based both on *stochastic local search* algorithms and on *complete systematic search*, see e.g. [30, 26, 24, 1, 25, 13]. Still, universally efficient methods are yet to be found.

Most successful satisfiability checkers assume that the input formulae are in *conjunctive normal form* (CNF) [27]. The reason for this is that it is simpler to develop efficient data structures and algorithms for CNF than for arbitrary formulae. Moreover, propositional formulae can be transformed in polynomial time (see e.g. [29]) into CNF while preserving the satisfiability of the instance. Therefore one usually employs a more general formula representation and then transforms the formula into CNF. However, such a polynomial time translation introduces auxiliary variables, and it should be noticed that an increase in the number of variables in an instance reflects in the worst–case exponentially to the performance of typical SAT checkers. On the other hand, using CNF makes efficient modelling of an application cumbersome.

In addition to being hard to use directly as a modelling language, by translating other representations to CNF one often hides information about the structure of the original problem. One way of representing propositional formulae in a more general, structure–preserving way is to use *Boolean circuits* (see e.g. [28]). Basically, Boolean circuits are acyclic directed graphs in which the nodes – representing sub-formulas of the instance – are called *gates*, and dependencies between different gates are represented by edges. Boolean circuits are interesting because they allow for a compact and natural representation in many domains, in which the representation can be simplified by sharing common subexpressions and by preserving natural structures and concepts of the domain. Boolean circuits can be translated into CNF using a standard translation, often referred to as *Tseitin's translation*, as it was first discussed in [33]. This translation introduces a new variable for each Boolean connective in the formula, resulting in a linear size CNF translation.

Recognising the factors that affect the difficulty of satisfiability checking, i.e. the time needed to determine whether an instance is satisfiable or not, giving a satisfying truth assignment in the positive case, is crucial if one is to find more efficient method for the task. The basis of most state–of–the–art SAT checkers today is the *Davis–Putnam procedure* (DPLL) [11, 12]. The efficiency of a typical DPLL based SAT checking system depends on

(i) the applied search space pruning techniques, e.g., *non-branching deduction rules*, *non-chronological backtracking* (see e.g. [25]), and *conflict-driven learning* (see e.g. [34]), and on

(ii) the *splitting rule* (i.e., on which Boolean variables to apply the *explicit cut* that induces branching, and what kind of *heuristics* is this decision based on).

Different approaches to measuring the efficiency of SAT checking methods can be considered. One can compare SAT checkers by *experimental evaluation*, i.e., investigate how long does it take for checkers to give an answer to different types of instances. There exists a variety of publicly available benchmark instance sets from different problem domains. While of practical importance, there are many difficulties concerned with such aspects as benchmark problem coverage and implementational issues in performing objective experimental evaluation. Another approach is *worst–case analysis* of SAT checking algorithms (see e.g. [10]), i.e., giving analytic proof of upper bounds on the running times of algorithms w.r.t. instance size. Heuristics applied in particular algorithms has a huge effect on the worst–case performance. Tight upper bounds are of great theoretical interest and an active area of research. Although there is potential for also breakthroughs in practise, upper bounds can be a misleading measure, as an algorithmic idea that is highly successful in practise, with very good average–case performance, does not stand out in such analysis.

A third approach, the one taken in this work, is to investigate how large are the minimal–size proofs for different formulae. This measure is called *proof complexity* (see e.g. [2]). Proof complexity is of our interest, as it allows one to differentiate the heuristic performance from the algorithmic idea and to consider how small proofs can be established assuming *optimal* heuristic behaviour. The relative proof complexity of SAT checking methods gives a way of proving in this sense the theoretical superiority of one method to another, or, showing that different methods cannot be compared to one another.

In this work we consider solving instances of Boolean circuit satisfiability. A *tableau method* for satisfiability checking that works directly with Boolean circuits has been developed [20] and implemented [19]. The idea is to avoid translating circuits to CNF. Tseitin's translation introduces a number of new variables linear to the size of the circuit. This results in increased computational complexity, as the performance of SAT checking algorithms depends in the worst–case exponentially on the number of variables in the instance. We employ Boolean circuits as the input format, thus allowing for a more structured representation than propositional formulae. Instead of standard (cut free) tableau techniques [8] the tableau method for Boolean circuits employs a direct cut rule combined with deterministic (non-branching) deduction rules. The aim is to achieve high performance while avoiding some computational problems in cut free tableaux [9, 8]. The rules in the method are related to the successful Davis–Putnam procedure, although DPLL assumes input in CNF. This gives us means of generalising the main results of this work to DPLL.

1.2 Scope and Contributions

In this work we study satisfiability checking methods for Boolean circuits. We focus on the splitting/cut rule. Namely, the research problem is:

> *How do restrictions on the use of the cut rule effect the proof complexity in Boolean circuit satisfiability checking based on tableaux?*

For instance, one may think that it is a good idea to restrict the cuts to the *input gates* only as they determine the values of all other gates. Therefore, the search space for a circuit with K gates and N input gates, $K \geq N$, would be 2^N instead of 2^K. This approach is proposed, for example, in [31, 14]. However, our results show that doing so will in the worst case result in exponentially larger proofs compared to the unrestricted cut rule. In addition to the input gate restricted cuts, we study several other natural locality based restrictions of the cut rule, e.g., "top-down" cuts that are made only on the children of the already determined gates and "bottom-up" cuts that can be applied on input gates and on the parents of the already determined gates. Our results show that restricting the cut in any of the considered ways can result in exponentially larger proofs than the unrestricted cut. In addition, we show that there are also exponential differences in the proof complexity between the restricted versions of the cut rule.

The tableau method we introduce is based on a subset of the rules in the method introduced in [20]. The set of rules in our method are closely related to those in the DPLL method. Thus the main results directly apply to SAT checkers for CNF formulae that are obtained from Boolean circuits by using Tseitin's translation. In addition to the proof complexity results, we show that the method we introduce is *sound* and *complete*, and discuss how and why the method relates to the Davis–Putnam method.

The main results presented in this work have been published and presented in an international forum [17].

1.3 Outline for the Rest of the Work

The rest of this work is organised as follows.

- Chapter 2: As preliminaries the concepts of propositional satisfiability, proof complexity, and polynomial simulation, and the resolution principle are introduced.

- Chapter 3: Boolean circuits and the related **NP**–complete Boolean circuit satisfiability problem are introduced. Sets of clauses are associated with Boolean circuits of specific kind.

- Chapter 4: A tableau method for Boolean circuit satisfiability checking called **BC** is introduced. The polynomial simulatability among variations of **BC** is discussed in the light of Boolean circuits generated from sets of clauses. The **BC** method and its variations are shown to be complete and sound proof systems for Boolean circuits. Additionally, the notion of resolution–boundedness is introduced, and **BC** and its variations are shown to have this property.

- Chapter 5: Preliminaries for the main results of the work are presented, i.e., certain circuit constructs are introduced.

- Chapter 6: The main results of the work are presented. Negative polynomial simulation results are shown to hold for each pair of the considered variations of the **BC** method. Furthermore, the relevance of the results is discussed in the light of the Davis–Putnam method.

- Chapter 7: This chapter sums up the content of the work. Related questions for further work are given.

2 PRELIMINARIES

As preliminaries the concepts of propositional satisfiability, proof complexity, and polynomial simulation, and the resolution principle are introduced.

2.1 Propositional Logic

First we introduce the *language of propositional logic*. Our definitions follows those of [27]. We use the following notation.

- $\neg$ stands for "not"(*negation*).
- $\vee$ stands for "or" (*disjunction*).
- $\wedge$ stands for "and" (*conjunction*).

The alphabet of propositional logic consists of the symbols

(i) $\neg, \vee, \wedge$ (*connectives*)[1],

(ii) (,) (*parentheses*), and

(iii) $a, a_1, a_2, \ldots, b, b_1, b_2, \ldots$ (*Boolean variables*).

Next we define what we mean by *propositions*, or *propositional formulae*.

Definition 2.1 (Propositions) *The set of* propositions *consists of exactly the following.*

(i) *Boolean variables are propositions.*

(ii) *If α and β are propositions, then $\neg\alpha$, $(\alpha \vee \beta)$, and $(\alpha \wedge \beta)$ are propositions.*

Generally, the shorthand $(\alpha_1 \vee \cdots \vee \alpha_k)$ is used for $(\cdots((\alpha_1 \vee \alpha_2) \vee \alpha_3) \cdots \vee \alpha_k)$, and similarly for $\wedge$. Moreover, if the outermost symbols in a formula are both parentheses, they are usually left out.

Next we define the semantics. The semantics is based on assigning truth values (`true` and `false`) to propositions. Let $\mathcal{V}$ be a finite set of Boolean variables. A *truth assignment* over $\mathcal{V}$ is a function $\tau : \mathcal{V} \to \{\texttt{true}, \texttt{false}\}$. A truth assignment τ is a *satisfying truth assignment* for a variable v if and only if $\tau(v) = \texttt{true}$. A truth assignment over $\mathcal{V}$ is extended to arbitrary proposition over $\mathcal{V}$ as follows.

- $\tau(\neg\alpha) = \texttt{true}$ if and only if $\tau(\alpha) = \texttt{false}$.
- $\tau((\alpha \vee \beta)) = \texttt{true}$ if and only if $\tau(\alpha) = \texttt{true}$ or $\tau(\beta) = \texttt{true}$.
- $\tau((\alpha \wedge \beta)) = \texttt{true}$ if and only if $\tau(\alpha) = \texttt{true}$ and $\tau(\beta) = \texttt{true}$.

[1]Usually, the connectives $\rightarrow$ (*implication*) and $\leftrightarrow$ (*equivalence*) are introduced, but they are generally not needed: $\alpha \rightarrow \beta$ is equivalent to $\neg\alpha \vee \beta$, while $\alpha \leftrightarrow \beta$ is equivalent to $(\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$.

If $\tau(\alpha) = \texttt{true}$ for some assignment τ, then the proposition α is *satisfiable*, and τ is a *satisfying truth assignment* for α. Otherwise, α is *unsatisfiable*. If $\tau(\alpha) = \texttt{true}$ for any τ, then α is *valid*.

For convenience we define some additional concepts. A *literal* is a Boolean variable v or its negation, $\neg v$. A *clause* over $\mathcal{V}$ is a disjunction of literals $\bigvee_{i\in\{1,\ldots,n\}} l_i$. The *length* of this clause is n. A clause of length one is a *unit clause*. A set of clauses $\varphi = \{C_1, \ldots, C_k\}$ is *satisfiable* if and only if there is a truth assignment that satisfies each C_i, $1 \leq i \leq k$. Otherwise it is *unsatisfiable*. Thus φ coincides with the *conjunctive normal form* (CNF) for propositional formulae [27], i.e. $\bigwedge_{i\in\{1,\ldots,k\}} C_i$. Notably, any propositional formula can be transformed in polynomial time (see e.g. [29]) into a CNF formula (a set of clauses) that preserves the satisfiability of the original one.

Example 2.1 *Let*

$$\mathrm{UNSAT}_{a,b} \overset{\mathrm{def}}{=} \{a \vee b, a \vee \neg b, \neg a \vee b, \neg a \vee \neg b\}$$

be a set of clauses over the Boolean variables $\{a, b\}$. *We have that* $\mathrm{UNSAT}_{a,b}$ *is unsatisfiable. If we remove any clause from* $\mathrm{UNSAT}_{a,b}$*, we get a satisfiable set of clauses. The one and only satisfying truth assignment of* $\mathrm{UNSAT}_{a,b} \setminus \{a \vee b\}$ *is* $\{a \rightarrow \texttt{false}, b \rightarrow \texttt{false}\}$. *Seen as a propositional formula in CNF,* $\mathrm{UNSAT}_{a,b}$ *is* $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$.

2.2 The Satisfiability Problem and NP-Completeness

The propositional satisfiability problem, SAT, is defined as follows.

Definition 2.2 (SAT)

Instance: A set of clauses φ.

Problem: Is φ *satisfiable?*

The propositional satisfiability problem is an archetypical **NP**-*complete* problem [28]. Informally, the acronym **NP** stands for the class of *decision problems*[2] for which it holds that the solution to any instance of the problem can be verified in polynomial time. A problem $P \in \mathbf{NP}$ is said to be **NP**-*complete*, if for any problem $P' \in \mathbf{NP}$ there is a polynomial time algorithm which, given an instance I' of P', outputs an instance of P to which the answer is "yes" if and only if the answer to I' is "yes". In this sense, **NP**-complete problems are the hardest problems in the class **NP**.

2.3 Propositional Proof Complexity and Simulation

A *propositional proof* is a certificate – seen as an instance of some general format for presenting proofs – for the validity of a proposition, or, equivalently, for the unsatisfiability of the negation of the proposition. A *propositional proof system* (see e.g. [2]) is then a polynomial–time computable predicate T such that for all propositions α it holds that α is unsatisfiable if and only if

[2]A decision problem is a problem to which the answer is either "yes" or "no".

there is a proof P for α such that $T(\alpha, P)$. If such a P exists, it is a *T-proof* for α.

We use the notion of *p-simulation* [7] to study the relative efficiency of proof systems. Let T be a proof system. The *proof complexity* (or *complexity* in short) of a proposition α in T is the minimum of $|P|$, where P is a T-proof for α and $|P|$ the *size*[3] of P. For any two proof systems T and T', we say that *T p-simulates T'*, denoted by $T \succeq T'$, if there is a polynomial $p(n)$ such that, for any α, if there is T'-proof for α of size n, then there is a T-proof for α of size at most $p(n)$. The relation $\succeq$ is transitive. If $T \succeq T'$ holds but $T' \succeq T$ does not, we write $T \succ T'$. If neither $T \succeq T'$ nor $T' \succeq T$ holds, we write $T \not\equiv T'$.

We denote by $\succeq_\chi$ the restricted form of p-simulation, in which $T \succeq_\chi T'$ holds if if there is a polynomial $p(n)$ such that, for any $x \in \chi$, if there is T'-proof for x of size n, then there is a T-proof for x of size at most $p(n)$. If both $T \succeq_\chi T'$ and $T' \succeq_\chi T$ hold, we write $T \equiv_\chi T'$.

2.4 Resolution

We now present a proof system called *resolution* [27]. In resolution, we see clauses as sets of literals, e.g., the clause $a \vee b \vee \neg c \vee b$ is seen as the set $\{a, b, \neg c\}$. Let φ be a set of clauses (seen as sets of literals), and assume that φ is unsatisfiable. A *resolution refutation* is a proof for the unsatisfiability of φ. It consists of a sequence of clauses $\mathcal{R} = (C_1, \dots C_t)$, where C_t is the empty set $\emptyset$, and each C_k, $1 \leq k < t$, is either in φ or is derived from two clauses C_i and C_j, where $1 \leq i \neq j < k$, by applying the *resolution rule*

$$\frac{\begin{array}{c} C \cup \{x\} \\ C' \cup \{\neg x\}, \end{array}}{C \cup C'}$$

where neither C nor C' contains the literals x or $\neg x$. In other words, the resolution rule states that the clause $C_k = C \cup C'$ can be inferred from clauses $C_i = C \cup \{x\}$ and $C_j = C' \cup \{\neg x\}$. Applying the resolution rule on the clauses $C \cup \{x\}$ and $C' \cup \{\neg x\}$ as above we say that we *resolve on* (the literal) x, while $C \cup C'$ is called the *resolvent*.

A *resolution tree refutation* for φ is a resolution refutation which can be presented as a labelled binary tree with the following properties [27].

(i) The root of the tree is the empty clause.

(ii) The leaves of the tree are labelled with clauses in φ.

(iii) If any non-leaf node n is labelled with clause C and its immediate successors n_1, n_2 are labelled with C_1, C_2, respectively, then C is a resolvent of C_1 and C_2.

Notably, one can always read a resolution refutation from a refutation tree resolution by simply constructing a sequence by taking the nodes in the tree in left–to–right, bottom–up order.

[3]Defining the *size* of a proof depends highly on the type of the proof system considered.

Example 2.2 *A resolution refutation for* $\mathrm{UNSAT}_{a,b}$ *is*

$$(\{a, b\}, \{a, \neg b\}, \{a\}, \{\neg a, b\}, \{\neg a, \neg b\}, \{\neg a\}, \emptyset).$$

A resolution tree refutation for $\mathrm{UNSAT}_{a,b}$ *is shown in Figure 1.*

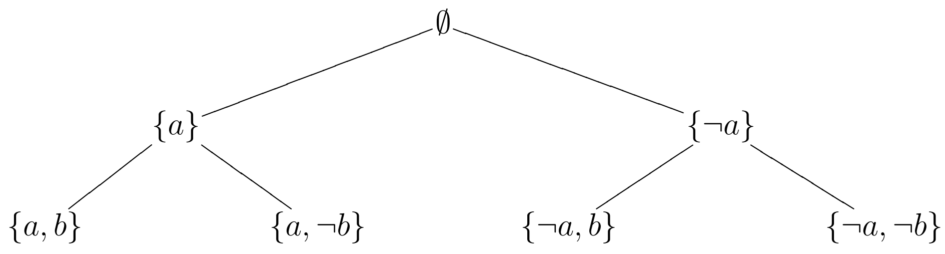

Figure 1: A resolution tree refutation for the set of clauses $\mathrm{UNSAT}_{a,b}$.

In order to have a measure on the complexity of proofs in resolution, we define the size of a resolution refutation as follows.

Definition 2.3 *The size of a resolution refutation* $\mathcal{R} = (C_1, \ldots C_t)$ *is* t.

Example 2.3 *The size of the resolution refutation in Example 2.2 is 7.*

Definition 2.4 *The size of a resolution tree refutation is the number of nodes in the tree.*

Example 2.4 *The size of the resolution tree refutation in Figure 1 is 7.*

3 BOOLEAN CIRCUITS

Boolean circuits and the related **NP**–complete Boolean circuit satisfiability problem are introduced. Sets of clauses are associated with Boolean circuits of specific kind.

3.1 Definition

Informally, a *Boolean circuit* (see e.g. [28]) is an acyclic directed graph in which the nodes are called *gates*. The gates can be divided into three categories[4]:

- *output gates* with incoming edges but no outgoing edges,
- *intermediate gates* with both incoming and outgoing edges, and
- *input gates* with outgoing edges but no incoming edges.

A Boolean function is associated with each output and intermediate gate.

Formally, we present a Boolean circuit $\mathcal{C}$ with the set of gates $\mathcal{V}$ as a set of equations of the form $v = f(v_1, \ldots, v_k)$, where $v, v_1, \ldots, v_k \in \mathcal{V}$ and f is a Boolean function. It is required that in the set of equations, each $v \in \mathcal{V}$ has at most one equation and that the equations are non-recursive.

For a Boolean circuit $\mathcal{C}$, we denote the set of gates appearing in $\mathcal{C}$ by $V(\mathcal{C})$. The edge relation for a Boolean circuit $\mathcal{C}$ is defined as $E(\mathcal{C}) = \{\langle v, v' \rangle \mid v' = f(\ldots, v, \ldots) \in \mathcal{C}\}$.

Example 3.1 *Graphically, the Boolean circuit*

$$\begin{aligned}
\{v &= \mathsf{and}(e, f, g, h), \\
e &= \mathsf{or}(a, b), \\
f &= \mathsf{or}(b, c), \\
g &= \mathsf{or}(a, d), \\
h &= \mathsf{or}(c, d), \\
c &= \mathsf{not}(a), \\
d &= \mathsf{not}(b)\}
\end{aligned}$$

is shown in Figure 2. In this circuit, a and b are input gates, c, d, e, f, g and h intermediate gates, and v is an output gate.

A *truth assignment* for a Boolean circuit $\mathcal{C}$ is a function $\tau : V(\mathcal{C}) \to \{\texttt{true}, \texttt{false}\}$. Assignment τ is *consistent* if $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ holds for each equation $v = f(v_1, \ldots, v_k)$ in $\mathcal{C}$. A *constrained Boolean circuit* $\langle \mathcal{C}, c^+, c^- \rangle$ is a Boolean circuit $\mathcal{C}$ with the restrictions that the gates in $c^+ \subseteq V(\mathcal{C})$ are `true` and those in $c^- \subseteq V(\mathcal{C})$ are `false`. If there is a consistent truth assignment that respects the constraints of a constrained Boolean

[4]We do not consider circuits in which there are trivial gates with no edges (neither incoming nor outgoing).

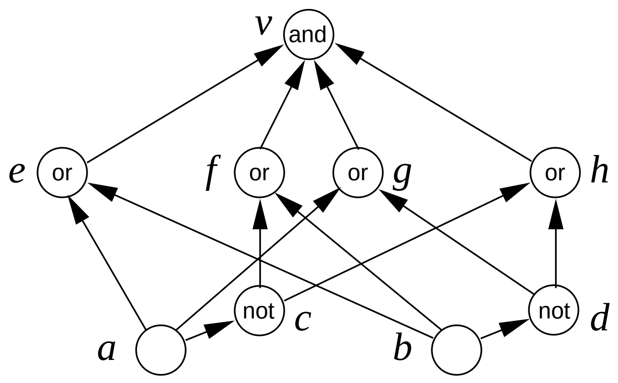

Figure 2: A Boolean circuit.

circuit, it is a *satisfying truth assignment* for the circuit, and the circuit is *satisfiable*. Otherwise the circuit is *unsatisfiable*.

Here, we are interested in CIRCUIT SAT, the *satisfaction problem* for constrained Boolean circuits, a problem closely related to SAT.

Definition 3.1 (CIRCUIT SAT)

Instance: *A constrained Boolean circuit* $\langle \mathcal{C}, c^+, c^- \rangle$.

Problem: *Is* $\langle \mathcal{C}, c^+, c^- \rangle$ *satisfiable?*

The CIRCUIT SAT problem is obviously **NP**-complete. Notice that any unconstrained circuit is trivially satisfiable, having 2^n satisfying truth assignments, where n is the number of input gates in the circuit.

In the following we consider the class of Boolean circuits in which the following three types of Boolean functions are allowed.

- $\mathsf{not}(v) = \texttt{true}$ if and only if v is $\texttt{false}$,
- $\mathsf{or}(v_1, \ldots, v_k) = \texttt{true}$ if and only if at least one v_i, $1 \leq i \leq k$, $k \geq 2$, is $\texttt{true}$, and
- $\mathsf{and}(v_1, \ldots, v_k) = \texttt{true}$ if and only if all v_i, $1 \leq i \leq k$, $k \geq 2$, are $\texttt{true}$.

Notice that it is straightforward to extend this class with additional Boolean functions such as xor and equivalence.

As a simple example of why Boolean circuits are a compact representation, consider the following.

Example 3.2 *Consider the Boolean circuit shown in Figure 3. An equivalent propositional formula, in which only such variables occur that correspond to the inputs in the circuit, can be constructed recursively top–down. For instance, the formula corresponding to gate* z_n *is* $v_{n+1} \vee_{z_n} w_{n+1}$*, and for gate* v_n *the corresponding formula is* $x_n \wedge_{v_n} (v_{n+1} \vee_{z_n} w_{n+1})$*. The resulting formula, corresponding to gate* v_1*, is of the form*

$$(x_1 \wedge_{v_1} ((\underbrace{\cdots}_{x_2\wedge_{v_2}((\cdots)\vee_{z_2}(\cdots))}) \vee_{z_1} (\underbrace{\cdots}_{((\cdots)\vee_{z_2}(\cdots))\wedge_{w_2} y_2}))) \vee_v (((\cdots) \vee_{z_1} (\cdots)) \wedge_{w_1} y_1).$$

In the formula the variables v_{n+1} and w_{n+1} both appear a number of times exponential to n. In the circuit, however, gates v_{n+1} and w_{n+1} appear only once.

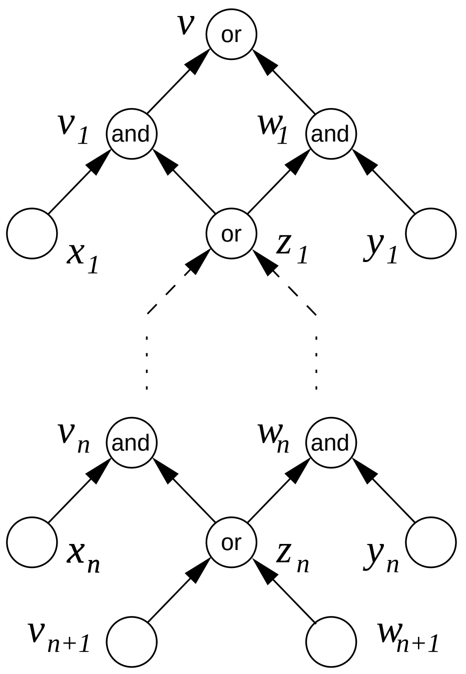

Figure 3: An example of why Boolean circuits are a compact representation.

3.2 Representing Sets of Clauses as Boolean Circuits

To relate a set of clauses with a Boolean circuit, we need a way to systematically generate a Boolean circuit from a given set of clauses. We do this by defining the *canonical Boolean circuit representation* $C(\varphi)$ for a set of clauses φ as follows. Let $\varphi = \{C_1, \ldots, C_k\}$, where each $C_i = \{l_{1_i}, \ldots l_{m_i}, \overline{l}_{1_i}, \ldots, \overline{l}_{n_i}\}$, and each l_{j_i} is a positive literal and $\overline{l}_{j_i}$ a negative literal. Now

$$\begin{aligned} C(\varphi) &= \{v = \mathsf{and}(o_1, \ldots, o_k\} \\ &\quad \cup \{o_i = \mathsf{or}(a_{l_{1_i}}, \ldots, a_{l_{m_i}}, b_{\overline{l}_{1_i}}, \ldots, b_{\overline{l}_{n_i}}\} \\ &\quad \cup \{b_{\overline{l}_{j_i}} = \mathsf{not}(a_{l_{j_i}})\}. \end{aligned}$$

By the canonicity of $C(\varphi)$ of a set of clauses φ, one can obviously reconstruct φ from $C(\varphi)$.

Example 3.3 *Recall the set of clauses* $\mathrm{UNSAT}_{a,b}$ *from Example 2.1. The canonical Boolean circuit representation* $C(\mathrm{UNSAT}_{a,b})$ *is shown in Figure 2.*

Given a set of clauses φ, we denote by $C_{\top}(\varphi)$ the circuit $C(\varphi)$ with the constraint that the output gate v of $C(\varphi)$ is `true`, i.e.,

$$C_{\top}(\varphi) = \langle C(\varphi), \{v\}, \emptyset \rangle.$$

The connection between φ and $C_{\top}(\varphi)$ is stated in the following proposition.

Proposition 3.1 *For any set of clauses φ, the set φ is satisfiable if and only if $C_{\top}(\varphi)$ is satisfiable.*

4 A TABLEAU METHOD

A tableau method for Boolean circuit satisfiability checking called $\mathbf{BC}$ is introduced. The polynomial simulatability among variations of $\mathbf{BC}$ is discussed in the light of Boolean circuits generated from sets of clauses. The $\mathbf{BC}$ method and its variations are shown to be complete and sound proof systems for Boolean circuits. Additionally, the notion of resolution–boundedness is introduced and $\mathbf{BC}$ and its variations are shown to have this property.

4.1 The $\mathbf{BC}$ Method

We concentrate on a tableau method for Boolean circuit satisfiability checking we call $\mathbf{BC}$. The $\mathbf{BC}$ method consists of the rules shown in Figure 4. It is a simplified version of the method introduced in [20].[5] [6]

Given a constrained Boolean circuit $\langle \mathcal{C}, c^+, c^- \rangle$, a $\mathbf{BC}$-tableau for it is a binary tree such that the root of the tree consists of the equations in $\mathcal{C}$ and the constraints; for each gate $v \in c^+$ a $\mathbf{T}v$ entry is added while for each $v \in c^-$ a $\mathbf{F}v$ entry is added. The other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, where $v \in V(\mathcal{C})$. The entries are generated by applying the rules in Figure 4 as in the standard tableau method [8].

We say that a branch B in a tableau has been *extended* to branch B', if B' consists of B to the leaf of which a sequence of entries generated by applying the rules (b)–(h) have been appended. By applying the explicit cut rule on a gate v in branch B, the branch is extended into two branches B' and B'', where B' (B'') consists of B to the leaf of which the entry $\mathbf{T}v$ ($\mathbf{F}v$) is appended. A tableau T' is an *immediate extension* of the tableau T if T' is T with the addition that one branch of T has been extended by applying some rule in the tableau method once. For each $v \in V(\mathcal{C})$, we say that the entry $\mathbf{T}v$ ($\mathbf{F}v$) can be *deduced* in the branch if the entry $\mathbf{T}v$ ($\mathbf{F}v$) can be generated by applying rules (b)–(h) only.

A branch in the tableau is *contradictory* if it contains both $\mathbf{F}v$ and $\mathbf{T}v$ entries for a gate $v \in V(\mathcal{C})$. Otherwise, the branch is *open*. A branch is *complete* if it is contradictory, or if there is a $\mathbf{F}v$ or a $\mathbf{T}v$ entry for each $v \in V(\mathcal{C})$ in the branch and the branch is closed under the rules (b)–(h). A tableau is *finished* if all the branches of the tableau are complete. A tableau is *closed* if all of its branches are contradictory. A closed $\mathbf{BC}$-tableau for a constrained circuit is called a $\mathbf{BC}$-*refutation* for the circuit.

Example 4.1 *For the circuit shown in Figure 2 with the constraint that the output gate v is* `true`*, a $\mathbf{BC}$-refutation is shown in Figure 5.*

4.2 Variations of $\mathbf{BC}$

We study variations of $\mathbf{BC}$ in which we restrict the application of the explicit cut rule to certain types of gates. Let $\langle \mathcal{C}, c^+, c^- \rangle$ be a constrained Boolean circuit. The considered variations of $\mathbf{BC}$ are the following.

[5]The method introduced in [20] has been implemented, see [19].

[6]In the method introduced in [20] rules e.g. for xor and equivalence are additionally provided.

$$\frac{v \in \mathcal{V}}{\mathbf{T}v \mid \mathbf{F}v}$$

(a) The explicit cut rule

$$\frac{\begin{array}{c} v = \mathsf{not}(v_1) \\ \mathbf{F}v_1 \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathsf{not}(v_1) \\ \mathbf{T}v_1 \end{array}}{\mathbf{F}v} \qquad \frac{\begin{array}{c} v = \mathsf{not}(v_1) \\ \mathbf{F}v \end{array}}{\mathbf{T}v_1} \qquad \frac{\begin{array}{c} v = \mathsf{not}(v_1) \\ \mathbf{T}v \end{array}}{\mathbf{F}v_1}$$

(b) "Up" rules for not (c) "Down" rules for not

$$\frac{\begin{array}{c} v = \mathsf{or}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{F}v} \quad \frac{\begin{array}{c} v = \mathsf{or}(v_1, \ldots, v_k) \\ \mathbf{T}v_i, i \in \{1, \ldots, k\} \end{array}}{\mathbf{T}v} \quad \frac{\begin{array}{c} v = \mathsf{or}(v_1, \ldots, v_k) \\ \mathbf{F}v \end{array}}{\mathbf{F}v_1, \ldots, \mathbf{F}v_k}$$

(d) "Up" rules for or (e) "Down" rule for or

$$\frac{\begin{array}{c} v = \mathsf{and}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_k \end{array}}{\mathbf{T}v} \quad \frac{\begin{array}{c} v = \mathsf{and}(v_1, \ldots, v_k) \\ \mathbf{F}v_i, i \in \{1, \ldots, k\} \end{array}}{\mathbf{F}v} \quad \frac{\begin{array}{c} v = \mathsf{and}(v_1, \ldots, v_k) \\ \mathbf{T}v \end{array}}{\mathbf{T}v_1, \ldots, \mathbf{T}v_k}$$

(f) "Up" rules for and (g) "Down" rule for and

$$\frac{\begin{array}{c} v = \mathsf{or}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_{j-1}, \mathbf{F}v_{j+1}, \ldots \mathbf{F}v_k \\ \mathbf{T}v \end{array}}{\mathbf{T}v_j} \quad \frac{\begin{array}{c} v = \mathsf{and}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_{j-1}, \mathbf{T}v_{j+1}, \ldots \mathbf{T}v_k \\ \mathbf{F}v \end{array}}{\mathbf{F}v_j}$$

(h) "Last undetermined child" rules for or and and

Figure 4: Tableau method **BC** for Boolean circuits.

- $\mathbf{BC}_{\mathrm{i}}$: Application of explicit cut is restricted to input gates (we call such cuts *input cuts*).

- $\mathbf{BC}_{\mathrm{td}}$: Application of explicit cut is restricted to output gates and those gates v for which there is a $\mathbf{T}v'$ or a $\mathbf{F}v'$ entry in the branch and $\langle v, v' \rangle \in E(\mathcal{C})$ (*top-down cuts*).

- $\mathbf{BC}_{\mathrm{bu}}$: Application of explicit cut is restricted to input cuts and gates v for which there is a $\mathbf{T}v'$ or a $\mathbf{F}v'$ entry in the branch and $\langle v', v \rangle \in E(\mathcal{C})$ (*bottom-up cuts*).

- $\mathbf{BC}_{\mathrm{i+td}}$: Application of explicit cut is restricted to input and top-down cuts.

- $\mathbf{BC}_{\mathrm{bu+td}}$: Application of explicit cut is restricted to bottom-up and top-down cuts.

1.	$v = \mathsf{and}(e, f, g, h)$	
2.	$e = \mathsf{or}(a, b)$	
3.	$f = \mathsf{or}(b, c)$	
4.	$g = \mathsf{or}(a, d)$	
5.	$h = \mathsf{or}(c, d)$	
6.	$c = \mathsf{not}(a)$	
7.	$d = \mathsf{not}(b)$	
8.	$\mathbf{T}v$	
9.	$\mathbf{T}e$	$(1, 8)$
10.	$\mathbf{T}f$	$(1, 8)$
11.	$\mathbf{T}g$	$(1, 8)$
12.	$\mathbf{T}h$	$(1, 8)$

13.	$\mathbf{T}a$	(Cut)	14.	$\mathbf{F}a$	(Cut)
15.	$\mathbf{F}c$	$(6, 13)$	20.	$\mathbf{T}b$	$(2, 9, 14)$
16.	$\mathbf{T}b$	$(3, 10, 15)$	21.	$\mathbf{T}d$	$(4, 11, 14)$
17.	$\mathbf{F}d$	$(7, 16)$	22.	$\mathbf{F}d$	$(7, 20)$
18.	$\mathbf{F}h$	$(5, 15, 17)$	23.	$\times$	$(21, 22)$
19.	$\times$	$(12, 18)$			

Figure 5: A **BC**-refutation for $C_{\top}(\mathrm{UNSAT}_{a,b})$.

4.3 Completeness and Soundness

Completeness and *soundness* [32] are two essential properties of proof systems. In the light of the proof systems considered in this work, by completeness we mean that there is a closed tableau for any unsatisfiable circuit. This is equivalent (by contraposition) to the fact that a complete open tableau for a circuit implies that the circuit is satisfiable. By soundness we mean that the existence of a closed tableau for a given circuit implies that the circuit is unsatisfiable. We now give proofs of the fact that the **BC** method and all its considered variations are complete and sound proof systems for constrained Boolean circuits.

Theorem 4.1 *The proof system that consists of the explicit cut rule restricted to input gates and the "up" rules for* not*,* or*, and* and *is complete for constrained Boolean circuits.*

Proof of Theorem 4.1. First we argue that, given any constrained circuit, we can generate a finished tableau with the given rules. Generally, we can apply the cut rule consecutively in every branch on each of the input gates. By doing so we have exactly either the entry $\mathbf{T}g$ or $\mathbf{F}g$ for each input gate g in every branch, and every branch is distinct. Thus in any branch we can then deduce an entry for all gates every immediate child of which is an input gate by applying the appropriate "up" rule. Proceeding recursively, we can deduce an entry for every gate in every branch. After this, each branch is obviously closed under the "up" rules. Notably, each branch is thus complete as no new entries can be deduced with the rules (b)–(h).

Now we argue that given a finished tableau with an open branch, the circuit in the root of the tableau must be satisfiable. In any complete branch we have an entry for each gate. If a complete branch is open, then it holds that we have exactly one of the entries $\mathbf{T}g$ and $\mathbf{F}g$ in the branch for each gate g. From the branch we construct a truth assignment τ for the circuit as follows. For each gate g with the entry $\mathbf{T}g$ ($\mathbf{F}g$) in the branch we assign $\tau(g) = \texttt{true}$ ($\tau(g) = \texttt{false}$). Now take an arbitrary equation $v = f(v_1, \ldots, v_k)$ in the circuit. One of the "up" rules was applied to deduce an entry for v from entries for $v_1, \ldots, v_k$. Thus by the construction of τ the branch contains entries for the truth values $\tau(v_1), \ldots, \tau(v_k), \tau(v)$, and we have that $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ must hold. Thus $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ must hold for any equation in the circuit, and hence τ must be a satisfying assignment. Therefore the circuit is satisfiable. By contraposition, the given proof system is complete. □

Furthermore, notice that with the top–down restricted cut rule we are able to apply the cut rule to input gates in any circuit by first systematically applying the cut rule on the gates in the circuit in a top–down fashion. As all the considered variations of the **BC** method contain the "up" rules for not, or, and and, we have the following corollary.

Corollary 4.1 $\mathbf{BC}_{\mathrm{i}}$, $\mathbf{BC}_{\mathrm{td}}$, $\mathbf{BC}_{\mathrm{i+td}}$, $\mathbf{BC}_{\mathrm{bu}}$, $\mathbf{BC}_{\mathrm{bu+td}}$, *and* **BC** *are complete proof systems for constrained Boolean circuits.*

We now address the soundness of the **BC** method. Some definition are needed in the proof of the following theorem. Let us call an entry $\mathbf{T}g$ ($\mathbf{F}g$) *consistent* under the truth assignment τ if $\tau(g) = \texttt{true}$ ($\tau(g) = \texttt{false}$). A branch B is *consistent* under τ if all entries of the type $\mathbf{T}g$ and $\mathbf{F}g$ are consistent under τ and, furthermore, τ is consistent for the circuit in the root of the tableau (i.e., $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ for each $v = f(v_1, \ldots, v_k)$ in the circuit). Tableau T is *consistent* under τ if at least one branch in T is consistent under τ. Particularly, we note that a closed tableau is not consistent under any truth assignment.

Theorem 4.2 **BC** *is a sound proof system for constrained Boolean circuits.*

Proof of Theorem 4.2. We argue that if a tableau T' is an immediate extension of the tableau T, then T' must be consistent under every truth assignment under which T is consistent. Now if T is consistent under a truth assignment τ, it must contain at least one branch B that is consistent under τ. Tableau T' was obtained from T by appending one or more entries to the leaf of some branch B' of T. If B' is distinct from B, then B is still a branch of T'. As B is consistent under τ it must hold that T' is also consistent under τ. On the other hand, assume that B' is identical to B, i.e., that B was the branch that was extended in T to obtain T'. There are the following cases to consider.

- If B was extended by applying the explicit cut rule on the gate v, then B was extended to two branches B_1, B_2. The branch B_1 is B extended with $\mathbf{T}v$ and B_2 is B extended with $\mathbf{F}v$. For any consistent truth assignment τ it must hold that either $\mathbf{T}v$ or $\mathbf{F}v$ is consistent under τ. Thus if B is consistent under τ, then either B_1 or B_2 is also consistent under τ.

- If B was extended by applying one of the "up" rules for not on $v = \mathsf{not}(v_1)$, then there are two cases to consider.

  (i) If B was extended with $\mathbf{T}v$ to obtain B_1, we must have $\mathbf{F}v_1$ in B. But as B is consistent under τ, v is restricted to have the opposite truth value to v_1 under τ, and thus B_1 must also be consistent under τ.

  (ii) If B was extended with $\mathbf{F}v$ to obtain B_1, we must have $\mathbf{T}v_1$ in B. But as B is consistent under τ, v is restricted to have the opposite truth value to v_1 under τ, and thus B_1 must also be consistent under τ.

- If B was extended by applying one of the "down" rules for not, the situation is similar to the one in which we have applied one of the "up" rules for not; the reader is invited to confirm these.

- If B was extended by applying one of the "up" rules for or on $v = \mathsf{or}(v_1, \ldots v_k)$, then there are two cases to consider.

  (i) If B was extended with $\mathbf{F}v$ to obtain B_1, we must have $\mathbf{F}v_i$ for every $1 \leq i \leq k$ in B. But as B is consistent under τ, v is restricted to be `true` if at least one of the v_i's is `true` under τ, and thus B_1 must also be consistent under τ;

  (ii) If B was extended with $\mathbf{T}v$ to obtain B_1, we must have $\mathbf{T}v_i$ for some $1 \leq i \leq k$ in B. But as B is consistent under τ, v is restricted to be `true` if at least one of the v_i's is `true` under τ, and thus B_1 must also be consistent under τ.

  The reader is invited to confirm that the situation is similar for the "up" rules for and.

- If B was extended by applying the "down" rule for or on a gate $v = \mathsf{or}(v_1, \ldots, v_k)$, then B was extended with $\mathbf{F}v_i$ for each $1 \leq i \leq k$ to obtain B_1, and we must have $\mathbf{F}v$ in B. But as B is consistent under τ, all the v_i's are restricted to `false` if v is `false` under τ, and thus B_1 must also be consistent under τ.

  Similarly for the "down" rule for and; the reader is invited to confirm this.

- If B was extended by applying the "last undetermined child" rule for or on $v = \mathsf{or}(v_1, \ldots, v_k)$, then B was extended with $\mathbf{T}v_k$ to obtain B_1, and we must have $\mathbf{T}v$ and $\mathbf{F}v_i$ for each $1 \leq i < k$ in B. But as B is consistent under τ, v is restricted to `true` if at least one of the v_i's is `true` under τ, and thus B_1 must also be consistent under τ.

  Again, the reader is invited to confirm that the situation is similar for the "last undetermined child" rule for and.

This shows that any immediate extension of a given tableau consistent under a given truth assignment is also consistent under the given truth assignment. Thus inductively we have that for any tableau T, if the root of the tableau – including the constraint entries – is consistent under a given truth assignment τ, then T is also consistent under τ. For any closed tableau T we must have that T is not consistent under any truth assignment τ, and hence the root of T is not consistent under τ. Thus τ cannot be a satisfying truth assignment for the constrained Boolean circuit in the root of the tableau. Therefore the $\mathbf{BC}$ method is sound. □

Directly by the restricted nature of the considered variations of the $\mathbf{BC}$ method we have the following corollary.

Corollary 4.2 $\mathbf{BC}_{\text{bu+td}}$*,* $\mathbf{BC}_{\text{bu}}$*,* $\mathbf{BC}_{\text{i+td}}$*,* $\mathbf{BC}_{\text{i}}$*, and* $\mathbf{BC}_{\text{td}}$ *are sound proof systems for constrained Boolean circuits.*

4.4 Relations Between Variations of BC

For analysing the complexity of $\mathbf{BC}$-refutations we need the notion of the size of a $\mathbf{BC}$-refutation. We define it as follows.

Definition 4.1 *The* size *of a* $\mathbf{BC}$*-refutation is the number of nodes in the closed tableau.*

Example 4.2 *The size of the* $\mathbf{BC}$*-refutation shown in Figure 5 is 14.*

The concepts of refutation and the size of a refutation in the restricted variations of $\mathbf{BC}$ are the same as for $\mathbf{BC}$.

An obvious ordering of $\mathbf{BC}$ and its restricted variations based on the p-simulation relation, resulting from the restricted nature of the variations, is shown in Figure 6.

$$\begin{tikzcd}
 & \mathbf{BC}_{\text{bu}} \arrow[dr, phantom, "\succeq" sloped] & \\
\mathbf{BC} \succeq \mathbf{BC}_{\text{bu+td}} \arrow[ur, phantom, "\succeq" sloped] \arrow[dr, phantom, "\succeq" sloped] & & \mathbf{BC}_{\text{i}} \\
 & \mathbf{BC}_{\text{i+td}} \arrow[ur, phantom, "\succeq" sloped] \arrow[dr, phantom, "\succeq" sloped] & \\
 & & \mathbf{BC}_{\text{td}}
\end{tikzcd}$$

Figure 6: An obvious ordering of $\mathbf{BC}$ and its variations based on the p-simulation relation.

It turns out that all the considered variations of the $\mathbf{BC}$ method are equivalent under the p–simulation relation under the set of canonical Boolean circuit representations of all sets of clauses. For the following, let Φ be the family of all sets of clauses, and $C_\top(\Phi) = \{C_\top(\varphi) \mid \varphi \in \Phi\}$.

Theorem 4.3 $\mathbf{BC} \equiv_{C_\top(\Phi)} \mathbf{BC}_{\text{i}}$.

Proof of Theorem 4.3. First notice that for any set of clauses φ, using the "down" rule for and we can deduce $\mathbf{T}g$ for all or gates g in $C_{\top}(\varphi)$. Thus we can assume that there is a minimal–size refutation for $C_{\top}(\varphi)$ in which the and rule is applied to deduce the entries concerning the or gates that are needed to achieve the closed tableau.

As we have the entry $\mathbf{T}v$ for the output gate v in the branch, it makes no sense to apply the cut rule on v. Otherwise, we would generate in one branch $\mathbf{F}v$, thus closing the branch, and in the other $\mathbf{T}v$, which we already had in the branch. The same reasoning applies for the or gates.

Now consider applying the cut rule on a not gate $g = \mathsf{not}(g')$. In the branch with $\mathbf{T}g$ ($\mathbf{F}g$) we can directly deduce $\mathbf{F}g'$ ($\mathbf{T}g'$). Thus this is equivalent to having applied the cut rule on g' and then deducing an entry for g. Thus we can limit the application of the cut rule to input gates. This shows $\mathbf{BC}_{\mathrm{i}} \succeq_{C_{\top}(\Phi)} \mathbf{BC}$. $\mathbf{BC} \succeq_{C_{\top}(\Phi)} \mathbf{BC}_{\mathrm{i}}$ holds trivially. $\square$

By further noticing that for any circuit in the family $C_{\top}(\Phi)$ input cuts can be p–simulated with top–down cuts by first applying the "down" rules for and on the output gate and then applying the cut rule on the input gate (after possible applying the "down" rule for not in between), we have the following corollary.

Corollary 4.3 $\mathbf{BC} \equiv_{C_{\top}(\Phi)} X$ *for all* $X \in \{\mathbf{BC}_{\mathrm{i}}, \mathbf{BC}_{\mathrm{td}}, \mathbf{BC}_{\mathrm{i+td}}, \mathbf{BC}_{\mathrm{bu}}\}$.

4.5 Resolution–Boundedness

To compare the proof complexity of resolution and tableau methods for Boolean circuits, we use the notion of *resolution–boundedness*, which is defined as follows.

Definition 4.2 (Resolution–boundedness) *A tableau method T for constrained Boolean circuits is* resolution–bounded *if there is a polynomial $p(n)$ such that, for any set of clauses φ, if there is a T-refutation for $C_{\top}(\varphi)$ of size n, then there is a resolution refutation for φ of size $p(n)$.*

We now show that the $\mathbf{BC}$ method is resolution–bounded.

Theorem 4.4 $\mathbf{BC}$ *is resolution–bounded.*

Proof of Theorem 4.4. Let φ be an unsatisfiable set of clauses. Following the lines of the proof of Theorem 4.3, we assume to have a $\mathbf{T}g$ entry for each or gate g in $C_{\top}(\varphi)$ in all branches in any $\mathbf{BC}$–refutation for $C_{\top}(\varphi)$, and that the cut rule is applied on input gates only.

From the $\mathbf{BC}$–refutation we form a *cut tree*. A cut tree is a binary tree in which the edges from a particular parent to its children are labelled with $\mathbf{T}g$ and $\mathbf{F}g$, where g is some *input* gate, i.e., there are entries for input gates only. A cut tree is formed by examining the refutation tableau top–down starting with single node in the tree.

- If the cut rule has next been applied on an input gate g in the tableau, then we insert children to the current node in the tree, labelling the edges to the children with $\mathbf{T}g$ and $\mathbf{F}g$, respectively. The effect of the cut rule is hence copied as such into the cut tree, and the branch with $\mathbf{T}g$ ($\mathbf{F}g$) in the cut tree will correspond to the branch in the tableau with $\mathbf{T}g$ ($\mathbf{F}g$).

- If the $\mathbf{T}g$ ($\mathbf{F}g$) entry has been deduced by applying the "last undetermined child" rule for or, then there are two possibilities.
  - If the entry is for an input gate, then we insert children to the current node in the tree, labelling the edges to the children with $\mathbf{T}g$ and $\mathbf{F}g$, respectively. The child node into which the edge labelled with $\mathbf{T}g$ ($\mathbf{F}g$) points to will correspond to the current branch in the tableau, while the other child node will be a leaf node in the cut tree.
  - If the entry is for a not gate $g = \mathsf{not}(g')$, then we insert children to the current node in the tree, labelling the edges to the children with $\mathbf{F}g'$ and $\mathbf{T}g'$, respectively. The child node into which the edge labelled with $\mathbf{F}g'$ ($\mathbf{T}g'$) points to will correspond to the current branch in the tableau, while the other child node will be a leaf node in the cut tree.
- All other entries in the tableau are disregarded forming the cut tree.

As an example, a cut tree corresponding to the **BC**–refutation in Figure 5 is shown in Figure 7.

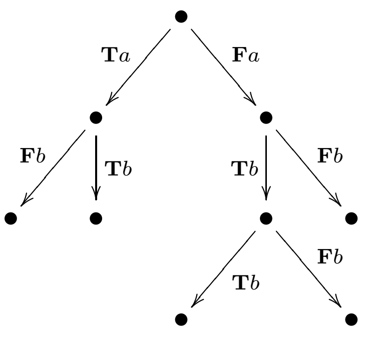

Figure 7: A cut tree corresponding to the **BC**-refutation shown in Figure 5.

The idea is that each application of the rules (b)–(h), in which a particular entry E on a gate g has been deduced, can be seen as a sequence of first applying the cut rule on g, and then closing the branch with the complementary entry of E by deducing E. This can be done because if we can deduce $\mathbf{T}g$ ($\mathbf{F}g$), then $\mathbf{F}g$ ($\mathbf{T}g$) will lead to contradiction.

For the following, let x_g denote the variable corresponding to the input gate g so that $\mathbf{T}g$ ($\mathbf{F}g$) corresponds to x_g ($\neg x_g$). Notice that applying the rules for not in **BC** correspond simply to negating the corresponding literal in φ. We can thus directly read the clause $x_{v_1} \vee \cdots \vee x_{v_m} \vee \neg x_{v'_1} \vee \cdots \vee \neg x_{v'_n}$ that corresponds to v from each equation $v = \mathsf{or}(v_1, \ldots, v_m, \overline{v}_1, \ldots, \overline{v}_n)$ in $C_\mathrm{T}(\varphi)$, where each v_i, $1 \leq i \leq m$, is an input gate and each $\overline{v}_i$, $1 \leq i \leq n$, is a not gate of the form $\overline{v}_i = \mathsf{not}(v'_i)$.

From a cut tree we can generate a corresponding resolution tree refutation in a straightforward manner. We know that each branch in the $\mathbf{BC}$–refutation is contradictory. Thus we must have labels on the edges in each branch of the cut tree that *falsify* some or gate $v = \mathsf{or}(v_1, \ldots, v_k)$, i.e., assigning the truth values corresponding to these entries to $v = \mathsf{or}(v_1, \ldots, v_k)$ does not satisfy the equation having the entry $\mathbf{T}v$ in the branch. Now we associate the clause corresponding to a falsified or gate in each branch with the leaf of the branch.

Resolution steps are taken as follows. Starting bottom–up from the leaf nodes, we examine which input gate g appears in the labels on the edges going to the parent node.

- If the variable x_g appears in both of the clauses associated with the clauses associated with the two leaf nodes, then we apply the resolution rule on these clauses resolving on x_g. The resolvent is then associated with the parent node.
- If x_g does not appear in both of the clauses associated with the two leaf nodes, then we associate with the parent node the clause in which x_g does not appear. If x_g does not appear in either one of the clauses, then the clause is selected arbitrarily among the two.

The process above is then repeated in a bottom–up fashion to all the nodes in the tree.

On each resolution step, x_g, the variable corresponding to the gate that appears in the labels on the edges going to the parent, does not appear in the resolvent. Thus the resolvent associated with the root of the cut tree will be empty. As an example, the resolution tree refutation corresponding to the cut tree in Figure 7, and thus to the $\mathbf{BC}$–refutation in Figure 5, is shown in Figure 1.

For each entry in the original $\mathbf{BC}$–refutation T we make at most one resolution step. Thus the resulting resolution tree refutation for φ is, in the worst–case, of linear size w.r.t. the size of T. $\square$

Again, by the restricted nature of the variants of the $\mathbf{BC}$ method, we have the following corollary.

Corollary 4.4 $\mathbf{BC}_{\mathrm{i}}$, $\mathbf{BC}_{\mathrm{td}}$, $\mathbf{BC}_{\mathrm{i+td}}$, $\mathbf{BC}_{\mathrm{bu}}$, *and* $\mathbf{BC}_{\mathrm{bu+td}}$ *are resolution–bounded.*

5 CIRCUIT GADGETS

Preliminaries for the main results of the work are presented, i.e., certain circuit constructs are introduced.

5.1 Pigeon-Hole Principle and the PHP_n^{n+1} Gadget

An example of a propositional formula with high proof complexity in many proof systems is the *pigeon-hole principle* PHP_n^m [18]. The pigeon-hole principle states that there is no injective mapping from a finite m-element set into a finite n-element set if $m > n$ (that is, m pigeons cannot sit in less than m holes so that every pigeon has its own hole). In the following we consider the case $m = n + 1$.

Definition 5.1 (PHP_n^{n+1})

$$\mathrm{PHP}_n^{n+1} \overset{\mathrm{def}}{=} \bigcup_{1 \le i \le n+1} \{P_i\} \cup \bigcup_{\substack{1 \le i \ne i' \le n+1, \\ 1 \le j \le n}} \{H_{i,i'}^j\},$$

where the clauses P_i and $H_{i,i'}^j$ are defined as

$P_i \overset{\mathrm{def}}{=} \bigvee_{j=1}^{n} x_{i,j}$ *and*

$H_{i,i'}^j \overset{\mathrm{def}}{=} \neg x_{i,j} \vee \neg x_{i',j}$,

and each $x_{i,j}$ is a Boolean variable with the following interpretation:

$$x_{i,j} = \texttt{true} \textit{ if and only if the } i^{\text{th}} \textit{ pigeon sits in the } j^{\text{th}} \textit{ hole.}$$

The P_i clauses state that each pigeon has to sit in some hole, while clauses $H_{i,i'}^j$ state that no two pigeons can sit in the same hole. The union of all the clauses P_i and $H_{i,i'}^j$ is obviously (by the pigeon-hole principle) unsatisfiable. In this encoding the number of connectives is polynomially bounded w.r.t. the number of Boolean variables.

The canonical Boolean circuit representation of PHP_n^{n+1} is graphically represented as shown in Figure 8(a). In Figure 8(b) a part of the circuit is shown in detail. We call $C(\mathrm{PHP}_n^{n+1})$ the PHP_n^{n+1} *gadget*. Notice that as PHP_n^{n+1} is unsatisfiable, so is $C_{\mathrm{T}}(\mathrm{PHP}_n^{n+1})$.
Formally the PHP_n^{n+1} gadget is the set of equations

$$\begin{aligned}
C(\mathrm{PHP}_n^{n+1}) \quad = \quad & \{v = \mathsf{and}(p_1, \ldots, p_{n+1}, h_{1,2}^1, \ldots, h_{n+1,n}^n)\} \\
& \cup \{p_i = \mathsf{or}(x_{i,1}, \ldots, x_{i,n}) \mid 1 \le i \le n+1\} \\
& \cup \{h_{i,i'}^j = \mathsf{or}(l_{i,j}, l_{i',j}) \mid 1 \le i \ne i' \le n+1, 1 \le j \le n\} \\
& \cup \{l_{i,j} = \mathsf{not}(x_{i,j}) \mid 1 \le i \le n+1, 1 \le j \le n\},
\end{aligned}$$

where $h_{1,2}^1, \ldots, h_{n+1,n}^n$ stands for all $h_{i,i'}^j$, where $1 \le i \ne i' \le n+1$ and $1 \le j \le n$.

For resolution the following theorem was first proven by Haken in 1985 [16].

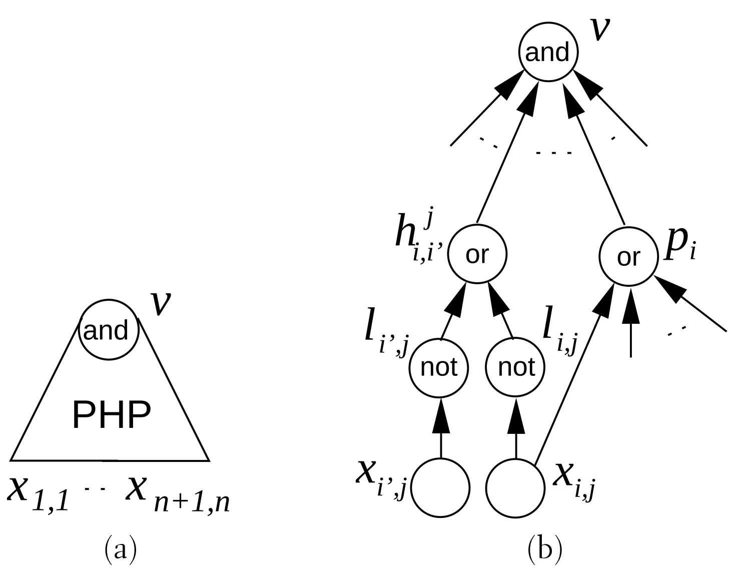

Figure 8: (a) The PHP_n^{n+1} gadget graphically, (b) a part of the PHP_n^{n+1} gadget in detail.

Theorem 5.1 *The proof complexity of* PHP_n^{n+1} *is exponential w.r.t.* n *for resolution.*

As $\mathbf{BC}$ and the variations we consider are resolution–bounded by Theorem 4.4 and Corollary 4.4, we have the following corollary.

Corollary 5.1 *The proof complexity of* $C_{\top}(\mathrm{PHP}_n^{n+1})$ *is exponential w.r.t.* n *for* $\mathbf{BC}_{\mathrm{i}}$, $\mathbf{BC}_{\mathrm{td}}$, $\mathbf{BC}_{\mathrm{i+td}}$, $\mathbf{BC}_{\mathrm{bu}}$, $\mathbf{BC}_{\mathrm{bu+td}}$, *and* $\mathbf{BC}$.

5.2 The TD Gadget

The structure of the TD_n *gadget* is shown in Figure 9(a). In the following we present the TD_n gadget graphically as shown in Figure 9(b). Formally the TD_n gadget is the set of equations

$$\begin{aligned}\mathrm{TD}_n \quad = \quad & \{v = \mathsf{or}(v_1, w_1)\} \\ & \cup \{v_i = \mathsf{and}(x_i, z_i) \mid 1 \leq i \leq n\} \\ & \cup \{w_i = \mathsf{and}(y_i, z_i) \mid 1 \leq i \leq n\} \\ & \cup \{z_i = \mathsf{or}(v_{i+1}, w_{i+1}) \mid 1 \leq i \leq n\}.\end{aligned}$$

The following lemma on the TD_n gadget will be useful in the proofs of our main results.

Lemma 5.1 *It is impossible to generate a polynomial size tableau with respect to* n *for* TD_n *with gate* v *constrained to* `true` *in* $\mathbf{BC}_{\mathrm{td}}$ *in which there is an entry for the gate* z_n *in every branch of the tableau.*

Proof of Lemma 5.1. The entry $\mathbf{T}v$ implies $\mathbf{T}v_1$ or $\mathbf{T}w_1$, but we cannot deduce one or the other. Thus we must apply the cut rule on either v_1 or w_1 in $\mathbf{BC}_{\mathrm{td}}$. Assume that we cut on v_1 (cutting on w_1 is symmetric). Now consider the branch in which we have $\mathbf{F}v_1$. Due to $v = \mathsf{or}(v_1, w_1)$ we must

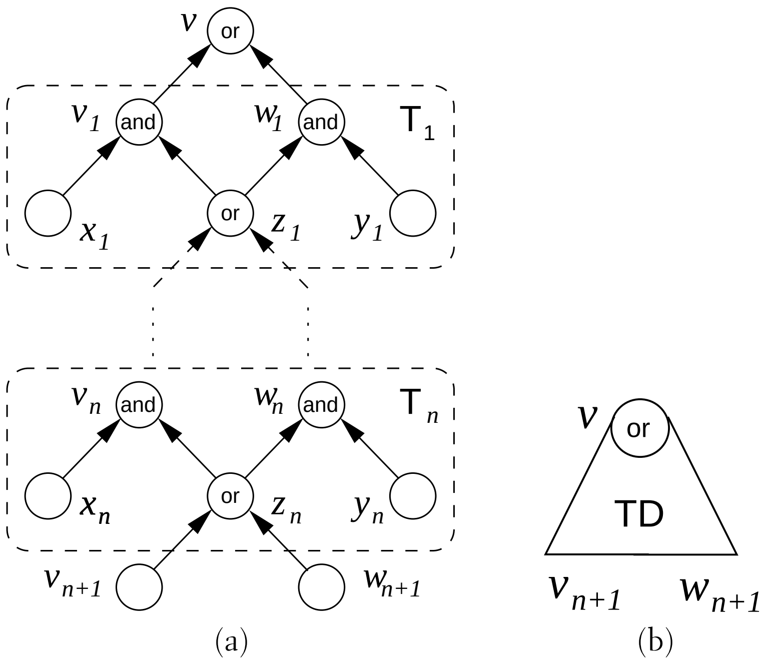

Figure 9: (a) The structure of the TD$_n$ gadget, (b) the TD$_n$ gadget graphically.

have $\mathbf{T}w_1$. Then from $w_1 = \mathsf{and}(y_1, z_1)$ we deduce $\mathbf{T}y_1$ and $\mathbf{T}z_1$ in the branch. In the branch where we have $\mathbf{T}v_1$ using the "down" rule for and we deduce $\mathbf{T}x_1$ and $\mathbf{T}z_1$. Nothing else can be deduced. Inductively on i, in order to have an entry for the gate z_i in every branch of the tableau, the tableau must contain at least 2^i branches, all of which remain open. This is because we must for each i apply the cut rule on either v_i or w_i. This is demonstrated in Figure 10. □

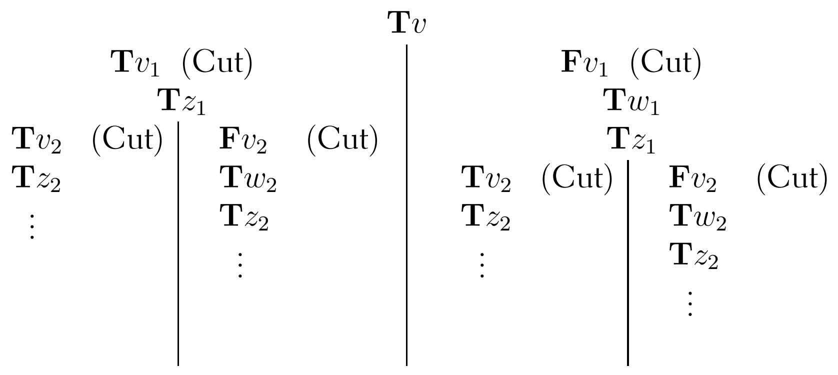

Figure 10: Why $\mathbf{BC}_{\text{td}}$-refutations corcerning the TD$_n$ gadget are large.

5.3 The UNSAT Gadget

Recall the set of clauses UNSAT$_{a,b}$ that appeared in Example 2.1. The structure of $C(\text{UNSAT}_{a,b})$ is shown in Figure 2. Graphically we present

$C(\mathrm{UNSAT}_{a,b})$ as shown in Figure 11. We call $C(\mathrm{UNSAT}_{a,b})$ the $\mathrm{UNSAT}_{a,b}$ *gadget*. The formal definition of $C(\mathrm{UNSAT}_{a,b})$ appears in Example 2.1.

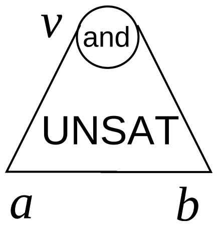

Figure 11: The $\mathrm{UNSAT}_{a,b}$ gadget graphically.

5.4 The XOR Gadget

The Boolean xor function

$$\mathsf{xor}(x, y) = (x \land \neg y) \lor (\neg x \land y)$$

evaluates to `true` if and only if exactly one of x, y is `true`. Based on the xor function we can construct a Boolean circuit, as shown in Figure 12, for which it holds that the output gate a evaluates to `true` if and only if $\mathsf{xor}(x, y)$ evaluates to `true`. When we use this circuit construct as a part of a circuit, we represent it graphically as an "xor gate" $\oplus$.

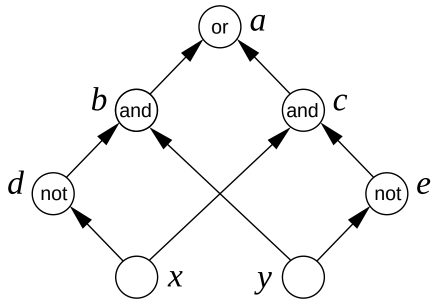

Figure 12: The Boolean function xor as a Boolean circuit.

Notice that we can deduce an entry for any one of a, x, y if we have entries for the other two in the branch. Furthermore, if we do not have entries for any of b, c, d, e, we can not deduce an entry for any one of a, x, y if we do not have entries for the other two in the branch. If we apply the cut rule on x, we can deduce exactly the entries $\mathbf{F}c$, $\mathbf{T}d$ in the branch with $\mathbf{F}x$, and $\mathbf{F}d$, $\mathbf{F}b$ in the branch with $\mathbf{T}x$. Especially, with bottom–up cuts we can then apply the cut in both of these branches on gate a. For gate y the situation is symmetric. Moreover, applying the cut rule on both x and y is equivalent to applying bottom–up cut on x and then on c in both branches, in the sense that we can then deduce an entry in all of these branches for a. Again, the situation for y, applying the cut then on b, is symmetric. This means that with bottom–up cuts we can forget about the inner structure of the circuit, always applying the cut rule on gates x, y only.

With top–down cuts, by applying the cut rule on gate a we can then deduce exactly the entries $\mathbf{F}b$, $\mathbf{F}c$ in the branch with $\mathbf{F}a$. If we then apply the cut rule on x or y in both branches, we can deduce entries for the rest of the gates in all of these branches. On the other hand, in the branch with $\mathbf{T}a$ we can deduce nothing. If we then apply the cut rule on b or c in both branches, we can deduce entries for the rest of the gates in all of the branches. In this case, in the following we will say that we apply the cut on x or y when actually we simulate this by applying the cut rule on b or c after which entries for x an y can be deduced. This lets us forget about the inner structure of $\oplus$, although it exists.

Using the "xor gate" we construct a circuit as shown in Figure 13(a). This construct, the XOR$_n$ *gadget*, has n layers X_i, $1 \leq i \leq n$, of xor gates, as shown in the figure. In the following, we present the XOR$_n$ gadget as shown in Figure 13(b).

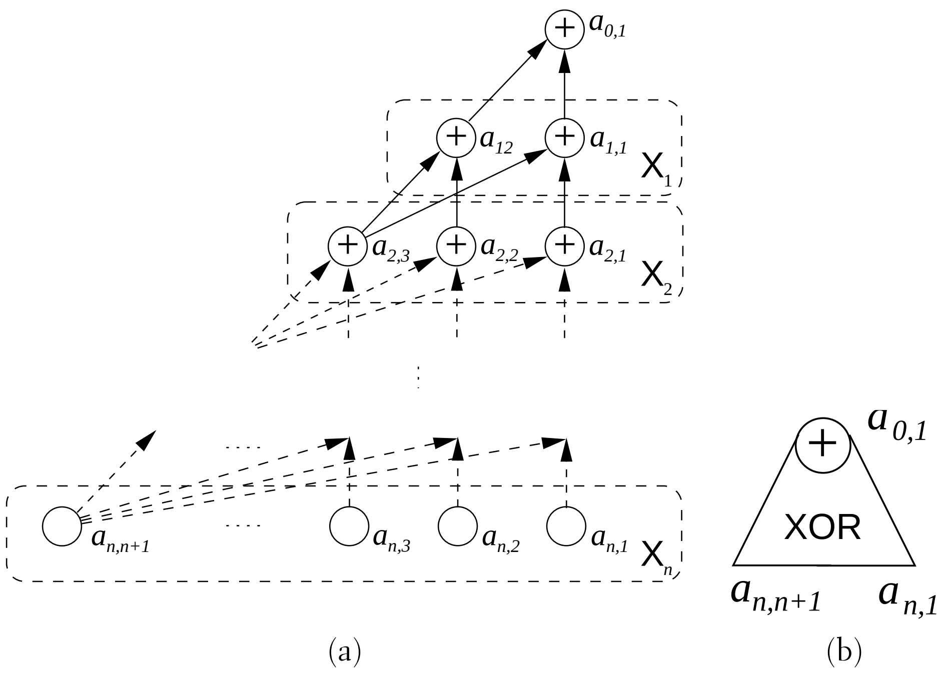

Figure 13: (a) The structure of the XOR$_n$ gadget, (b) the XOR$_n$ gadget graphically.

Formally, the XOR$_n$ gadget is the set of equations

$$\begin{aligned}
\mathrm{XOR}_n = \; & \{a_{i,j} = \mathsf{or}(b_{i,j}, c_{i,j}) \mid 0 \leq i < n, 1 \leq j \leq i+1\} \\
& \cup \{b_{i,j} = \mathsf{and}(d_{i,j}, a_{i+1,j}) \mid 0 \leq i < n, 1 \leq j \leq i+1\} \\
& \cup \{c_{i,j} = \mathsf{and}(e_{i,j}, a_{i+1,i+2}) \mid 0 \leq i < n\} \\
& \cup \{d_{i,j} = \mathsf{not}(a_{i+1,i+2}) \mid 0 \leq i < n\} \\
& \cup \{e_{i,j} = \mathsf{not}(a_{i+1,j}) \mid 0 \leq i < n, 1 \leq j \leq i+1\}
\end{aligned}$$

The number of gates in the gadget is polynomial with respect to n, namely $\Theta(n^2)$.[7]

[7]For a given n, we have $5 \cdot \sum_{i=1}^{n+1} i = \frac{5(n+1)(n+2)}{2}$ gates, giving $\Theta(n^2)$.

The following two lemmas on the XOR_n gadget will be useful in the proofs of our main results.

Lemma 5.2 *It is impossible to generate a polynomial size tableau with respect to n for XOR_n in $\mathbf{BC}_{\mathrm{bu}}$ in which there is an entry for the output gate $a_{0,1}$ in every branch of the tableau.*

Proof of Lemma 5.2. After applying the cut rule on a single $a_{n,j}$ we cannot yet deduce entries for any other $a_{i,j'}$. Thus we can only either apply the cut rule

(a) on one of the other gates on level n, or

(a) on any of the gates on level $n-1$ that is the parent of $a_{n,j}$ on which we applied the cut rule.

Consider case (a). Assume that we first applied the cut rule on some $a_{n,j}$, where $1 \leq j \leq n$. If we then apply the cut rule on $a_{n,n+1}$, we can deduce an entry for $a_{n-1,j}$, but not for any other $a_{i,j'}$. If we first applied the cut rule on $a_{n,n+1}$, we then apply the cut rule on some $a_{n,j}$, and can similarly deduce an entry only for $a_{n-1,j}$. Either way, in case (i) we can deduce exactly one entry for a single gate on level $n-1$, and have a similar situation on level $n-1$ to the one we were in on level n before applying the second cut. Most noticeably, no branch closes. Inductively, "climbing up" the circuit bottom–up from level n to level 1 we thus end up with a tableau with exponential number of entries w.r.t. n.

For case (b), by applying the cut rule on a particular parent $a_{n-1,i}$ of $a_{n,j}$, we can deduce an entry for the other child of $a_{n-1,i}$, say $a_{n,k}$, but not for any other $a_{i',j'}$. It must hold that either $k = n+1$ or $j = n+1$. By applying the cut rule on any other $a_{n,l}$, we can deduce $a_{n-1,l}$, but not for any other $a_{i',j'}$, and are in a similar situation on level $n-1$ to that we described in case (a) on level n. Moreover, we have to apply the cut rule on all gates on level n before we can deduce entries for all gates on level $n-1$.

Combining the ideas from cases (a) and (b), we notice that having entries for $a_{i,j}$ and $a_{i,j'}$, the only other gates $a_{i',j''}$ for which we can at most deduce entries are (i) $a_{i-1,j}$ (if $j' = i+1$), and, given that we have also an entry for $a_{i+1,i+2}$, (ii) $a_{i+1,j}$ and $a_{i+1,j'}$.

We will thus in any case have to apply the cut rule a number of times linear to n and end up with a tableau with exponential number of entries w.r.t. n before reaching level 1. □

Lemma 5.3 *Given that the output gate $a_{0,1}$ of XOR_n is constrained to* `true`*, it is impossible to generate a polynomial size tableau with respect to n for XOR_n in $\mathbf{BC}_{\mathrm{td}}$ in which there is an entry for some gate $a_{n,i}$, $1 \leq i \leq n+1$ in every branch of the tableau.*

Proof of Lemma 5.3. By $\mathbf{T}a_{0,1}$ we may apply the cut rule on either $a_{1,1}$ or $a_{1,2}$. If we apply the cut rule on $a_{1,1}$, in the branch with $\mathbf{T}a_{1,1}$ we can deduce $\mathbf{F}a_{1,2}$, and in the branch with $\mathbf{F}a_{1,1}$ we can deduce $\mathbf{T}a_{1,2}$, but no entries for any other $a_{i,l}$. By a symmetric argument, we can deduce $\mathbf{T}a_{1,1}$ in one branch and $\mathbf{F}a_{1,1}$ in the other if we apply the cut rule on $a_{1,2}$. In both cases, we double the number of open branches. Now we may apply the cut rule to any of the gates on level 2 of the XOR gadget. By a similar argument, after applying the cut rule on one of the three gates, we can deduce an entry for the other two gates in each branch. Still, none of the branches close. Inductively, on level i we always double the number of open branches. $\square$

6 MAIN RESULTS

The main results of the work are presented. Negative polynomial simulation results are shown to hold for each pair of the considered variations of the $\mathbf{BC}$ method. A road map to the results presented in the first part of this chapter is shown below. Furthermore, the relevance of the results is discussed in the light of the Davis–Putnam method.

$$\begin{tikzcd}
 & & \mathbf{BC}_{\mathrm{bu}} \arrow[dl, phantom, "\succeq^{6.5}"] \arrow[dr, phantom, "\succeq^{6.1}"] & \\
\mathbf{BC} \arrow[r, phantom, "\overset{6.6}{\succeq}"] & \mathbf{BC}_{\mathrm{bu+td}} \arrow[dr, phantom, "\succeq^{6.4}"] & & \mathbf{BC}_{\mathrm{i}} \arrow[dl, phantom, "\succeq^{6.2}"] \\
 & & \mathbf{BC}_{\mathrm{i+td}} \arrow[dr, phantom, "\succeq^{6.3}"] & \\
 & & & \mathbf{BC}_{\mathrm{td}}
\end{tikzcd}$$

6.1 $\mathbf{BC}_{\mathrm{bu}}$ vs $\mathbf{BC}_{\mathrm{i}}$

We now show that $\mathbf{BC}_{\mathrm{i}}$ cannot p–simulate $\mathbf{BC}_{\mathrm{bu}}$. The proof utilises the UNSAT and PHP_n^{n+1} gadgets and the resolution–boundedness of the variations of the $\mathbf{BC}$ method.

Theorem 6.1 $\mathbf{BC}_{\mathrm{bu}} \succ \mathbf{BC}_{\mathrm{i}}$.

Proof of Theorem 6.1. Consider the family of circuits of the type shown in Figure 14 with the constraint that the output gate v is `true`. Any circuit in the family is obviously unsatisfiable.

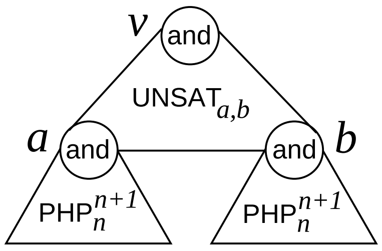

Figure 14: Circuit for Theorems 6.1 and 6.2.

For an arbitrary n, for $\mathbf{BC}_{\mathrm{bu}}$ we can construct a constant size refutation as follows. First, deduce $\mathbf{T}e$, $\mathbf{T}f$, $\mathbf{T}g$, $\mathbf{T}h$ from $\mathbf{T}v$. Then apply (say, in the PHP_n^{n+1} gadget on the left) the cut rule first on one of the input gates $x_{i,j}$, and deduce an entry for $l_{i,j}$. After this, apply the cut rule on $h_{i,i'}^{j}$ in both of the induced branches. Now we have induced four branches in total, having in each branch a constant number of entries, and can apply the cut rule on a in each branch. After having an entry on a, each branch can be closed in a constant number of steps similarly to the refutation shown in Figure 5. The generated closed tableau will by the above be of constant size.

Notice that to generate a refutation we need to reach the UNSAT gadget, i.e., it is impossible to generate a contradiction in all the branches of a tableau without having an entry for some of the gates in the UNSAT gadget in the tableau.

Now consider $\mathbf{BC}_{\mathrm{i}}$. From $\mathbf{T}v$ we can deduce $\mathbf{T}e$, $\mathbf{T}f$, $\mathbf{T}g$, and $\mathbf{T}h$, but nothing else. As PHP_n^{n+1} is unsatisfiable, it is impossible to deduce $\mathbf{T}a$ or $\mathbf{T}b$ with "up" rules as a and b are and gates. Thus we can only have $\mathbf{F}a$ and $\mathbf{F}b$ entries in any branch. Now assume that there is a $\mathbf{BC}_{\mathrm{i}}$-refutation of polynomial size w.r.t. n for this circuit. A closed tableau can only be achieved after deducing an entry for gate a or gate b, as we have no constraints on the PHP_n^{n+1} parts of the circuit. Thus we must have either $\mathbf{F}a$ or $\mathbf{F}b$ in every branch. But if we have $\mathbf{F}a$ or $\mathbf{F}b$ in every branch, then we could construct a $\mathbf{BC}_{\mathrm{i}}$-refutation of polynomial size w.r.t. n for $C_{\top}(\mathrm{PHP}_n^{n+1})$. This is in contradiction with Corollary 5.1. Thus all $\mathbf{BC}_{\mathrm{i}}$-refutations will be of exponential size w.r.t. n. □

6.2 $\mathbf{BC}_{\mathrm{i+td}}$ vs $\mathbf{BC}_{\mathrm{i}}$

We now proceed to show that $\mathbf{BC}_{\mathrm{i}}$ cannot p–simulate $\mathbf{BC}_{\mathrm{i+td}}$. In the proof, we re-use the ideas used in the proof of Theorem 6.1.

Theorem 6.2 $\mathbf{BC}_{\mathrm{i+td}} \succ \mathbf{BC}_{\mathrm{i}}$.

Proof of Theorem 6.2. Consider again the family of circuits of the type shown in Figure 14 with the constraint that the output gate v is `true`. We have that any circuit in the family is unsatisfiable.

For $\mathbf{BC}_{\mathrm{i+td}}$ we can construct a constant size refutation by first deducing $\mathbf{T}e$, then applying the cut rule on gate a, and then closing each branch similarly to the refutation shown in Figure 5. For $\mathbf{BC}_{\mathrm{i}}$, all $\mathbf{BC}_{\mathrm{i}}$-refutations will be of exponential size w.r.t. n as argued in the proof of Theorem 6.1. □

6.3 $\mathbf{BC}_{\mathrm{i+td}}$ vs $\mathbf{BC}_{\mathrm{td}}$

Next we show that $\mathbf{BC}_{\mathrm{td}}$ cannot p–simulate $\mathbf{BC}_{\mathrm{i+td}}$. The proof of the following theorem is based on a circuit constructed from two UNSAT gadgets and a TD_n gadget.

Theorem 6.3 $\mathbf{BC}_{\mathrm{i+td}} \succ \mathbf{BC}_{\mathrm{td}}$.

Proof of Theorem 6.3. Consider the family of circuits of the type shown in Figure 15 with the constraint that the output gate v is `true`.

Any circuit in this family is obviously unsatisfiable. For $\mathbf{BC}_{\mathrm{i+td}}$ we can construct a refutation of linear size w.r.t. n as follows. First apply consecutively the cut rule on gates a_1, b_1, a_2, b_2 in each branch. This induces 16 branches, a constant number. We then have an entry for each of a_1, b_1, a_2, b_2 in every branch. Now we can deduce an entry for v_{n+1} and w_{n+1} in each branch. As $C_{\top}(\mathrm{UNSAT}_{a,b})$ is unsatisfiable, we can only deduce $\mathbf{F}v_{n+1}$ and $\mathbf{F}w_{n+1}$. This can clearly be done in a constant number of steps. From the entries $\mathbf{F}v_{n+1}$, $\mathbf{F}w_{n+1}$ we can then deduce $\mathbf{F}z_n$, and then $\mathbf{F}v_n$, $\mathbf{F}w_n$. Proceeding recursively, we can thus deduce $\mathbf{F}v$ in every branch with a linear number of steps w.r.t. n.

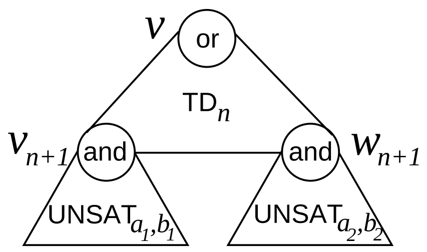

Figure 15: Circuit for Theorem 6.3.

Notice that to generate a refutation we need to reach the UNSAT gadgets, as in the proof of Theorem 6.1. But by Lemma 5.1, before reaching the gate z_n from top–down, we already must have generated a tableau with exponentially many entries w.r.t. n. Every $\mathbf{BC}_{\mathrm{td}}$-refutation is thus of exponential size w.r.t. n for this family of circuits. □

6.4 $\mathbf{BC}_{\mathrm{bu+td}}$ vs $\mathbf{BC}_{\mathrm{i+td}}$

In this subsection we show that $\mathbf{BC}_{\mathrm{i+td}}$ cannot p–simulate $\mathbf{BC}_{\mathrm{bu+td}}$. Using the ideas in the proof of Theorem 6.1, we construct a circuit from three circuits similar to the one used in the proofs of Theorems 6.1 and 6.2, an XOR$_n$ gadget, and an expander sub-circuit that connects the former four. The expander circuit is an example of a simple nontrivial circuit in which deduction can be propagated through the circuit in a straightforward fashion. It is applied here so that trivial simplification of the circuit is not possible. Lemma 5.3 is also applied.

Theorem 6.4 $\mathbf{BC}_{\mathrm{bu+td}} \succ \mathbf{BC}_{\mathrm{i+td}}$.

Proof of Theorem 6.4. Consider the family of circuits of the type shown in Figure 16 with the constraint that the output gate v is `true`.

Any circuit in the family is unsatisfiable. For $\mathbf{BC}_{\mathrm{bu+td}}$ we can construct a refutation of polynomial size w.r.t. n as follows. First apply the bottom–up strategy introduced in the proof of Theorem 6.1 to generate entries for the and gates $a_1, a_2, a_3, b_1, b_2, b_3$ in every branch. As in the proof of Theorem 6.1, this can be done having a constant number of entries in the tableau. Then it is straightforward to deduce entries for v_1, v_2, v_3 in each branch. Furthermore, as $C_{\top}(\mathrm{UNSAT}_{a,b})$ is unsatisfiable, we must then have $\mathbf{F}v_1, \mathbf{F}v_2, \mathbf{F}v_3$ in every branch. Now in an arbitrary branch, it is straightforward to deduce the entries $\mathbf{F}a_{n,j}$ for all $1 \leq j \leq n+1$, generating only a number of entries in the order of n^2. Continuing on, generating only a number of entries in the order of n^2, deducing recursively $\mathbf{F}a_{i-1,j}$ from $\mathbf{F}a_{i,j}$ and $\mathbf{F}a_{i,i+1}$ we can at last deduce $\mathbf{F}v$. As we have in total a constant number of branches and in each branch a polynomial number of entries w.r.t. n, we clearly have a $\mathbf{BC}_{\mathrm{bu+td}}$-refutation of polynomial size w.r.t. n.

Again, to generate a refutation we need to reach the UNSAT gadgets. With input cuts, this results in a refutation of exponential size w.r.t. n, as argued in the proof of Theorem 6.1. By Lemma 5.3, any top-down approach will also result in a refutation of exponential size w.r.t. n. □

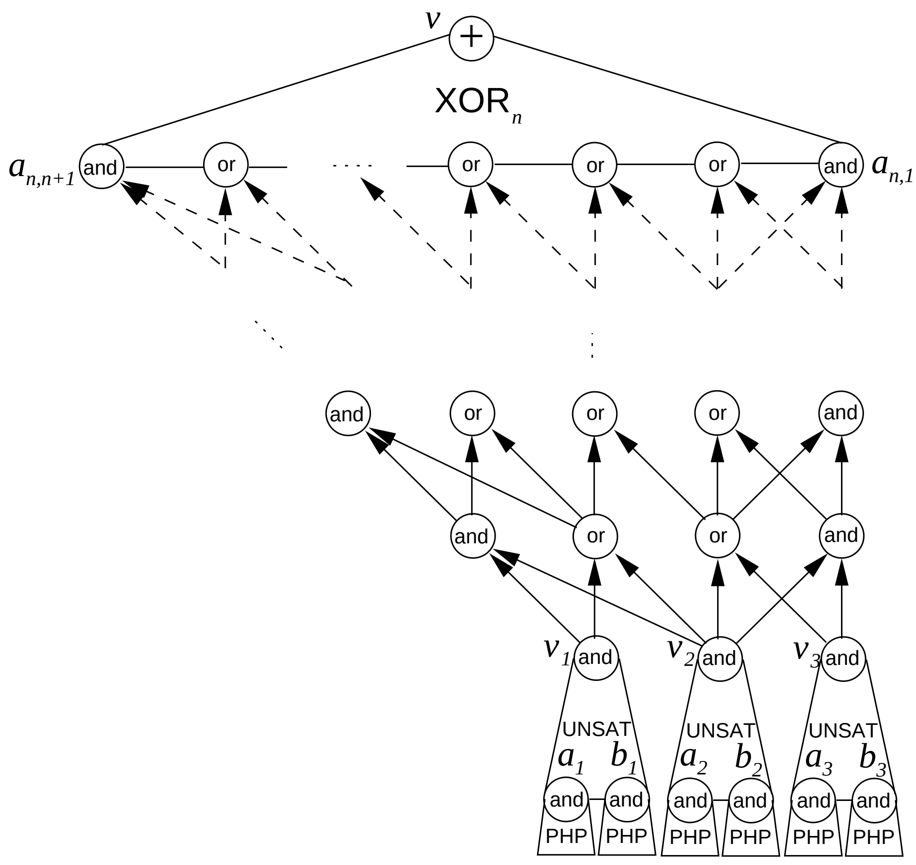

Figure 16: Circuit for Theorem 6.4.

6.5 $\mathbf{BC}_{\text{bu+td}}$ vs $\mathbf{BC}_{\text{bu}}$

Next we show that $\mathbf{BC}_{\text{bu}}$ cannot p–simulate $\mathbf{BC}_{\text{bu+td}}$. In addition to ideas in used in the proof of Theorem 6.2, we use a circuit constructed from a pair of XOR$_n$ gadgets and an UNSAT gadget. We also apply Lemma 5.2.

Theorem 6.5 $\mathbf{BC}_{\text{bu+td}} \succ \mathbf{BC}_{\text{bu}}$.

Proof of Theorem 6.5. Consider the family of circuits of the type shown in Figure 17 with the constraint that the output gate v is `true`. As $C_{\top}(\text{UNSAT}_{a,b})$ is unsatisfiable, any circuit in this family is also unsatisfiable.

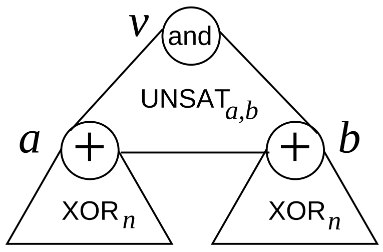

Figure 17: Circuit for Theorem 6.5.

As already described in the proof of Theorem 6.2, for $\mathbf{BC}_{\text{bu+td}}$ we can construct a constant size refutation top–down by first deducing $\mathbf{T}e$, then applying the cut rule on gate a, and closing each branch similarly to the refutation shown in Figure 5.

It is impossible to generate a refutation without reaching the UNSAT gadgets, as in the previous proofs in which we had an UNSAT gadget as a part of the circuit. By Lemma 5.2, in order to reach the UNSAT gadget, we must generate a tableau with exponential number of branches w.r.t. n. Thus any $\mathbf{BC}_{\mathrm{bu}}$-refutation for any circuit in this family must be of exponential size w.r.t. n. □

6.6 $\mathbf{BC}$ vs $\mathbf{BC}_{\mathrm{bu+td}}$

Now we proceed by showing that $\mathbf{BC}_{\mathrm{bu+td}}$ cannot p–simulate $\mathbf{BC}$. The proof uses $n+1$ UNSAT gadgets and $2n+3$ XOR$_n$ gadgets, and applies Lemmas 5.2 and 5.3.

Theorem 6.6 $\mathbf{BC} \succ \mathbf{BC}_{\mathrm{bu+td}}$.

Proof of Theorem 6.6. Consider the family of circuits of the type shown in Figure 18 with the constraint that the output gate v is `true`.

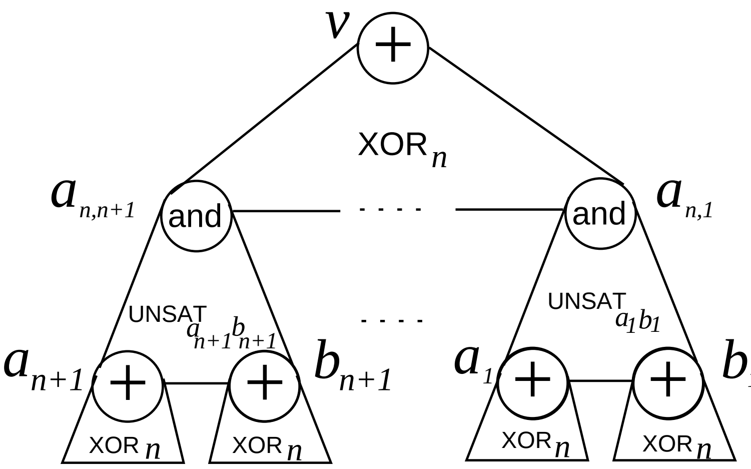

Figure 18: Circuit for Theorem 6.6.

For $\mathbf{BC}$ we can construct a refutation of polynomial size w.r.t. n as follows. First apply the cut rule on $a_{n,1}$. In the branch in which we have $\mathbf{T}a_{n,1}$, apply the cut rule on a_1. Similarly to the refutation in Figure 5, we can close both the branch in which we have $\mathbf{T}a_1$ and the one in which we have $\mathbf{F}a_1$. In the branch in which we have $\mathbf{F}a_{n,1}$, recursively on i, cut first on $a_{n,i}$ and then in the branch in which we have $\mathbf{T}a_{n,i}$, cut on a_i and again close both of the induced branches. This idea is shown in Figure 19. As the refutation in Figure 5 is of constant size, we end up with a tableau of polynomial size w.r.t. n in which there is a single open branch with the entries $\mathbf{F}a_{n,i}$ for all $1 \leq i \leq n+1$.

After this, generating only number of entries in the order of n^2, deducing recursively $\mathbf{F}a_{i-1,j}$ from $\mathbf{F}a_{i,j}$ and $\mathbf{F}a_{i,i+1}$ we can at last deduce $\mathbf{F}v$, thus generating a $\mathbf{BC}$-refutation of polynomial size w.r.t. n.

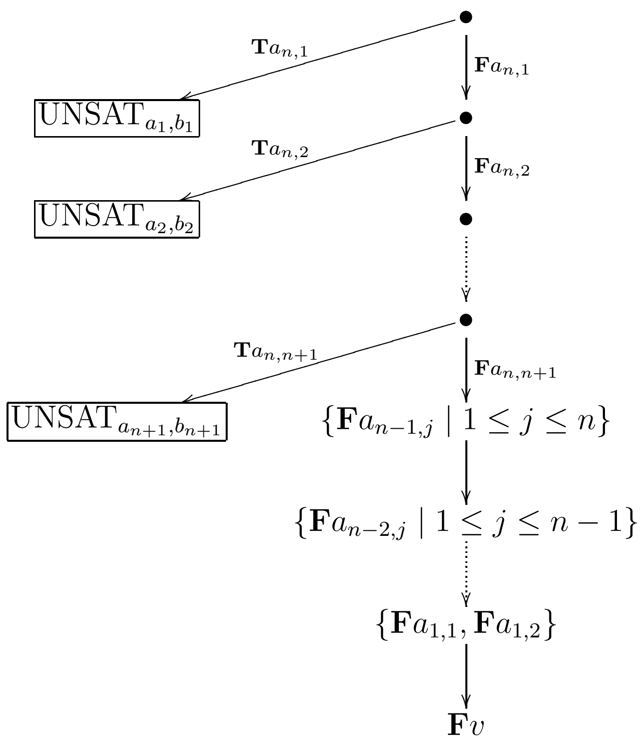

Figure 19: How to generate a polynomial size $\mathbf{BC}$-refutation for the circuit shown in Figure 18.

Again, to generate a refutation we need to reach the UNSAT gadgets. By Lemma 5.2, any bottom–up approach will result in a refutation of exponential size w.r.t. n. By Lemma 5.3, this applies also for any top–down approach. Thus any $\mathbf{BC}_{\text{bu+td}}$-refutation will be of exponential size w.r.t. n for any circuit in this family. □

6.7 $\mathbf{BC}_{\text{i}}$ vs $\mathbf{BC}_{\text{td}}$

We now turn to show that $\mathbf{BC}_{\text{i}}$ and $\mathbf{BC}_{\text{td}}$ are incomparable under the p–simulation relation. The proof draws heavily on the proofs of Theorems 6.1 and 6.3.

Theorem 6.7 $\mathbf{BC}_{\text{i}} \not\equiv \mathbf{BC}_{\text{td}}$.

Proof of Theorem 6.7. Consider again the family of circuits shown in Figure 14. In the proof of Theorem 6.1 it is shown that all $\mathbf{BC}_{\text{i}}$-refutations for any circuit in this family are of exponential size w.r.t. n. For an idea of how to generate a $\mathbf{BC}_{\text{td}}$-refutation of constant size we again refer the reader to the refutation shown in Figure 5.

On the other hand, consider the family of circuits shown in Figure 15. It is shown in the proof of Theorem 6.3 that all $\mathbf{BC}_{\text{td}}$-refutations for any circuit in this family are of exponential size w.r.t. n, while in the same proof it is described how to construct a linear size refutation for an arbitrary circuit in the family applying the cut rule only on input gates. □

6.8 $\mathbf{BC}_{\mathrm{bu}}$ vs $\mathbf{BC}_{\mathrm{td}}$

Using ideas from the proofs of Theorems 6.5 and 6.7, in this subsection we show that $\mathbf{BC}_{\mathrm{bu}}$ and $\mathbf{BC}_{\mathrm{td}}$ are incomparable under the p–simulation relation.

Theorem 6.8 $\mathbf{BC}_{\mathrm{bu}} \not\equiv \mathbf{BC}_{\mathrm{td}}$.

Proof of Theorem 6.8. As explained in the proof of Theorem 6.7, for an arbitrary circuit in the family of circuits shown in Figure 15, there is a refutation of linear size w.r.t. n in which the cut rule is applied only on input gates of the circuit, while all $\mathbf{BC}_{\mathrm{td}}$-refutations for the circuit are of exponential size w.r.t. n.

In the proof of Theorem 6.5 the family of circuits shown in Figure 17 is introduced. For this family it holds that applying the cut rule in a top–down fashion there is a refutation of constant size for an arbitrary circuit in the family, while by applying the cut rule in bottom–up fashion only will always result in a exponential size refutation w.r.t. n. □

6.9 $\mathbf{BC}_{\mathrm{bu}}$ vs $\mathbf{BC}_{\mathrm{i+td}}$

As the last one of the main theorems of this work, we argue that $\mathbf{BC}_{\mathrm{bu}}$ and $\mathbf{BC}_{\mathrm{i+td}}$ are incomparable under the p–simulation relation.

Theorem 6.9 $\mathbf{BC}_{\mathrm{bu}} \not\equiv \mathbf{BC}_{\mathrm{i+td}}$.

Proof of Theorem 6.9. It is shown in the proof of Theorem 6.5 that every $\mathbf{BC}_{\mathrm{bu}}$-refutation for an arbitrary member of the family of circuits shown in Figure 17 is of exponential size w.r.t. n. For $\mathbf{BC}_{\mathrm{i+td}}$ we can construct a constant size refutation for these circuits similarly to the refutation described in the proof of Theorem 6.2.

On the other hand, now consider the family of circuits shown in Figure 20 with the constraint that the output gate v is `true`. Notice that such a circuit consists of a TD_n gadget from the input gates of which hang two sub-circuits equivalent to the circuit in Figure 14. Combining Lemma 5.1 and the reasoning presented in the proof of Theorem 6.1, we have that every $\mathbf{BC}_{\mathrm{i+td}}$-refutation for an arbitrary circuit in this family is of exponential size w.r.t. n, as it is impossible to reach the UNSAT gadgets both top–down and bottom–up without generating an exponential number of entries in the tableau.

For $\mathbf{BC}_{\mathrm{bu}}$, we can generate a refutation of linear size w.r.t. n as follows. It is discussed in the proof of Theorem 6.1 how one can apply the cut on gate a in the circuit in Figure 14. What we can do here is to cut through the PHP_n^{n+1} circuits similarly as in the proof of Theorem 6.1. Then we can apply the cut rule on each gate a_1, b_1, a_2, b_2 in each branch. After this, it is straightforward to deduce an entry for gates v_{n+1} and w_{n+1} in every branch. Due to the unsatisfiability of $C_{\top}(\mathrm{UNSAT}_{a,b})$, with this bottom–up approach it is only possible to deduce $\mathbf{F}v_{n+1}, \mathbf{F}w_{n+1}$. At this point we note that as the UNSAT gadget with PHP_n^{n+1} gadgets hanging from the input gates has a constant number of gates, we have so far obviously generated a tableau with only constant number of entries. Having $\mathbf{F}v_{n+1}, \mathbf{F}w_{n+1}$ in each branch,

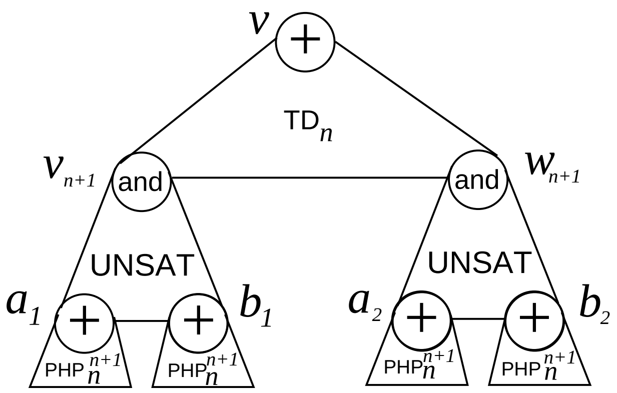

Figure 20: Circuit for Theorem 6.9.

it is possible to deduce $\mathbf{F}v$ by generating only a linear number of entries w.r.t. n, as explained in the proof of Theorem 6.3. Thus we can generate a $\mathbf{BC}_{\mathrm{bu}}$-refutation of linear size w.r.t. n for any member of the family of circuits considered. □

6.10 Corollaries

Combining Theorems 6.1 – 6.9, by the transitivity of $\succeq$, the resulting ordering of $\mathbf{BC}$ and its restricted variations based on the p–simulation relation is shown in Figure 21.

$$\begin{tikzcd}[column sep=small]
 & & \mathbf{BC}_{\mathrm{bu}} & & \\
\mathbf{BC} \arrow[r, phantom, "\succ"] & \mathbf{BC}_{\mathrm{bu+td}} \arrow[ur, phantom, "\prec" sloped] \arrow[rr, phantom, "\not\equiv"] \arrow[dr, phantom, "\succ" sloped] & & \mathbf{BC}_{\mathrm{i}} \arrow[ul, phantom, "\succ" sloped] \arrow[dl, phantom, "\prec" sloped] & \\
 & & \mathbf{BC}_{\mathrm{i+td}} \arrow[r, phantom, "\not\equiv"] \arrow[dr, phantom, "\succ" sloped] & & \\
 & & & \mathbf{BC}_{\mathrm{td}} &
\end{tikzcd}$$

Figure 21: Summary of the ordering of $\mathbf{BC}$ and its restricted variations based on the p–simulation relation. The case $\mathbf{BC}_{\mathrm{bu}} \not\equiv \mathbf{BC}_{\mathrm{td}}$ is omitted from the picture for clarity.

We have thus shown that no two variations of the $\mathbf{BC}$ method are equal using proof complexity as the measure. Notice that the results obviously hold for extended classes of Boolean circuits if the set of rules involving and, or, and not gates remains unchanged in the tableau method.

6.11 Relevance to the Davis–Putnam Method

Given an arbitrary constrained Boolean circuit $\mathcal{C} = \langle C, c^{+}, c^{-}\rangle$, one can translate $\mathcal{C}$ into a propositional formula in CNF of linear–size w.r.t. the size of the circuit that is satisfiable if and only if $\mathcal{C}$ is. The standard way of doing the translation is often referred to as Tseitin's translation, first discussed in [33]. This translation introduces a new variable v_g for each gate $g \in \mathcal{C}$ and captures the functional dependencies in $\mathcal{C}$ by clauses, the output being thus in CNF. More precisely,

- each gate $g \in c^+$ (c^-) is translated into the unit clause $\{v_g\}$ ($\{\neg v_g\}$),
- each $g = \mathsf{not}(g_1) \in C$ is translated into the set of clauses

$$\{\{v_g \vee v_{g_1}\}, \{\neg v_g \vee \neg v_{g_1}\}\},$$

- each $g = \mathsf{or}(g_1, \ldots g_k) \in C$ is translated into the set of clauses

$$\{\{v_{g_1} \vee \cdots \vee v_{g_k} \vee \neg v_g\}\} \cup \bigcup_{i \in \{1,\ldots,k\}} \{\{v_g \vee \neg v_{g_i}\}\},$$

  and

- each $g = \mathsf{and}(g_1, \ldots g_k) \in C$ is translated into the set of clauses

$$\{\{\neg v_{g_1} \vee \cdots \vee \neg v_{g_k} \vee v_g\}\} \cup \bigcup_{i \in \{1,\ldots,k\}} \{\{\neg v_g \vee v_{g_i}\}\}.$$

The CNF formula output by the translation is the union of the sets of clauses above. With gates with only finite fan–in, i.e., gates with only finitely many direct children, the CNF is obviously of linear size in the number of gates, constrained gates, and edges in $\mathcal{C}$.

It turns out that the main results of this work apply to the Davis–Putnam method (DPLL) [12, 11] in the case that the input is translated into CNF using Tseitin's translation. This is due to the fact that the proof complexity of an arbitrary Boolean circuit $\mathcal{C}$ for $\mathbf{BC}$ is always within a polynomial w.r.t. the proof complexity of the CNF formula that results from Tseitin's translation from $\mathcal{C}$ for DPLL.

We assume that the reader is familiar with the basic DPLL method. A DPLL–refutation can be abstractly seen as a tableau in which the entries are sets of clauses. There are two rules in the DPLL method which can be applied to generate tableau entries.

(i) *Splitting rule*, which *splits* φ into two sets, $\varphi' = \varphi \cup \{\{v\}\}$ and $\varphi'' = \varphi \cup \{\{\neg v\}\}$, where v is some variable that appears in φ. Now obviously φ is satisfiable if and only if φ' or φ'' is.

(ii) *Unit propagation*, which, given that there is a unit clause $\{v\} \in \varphi$, propagating on $\{v\}$ transforms each clause of the form $\{\neg v \vee v_1 \vee \cdots \vee v_k\}$ into $\{v_1 \vee \cdots \vee v_k\}$, and removes all clauses that contain v from φ.

A branch in a tableau is contradictory if there are both of the unit clauses $\{a\}$ and $\{\neg a\}$ in the branch for some variable a. Rest of the terminology concerning DPLL–tableaux is synonymous in an obvious way with that of $\mathbf{BC}$–tableaux.

For the following, let φ be a set of clauses that is obtained from a Boolean circuit using Tseitin's translation. The idea here is to show that DPLL for

φ can p–simulate $\mathbf{BC}$ for the original circuit, and vice versa. In fact, any DPLL–refutation can be interpreted as a $\mathbf{BC}$–refutation, and vice versa. Especially, we argue that unit clauses in a DPLL–refutation generated by applying the two rules above correspond exactly to non–root entries of the corresponding $\mathbf{BC}$–refutation.

Obviously, the splitting rule is equivalent to the cut rule in $\mathbf{BC}$; adding the unit clause $\{v_g\}$ ($\{\neg v_g\}$), is equivalent to extending a branch with $\mathbf{T}g$ ($\mathbf{F}g$) by applying the cut rule (and deducing $\mathbf{T}g'$ by applying the "up" rule for not on $g' = \mathsf{not}(g)$), and vice versa.

Clearly, having a unit clause $\{v_g\}$ ($\{\neg v_g\}$) amounts to having $\mathbf{T}g$ ($\mathbf{F}g$) in the branch in $\mathbf{BC}$. Deducing a unit clause $\{v_g\}$ by unit propagation from a clause $\{v_g \vee v_{g_1} \vee \cdots \vee v_{g_k}\}$ requires propagating on each of the unit clauses $\{\neg v_{g_i}\}$, where $1 \leq i \leq k$. Thus to deduce a new unit clause we need to have all $\{\neg v_{g_i}\}$'s in φ. Next we argue that unit clauses that can be generated by applying unit propagation on φ correspond one–to–one to the non–root entries generated by applying the rules (b)–(h) in a corresponding $\mathbf{BC}$–refutation for the original circuit. Consider the different gate types in Boolean circuits.

- $g = \mathsf{not}(g')$: This is translated into $\{\{g \vee g'\}, \{\neg g \vee \neg g'\}\}$.
  - If we have $\{g\} \in \varphi$, then unit propagation removes the clause $\{g \vee g'\}$ and transforms $\{\neg g \vee \neg g'\}$ into $\{\neg g'\}$. This is equivalent to deducing $\mathbf{F}g'$ from $\mathbf{T}g$ by applying the "down" rule for not in $\mathbf{BC}$.
  - The other cases $\{\neg g\} \in \varphi$, $\{g'\} \in \varphi$, and $\{\neg g'\} \in \varphi$ are very similar, and thus left for the reader to confirm.

- $g = \mathsf{or}(g_1, \ldots, g_k)$: This is translated into

$$\{\{v_{g_1} \vee \cdots \vee v_{g_k} \vee \neg v_g\}, \{v_g \vee \neg v_{g_1}\}, \ldots, \{v_g \vee \neg v_{g_k}\}\}.$$

  - If we have $\{v_{g_i}\} \in \varphi$ for some i, then unit propagation removes the clause $\{v_{g_1} \vee \cdots \vee v_{g_k} \vee \neg v_g\}$, and transforms $\{v_g \vee \neg v_{g_i}\}$ into $\{v_g\}$. Now unit propagation on $\{v_g\}$ removes all the rest $\{v_g \vee \neg v_{g_j}\}$, $j \neq i$. This is equivalent to deducing $\mathbf{T}g$ from $\mathbf{T}g_i$ by applying the "up" rule for or in $\mathbf{BC}$.
  - If we have $\{\neg v_g\} \in \varphi$, then unit propagation removes the clause $\{v_{g_1} \vee \cdots \vee v_{g_k} \vee \neg v_g\}$, and transforms each $\{v_g \vee \neg v_{g_i}\}$ into $\{\neg v_{g_i}\}$. This is equivalent to deducing $\mathbf{F}g$ from all $\mathbf{F}g_i$, $1 \leq i \leq k$, by applying the "up" rule for or in $\mathbf{BC}$.
  - If we have $\{v_g\} \in \varphi$, then unit propagation removes all $\{v_g \vee \neg v_{g_i}\}$ and transforms $\{v_{g_1} \vee \cdots \vee v_{g_k} \vee \neg v_g\}$ into $\{v_{g_1} \vee \cdots \vee v_{g_k}\}$. Then to deduce $\{v_{g_i}\}$ for some i we still need to have $\{\neg v_{g_j}\} \in \varphi$ for all $j \neq i$. Thus deducing $\{v_{g_i}\}$ is equivalent to deducing $\mathbf{T}g_i$ from $\mathbf{T}v_g$ and all $\mathbf{F}v_{g_j}$ using the "last undetermined child" rule for or in $\mathbf{BC}$.
  - If we have $\{\neg v_{g_i}\} \in \varphi$ for some i, then unit propagation removes the clause $\{v_g \vee \neg v_{g_i}\}$ and transforms $\{v_{g_1} \vee \cdots \vee v_{g_k} \vee \neg v_g\}$ into $\{v_{g_1} \vee \cdots \vee v_{g_{i-1}} \vee v_{g_{i+1}} \vee \cdots \vee v_{g_k} \vee \neg v_g\}$. Then to deduce

(i) $\{v_{g_j}\}$ for some $j \neq i$ we still need to have $\{v_g\}$ and $\{\neg v_{g_l}\} \in \varphi$ for all $l \neq j$. Thus deducing $\{v_{g_j}\}$ is equivalent to deducing $\mathbf{T}g_j$ from $\mathbf{T}v_g$ and all $\mathbf{F}v_{g_l}$ using the "last undetermined child" rule for or in $\mathbf{BC}$.

(ii) $\{v_g\}$ we still need to have $\{v_{g_j}\}$ for some $j \neq i$, which is equivalent to deducing $\mathbf{T}g$ from $\mathbf{T}g_j$ by applying the "up" rule for or in $\mathbf{BC}$.

(iii) $\{\neg v_{g_j}\}$ we still need to have $\{\neg v_g\}$, which is equivalent to deducing $\mathbf{F}g_j$ from $\mathbf{F}g$ by applying the "down" rule for or in $\mathbf{BC}$.

(iv) $\{\neg v_g\}$ we still need to have $\{v_{g_j}\}$ for all $j \neq i$, which is equivalent to deducing $\mathbf{F}g$ from all $\mathbf{F}v_l$, $1 \leq l \leq k$, using the "up" rule for or in $\mathbf{BC}$.

- $g = \mathsf{and}(g_1, \ldots, g_k)$: This is translated into

$$\{\neg v_{g_1} \vee \cdots \vee \neg v_{g_k} \vee v_g\}, \{\neg v_g \vee v_{g_1}\}, \ldots, \{\neg v_g \vee v_{g_k}\}.$$

  The deduction here is, in a sense, the dual of the one presented for the or gates, and thus rather similar to the discussion above. This is left for the reader to confirm.

Thus unit propagation is equivalent to the set of rules consisting of the rules (b)–(h) in $\mathbf{BC}$.

7 CONCLUSION

This work addresses the question of how restrictions on the use of the cut rule effect the proof complexity in Boolean circuit satisfiability checking based on tableaux. The tableau method in question consists of a complete and sound subset of the rules in the method introduced in [20].

The results show that restricting the application of the cut rule in any of the natural locality based ways considered (input cuts, top–down cuts, bottom–up cuts, input and top–down cuts) increases the proof complexity exponentially. Moreover, there are exponential differences between the proof complexity of all the restricted methods. The proofs rely on the resolution–boundedness of the methods and on properties of certain circuit gadgets such as a Boolean circuit representation of the well–known pigeon–hole principle.

The introduced tableau method can be seen as a generalisation of the Davis–Putnam method for CNF formulas obtained from Boolean circuits using Tseitin's translation. Thus the results show that locality based cut restrictions – such as splitting on the input gates only – have a worst–case exponential effect on the sizes of proofs in Davis–Putnam based satisfiability checkers, contradicting the common belief based on empirical results (see e.g. [31, 14]).

7.1 Further Work

We now present some further directions of research based on this work.

- **Empiric correspondence.** Can empiric correspondence be shown to backup the theoretical results, i.e., could we see the difference in restricting the application of the cut / splitting rule in e.g. state–of–the–art satisfiability solvers?

  In some satisfiability checkers such as `zChaff` [26] one can restrict the case splittings to a given *static* subset of variables. But in a more general view, the question of how to integrate different locality–based cut restrictions acting on a *dynamically* selected subset of variables into existing solvers in not a trivial one.

- **Cut heuristics.** The question of empiric correspondence leads thus to the problem of developing efficient cut heuristics, i.e., general methods for choosing gates on which the cut rule is applied. Development of efficient cut heuristics is a nontrivial task; the total number of gates in a circuit can be enormous compared to the number of input gates, so we should be able to select a small subset of the gates to which to apply the cut in the circuit. This problem has two sides.

  1. *How to determine on which gates we do not need to apply the cut rule.* A trivial example of this is that having $x = \mathsf{not}(y)$ implies that it is not necessary to apply the cut on both x and y. The question remains, are there general rules on what gates should be included in this set. The graphical view of Boolean circuit suggests e.g. that some kind of graph–analytic methods could be applied in identifying gates that should be in this set. For example, in the

main proofs of this work we considered circuits in which certain bottleneck gates are visible.

2. *How to select the gate on which the cut rule is applied next.* Having a subset of gates for candidates on which to apply the cut rule, how to *dynamically* choose one gate of these on which the cut is applied next in order to gain maximally from the cut?

- **Further deduction / pruning rules.** What happen to the proof complexity when different deduction / pruning rules such as *one–step lookahead*, *equivalence reasoning*, or *cone–of–influence* (see e.g. [20]) are introduced?

- **Learning.** We should be able to *learn* the reasons for contradictions during the satisfiability search, producing efficient further constraints to guide the search. The question is: do the main results of this work apply between the restricted variants of **BC** if different learning schemes (see e.g. [34, 26, 25, 1, 13]) are introduced?

Acknowledgements

This report is a reprint of my Master's thesis, with typos corrected only. I am greatly indebted to Professor Ilkka Niemelä and Doctor Tommi A. Junttila, my thesis supervisor and instructor, respectively. Without their knowledge, patience, and inspiration this work would not be as such. In addition I thank Emilia Oikarinen for numerous valuable comment on the work. The financial support of Academy of Finland under the project *Applications of Rule–Based Constraint Programming* (#53695) is gratefully acknowledged.

BIBLIOGRAPHY

[1] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[2] Paul Beame and Toniann Pitassi. Propositional proof complexity: past, present, and future. *Bulletin of the European Association for Theoretical Computer Science*, 65:66–89, June 1998.

[3] Armin Biere and Wolfgang Kunz. SAT and ATPG: Boolean engines for formal hardware verification. In *Proceedings of 20th IEEE/ACM International Conference on Computer Aided Design (ICCAD'02), San Jose CA, USA, November 2002*. IEEE Press, 2002.

[4] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer-Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer, 2001.

[5] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, July 2001.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference record of third annual ACM Symposium on Theory of Computing: papers presented at the symposium, Shaker Heights, Ohio, May 3–5, 1971*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[7] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 1979.

[8] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[9] Marcello D'Agostino and Marco Mondadori. The taming of the cut: Classical refutations with analytic cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.

[10] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2 - 2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, October 2002.

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

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

[13] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[14] Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 948–953. AAAI Press, July 26–30 1998.

[15] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: a survey. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.

[16] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2–3):297–308, 1985.

[17] Matti Järvisalo, Tommi A. Junttila, and Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for Boolean circuits. AI&M 15–2004, 8th International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, USA, January 4–6 2004. Proceedings available at `http://rutcor.rutgers.edu/%7Eamai/aimath04/`.

[18] Stasys Jukna. *Extremal Combinatorics: with Applications in Computer Science*. Springer-Verlag, Heidelberg, Germany, 2001.

[19] Tommi A. Junttila. BCSat 0.3 – a satisfiability checker for Boolean circuits. Computer program, 2001. Available at `http://www.tcs.hut.fi/Software/`.

[20] Tommi A. Junttila and Ilkka Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic – CL 2000; First International Conference*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 553–567, London, UK, 2000. Springer-Verlag.

[21] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359 – 363, 1992.

[22] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 1194–1201, Portland, Oregon, July 1996.

[23] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[24] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 366–371, Nagoya, Japan, August 23–29 1997.

[25] João P. Marques-Silva and Karem A. Sakallah. GRASP: a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, pages 220–227, 1997.

[26] 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*, pages 530–535. ACM, 2001.

[27] Anil Nerode and Richard A. Shore. *Logic for Applications*. Springer–Verlag, New York, Washington, USA, 2nd edition, 1997.

[28] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[29] David A. Plaisted and Steven A. Greenbaum. A structure–preserving clause form translation. *Journal of Symbolic Computation*, 2:193–304, 1986.

[30] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. American Mathematical Society, 1996.

[31] Ofer Shtrichman. Tuning SAT checkers for Bounded Model Checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification – CAV 2000; 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494, Chicago, IL, USA, 2000. Springer-Verlag.

[32] Raymond M. Smullyan. *First–Order Logic*. Springer–Verlag, Heidelberg, Germany, 1968.

[33] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Heidelberg, Germany, 1983.

[34] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285, Los Alamitos, CA, November 4–8 2001. IEEE Computer Society.

[35] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer-Verlag, July 27-30 2002.

ISBN 951-22-7020-X
ISSN 1457-7615