Chapter 1

Proof Complexity
Sam Buss and Jakob Nordström

Dear reviewers, This is a rough, but largely complete, version of the article. We will continue working on the paper, and substantial updates are still expected. But comments are greatly appreciated at any point. – Sam and Jakob

1.1. Introduction

The satisfiability problem (SAT) — i.e., to determine whether a given a formula in propositional logic has a satisfying assignment or not — is of central importance to both the theory of computer science and the practice of automatic theorem proving and proof search. Proof complexity — i.e., the study of the complexity of proofs and the difficulty of searching for proofs — joins the theoretic and practical aspects of satisfiability.

For theoretical computer science, SAT is the canonical $\text{NP}$-complete problem, even for conjunctive normal form (CNF) formulas [Coo71, Lev73]. In fact, SAT is very efficient at expressing $\text{NP}$-complete problems in that many of the standard $\text{NP}$-complete problems, including the question of whether a Turing machine halts within $n$ steps, have very efficient, almost linear time, reductions to the satisfiability of a CNF formula. Furthermore, many theoreticians tend to believe that the Strong Exponential Time Hypothesis (SETH) holds. SETH conjectures that any algorithm for solving CNF SAT must have worst case runtime $\approx 2^n$ on instances of CNF SAT involving $n$ variables [IP01, CIP09]. This hypothesis has been widely studied in recent years, and serves as a basis for proving conditional hardness results for other problems. In other words, CNF SAT serves as the canonical hard decision problem, and is frequently conjectured to require exponential time to solve. In any event, CNF SAT is thus widely conjectured to be highly infeasible.

In contrast, for practical theorem proving, CNF SAT is the core method for encoding and solving problems. On one hand, the expressiveness of CNF formulas means that a large variety of problems can be faithfully and straightforwardly encoded.

1 Formally, these reductions run in quasilinear time, i.e. time $n(\log n)^k$ for some constant $k$. For these quasilinear time reductions of the Turing machine halting problem to CNF SAT, see [Sch78, PF79, Rob79, Rob91, Coo88].
translated into CNF SAT problems. On the other hand, the message that SAT is hard to solve does not seem to have reached practitioners of SAT solving; instead, there has been enormous improvements in performance in SAT algorithms over the last six decade. Amazingly, state-of-the-art algorithms for deciding satisfiability — so-called SAT solvers — can routinely handle real-world instances involving hundreds of thousands or even millions of variables. It is a dramatic development that SAT solvers can often run in (close to) linear time!

Thus, theoreticians view SAT as being infeasible, while practitioners view it as being (often) feasible. There is no contradiction here. First, it is possible construct tiny formulas with just a few hundred variables that are totally beyond reach for even the best of today’s solvers. Conversely, the large instances which are solved by SAT solvers are based on problems which are “easy” but very large, for instance arising from software or hardware verification. What is mysterious is why these underlying easy but large problems remain easy after being translated to instances of SAT. Most SAT solvers are general-purpose and written in a very generic way that does not seem to exploit special properties of the underlying problem. Nonetheless, although SAT solvers will sometimes fail miserably, they succeed much more frequently than might be expected. This raises the questions of how practical SAT solvers can perform so well on many large problem and of what distinguishes problems that can be solved by SAT solvers from problems that cannot.

The best current SAT solvers are based on conflict-driven clause learning (CDCL) [MS99, MMZ+01]. Some solvers also incorporate elements of algebraic reasoning (e.g., Gaussian elimination) and/or geometric reasoning (e.g., linear inequalities), or use algebraic or geometric methods as the foundation rather than CDCL. Another augmentation of CDCL that has attracted much interest is extended resolution. How can we analyze the power of such algorithms? Our best approach is to study the underlying methods of reasoning and what they are able or unable to do in principle. This leads to the study of proof systems such as resolution, Frege systems, extended resolution, Nullstellensatz, polynomial calculus, cutting planes, et cetera. Proof complexity, as initiated in modern form by [CR79, Rec70], studies these system mostly from the viewpoint of complexity of static, completed proofs. With a few exceptions (notably automatizability), research in proof complexity ignores the constructive, algorithmic aspects of SAT solvers. Nonetheless, proof complexity has many useful implications and spinoffs for practical SAT solvers.

This article is intended as an overview of the connection between SAT solving and proof complexity aimed at readers who wish to become more familiar with either (or both) of these areas. We focus on the proof systems underlying current approaches to SAT solving. Our goal is first to explain how SAT solvers can be viewed as corresponding to proof systems and second to review some of the complexity results known for these proof systems. We will discuss resolution (corresponding to CDCL), polynomial calculus (corresponding to algebraic Gröbner basis computations), cutting planes (corresponding to geometric or so-called pseudo-Boolean solving), extended resolution (corresponding to DRAT), and will also briefly touch on Frege systems and quantified Frege systems. Lower

---

2A similar technique for constraint satisfaction problems (CSPs) was developed in [BS97].
bounds on the complexity of proofs in these proof systems show fundamental limitations on what one can hope for SAT solvers to achieve. Conversely, upper bounds on proof complexity can be viewed as challenges, for instance when actual SAT solver performance does not match what theory suggests should be possible.

We do not have space to cover all of proof complexity. Other nice surveys include [BP98a, Seg07] and the forthcoming book [Kra19]. Additionally, we would like to mention the authors’ own surveys [Bus12, Bus99, Nor13]. The present article is an adapted from and partially overlaps with the second author’s survey [Nor15], but has been thoroughly rewritten and substantially expanded with new material.

1.1.1. Outline of This Survey

The rest of this chapter is organized as follows. Section 1.2 presents a quick review of preliminaries. We discuss the resolution proof system and describe the connection to CDCL SAT solvers in Section 1.3 and then give an overview of some of the proof complexity results known for resolution in Section 1.4. In Section 1.5 we consider the algebraic proof systems Nullstellensatz and polynomial calculus, and also briefly touch on algebraic SAT solving. In Section 1.6 we move on to the geometric proof system cutting planes and the connections to conflict-driven pseudo-Boolean solving, after which we give an overview of what is known in proof complexity about different flavours of the cutting planes proof system in Section 1.8. We review extended resolution and DRAT in Section 1.9 and then continue to Frege proof systems, constant-depth Frege systems and quantified Frege systems in Sections 1.10, 1.11, and Section 1.12, respectively. Some concluding remarks are presented in Section 1.13.

1.2. Preliminaries

A Boolean variable $x$ ranges over values true and false; unless otherwise stated, we identify 1 with true and 0 with false. A literal $a$ over a Boolean variable $x$ is either the variable $x$ itself (a positive literal) or its negation $\overline{x}$ (a negative literal). We define $\overline{\overline{x}} = x$. It will sometimes be convenient to use the alternative notation $x^\sigma$, $\sigma \in \{0, 1\}$, for literals, where $x^1 = x$ and $x^0 = \overline{x}$. In other words, $x^\sigma$ is the literal that evaluates to true under the assignment $x = \sigma$.

A clause $C = a_1 \lor \cdots \lor a_k$ is a disjunction of literals over distinct variables. The empty clause, with $k = 0$, is denoted $\bot$. By convention clauses are not tautological; i.e., do not contain any variable and its negation. A clause $C'$ subsumes another clause $C$ if every literal from $C'$ also appears in $C$. In this case, $C'$ is at least a strong a clause as $C$. A $k$-clause is a clause that contains at most $k$ literals. A CNF formula $F = C_1 \land \cdots \land C_m$ is a conjunction of clauses. $F$ is a $k$-CNF formula if it consists of $k$-clauses. We think of clauses and CNF formulas as sets: the order of elements is irrelevant and there are no repetitions.

The width of a clause is the number of literals in the clause. Thus a $k$-clause has width $\leq k$. For simplicity of exposition, we will sometimes tacitly assume

---

3It can be noted, though, that SAT solvers generally implement formulas as multisets of clauses.
that formulas are $k$-CNF formulas for some fixed $k$. In particular, unless otherwise stated $k$ is assumed to be some arbitrary but fixed constant in what follows. There is a standard way to turn any CNF formula $F$ into 3-CNF by converting every clause of width $w > 3$
\[ a_1 \lor a_2 \lor \cdots \lor a_w \] (1.1a)
into the set of $w - 2$ many 3-clauses
\[ \{a_1 \lor a_2 \lor y_2 \} \cup \{y_{j-1} \lor a_j \lor y_j \mid 3 \leq j < w \} \cup \{y_{w-2} \lor a_{w-1} \lor a_w \}, \] (1.1b)
where the $y_j$'s denote new variables. This conversion to 3-CNF often does not change much from a theoretical point of view; however, there are some notable exceptions to this rule, which we will mention later.

A truth assignment is any mapping from variables to truth values 0 or 1. We allow truth assignments to be partial, i.e., with some variables left unassigned. A truth assignment is total if it assigns values to all variables under consideration. We represent a (partial) truth assignment $\rho$ as the set of literals set to true by $\rho$.

We write $\rho(x^\sigma) = 1$ if $x^\sigma \in \rho$, and write $\rho(x^\sigma) = 0$ if $x^{1-\sigma} \in \rho$. If $\rho$ does not assign any truth value to $x$, we write $\rho(x^\sigma) = \ast$. The assignment $\rho$ satisfies a clause $C$ provided it sets at least one literal of $C$ true; it falsifies $C$ provided it sets every literal in $C$ false. It satisfies a formula $F$ provided it satisfies every clause in $F$.

A formula $F$ is satisfiable provided some assignment satisfies it; otherwise it is unsatisfiable. $F$ logically implies a clause $C$ provided that every total truth assignment $\rho$ which satisfies $F$ also satisfies $C$. In this case, we write $F \Vdash C$. Since $\bot$ is unsatisfiable, $F \Vdash \bot$ is equivalent to $F$ being unsatisfiable.

Proof complexity often deals with asymptotic upper and lower bounds, generally expressed by using “big-O” notation. The commonly used notations include $f(n) = O(g(n))$, $f(n) = o(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$, and $f(n) = \Theta(g(n))$, where $f(n)$ and $g(n)$ are always nonnegative. We write $f(n) = O(g(n))$ to express that $\exists c > 0 \forall n \ f(n) \leq c \cdot g(n)$, i.e., that $f$ grows at most as quickly as $g$ asymptotically. We write $f(n) = \Theta(g(n))$ to express that $\forall c > 0 \exists n_0 \forall n \geq n_0 \ f(n) \leq c \cdot g(n)$, i.e., that $f$ grows strictly more slowly than $g$. When $g(n) > 0$ always holds, $f(n) = o(g(n))$ is equivalent to $\lim_{n \to \infty} f(n)/g(n) = 0$. The notations $f(n) = \Omega(g(n))$ and $f(n) = \omega(g(n))$, mean that $g(n) = O(f(n))$ and $g(n) = o(f(n))$, respectively, or, in words, that $f$ grows at least as fast or strictly faster than $g$. We write $f(n) = \Theta(g(n))$ to denote that both $f(n) = O(g(n))$ and $g(n) = O(f(n))$ hold, i.e., that asymptotically speaking $f$ and $g$ are essentially the same function up to a constant factor. Big-O notation is often used in subexpressions. For example, we write $f(n) = 2^{(1-o(1))n}$ to mean that for any fixed $\epsilon > 0$, we have $f(n) > 2^{(1-\epsilon)n}$ for all sufficiently large $n$.

We typically use $N$ to denote the size of a formula $F$, namely the total number of occurrences of literals in $F$. (For a $k$-CNF formula, $N$ may instead denote the number of clauses in $F$; this differs from the size of $F$ by at most a linear factor, in fact by only a constant factor if clauses have constant width.)

One of the central tasks of proof complexity is to prove upper or lower bounds on the complexity of proofs. That is, for a fixed proof system $P$ and a class $F$ of unsatisfiable formulas, we may let $f(N)$ denote the worst-case complexity (say,
the size or the length — these notions will be defined more precisely in the ensuing sections for each individual proof system) of \( \mathcal{P} \)-refutations of formulas \( F \in \mathcal{F} \) of size \( N \); then we wish to provide sharp bounds on the growth rate of \( f(N) \). Small upper bounds on \( f(N) \) indicate that \( \mathcal{P} \) is an efficient proof system for \( \mathcal{F} \)-formulas. Small upper bounds also suggest the possibility of efficient search procedures for finding refutations of formulas \( F \in \mathcal{F} \). Large lower bounds on \( f(N) \) indicate that \( \mathcal{P} \) cannot be efficient for all \( \mathcal{F} \)-formulas.

A satisfiable formula always has a short certificate, namely a satisfying truth assignment. For this reason, it often makes sense to focus on unsatisfiable formulas when proving complexity results. Most of the proof systems relevant for SAT solvers, (such as resolution, cutting planes, polynomial calculus, et cetera) are refutation systems; that is, they produce refutations of unsatisfiable formulas \( F \).

Some propositional proof systems (such as Frege systems) are proof systems and produce proofs of valid formulas instead of refutations. There is a duality between refutations and proofs, in that \( F \) is unsatisfiable iff its negation, the DNF formula \( \neg F \) is valid. We frequently use the term proof system for both refutation systems and proof systems; in most the cases in this article, they are actually refutation systems. When discussing refutation systems, we will frequently use the term proof (of unsatisfiability) for \( F \) to refer to a refutation of \( F \).

where \( n \leq m \) are assumed to be integers. We use the notation \( [n] = \{1, 2, \ldots, n\} \) for \( n \) an integer. We use \( \mathbb{N} = \{0, 1, 2, 3, \ldots\} \) to denote the set of all natural numbers and \( \mathbb{N}^+ = \mathbb{N} \setminus \{0\} \) the set of positive integers.

1.3. Resolution and CDCL SAT solvers

The resolution proof system [Bla37, DP60, DLL62] is a refutation system that works directly with clauses. A resolution derivation \( \pi \) of a clause \( D \) from a CNF formula \( F \) is a sequence of clauses \( \pi = (D_1, D_2, \ldots, D_{L-1}, D_L) \) such that \( D = D_L \) and each clause \( D_i \) is either

1. an axiom clause \( D_1 \in F \), or
2. a clause of the form \( D_i = B \lor C \) derived from clauses \( D_j = B \lor x \) and \( D_k = C \lor \bar{x} \) for \( j,k < i \) by the resolution rule

\[
\begin{align*}
B \lor x & \quad B \lor C \\
C \lor \bar{x} & \quad B \lor C 
\end{align*}
\]

(1.2)

We call \( B \lor C \) the resolvent over \( x \) of \( B \lor \bar{x} \) and \( C \lor \bar{x} \).

We write \( \pi : F \vdash D \) to denote that \( \pi \) is a derivation of \( D \) from \( F \); we write \( F \vdash \bot \) to denote that some such \( \pi \) exists. When \( D \) is the empty clause \( \bot \), we call \( \pi \) a resolution refutation of \( F \), and write \( \pi : F \vdash \bot \). The length, or size, of a resolution derivation/refutation is the number of clauses in it.

A resolution derivation \( \pi \) can be represented as a list of clauses annotated with explanations for each clause how it was obtained. This is illustrated in Figure 1.1a for refutation of the CNF formula \( F \)

\[
(x \lor y) \land (x \lor \bar{y} \lor z) \land (\bar{x} \lor z) \land (y \lor \bar{z}) \land (\bar{y} \lor \bar{z}) .
\]

(1.3)

A derivation \( \pi \) can also be represented by a directed acyclic graph (DAG) \( G_\pi \) in the following way. The vertices of \( G_\pi \) are \( \{v_1, v_2, \ldots, v_L\} \), with vertex \( v_i \) labelled
Chapter 1. Proof Complexity

1. \( x \lor y \) Axiom
2. \( x \lor \neg y \lor z \) Axiom
3. \( \neg x \lor z \) Axiom
4. \( y \lor \neg z \) Axiom
5. \( \neg y \lor \neg z \) Axiom
6. \( \neg x \) Res(4, 5)
7. \( \neg x \) Res(3, 6)
8. \( x \lor \neg y \) Res(2, 6)
9. \( x \) Res(1, 8)
10. \( \bot \) Res(7, 9)

(a) Resolution refutation as an annotated list. (b) Resolution refutation as a DAG.

Figure 1.1: Resolution refutation of the CNF formula (1.3).

by the clause \( D_i \). The sources (also called leaves) of the DAG are vertices labelled with the axiom clauses in \( F \). Without loss of generality there is a unique sink and it is labelled with \( D_L \). If \( \pi \) is a refutation, the sink is labelled with \( \bot \). Every vertex that is not a source has indegree two and is the resolvent of its two predecessors. See Figure 1.1b for an example.

A resolution refutation \( \pi \) is tree-like if \( G_\pi \) is a tree, or equivalently, if every clause \( D_i \) in the refutation is used at most once as an hypothesis in an application of the resolution rule. Note that it is permitted that clauses are repeated in the sequence \( \pi = (D_1, D_2, \ldots, D_{L-1}, D_L) \), so different vertices in \( G_\pi \) can be labelled by the same clause if need be (though every repetition counts towards the size of the refutation). The refutation in Figure 1.1b is not tree-like since the clause \( \neg x \) is used twice, but it could be made tree-like if we added a second derivation of \( \neg x \) from \( y \lor \neg z \) and \( \neg y \lor z \). More generally, any derivation can be converted into a tree-like derivation by repeating subderivations, but possibly at the cost of an exponential increase in the size of the proof [Tse68b, BIW04].

Soundness and completeness are fundamental properties for resolution, and indeed for almost any self-respecting proof system. For resolution, they are as follows.

**Theorem 1.3.1.** Let \( F \) be a CNF formula and \( C \) be a clause.

- (Soundness) If \( F \vdash C \), then \( F \models C \). In particular, if \( F \) has a resolution refutation, then \( F \) is unsatisfiable.
- (Completeness) If \( F \models C \), then there is a clause \( C' \subseteq C \) such that \( F \vdash C' \). In particular, if \( F \) is unsatisfiable, then \( F \) has a resolution refutation.

For technical reasons it is sometimes convenient to allow also the weakening rule:

\[
\frac{B}{B \lor C}, \tag{1.4}
\]

Any other sinks are parts of superfluous subderivations that can be removed.
which allows inferring a subsumed clause (i.e., a strictly weaker clause) from an already derived clause. Resolution with weakening is also sound and complete; in fact, completeness holds in the stronger sense that if $F \models C$, then $C$ has a resolution-plus-weakening derivation from $F$. It is not hard to show that weakening inferences in a resolution refutation can be eliminated without increasing the complexity of the proof. This holds for all the notions of complexity we discuss later, including length, width, and space.

An important restricted form of resolution is regular resolution. A resolution refutation $\pi$ is regular provided that no variable is resolved on more than once along any path in the DAG $G_\pi$. For example, the refutation of Figure 1.1 is regular. The soundness of regular resolution is immediate from the soundness of resolution. Regular resolution is also complete \cite{DP60}. However, \cite{AJPU07, Urq11} have shown that there is a family of CNF formulas $F_n$ which have polynomial size resolution refutations, but are such that, letting $c = c(n)$ be the length of $F_n$ (i.e., the number of clauses in $F_n$), the shortest regular resolution refutation of $F_n$ has size $2^{c - o(1)}$. This gives an exponential separation between regular resolution and (general) resolution. In particular, regular resolution does not “simulate” resolution, as we define next.

Definition 1.3.2. Let $\mathcal{P}$ and $\mathcal{P}'$ be propositional proof systems. For $F$ an unsatisfiable (CNF) formula, define $c(F)$ and $c'(F)$ to equal the size of the shortest $\mathcal{P}$-proof (respectively, $\mathcal{P}'$-proof) of $F$, where size is measured in terms of the number of symbols in the proofs. We say that $\mathcal{P}'$ polynomially simulates $\mathcal{P}$ provided that $c'(F) = c(F)^{O(1)}$.

In other words, shortest $\mathcal{P}'$-proofs are at most polynomially bigger than $\mathcal{P}$-proofs.\footnote{In the literature, the terms “simulates”, “p-simulates” and “polynomially simulates” are used more or less interchangeably, but sometimes coupled with the priviso that there is a polynomial time algorithm which can produce a $\mathcal{P}'$-proof given any $\mathcal{P}$-proof as input.}

As already mentioned, resolution can simulate resolution without weakening. On the other hand, tree-like resolution cannot simulate resolution, or even regular resolution \cite{Tse68a}.

Another interesting restriction on resolution which is relevant for CDCL SAT solvers is trivial resolution. A resolution derivation $\pi$ is trivial derivation of $D$ from $F$ if it can be written as a sequence of clauses $\pi = (D_1, \ldots, D_L = D)$ such that

1. $\pi$ is input (and hence linear), i.e., $D_1 \in F$, and for all $i \geq 1$ it holds that $D_{2i} \in F$ and that $D_{2i+1}$ is the resolvent of the latest derived clause $D_{2i-1}$ with the axiom $D_{2i}$.
2. $\pi$ is regular, i.e., no variable is used more than once as the resolution variable.

The sequence $\pi$ is a trivial resolution refutation provided $D_L$ is $\bot$. Trivial resolution is sound but not complete. For example (and jumping ahead a bit), the formula \cite{L3} is not refutable by unit propagation and thus does not have a trivial resolution refutation.
Chapter 1. Proof Complexity

A unit clause is a clause of width 1. A unit resolution refutation is a resolution refutation in which each resolution inference has at least one unit clause as a hypothesis. Unit resolution is sound but not complete. Indeed, it is very closely connected to trivial resolution [BKS04]. For a CNF formula, and \( C = a_1 \lor \cdots \lor a_k \) a clause, there is a trivial resolution derivation of \( C \) from \( F \) if and only if there is a unit resolution refutation of \( F \cup \{ \overline{a}_1, \overline{a}_2, \cdots, \overline{a}_k \} \), namely, of \( F \) plus \( k \) many unit clauses. In particular, \( F \) has a trivial resolution refutation exactly when \( F \) has a unit resolution refutation.

1.3.1. DPLL, CDCL and resolution

The most successful present-day SAT solvers are based on conflict-driven clause learning (CDCL) search procedures. CDCL algorithms have been highly refined and augmented with both practical optimizations and sophisticated inference techniques; however, they are based on four main conceptual ingredients:

(a) **DPLL**, a depth-first search procedure for a satisfying assignment.

(b) **Unit propagation** is typically used to guide the DPLL search procedure.

(c) A **clause learning** algorithm which attempts to prune the search space by inferring ("learning") clauses whenever the search procedure falsifies (finds a "conflict") and has to backtrack.

(d) A **Restart** strategy for stopping a depth-first search and starting a new one.

We will discuss these four ingredients one at a time. Our treatment will be informal, without including all necessary details, as we presume most readers are familiar with the concepts. For a more detailed treatment, see the chapters on Complete Algorithms [DP09] and CDCL Solvers [MLM09] in this handbook.

The first ingredient, DPLL, is named after the four collective authors, Davis, Putnam, Logeman and Loveland, of two of the primary sources for resolution and automated solvers, namely [DP60] and [DLL62]. The DPLL procedure is given as input a CNF formula \( F \), i.e., a set of clauses. Its task is to determine if \( F \) is satisfiable, and if so to find a satisfying assignment. The algorithm maintains a partial truth assignment \( \rho \) and run recursively. Initially, \( \rho \) is the empty assignment with all variables unset. Each time the recursive procedure is called it does the following steps:

- If the partial assignment \( \rho \) falsifies some clause of \( F \), it backtracks by returning "false".
- If \( \rho \) satisfies \( F \), it terminates and outputs \( \rho \) as a satisfying assignment.
- It picks some unset literal, \( a \), called the decision literal.
- It extends \( \rho \) to set \( a \) true, and makes a recursive call.
- If that call returns (if so, it returns "false"), then \( \rho \) is updated to set \( a \) false and another recursive call is made.
- If that call also returns, then there is no satisfying assignment extending \( \rho \).

The routine removes \( a \) from the domain of \( \rho \) and returns "false".

When the top-level procedure returns "false", \( F \) must be unsatisfiable.

There is an exact correspondence between the DPLL search procedure and
Chapter 1. Proof Complexity

In one direction, any tree-like regular resolution refutation $\pi$ containing $S$ occurrences of clauses, corresponds to a DPLL search tree with $S$ nodes (each node an invocation of the recursive procedure). This is done by letting the DPLL search procedure traverse the proof tree $G_{\pi}$ starting at $\bot$ in a depth-first fashion. When the traversal of $G_{\pi}$ is at a node labelled with clause $C$, the partial assignment $\rho$ falsifies $C$. Conversely, any DPLL search with $S$ calls to the recursive procedure gives rise to a tree-like resolution refutation with $\leq S$ clauses. For this, the tree-like refutation is formed from the leaves starting with clauses falsified by the deepest recursive calls. Pairs of recursive calls are joined by resolution inferences as needed (possibly with parts of the DPLL search tree being pruned away).

The second ingredient is unit propagation. Unit propagation can be viewed as a way to guide the DPLL search procedure, but it becomes much more important when used with clause learning, as will be discussed momentarily. Suppose that $F$ contains a clause $C = a_1 \lor \cdots \lor a_k$ and that the current partial assignment $\rho$ has set all but one of the literals in $C$ false, and that the remaining literal $a_i$ is unset. Then unit propagation is the operation of setting $a_i$ true and adding it to $\rho$. This can be done without loss of generality, since setting $a_i$ false would falsify $C$.

Unit propagation is directly related to the unit resolution discussed earlier. Let $F'$ be the CNF formula consisting of $F$ plus the unit clauses $a$ for all literals $a$ such that $\rho(a) = 0$. Then unit propagation starting with $F$ and the partial assignment $\rho$ will falsify some clause of $F$ if and only if $F'$ has a unit resolution refutation.

With unit propagation, the recursive procedure for the DPLL algorithm is as follows:

- It saves the assignment $\rho$ as $\rho_0$. Then $\rho$ is extended by unit propagation for as long as possible.
- If $\rho$ falsifies some clause of $F$, it backtracks by restoring $\rho$ to equal $\rho_0$ and returning “false”.
- If $\rho$ satisfies $F$, it terminates and outputs $\rho$ as a satisfying assignment.
- It picks some unset literal, $a$, called the decision literal.
- It extends $\rho$ to set $a$ true, and makes a recursive call.
- If that call returns (if so, it returns “false”), then $\rho$ is updated to set $a$ false and another recursive call is made.
- If that call also returns, there is no satisfying assignment extending $\rho$. The routine restores $\rho$ to equal $\rho_0$ and returns “false”.

Values of $\rho$ can be set either as decision literals or by unit propagation. Each set literal is assigned a decision level: the decision level is incremented when setting a decision literal and the unit propagated literals are given the decision level of the last decision literal.

The third ingredient is a crucial one: the use of clause learning to infer new clauses. This modifies the first step of DPLL algorithm to also add clauses

---

8Any tree-like resolution refutation can be converted to be a regular, tree-like refutation by pruning away parts of the proof as needed to make it regular. Thus, this is also an exact correspondence between the DPLL search procedure and shortest tree-like refutations.
Chapter 1. Proof Complexity

to the formula; namely, when \( p \) falsifies some clause of \( F \), then *clause learning* is used to infer one or more clauses and add them to \( F \). The intent is that the learned clauses will be used in the future to prune the search space and avoid again searching the same impossible settings to literals. The most common clause learning methods will infer only clauses \( C \) which can be derived from \( F \) by trivial resolution. This is based on the equivalence between trivial resolution refutations and unit resolution refutations and the fact that DPLL/CDCL uses unit propagation to find contradictions.

The most common way of finding a clause \( C \) to learn is the first-UIP (1UIP) method, as is illustrated in Figure 1.2. But for purposes of the connection between clause learning and trivial resolution, the key point is that the learned clause is chosen to be a clause \( C = a_1 \lor \ldots \lor a_k \) with the two following properties:

1. \( C \) is falsified by \( p \), i.e., \( \rho(a_i) = 0 \) for each \( a_i \) in \( C \), and
2. there is a unit resolution refutation of \( F \) plus the \( k \) unit clauses \( \{a_i \mid i \in [k]\} \).

When \( C \) is chosen in this way, \( C \) is called a *reverse unit propagation (RUP)* or an *asymmetric tautology (AT)* for \( F \).

By the correspondence between trivial resolution and unit resolution, this is equivalent to there being a trivial resolution derivation of \( C \) from \( F \).

A third property that typically holds for a learned clause \( C \) is

3. (UIP property.) \( C \) contains exactly one literal \( a_i \) that was set false at the top decision level; the rest of the literals in \( C \) were set false at lower decision levels. The literal \( a_i \) is called the *UIP literal.*

When \( C \) is chosen so that \( a_i \) is as “close as possible to the contradiction”, then this is called first-UIP (1UIP) learning, and \( a_i \) is called the 1UIP literal. Being as close as possible to the contradiction means that there is no other UIP literal \( b \) such that \( b \) is implied by unit propagation from \( F \) and the unit clauses expressing that \( C \) is false. There is always at least one UIP literal, since the decision literal the top level will be a UIP (if there is a contradiction).

An example of first-UIP clause learning is given in Figure 1.2. This shows a decision literal \( x \), three literals \( a, b, c \) set true at lower decision levels and the conflict graph at the decision level of \( x \). The first-UIP literal is \( y \); the dashed line shows the portion of the conflict graph that depends on \( y \): the learned clause is \( \pi \lor \tilde{b} \lor \tau \lor \gamma \) as \( a, b, c, y \) are the four literals that “support” this portion of the conflict graph. Figure 1.3 shows the unit propagations caused by setting \( x \) to 1 and the corresponding trivial resolution derivation of the learned clause. It uses the notation \( x \overset{\text{dec}}{\leftarrow} 1 \) to show \( x \) being set true as a decision literal; and \( \ell \overset{C}{\leftarrow} \sigma \) to show the literal \( \ell \) being set to the truth value \( \sigma \) by unit propagation using clause \( C \).

A nice feature of first-UIP clause learning is that the learned clause is asserting in that it contains exactly one literal \( a_i \) which has not been set false at a lower decision level. This literal can now immediately be set true by unit propagation. For example, the clause learned in Figure 1.2 is asserting: \( a \) and \( b \) have been set at lower levels, \( y \) can be inferred from the learned clause by unit propagation using clause \( C \).

\[\text{See } \{\text{GN03, Van08}\} \text{ for the original definitions of RUP; the modern terminology AT is discussed in } \{\text{HJB10, JHB12}\}.\]
Chapter 1. Proof Complexity

Figure 1.2: Example of a conflict graph and first-UIP learning formed from the eight clauses $\neg x \lor z$, $\neg x \lor \neg y$, $\neg y \lor t$, $\neg y \lor \neg u$, $\neg a \lor w$, $\neg a \lor \neg v$, $\neg b \lor \neg c$, and $\neg c \lor w$. The decision literal is $x$, the first-UIP literal is $y$, and the literals $a, b, c$ are from lower decision levels. The learned clause is $\neg b \lor \neg c \lor y$ (see Figure 1.3).

The unit propagation which set $c$ true does not affect the learned clause since $c$ was inferred at an earlier decision level. However, clause minimization based on self-subsumption [SB09, HS09] can take this into account to learn the smaller clause $\neg b \lor \neg c$. The most direct way to incorporate both unit propagation and clause learning (but not backjumping or restarts), in DPLL is as follows. We call this algorithm DPLL with Clause Learning (DPLL-CL). DPLL-CL recursively performs the following:

- The assignment $\rho$ is saved as $\rho_0$.
- Loop:
  - Extend $\rho$ by unit propagation.
  - If $\rho$ falsifies some clause of $F$, then
    - optionally learn one or more clauses $C$ and add them to $F$, and
    - restore $\rho$ to equal $\rho_0$ and return “false”.
  - If $\rho$ satisfies $F$, terminate and output $\rho$ as a satisfying assignment.
  - Pick some unset literal, $a$, as the decision literal.
  - Extend $\rho$ to set $a$ true, and make a recursive call to DPLL-CL.
  - If that call returns, unset the value of $a$ (so $a$ is no longer a decision literal). Then extend $\rho$ by unit propagation.
  - If the value of $a$ was set by unit propagation (if so, it is false), continue (with the next loop iteration).
  - Otherwise, update $\rho$ to set $a$ false as a decision literal.
  - Make another recursive call to DPLL-CL.
  - If that call also returns, there is no satisfying assignment extending $\rho_0$. Restore $\rho$ to equal $\rho_0$ and return “false”.

The DPLL-CL algorithm above has been formulated to be faithful to the idea of DPLL as a depth-first search procedure, but updated with unit propagation.
(a) The unit propagations triggered by setting $x$ true.

(b) The trivial resolution derivation of $a \lor b \lor c \lor y$ (written bottom-up).

Figure 1.3: The unit propagation derivation of a contradiction shown in Figure 1.2 and the trivial resolution derivation of the learned clause. On the left, in (a), the variables are listed in order in which they were set, namely $x$ as a decision literal, then $z$, $y$, $t$, $u$, $s$, $v$, and $w$ by unit propagation, until finding a contradiction $\perp$.

The 1UIP clause $\pi \lor b \lor \pi \lor \pi \lor y$ is found by traversing backwards through the unit propagated literals starting with the clause $t \lor \pi \lor \pi \lor y$ that was falsified. Each time a literal is reached that is set false in the current clause, a resolution inference is implicitly applied to update the current clause. (Note how $s$ is skipped as it does not appear negated in $b \lor \pi \lor t \lor \pi \lor y$.) Upon obtaining a clause (in this example, $\pi \lor b \lor \pi \lor \pi \lor y$) that contains only a single literal (in this example, $y$) at the highest decision level, we have reached the 1UIP literal and have obtained a trivial resolution refutation of the 1UIP learned clause, as shown in (b) on the right.
Chapter 1. Proof Complexity

and clause learning. Like the basic DPLL algorithm, it acts to set a literal $a$ first true and then false. The first recursive call to DPLL-CL has the decision literal $a$ set true. If that call returns, the newly learned clauses may already be enough to set $a$ false by unit propagation. But if not, $a$ can be set false as a decision literal anyway. For example, see the 1UIP clauses learning of Figures 1.2 and 1.3: there the learned clause allows $y$ to be set false by unit propagation, but it does not allow $\pi$ to be derived by unit propagation. Nonetheless $\pi$ is a consequence of the literals set in $\rho$.

The DPLL-CL algorithm as described above still lacks an important component of the CDCL algorithm, namely “backjumping”. Backjumping, also called nonchronologic backtracking, means backtracking multiple decision levels at once. For an example of a common way to use backjumping, refer to the conflict graph and 1UIP learned clause of Figures 1.2 and 1.3. There, $x$ has been set as a decision literal with decision level $lev(x)$. The literals $a$, $b$ and $c$ were set with strictly lower decision levels $lev(a)$, $lev(b)$ and $lev(c)$; let $L$ denote the maximum of these three levels. Once the 1UIP clause $\pi \lor b \lor c \lor y$ has been learned, the literal $y$ can be set true with decision level $lev(y) = L$ by unit propagation. To readily take advantage of this, the CDCL algorithm uses backjumping (nonchronological backtracking) to backtrack to decision level $L$.

Another reason for backtracking is that it can be viewed a miniature version of restarting. As discussed below, a restart consists of backtracking to level 0, and restarts are very important for improving the efficiency of CDCL.

We give an abstract formulation of the CDCL next. Unlike our earlier algorithms, it is not implemented as a recursive procedure; this is because backjumping allows backtracking multiple decision levels at once. Instead, the CDCL algorithm maintains a current decision level $L$. Whenever a literal $b$ is set, either as a decision literal or by unit propagation, it is given the current decision as its level, denoted $lev(b)$. The CDCL algorithm is as follows; its input is a formula $F$.

- **Initialize**: Set $L = 0$ and $\rho$ to be the empty assignment.
- **Loop**:
  - Extend $\rho$ as long as possible by unit propagation.
  - If $\rho$ satisfies all clauses of $F$, return $\rho$ as a satisfying assignment.
  - If $\rho$ falsifies some clause of $F$, then
    - If $L = 0$, return “Unsatisfiable”. Otherwise,
    - optionally learn one or more clauses $C$ and add them to $F$,
    - choose a backjumping level $L' < L$,
    - unassign all literals set at levels $> L'$, set $L = L'$, and
    - continue (with the next iteration of the loop).
  - Pick some unset literal, $a$, as the decision literal.
  - Increment $L$, and set $a$ true.
  - Continue (with the next iteration of the loop)

The CDCL algorithm as described above does not specify how to implement clause learning and choose the backjumping level $L'$, but a typical implementation would learn a 1UIP clause; this clause is asserting in that it allows new literal to

---

8This will not generally be true in the CDCL algorithm described next, since CDCL allows backjumping to backtrack multiple decision levels.
be set by unit propagation, and the backjump level $L'$ is set to equal the minimum decision level at which this literal can be unit propagated.

The partial assignment $\rho$ will set a literal $a$ true at level $0$ if and only if the unit clause $a$ is in $F$. It follows that the CDCL algorithm returns “Unsatisfiable”, then the current formula $F$ is unsatisfiable. If clause learning only learns clauses that are consequences of earlier clauses, then also the original (input) formula $F$ is unsatisfiable.

If the CDCL algorithm uses 1UIP clause learning, or other clause learning schemes based on the conflict graph, then the learned clauses will be derivable by trivial resolution derivations (by the correspondence between trivial resolution and unit resolution). Thus, if the CDCL algorithm returns “Unsatisfiable”, the original formula $F$ has a resolution refutation of size polynomial in the number of clauses learned by the CDCL algorithm. Figure 1.4a shows an example of a full CDCL refutation for the formula

$$F = (\overline{u} \lor w) \land (u \lor x \lor y) \land (x \lor \overline{y} \lor z) \land (\overline{y} \lor z) \land (x \lor \overline{z}) \land (u \lor w) \land (u \lor w) \ (1.5)$$

based on 1UIP clause learning. As shown, the CDCL search finds three conflicts. The first conflict is found after $\overline{w}$ and $\overline{x}$ are chosen as decision literals at decision levels 1 and 2. The resulting learned clause $u \lor x$ is asserting, so $x$ is set true by unit propagation at level 1. The second conflict learns the unit clause $\overline{x}$, which is an asserting clause setting $x$ false at level 0. The CDCL solver now performs backjumping (nonchronological backtracking); namely, it backtracks to unset all values set at decision levels above the level of the asserted literal $x$. In this case, backjumping involves backtracking out of decision level 1, unassigning $w$ and $u$, so that $w$ is no longer a decision literal. The third conflict arises at level 0, and completes the CDCL procedure.

Figure 1.4b shows the corresponding resolution refutation of $F$. Note that each conflict forms a trivial resolution derivation relative to the current CNF formula $F$ as augmented with learned clauses.

The fourth ingredient for CDCL algorithms is the use of restarts. A restart consists of halting the CDCL search, while preserving the clauses learned from $F$ so far, then starting a new CDCL search. The new CDCL search can use different decision literals, and this may provide an advantage in searching for either a satisfying assignment or a refutation. It might perhaps seem somewhat surprising, but in practice, the use of restarts is highly beneficial for CDCL solvers. As is discussed later in this section, there are also theoretical reasons why restarts may be beneficial. There are several intuitions of why restarts might be useful; but ultimately, it comes down to the fact that doing a restart allows the CDCL

---

9Inprocessing rules, including the RAT inferences discussed later, can learn clauses that are not consequences of the current formula $F$. Some of these inferences, such as the pure literal inference are benign. A literal is pure if it does not appear negatively in any clause, and any pure literal may be introduced as a unit clause without changing satisfiability. This justified by the fact that any satisfying assignment will w.l.o.g. satisfy all pure literals. However, some inprocessing rules can change the satisfying assignments for $F$ in essential ways. In this case, it is necessary to keep track of how satisfying assignments are modified. For this, see for instance, the discussion of “solution reconstruction” in [JHB12].

10This already fits into the CDCL algorithm described earlier, as setting the new (backjumping) decision level $L'$ equal to 0 causes a restart.
Chapter 1. Proof Complexity

(a) A CDCL refutation of the CNF formula (1.5). Decision variables are shown in diamonds. Unit propagations are shown in rectangles. The notation $x \leftarrow C$ means unit propagation using clause $C$ sets $x$ to 0. Dashed diamonds and boxes indicate assignments that were preserved after the previous conflict. Darker ovals indicate learned clauses.

(b) The corresponding resolution refutation.

Figure 1.4: A complete run of the CDCL procedure showing the unsatisfiability of the CNF formula (1.5), and a resolution refutation.
search to try out a different choice of decision literals. There has been extensive investigations of when to do restarts and how to choose decision literals, such as using VSIDS \[ MMZ^01 \] or phasing-saving \[ PD07 \]. The details of restart strategies are beyond the scope of this survey; but a good discussion can be found in \[ AST12, Hua07 \].

Yet another important ingredient of CDCL solvers is the use of clause deletion. Clause deletion means removing some of the learned clauses. This helps reduce memory usage and, even more importantly, allows unit propagation to run faster. The disadvantage is that clause deletion might delete clauses that would have been useful in the future. Some theoretical tradeoffs about the efficiency of clause deletion will be discussed later in the context of resolution space in Section 1.4.3.

The output of a SAT solver given a CNF formula \( F \) will be either a satisfying assignment for \( F \) or the assertion that \( F \) is unsatisfiable. In the second case, it can be useful to not just assert \( F \) is unsatisfiable, but also to produce a refutation of \( F \). Of course, the execution of the SAT solver can serve as a kind of proof; but this is not optimal, first because there could be bugs in the design or implementation of the SAT solver, and second because it does not correspond to a useful proof system. A better option is to output a resolution refutation: this is always possible for the CDCL constructions we have discussed so far, as illustrated in Figure 1.4; however, the resolutions proofs may be excessively long. A way to get more concise certificate (see \[ GN03, Van08, HJW13a \]) is to use a proof trace consisting of the sequence of learned clauses generated during the CDCL search (preferably with unneeded learned clauses omitted). For instance, in the example in Figure 1.4 the proof trace is just the sequence of clauses \( u \lor x, x, \bot \). Such a proof is called a RUP proof, since each clause follows by reverse unit propagation from \( F \) plus the preceding clauses in the sequence. The property of being RUP clause is checkable in polynomial time using unit propagation; thus the correctness of a RUP proof trace can be checked in time polynomially bounded by the size of \( F \) and the total size of the proof trace.

Nowadays, RUP proof traces have been supplanted by more sophisticated DRAT proofs \[ HHJW13 \]. DRAT proofs generalize RUP proofs, and also can simulate a wider range of preprocessing and inprocessing techniques. A very important aspect of modern solvers is that, before the main CDCL search algorithm starts, extensive preprocessing of the input is performed. Inprocessing uses a number of techniques which, although known to be theoretically very bad in the worst cases, can be very helpful in practice. Some solvers, such as Lingeling \[ Bie10 \], even interleave preprocessing techniques with CDCL search; this is known as inprocessing \[ JHB12 \]. Most, though not all, of these preprocessing and inprocessing techniques can also be formalized within resolution and hence RUP proofs. Some of them, however, require DRAT. In fact, DRAT proofs can even simulate extended resolution \[ 11 \] as will be discussed later in Section 1.9.

\[ 11 \text{It is important to note, however, that the power of DRAT proofs to simulate extended resolution is (at least as of yet) a purely theoretical result. Many reasoning techniques that, while powerful, fall well short of extended resolution, cannot currently be formalized in an efficient way in DRAT, two notable examples being cardinality reasoning and Gaussian reasoning. While it is possible in principle to handle cardinality reasoning or Gaussian reasoning in DRAT \[ HBH15, BT15 \], practical attempts in present day CDCL solvers have so far failed (but see}
We next turn to the question of the complexity of proofs as generated by CDCL solvers. As discussed above, the DPLL method corresponds to tree-like resolution, which can be exponentially worse than (general, dag-like) resolution. Since CDCL is only looking for structurally very restricted proofs, it is natural to ask how efficient CDCL proof search can be compared to the best possible general resolution proof. (Henceforth, when we talk about CDCL, we mean CDCL with restarts, unless explicitly stated otherwise.) Note that if the formula is exponentially hard for resolution, then basic CDCL search cannot be expected to run fast since it is searching for a resolution proof. But what we can do is to benchmark the solver against the best possible outcome, i.e., the most efficient proof. Thus, we can ask whether a CDCL search strategy can implement efficient proof search in the sense that, for unsatisfiable formulas, the CDCL algorithm finds a resolution refutation which is no more than polynomially longer than, say, than the shortest possible resolution refutation.

This turns out to be a deep and difficult question to answer. What we know so far is that one probably should not expect a fully constructive affirmative answer, since this would imply that several complexity classes in parameterized complexity which are believed to be different in fact collapse to one and the same complexity class [ARS08]. What we can do instead is to seek bounds on the proof-theoretic strength of so-called nondeterministic solvers that are assumed to somehow magically make optimal choices for certain heuristics during the search. A line of works including [BKS04, HBPV08, BHJ08] culminated in the papers [PD11, AFT11]. [PD11] showed that nondeterministic CDCL viewed can be as efficient as the resolution proof system except possibly for a polynomial blow-up. More technically speaking, the CDCL solver is allowed to magically choose decision variables and polarities for these variables, and also has to keep every single clause ever learned during the search. It also has to make somewhat frequent restarts, but not more frequent than what is already standard in state-of-the-art solvers, and has to use some so-called asserting clause learning scheme (such as, e.g., 1UIP, but any asserting scheme will be good enough). With these assumptions, what [PD11] shows is that a CDCL solver can decide unsatisfiability of a CNF formula in time polynomial in the shortest resolution proof of unsatisfiability of that formula. One possible way of interpreting this result might be that if the decision heuristic is good enough, then CDCL solvers at least have the potential to run fast on any formulas that possess short resolution proofs.

The construction of [PD11] showing that CDCL simulates resolution can be summarized as follows. Suppose that the formula has a resolution refutation \( \pi = (D_1, \ldots, D_L) \) with \( D_L = \bot \). Intuitively, the goal is for the CDCL search to successively learn the clauses \( D_i \) for \( i = 1, 2, \ldots \) until it reaches \( D_L = \bot \), which corresponds to a conflict at decision level 0 (see also the example in Figure 1.4). This may not always be possible, but a key insight in [PD11] is that the CDCL solver can instead learn other clauses which together are as powerful as \( D_i \). For [HKSB17] for automatic DRAT proofs of the pigeonhole principle).

\(^{12}\)Note that the word “nondeterministic” is used here in the technical sense of the definition of the complexity class \( \text{NP} \), where a computation can split into two branches at every step and succeeds in finding a solution if one of the (potentially exponentially many) branches does so. It thus has a very different meaning from “randomized,” where the requirement is that the successful branch should be found with high probability.
this, using a definition from [AFT11] we say that a set of clauses $F'$ absorbs a clause $D = a_1 \lor \cdots \lor a_k$ provided that setting any $k-1$ literals of $D$ false allows either the remaining literal of $D$ or the empty clause to be derived by unit propagation from the clauses of $F'$. What this means is that it seems to an outside observer that the set of clauses $F'$ contains $D$, since the same unit propagations can be observed as if $D$ was in the set. The construction of [PD11] allowing CDCL search to simulate the resolution refutation $\pi$ is based on showing that the CDCL search can learn clauses to absorb, successively, the clauses $D_1, \ldots, D_L$.

To absorb the $i$-th clause $D_i$, the CDCL search repeatedly selects one literal out of $D_i$ and sets the rest of the literals of $D_i$ false (if possible). If a conflict is found, a new learned clauses is added to the clause database $F'$ and the CDCL restarts. If no conflict is found, then a restart is done anyway, and a new literal from $D_i$ is selected. It can be shown that each $D_i$ can be absorbed by only polynomially many such steps.

In independent work, [AFT11] obtained an alternative, more effective version of this result by showing that if a formula $F$ has a resolution refutation in bounded width (i.e., where every clause contains only a constant number of literals), then CDCL using a decision strategy with enough randomness will decide $F$ efficiently.

At first sight this might not seem so impressive — after all, exhaustive search in bounded width also runs fast — but the point is that a CDCL solver is very far from doing exhaustive width search and does not care at all about the existence or non-existence of resolution refutations with only small clauses.

A downside of both of these results is that it is crucial for the SAT solver never to delete clauses. This is a very unrealistic assumption, since modern solvers throw away 90-95% of the clauses learned during search. It would be nice to extend the model of CDCL in [AFT11, PD11] to capture memory usage in a more realistic way, and then study the question of whether CDCL can simulate resolution efficiently with respect to both time and space — an attempt in this direction was made in [ELJ16]. The other downside of these results is that the simulation of resolution by CDCL depends crucially on the choice of decision literals. However, the correct choice of decision literals is made in a highly non-constructive way — namely, it consists of learning the clauses in the (unknown) shortest resolution refutation one at a time in the order they appear in the refutation. In other words, the usual heuristics for decision literals (such as VSIDS) do not come into play at all for the simulation of resolution by CDCL. Similarly, the concrete restart heuristic used plays no role — all that matters is that restarts are frequent enough. The way the construction goes, it can be shown that the solver makes progress towards its goal at the first conflict after every restart, but then it might not do anything useful until the next restart happens. One can show, though, that the solver also cannot do anything harmful while waiting for the next restart (but for this to hold it is crucial that the solver never deletes any learned clauses).

It is still an open problem to establish theoretical results about how the commonly used heuristics for decision literals and restarts contribute (or sometimes fail to contribute) to the success of CDCL search.

An alternate open problem is to understand the power of CDCL without restarts. The core ideas of CDCL are clause learning combined with backtracking and so, in spite of the usefulness of restarts, it is important to understand the
strength of CDCL without restarts. In particular, does CDCL without restarts simulate resolution? There are two related approaches to modeling CDCL without restarts as a proof system. The first is pool resolution [Van05]; the second is RegWRTI [BHJ08].

A pool resolution refutation consists of a resolution refutation $\pi$ which has a regular depth-first traversal; in other words, there is a depth-first traversal of the dag $G_\pi$ such that at each point during the traversal, the path from the current clause back to the empty clause does not resolve more than once on any given variable. The intuition is that a regular depth-first traversal corresponds to the CDCL search. Then, if clauses are learned as they are traversed, they do not need to be traversed again. And, finally, the fact that CDCL never chooses a decision literal which has already been set and that clause learning only learns clauses containing negated literals imposes a regularity condition for each path in the depth-first search.

The RegWRTI system is similar in spirit to pool resolution, but better approximates the way clause learning works in CDCL solvers (without restarts). The definition of RegWRTI is a bit complicated, so we omit it here. It open whether pool resolution or RegWRTI simulate resolution; indeed, several candidates for separation have failed to give a separation, see [BB12, BBJ14, BK14].

Pool resolution and RegWRTI do not fully capture CDCL without restarts, as they do not incorporate self-subsumption during clause learning. As an example, see Figure 1.2 where the 1UIP clause $a \lor b \lor c \lor \overline{y}$ can be resolved with the clause $a \lor b \lor \overline{c}$ to yield a better learned clause $a \lor b \lor \overline{y}$. Since this resolves on the literal $c$ even though $c$ has been set false, it does not fit the framework of pool resolution or RegWRTI. It is possible that pool resolution or especially RegWRTI can simulate clause learning augmented with self-subsumption, but this is an open question.

1.4. Resolution and Proof Complexity

The previous section described the close connections between resolution and SAT solvers based on CDCL search. This allows us to study the complexity of CDCL proof search by giving upper and lower bounds on the complexity of resolution refutations. A lower bound on resolution proof length gives a corresponding lower bound on CDCL proof search, since the execution of a SAT solver can be lower bounded in terms of the associated resolution refutation. Conversely, an upper bound on resolution refutation lengths points to at least the possibility of good CDCL search algorithms. There are other ways to measure resolution proof complexity beside sheer length. Notably, bounding the width or the space of resolution refutations can shed light on the effectiveness of different clause learning and clause forgetting strategies. In this section, we review what is known about these different proof complexity measures for resolution.

1.4.1. Resolution Length

We start by recalling that the length (also referred to as the size) of a resolution refutation is the number of clauses in it counted with repetitions (which is relevant for tree-like or regular refutations). In general, proof length/size is the most
fundamental measure in proof complexity, and as just discussed, lower bounds for resolution length imply lower bounds on CDCL solver running time.

Any CNF formula of size \( N \) can be refuted in resolution in length \( \exp(O(N)) \), and there are formulas for which matching \( \exp(\Omega(N)) \) lower bounds are known. Let us discuss some examples of formulas known to be hard with respect to resolution length.

Our first example is the pigeonhole principle (PHP), which says that “\( m \) pigeons do not fit into \( n \) holes if \( m > n \).” This is arguably the single most studied combinatorial principle in all of proof complexity (see \cite{Raz02} for a survey). When written as an unsatisfiable CNF formula, this becomes the claim that, on the contrary, \( m > n \) pigeons do fit into \( n \) holes. To encode this, one uses variables \( p_{i,j} \) to denote “pigeon \( i \) goes into hole \( j \),” and write down the following clauses, where \( i \neq i' \) range over \( 1, \ldots, m \) and \( j \neq j' \) range over \( 1, \ldots, n \):

\[
\begin{align*}
    p_{i,1} &\lor p_{i,2} \lor \cdots \lor p_{i,n} & \text{[every pigeon } i \text{ gets a hole]} \\
    \overline{p}_{i,j} &\lor \overline{p}_{i',j} & \text{[no hole } j \text{ gets two pigeons } i \neq i']
\end{align*}
\]

There are also variants where one in addition has “functionality” and/or “onto” axioms

\[
\begin{align*}
    \overline{p}_{i,j} &\lor \overline{p}_{i,j'} & \text{[no pigeon } i \text{ gets two holes } j \neq j'] \\
    p_{1,j} &\lor p_{2,j} \lor \cdots \lor p_{m,j} & \text{[every hole } j \text{ gets a pigeon]}
\end{align*}
\]

In a breakthrough result, Haken \cite{Hak85} proved that the PHP formula consisting of clauses (1.6a) and (1.6b) requires length \( \exp(\Omega(n)) \) in resolution for \( m = n + 1 \) pigeons, and his proof can be extended to work also for the onto FPHP formulas consisting of all clauses (1.6a)–(1.6d). Later work \cite{Raz04a, Raz03, Raz04b} has shown that all of the PHP formula variants remain hard even for arbitrarily many pigeons \( m \), requiring resolution length \( \exp(\Omega(n^\delta)) \) for some \( \delta > 0 \). What this means, intuitively, is that the resolution proof system really cannot count — even faced with the preposterous claim that infinitely many pigeons can be mapped in a one-to-one fashion into a some finite number \( n \) of holes, resolution cannot refute this claim with a proof of length polynomially bounded in the number of holes.

Since PHP formulas have size \( N = \Theta(n^3) \), Haken’s lower bound is only of the form \( \exp(\Omega(\sqrt[3]{N})) \) expressed in terms of formula size, however, and so does not quite match the \( \exp(O(N)) \) worst-case upper bound. The first truly exponential lower bound on length was obtained for Tseitin formulas (an example of which is shown in Figure 1.5), which encode (the negation of) the principle that “the sum of the vertex degrees in a graph is even.” Here the variables correspond to the edges in an undirected graph \( G \) of bounded degree. Every vertex in \( G \) is labelled 0 or 1 so that the sum of the vertex labels is odd. The Tseitin formula for \( G \) is the CNF formula which is the conjunction of the set of clauses expressing that, for each vertex of \( G \), the parity of the number of true edges incident to that vertex is equal to the vertex label. See Figure 1.5b, which displays the formula corresponding to the labelled graph in Figure 1.5a.

If we sum over all vertices, the parity of the number of all true edges should be an odd number by the construction of the labelling. However, since such a sum
counts each edge exactly twice it has to be even. Thus, the Tseitin formulas are indeed unsatisfiable. Urquhart \cite{Urq87} established that Tseitin formulas require resolution length exp($\Omega(N)$) if the underlying graph is a well-connected so-called expander graph (which holds asymptotically almost surely for a random regular graph of bounded degree, for instance). Intuitively, this shows that not only is resolution unable to to count efficiently in general, but it cannot even do so mod 2.

Another example of exponentially hard formulas are random $k$-CNF formulas, which are generated by randomly sampling $\Delta n^k$-clauses over $n$ variables for some large enough constant $\Delta$ depending on $k$. For instance, $\Delta \gtrapprox 4.5$ is sufficient to get unsatisfiable 3-CNF formulas asymptotically almost surely \cite{DBM00} (see also the chapter on Random Satisfiability \cite{Ach09} in this handbook). Chvátal and Szemerédi \cite{CS88} established that resolution requires length exp($\Omega(N)$) to refute such formulas (again asymptotically almost surely).

By now strong lower bounds have been shown for formulas encoding tiling problems \cite{Ale04,DR01}, $k$-colourability \cite{BCMM05}, independent sets and vertex covers \cite{BIS07}, and many other combinatorial principles. We conclude our discussion of resolution length by mentioning one recent addition to this long list, namely the subset cardinality formulas studied in \cite{Spe10,VS10,MN14} (also known as zero-one design or sgen formulas).

To construct these formulas, we start with an $n \times n$ $(0,1)$-matrix with 4 non-zero entries in each row and column except that one extra non-zero entry is added to some empty cell (as in Figure 1.6a, where the extra 1 in the bottom row is in bold face). The variables of the formula are the non-zero entries of the matrix, yielding a total of $4n+1$ variables. For each row of 4 ones in the matrix, we write down the natural 3-CNF formula encoding the positive cardinality constraint that at least 2 variables must be true (as in the first set of clauses in Figure 1.6b), and for the row with 5 ones the 3-CNF formula encoding that a strict majority of 3 variables must be true. For the columns we instead encode negative cardinality constraints that the number of false variables is at least 2 and 3, respectively (see the last set of clauses in Figure 1.6b). The formula consisting of the conjunction of all these clauses must be unsatisfiable, since a strict majority of the variables

(a) Graph with odd labelling.  
(b) Corresponding Tseitin formula.
Chapter 1. Proof Complexity

(a) Matrix with row and column constraints. (b) Cardinality constraints in CNF.

Figure 1.6: Matrix and (fragment of) corresponding subset cardinality formula.

cannot be true and false simultaneously. We will have reason to return to these formulas below when we discuss connections between CDCL and resolution, and also when discussing cutting planes and pseudo-Boolean solving.

It was shown empirically in [Spe10, VS10] that these formulas are very hard for CDCL solvers, but there was no analysis of the theoretical hardness. This was done by [MN14], who showed that subset cardinality formulas are indeed exponentially hard if the underlying matrix is an expander (informally, if every small-to-medium set of rows has non-zero entries in many distinct columns).

1.4.2. Resolution Width

A second complexity measure in resolution that is almost as well studied as length, is width, measured as the size of a largest clause in a resolution refutation. It is clear that the width needed to refute a formula is never larger than the number of variables $n$, which is in turn less than the total formula size $N$. It also easy to see that an upper bound $w$ on resolution width implies an upper bound $O(n^w)$ on resolution length, simply because the total number of distinct clauses of width at most $w$ over $n$ variables is less than $(3n)^w$. Incidentally, this simple counting argument turns out to be essentially tight, in that there are $k$-CNF formulas refutable in width $w$ that require resolution length $n^\Omega(w)$, as shown by [ALN16].

Much less obviously, however, and much more interestingly, strong enough width lower bounds imply strong length lower bounds. Ben-Sasson and Wigderson [BW01] (using methods based on [CEI96, IPS99]) showed that for a $k$-CNF formula over $n$ variables it holds that

$$\text{refutation length} \geq \exp\left(\Omega\left(\frac{(\text{refutation width} - k)^2}{n}\right)\right)$$

(1.7)

where “refutation length” and “refutation width” mean the minimum length and minimum width, respectively, of any resolution refutations of the formula (note that these two minima could potentially be realized by different refutations).
Chapter 1. Proof Complexity

The inequality (1.7) implies that if one can prove that a formula requires width $\omega(\sqrt{n \log n})$, this immediately yields a superpolynomial length lower bound, and a width lower bound $\Omega(N)$ in terms of the formula size $N$ (which is lower-bounded by the number of variables $n$) implies a truly exponential $\exp(\Omega(N))$ length lower bound. Almost all known lower bounds on resolution length can be derived via width lower bounds in this way (in particular, essentially all the bounds discussed in Section 1.4.1 although the ones predating [BW01] were originally not obtained in this way).

For tree-like resolution refutations of $k$-CNF formulas, the paper [BW01] proved a sharper version

$$\text{tree-like refutation length} \geq 2^{\text{refutation width}} - k$$

(1.8)

of the bound in (1.7) for general resolution. This means that for tree-like resolution, even width lower bounds $\omega(\log N)$ yield superpolynomial length lower bounds. For general resolution, however, a width lower bound even as large as $\Omega(\sqrt{n \log n})$ does not imply any length lower bound according to (1.7). This raises the question of whether it is possible to improve the analysis so that (1.7) can be strengthened to something closer to (1.8) also for general resolution. Bonet and Galesi [BG01] showed that this is not the case by studying another interesting combinatorial benchmark formula, which we describe next.

The ordering principle says that “every finite (partially or totally) ordered set $\{e_1, \ldots, e_n\}$ has a minimal element.” To encode the negation of this statement in CNF, we use variables $x_{i,j}$ to denote “$e_i < e_j$” and write down the following clauses (for $i \neq j \neq k \neq i$ ranging over 1, ..., $n$):

$$x_{i,j} \lor x_{j,i} \quad \text{[anti-symmetry; not both } e_i < e_j \text{ and } e_j < e_i\text{]} \quad (1.9a)$$

$$x_{i,j} \lor x_{j,k} \lor x_{i,k} \quad \text{[transitivity; } e_i < e_j \text{ and } e_j < e_k \text{ implies } e_i < e_k\text{]} \quad (1.9b)$$

$$\lor_{1 \leq i \leq n, i \neq j} x_{i,j} \quad \text{[} e_j \text{ is not a minimal element]} \quad (1.9c)$$

One can also add axioms

$$x_{i,j} \lor x_{j,i} \quad \text{[totality; either } e_i < e_j \text{ or } e_j < e_i\text{]} \quad (1.9d)$$

to specify that the ordering has to be total. This yields a formula over $\Theta(n^2)$ variables of total size $N = \Theta(n^3)$. (We remark that variants of this formula also appear under the name least number principle formula or graph tautology in the literature.)

It was conjectured in [Kri85] that these formulas should be exponentially hard for resolution, but Stålmarck [Stå96] showed that they are refutable in length $O(N)$ (even without the clauses (1.9d)). As the formula is described above, it does not really make sense to ask about the refutation width, since already the axiom clauses (1.9c) have unbounded width. However, one can convert the formula to 3-CNF by applying the transformation from (1.1a) to (1.1b) to the wide axioms (1.9c), and for this version of the formula [BG01] established

13To be precise, the exceptions are [Ale04, DR01, Raz04a, Raz03, Raz04b], where the number of variables $n$, and hence the formula size $N$, is at least as large as the refutation width squared, and where other methods must therefore be used to prove lower bounds on resolution length.
a width lower bound $\Omega(\sqrt[3]{N})$ (which is tight, and holds even if the axioms (1.9d) are also added). This shows that even polynomially large resolution width does not necessarily imply any length lower bounds for general resolution (and, in view of (1.8), it provides an exponential separation in proof power between general and tree-like resolution, although even stronger separations are known [BIW04]).

### 1.4.3. Resolution Space

The study of the space complexity of proofs, which was initiated in the late 1990s, was originally motivated by considerations of SAT solver memory usage, but has also turned out to be of intrinsic interest for proof complexity. Space can be measured in different ways — here we focus on the most well studied measure of clause space, which is the maximal number of clauses needed in memory while verifying the correctness of a resolution refutation. Thus, in what follows below “space” will always mean “clause space.”

The space usage of a resolution refutation at step $t$ is the number of clauses at steps $\leq t$ that are used at steps $\geq t$. Returning to our example resolution refutation in Figure 1.1, the space usage at step 8 is 5 (the clauses in memory at this point are clauses 1, 2, 6, 7, and 8). The space of a proof is obtained by measuring the space usage at each step in the proof and taking the maximum. Phrased differently, one can view the formula as being stored on a read-only input tape, from where the axiom clauses can be read into working memory. The resolution rule can only be applied to clauses currently in working memory, and if a clause has been erased from working memory, then it is gone, and will have to be rederived again if it is to be used again (or read again from the input tape, in the case of axiom clauses). Then space measures how many clauses are used in working memory to perform the resolution refutation. Incidentally, it is not hard to see that the proof in Figure 1.1 is not optimal when it comes to minimizing space. We could do the same refutation in space 4 instead by processing the clauses in the order 4, 5, 6, 3, 7, 2, 8, 1, 9, 10. (In fact, it is even conceivable that if minimizing space is all we care about, then it might be beneficial to forget clauses and rederive them later, even if it means repeating the same steps in the resolution refutation multiple times. This indeed turns out to be the case, as discussed in Section 1.4.4 below.)

Perhaps somewhat surprisingly, any unsatisfiable CNF formula of size $N$ can always be refuted in resolution space at most $N + O(1)$ as shown by [ET01], though the resolution refutation thus obtained might have exponential length. Lower bounds on space were subsequently shown for PHP formulas and Tseitin formulas [ABRW02, ET01] and for random $k$-CNF formulas [BG03]. For the latter two formula families the (optimal linear) lower bounds matched exactly previously known

---

14Note, though, that this measure underestimates the actual memory usage, since storing a clause requires more than a constant amount of memory (this is similar to how we ignore the size of clauses when defining the length and size of resolution proofs to be the total number of clauses in the proof). For completeness, we mention that there is also a measure total space, counting the total number of literals in memory (with repetitions), which has been studied in, e.g., [ABRW02, BHT10, BHH15].

15This space upper bound is obtained by simply running CDCL (or even DPLL) as described in Section 1.3.1 with some arbitrary but fixed variable ordering.
known width lower bounds, and also the proof techniques had a very similar flavour. This led to the question of whether there was some deeper connection hidden here. In a very elegant paper, [AD08] confirmed this by showing that the inequality
\[ \text{refutation space} \geq \text{refutation width} + O(1) \] (1.10)
holds for resolution refutations of \( k \)-CNF formulas. The proof of (1.10) is beautiful but uses a somewhat non-explicit argument based on finite model theory. A more explicit proof, which works by simple syntactic manipulations to construct a small-width refutation from a small-space refutation, was presented in [FLM+15].

Since for all formulas studied up to [AD08] the width and space complexity measures turned out to actually coincide, it is natural to ask whether (1.10) could be strengthened to an asymptotic equality. The answer to this question is no. As shown in the sequence of works [Nor09a, NH13, BN08], there are formulas that can be refuted in width \( O(1) \) and length \( O(N) \) but require space \( \Omega(N/\log N) \) (i.e., formulas that are maximally easy for width but exhibit worst-case behaviour for space except for a log factor, and this result is tight since it can be shown that any formula refutable in length \( O(N) \) can also be refuted in space \( O(N/\log N) \)).

These formulas are pebbling contradictions (also called pebbling formulas) encoding so-called pebble games on bounded fan-in DAGs, which for the purposes of this discussion we additionally require to have a unique sink. In the “vanilla version” of the formula (illustrated in Figure 1.7), there is one variable associated to each vertex and clauses encoding that

- the source vertices are all true;
- if all immediate predecessors are true, then truth propagates to the successor;
- but the sink is false.

There is an extensive literature on pebbling space and time-space trade-offs from the 1970s and 80s, (with [Pip80] giving an excellent overview of this area, and the upcoming survey [Nor19] covering some more recent developments). Pebbling contradictions have been useful before in proof complexity in various contexts, e.g., in [RM99, BEGJ00, BW01]. Since pebbling contradictions can be shown to be refutable in constant width but there are graphs for which the pebble game
requires large space, one could hope that the pebbling properties of such DAGs would somehow carry over to resolution refutations of pebbling formulas and help us separate space and width.

Unfortunately, this hope cannot possibly materialize — a quick visual inspection of Figure 1.7b reveals that this is a Horn formula (i.e., having at most one positive literal in each clause), and such formulas are maximally easy for length, width, and space simultaneously, since they are decided by unit propagation. However, we can modify these formulas by substituting for every variable \( x \) an exclusive or \( x \oplus y \) of two new variables, and then expand to CNF in the canonical way to get a new formula. This process is called XOR-ification or XOR-substitution and is perhaps easiest to explain by example. Performing this substitution in the clause

\[
\overline{x} \lor y
\]

we obtain the formula

\[
\overline{(x_1 \oplus x_2) \lor (y_1 \oplus y_2)}
\]

which when expanded out to CNF becomes

\[
(x_1 \lor \overline{x}_2 \lor y_1 \lor y_2) \\
\land (x_1 \lor \overline{x}_2 \lor \overline{y}_1 \lor \overline{y}_2) \\
\land (\overline{x}_1 \lor x_2 \lor y_1 \lor y_2) \\
\land (\overline{x}_1 \lor x_2 \lor \overline{y}_1 \lor \overline{y}_2)
\]

As another example, applying XOR-substitution to Figure 1.7b yields the formula in Figure 1.8.

Using such XOR-substitution, it turns out that the pebbling contradiction inherits the time-space trade-offs of the pebbling DAG in terms of which it is defined [BN08, BN11] (and there is nothing magical with XOR — this can be shown to work also for substitution with other Boolean functions that have the “right properties”). Now the strong space-width separation described above is obtained by plugging in the pebbling DAGs studied in [PTC77, GT78].

1.4.4. Resolution Trade-offs

In the preceding sections, we have seen that for all the complexity measures of length, width, and space there are formulas which are maximally hard for these measures. Suppose, however, that we are given a formula that is guaranteed to be easy for two or more of these measures. Can we then find a resolution refutation that optimizes these complexity measures simultaneously? Or are there trade-offs, so that minimizing one measure must cause a sharp increase in the other measure? Such questions about trade-offs have a long history in computational complexity theory, but it seems that Ben-Sasson [Ben09] was first to raise the issue in the context of proof complexity.

It should be noted that this kind of trade-off questions need to be phrased slightly more carefully in order to be really interesting. As observed in [Nor09a], it is often possible to obtain trade-off results simply by gluing together two formulas \( F \) and \( G \) over disjoint sets of variables which have different proof complexity
properties, and then obtain a trade-off result from the fact that any proof of unsatisfiability has to refute either $F$ or $G$. In order to eliminate such examples and obtain formulas that have inherent trade-off properties, we can additionally require that the formulas in question should be minimally unsatisfiable, i.e., that if any clause in the formula is removed, then the residual formula is satisfiable. For most of the trade-off results we consider here, the formulas are of this flavour. Also, it can be noted that for the strongest trade-off results discussed below, the trick of gluing together disjoint formulas cannot yield trade-offs with so strong parameters anyway.

The first trade-off result in proof complexity seems to have been obtained in [Ben99], where a strong space-width trade-off was established. Namely, there are formulas for which

- there are refutations in width $O(1)$;
- there are also refutations in space $O(1)$;
- but optimizing one measure causes (essentially) worst-case behaviour for the other.

This holds for the “vanilla version” of the pebbling contradictions in Figure 1.7b (if one again uses the graphs studied in [PTC77], [GT78]). Using techniques from [Raz16], this space-width trade-off was strengthened in [BNT16] to show that there are formulas for which resolution refutations in width $w$ require space almost $n^w$, i.e., far above the linear worst-case upper bound for space.

Regarding trade-offs between length and space, it was shown in [BNT11, BBI16, BNT13] that there are formulas that

- can be refuted in short length;

Figure 1.8: Pebbling contradiction in Figure 1.7b with XOR-substitution.
can be refuted in small space;
• but even slightly optimizing one measure causes a dramatic blow-up for the other.

This holds for substituted pebbling formulas over DAGs with strong time-space trade-offs (as in, e.g., \cite{CSS0, CSS2, LT82, Nor12}) and for Tseitin formulas over long, narrow rectangular grids.\footnote{To be precise, the results in \cite{BN11} require that one adds two copies of every edge in the grid graph, which corresponds to XOR-substitution in the Tseitin formula, but it can be shown that this substitution can be eliminated with some extra work.} What a length-space trade-off theorem says is that if a resolution is of short length, then at some point during the refutation a lot of space is being used. The theorem does not a priori rule out, though, that this high space usage could be an isolated spike, and that most of the time the refutation uses very little space. A more recent work \cite{AdRNV17} established more robust \emph{cumulative} length-space trade-offs, exhibiting formulas where any short proof has to use a lot of space essentially the whole time.

For length versus width, we know that short refutation length implies small refutation width by (1.7). The proof of this inequality works by transforming a given short refutation into a narrow one, but the length blows up exponentially in the process. Thapen \cite{Tha16} showed that this blow-up is unavoidable by exhibiting formulas for which there exist resolution refutations in short length, but for which any refutation in width as guaranteed by (1.7) has to be exponentially long. These formulas are slightly tricky to describe, however, and so we do not do so here. A technical issue with Thapen’s result is that for all other trade-offs discussed above there are \(k\)-CNF formulas (for \(k = O(1)\)) that exhibit this behaviour, but Thapen’s formulas have clauses of logarithmic width. It would be nice to bring this down to constant width if possible. We also want to mention in this context that in a recent, very intriguing work \cite{Raz16} obtained doubly exponential size-width trade-offs in tree-like resolution (this is measured in the number of variables in the formulas, which have exponential size and polynomial width).

\subsection*{1.4.5. Theoretical Complexity Measures and Hardness in Practice}

The next topic we discuss is whether practical hardness for CDCL is in any way related to the complexity measures of resolution length, width, and space. One interesting observation in this context is that it follows from the results reviewed in Section 1.4 — if we “normalize” length by taking a logarithm since it can be exponential in the formula size \(N\) whereas the worst-case upper bounds for width and space are linear — that for any \(k\)-CNF formula the inequalities

\[
\log(\text{refutation length}) \lesssim \text{refutation width} \lesssim \text{refutation space}
\] (1.12)

hold. Thus, length, width, and space form an hierarchy of increasingly strict hardness measures. Let us briefly discuss the measures again in this light:

• We know that length provides a lower bound on CDCL running time\footnote{Except if some non-resolution-based preprocessing or inprocessing techniques happen to be very successful.} and that CDCL polynomially simulates resolution \cite{PD11}. However, the results
Chapter 1. Proof Complexity

in [AR08] suggest that short resolution proofs should be intractable to find in the worst case.

• Regarding width, searching for proofs in small width is apparently a well-unknown heuristic in AI community, and [AFT11] proved that CDCL has the potential to run fast if such proofs exist.

• As to space, memory consumption is an important bottleneck for SAT solvers in practice, and space complexity results provide lower bounds on CDCL clause database size. One downside of this is that the bounds can be at most linear, and the solver would certainly use a linear amount of memory just to store the input. However, it is important to note that the space lower bounds hold even in a model where the solvers knows exactly which clauses it needs to keep. It could be argued that in reality probably much more memory than the bare minimum should be needed.

Are width or even space lower bounds relevant indicators of CDCL hardness? Or could it be true in practice that CDCL does essentially as well as resolution with respect to length/running time? These are not mathematically well-defined questions, since state-of-the-art CDCL solvers are moving targets, but perhaps it could still possible to perform experiments and draw interesting conclusions? Such an approach was proposed by [ABLM08], and [JMNŽ12] performed what seems to have been the first systematic attempt to implement this program.

In view of the discussion above it seems too optimistic that length complexity should be a reliable indicator of CDCL hardness. [JMNŽ12] therefore focused on comparing width and space by running extensive experiments on formulas with constant width complexity (and linear length complexity) but varying space complexity to see whether running time correlated with space. These experiments produced lots of interesting data, but it seems fair to say that the results are inconclusive. For some families of formulas the correlation between running time and space complexity looks very nice, but for other formulas the results seem quite chaotic.

In fact, one problem is that formulas with low width complexity and varying space complexity are hard to find — pretty much the only known examples are the substituted pebbling formulas discussed in Section 1.4.3. Thus, it is not even clear whether the experiments measured differences in width and space complexity or some other property specific to these particular formulas. This problem seems inherent, however. One cannot just pick arbitrary benchmark formulas and compute the width and space complexity for them before running experiments, since deciding width is EXPTIME-complete [Ber12] and deciding space appears likely to be PSPACE-complete.

1.4.6. Using Theory Benchmarks to Shed Light on CDCL Heuristics

Although modern CDCL solvers are routinely used to solve real-world instances with hundreds of thousands or even millions of variables, it seems fair to say that it is still very poorly understood how these solvers can be so unreasonably effective. As should be clear from the description in Section 1.3 the basic architecture of CDCL solvers is fairly simple, but the secret behind the impressive performance of state-of-the-art solvers lies in a careful implementation of the basic CDCL
algorithm with highly optimized data structures, as well in the use of dozens of sophisticated heuristics.

Unfortunately, many of these heuristics interact in subtle ways, which makes it hard to assess their relative importance. A natural approach to gain a better understanding would be to collect real-world benchmarks and run experiments on an instrumented CDCL solver with different parameter settings to study how they contribute to overall performance, as proposed in [LM02] [KSM11]. It seems quite tricky to implement this idea in a satisfactory way, however. The set of available benchmarks is quite limited, and is also a highly heterogeneous collection in terms of formula properties. For this reason it turns out to be challenging to obtain statistically significant data which would admit drawing general conclusions.

The recent paper [EGG+18] instead put forth the proposition that a better understanding of CDCL could be obtained by running experiments on carefully chosen theoretical benchmarks. By tuning various parameters, one can study what impact each heuristic has on performance and how this correlates with the theoretical properties of the formulas. An obvious objection is that it is very unclear why such a study of crafted benchmarks should have any practical relevance, but some arguments in favour of this approach given in [EGG+18] are as follows:

- The benchmarks are scalable, meaning that one can generate “the same” formula for different sizes and study how performance scales as the instance size increases. (This simple but powerful idea was perhaps first articulated in an applied SAT solving setting in [PV05].)
- The benchmarks are chosen to have different extremal properties in a proof-complexity-theoretic sense, meaning that they can be viewed as challenging benchmarks for different heuristics for variable decisions, clause deletions, restarts, et cetera.
- Finally, in contrast to most combinatorial benchmarks traditionally used in the SAT competitions [18] which are known to be very hard for resolution, the benchmarks in [EGG+18] have been constructed so as to be easy in the sense of having very short resolution proofs of unsatisfiability that solver could potentially find. In view of this, it can be argued that the performance of the solver provides a measure of the quality of the proof search and how it is affected by different heuristics.

Below follow the main conclusions reported in [EGG+18] after comparing the empirical results with theoretical properties of the benchmarks:

1. Learned clauses are absolutely critical for performance. While the information gathered during conflict analysis is important for guiding other heuristics, the solvers crucially need the exponential increase in reasoning power afforded by also storing the learned clauses to go from tree-like (DPLL-style) to general resolution proofs.

2. While the mathematical question of whether restarts are just a helpful heuristic or are fundamentally needed for CDCL solvers to harness the full power of resolution remains wide open, the experimental results provide
some circumstantial evidence that the latter might be the case. Also, adaptive restarts as in [AS12] often work markedly better than the fixed-interval so-called Luby sequence restarts in [ES04].

3. For formulas inspired by time-space trade-off results, too aggressive clause erasure can incur a stiff penalty in running time also in practice. And when memory is tight, the LBD (literal block distance) heuristic [AS09] often does a particularly good job at identifying useful clauses.

4. For VSIDS variable decisions [MMZ+01] the choice of decay factor can sometimes be vitally important. It is not at all clear why, but one hypothesis is that this might be connected to whether the proof search needs to find DAG-like proofs or whether tree-like proofs are good enough. We can also see that VSIDS decisions can sometimes go badly wrong for easy but tricky formulas, which suggests that there is room for further improvements in variable selection heuristics.

It should perhaps be emphasized that none of these findings should be considered to be a priori obvious, since proof complexity is inherently non-constructive whereas CDCL is about algorithmic proof search.

Another point to be emphasized is that the above findings are in no way rigorous results, but more a way of collecting circumstantial evidence for connections between theory and practice. Much work still remains to gain a more solid, mathematical understanding of when and why CDCL solvers work.

1.5. Algebraic Proof Systems

We now switch topics to algebraic proof systems, where formulas are translated to polynomials so that questions of satisfiability or unsatisfiability can be answered using algebraic methods of reasoning.

In what follows, we will let $F$ be a field (which will usually be the finite field $GF(2)$ with two elements $\{0, 1\}$ in practical SAT solving applications but can be any field from the point of view of proof complexity) and $\vec{x} = \{x_1, \ldots, x_n\}$ be a set of variables. A monomial $m$ is a product of variables $m = \prod_{i=1}^{n} x_i^{e_i}$. If $e_i \in \{0, 1\}$ for all $i$ we say that the monomial is multilinear. A term $t = \alpha m$ is a monomial $m$ multiplied by some field element $\alpha \in F$.

1.5.1. Nullstellensatz

As our first example of an algebraic proof system we discuss Nullstellensatz introduced by Beame et al. [BIK+94]. A Nullstellensatz refutation of a set of polynomials $\mathcal{P} = \{p_i(\vec{x}) \mid i \in [m]\}$ in the polynomial ring $F[\vec{x}]$ is a syntactic equality

$$\sum_{i=1}^{m} r_i(\vec{x}) \cdot p_i(\vec{x}) + \sum_{j=1}^{n} s_j(\vec{x}) \cdot (x_j^2 - x_j) = 1 \quad (1.13)$$

(where $r_i, s_j$ are also polynomials in $F[\vec{x}]$). In algebraic language, what this shows is that the multiplicative identity 1 of the field $F$ lies in the polynomial ideal generated by $\mathcal{P} \cup \{x_j^2 - x_j \mid j \in [n]\}$. By (a slight extension of) Hilbert’s
Chapter 1. Proof Complexity

Nullstellensatz, such a refutation \((1.13)\) exists if and only if there is no common \(\{0, 1\}\)-valued root for the set of polynomials \(P\).

Nullstellensatz can also be viewed as a proof system for certifying the unsatisfiability of CNF formulas. In this setting we first translate clauses of the form

\[
C = \bigvee_{x \in P} x \lor \bigvee_{y \in N} \overline{y}
\]

(1.14a)

to polynomials

\[
p(C) = \prod_{x \in P} (1 - x) \cdot \prod_{y \in N} y .
\]

As a concrete example, \(D = x \lor y \lor z\) gets translated to \(p(D) = (1 - x)(1 - y)z = z - yz - xz + xyz\)\(^{19}\) Then a Nullstellensatz refutation of \(\{p(C_i) \mid i \in [m]\}\) can be viewed as a refutation of the CNF formula \(F = \bigwedge_{i=1}^{m} C_i\). This is so since an assignment to the variables in \(F\) (where we think of 1 as true and 0 as false) is satisfying precisely if the all the polynomials \(\{p(C_i) \mid i \in [m]\}\) vanish, and a Nullstellensatz refutation rules out that such a satisfying assignment exists.

The size of a Nullstellensatz refutation is defined to be the total number of monomials encountered when all products of polynomials are expanded out as linear combinations of monomials. To be more precise, let \(mSize(p)\) denote the number of monomials in a polynomial \(p\) written as a linear combination of monomials (so that for our example clause \(D\) above we have \(mSize(p(D)) = 4\)). Then the size of a Nullstellensatz refutation of the form \((1.13)\) is

\[
\sum_{i=1}^{m} mSize(r_i(\vec{x})) \cdot mSize(p_i(\vec{x})) + \sum_{j=1}^{n} 2 \cdot mSize(s_j(\vec{x})) .
\]

(1.15)

We remark that this is not the only possible way of measuring size, however. It can be noted that the definition \((1.15)\) is quite wasteful in that it forces us to represent the proof in a very inefficient way. Other papers in the so-called semialgebraic proof complexity literature, such as [GHP02, KI06, DMR09], instead define size in terms of the polynomials in isolation, more along the lines of

\[
\sum_{i=1}^{m} (mSize(r_i(\vec{x})) + mSize(p_i(\vec{x}))) + \sum_{j=1}^{n} (mSize(s_j(\vec{x}))) + 2
\]

(1.16)
or as the bit size or “any reasonable size” of the representation of all polynomials \(r_i(\vec{x}), p_i(\vec{x}), s_j(\vec{x})\). However, our definition \((1.15)\) is consistent with the general definition of size for so-called algebraic and semialgebraic proof systems.

\(^{19}\)Note that in this translation we are thinking of 1 as \(true\) and 0 as \(false\). It can be argued that in the context of algebraic proof systems a more natural convention is to flip the values and identify 0 with \(true\) and 1 with \(false\), just as a clause evaluating to \(true\) is identified with the corresponding polynomial evaluating to 0. If we adopt this flipped convention, then \(x \lor y \lor \overline{z}\) would be translated to \(xyz(1 - z) = xy - xyz\). There is no consensus on this matter in the algebraic proof complexity literature, however. For simplicity, we will identify 1 with \(true\) and 0 with \(false\) throughout this survey.
Chapter 1. Proof Complexity

33

in [ALN16, Ber18, AO19], and, in particular, matches the size definition in the other algebraic proof systems that will be discussed later in this section.\(^{20}\)

A much more well-studied measure for Nullstellensatz than size, however, is degree, which is defined as \(\max\{\deg(r_i(x_i) \cdot p_i(x)), \deg(s_j(x) \cdot (x_j^2 - x_j))\}\). In order to prove a lower bound \(d\) on Nullstellensatz degree for refuting \(P\), one can construct a \(d\)-design, which is a map \(D\) from degree-\(d\) polynomials in \(F[x]\) to \(F\) such that

1. \(D\) is linear, i.e., \(D(\alpha p + \beta q) = \alpha D(p) + \beta D(q)\) for \(\alpha, \beta \in F\);
2. \(D(1) = 1\);
3. \(D(rp) = 0\) for all \(p \in P\) and \(r \in F[x]\) such that \(\deg(rp) \leq d\);
4. \(D(x^2s) = D(xs)\) for all \(s \in F[x]\) such that \(\deg(s) \leq d - 2\).

Designs provide a characterization of Nullstellensatz degree in that there is a \(d\)-design for \(P\) if and only if there is no Nullstellensatz refutation of \(P\) in degree \(d\) (this is clearly spelled out in [Bus98] but is mentioned there to have been known before). The only-if direction is clear—applying \(D\) to a purported Nullstellensatz refutation of the form (1.13) yields 0 on the left-hand side but 1 on the right-hand side. The if-direction requires more work, but follows from linear programming duality.

Lower bounds on Nullstellensatz degree have been proven for formulas encoding combinatorial principles such as the pigeonhole principle [BCE+98] and pebbling contradictions [BCIP02], which have been described above, and also for other formulas encoding the induction principle [BP98b], house-sitting principle [CEI96, Bus98], and matching [BIK+97], which we will not discuss further in this survey.

It seems fair to say that research in algebraic proof complexity soon moved on from Nullstellensatz to stronger systems such as polynomial calculus [CE96], which we will discuss shortly. Briefly (and somewhat informally), the difference is that in polynomial calculus the proof that 1 lies in the ideal generated by \(P \cup \{x_j^2 - x_j \mid j \in [n]\}\) can be constructed dynamically by a step-by-step derivation, which can make it possible to decrease both degree and size significantly. However, Nullstellensatz has seen somewhat of a revival in a recent line of works [RPRC16, PRI17, PR18, CKRS19, dRMN+19] showing that Nullstellensatz degree lower bounds can be “lifted” to lower bounds in stronger computational models. (We will briefly discuss lifting and how it has been used in proof complexity in Sections 1.7 and 1.8.) The size complexity measure for Nullstellensatz has also received attention in recent papers such as [Ber18, AO19, dRMNR19].

When proving lower bounds for algebraic proof systems it is often convenient to consider a multilinear setting where refutations are presented in the quotient ring \(\mathbb{F}[x]/\langle x_j^2 - x_j \mid j \in [n]\rangle\). Since the Boolean axioms \(x_j^2 - x_j\) are no longer

\(^{20}\)We remark that in the end the difference is not too important, since the two measures (1.15) and (1.16) are at most a square apart, and for the size measure we typically focus on the distinction between polynomial and superpolynomial. In addition, and more importantly, when we are dealing with \(k\)-CNF formulas with \(k = O(1)\), as we are mostly doing in this survey, then the two size definitions are the same up to a constant factor \(2^k\). We refer the reader to Section 2.4 in [AH18] for a more detailed discussion of the definition of proof size in algebraic and semialgebraic proof systems.
needed, the refutation \( (1.13) \) can be written simply as
\[
\sum_{i=1}^{m} r_i(\vec{x}) \cdot p_i(\vec{x}) = 1, \tag{1.17}
\]
where we assume that all results of multiplications are implicitly multilinearized. It is clear that any refutation on the form \( (1.13) \) remains valid after multilinearization, and so the size and degree measures can only decrease in a multilinear setting.

1.5.2. Polynomial Calculus

The proof system \textit{polynomial calculus} (PC) was introduced in \cite{CEI96} to model \textit{Gröbner} basis computations (and was originally called the “\textit{Gröbner} basis proof system,” although by now the name polynomial calculus is firmly established). As in Nullstellensatz, the set-up is that we have a set of polynomials \( P \) over variables \( x_1, \ldots, x_n \), where the polynomial coefficients live in some field \( \mathbb{F} \). The goal is to prove that the polynomials \( P \cup \{ x_j^2 - x_j \mid j \in [n] \} \) has no common root. When operating with CNF formulas, we first translate the clauses to polynomials using the transformation from (1.14a) to (1.14b).

It is important to observe that from an algebraic point of view the variables can take as values any elements in the field \( \mathbb{F} \). Hence, we need to add constraints enforcing 0/1 assignments. We are also allowed to take linear combinations of polynomials, or to multiply a polynomial by any monomial, since any common root for the original polynomials is preserved under such operations. This leads to the following set of derivation rules for polynomial calculus:

\[
\begin{align*}
\text{Boolean axioms} & \quad x_j^2 - x_j \quad \tag{1.18a} \\
\text{Linear combination} & \quad \frac{p}{\alpha p} + \frac{q}{\beta q} \quad (\alpha, \beta \in \mathbb{F}) \quad \tag{1.18b} \\
\text{Multiplication} & \quad \frac{p}{mp} \quad (m \text{ any monomial}) \quad \tag{1.18c}
\end{align*}
\]

A PC refutation of \( P \) also allows polynomials from \( P \) as axioms; the refutation ends when 1 has been derived, showing that the polynomial equations have no common root, or equivalently when \( P \) is an translation of a CNF, that the CNF is unsatisfiable. The polynomial calculus proof system is sound and complete, not only for CNF formulas but for inconsistent systems of polynomial equations in general. As for Nullstellensatz, we can consider the setting where all polynomials are multilinear, because any higher powers of variables can always be eliminated by using the Boolean axioms \( (1.18a) \).

To define the complexity measures of \textit{size}, \textit{degree}, and \textit{space}, we write out polynomials in a refutation as linear combinations of monomials. Then the \textit{size} of a refutation, which is the analogue of resolution length, is the total number of monomials in the refutation (counted with repetitions), the \textit{degree}, corresponding to resolution width, is the largest degree of any monomial in it, and the
(monomial) space, which is the analogue of resolution (clause) space, is the maximal number of monomials in memory at any point during the refutation (again counted with repetitions). One can also define a length measure for polynomial calculus, which is the number of derivation steps, but this can be exponentially smaller than the size, which is the more relevant measure to study here.\(^{21}\)

The representation of a clause \(\bigvee_{i=1}^{n} x_i\) as a polynomial in PC is \(\prod_{i=1}^{n} (1-x_i)\), which means that the number of monomials is exponential in the clause width. This problem arises only for positive literals, however—a large clause with only negative literals is translated to a single monomial. To get a cleaner and more symmetric treatment, in \([ABRW02]\) the proof system polynomial calculus (with) resolution, or PCR for short, was introduced. The idea is to add an extra set of parallel formal variables \(x'_j, j \in [n]\), so that positive and negative literals can both be represented in an efficient fashion.

Lines in a PCR proof are polynomials over the ring \(\mathbb{F}[x_1, x'_1, \ldots, x_n, x'_n]\), where as before \(\mathbb{F}\) is some field. We have all the axioms and rules of PC plus the axiom

\[
\text{Negation} \quad x_j + x'_j - 1. \tag{1.18d}
\]

Size, length, and degree are defined as for polynomial calculus, and the (monomial) space of a PCR refutation is again the maximal number of monomials in any configuration counted with repetitions.

It is important to understand that from an algebraic point of view the variables \(x_j\) and \(x'_j\) are completely independent. The point of the complementarity rule, therefore, is to force \(x_j\) and \(x'_j\) to take opposite values in \(\{0, 1\}\), so that they respect the intended meaning of negation. It is worth noting that in actual Gröbner basis calculations one would not have both \(x_j\) and \(x'_j\), so the introduction of “twin variables” is just to get a nicer proof system from a theoretical point of view. Our example clause \(D = x \lor y \lor z\) is rendered as \(x'y'z\) in PCR.

One gets the same degree bounds for PCR as in PC (just substitute \(1-x\) for \(x'\)), but one can potentially avoid an exponential blow-up in size measured in the number of monomials (and thus also for space). There are \(k\)-CNF formulas for which PCR is exponentially more powerful than PC with respect to size \([LN19]\).

In PCR, monomial space is a natural generalization of clause space since every clause translates into a monomial as just explained in the example above.

Clearly, PC and PCR are very closely related, and in what follows we will sometimes be a bit sloppy and write just “polynomial calculus” when the distinction between the two is not important. We write “polynomial calculus resolution” or “PCR” to highlight when a claim only holds for polynomial calculus with “twin variables” for positive and negative literals.

### 1.5.3. Nullstellensatz, Polynomial Calculus, and Resolution

Polynomial calculus resolution can simulate resolution efficiently with respect to length/size, width/degree, and space simultaneously simply by mimicking refu-

\(^{21}\)In fact, if we consider the multilinear setting, where there are no Boolean axioms and instead multiplication is defined to return the multilinearized result, then it is not hard to show that any CNF formula with \(m\) clauses over \(n\) variables can be refuted in polynomial calculus in length \(O(mn)\). See, e.g., \([MIN15]\) for a proof of this fact.
Chapter 1. Proof Complexity

(a) Resolution step.

(b) Corresponding PCR derivation.

Figure 1.9: Example of simulation of resolution step by PCR.

36

Chapter 1. Proof Complexity

(a) Resolution step. (b) Corresponding PCR derivation.

Figure 1.9: Example of simulation of resolution step by PCR.

Polynomial calculus can be strictly stronger than resolution with respect to size and degree. For instance, over GF(2) it is not hard to see that Tseitin formulas can be refuted in size $O(N \log N)$ and degree $O(1)$ by doing Gaussian elimination. Another example are the onto FPHP formulas (1.6a)–(1.6d), which were shown to be easy in [Rii93]. It remains open whether such separations can be found also for space, however.

Open Problem 1.1. Prove (or disprove) that polynomial calculus resolution is strictly stronger than resolution with respect to space.

The proof systems Nullstellensatz and PC are incomparable to resolution with respect to size/length—there are formulas for which both Nullstellensatz and PC are exponentially more efficient than resolution, and other formulas for which resolution is exponentially better.

1.5.4. Size and Degree for Nullstellensatz and Polynomial Calculus

A lot of what is known about length versus width in resolution carries over to size versus degree in polynomial calculus, whereas Nullstellensatz is mostly different. It is not hard to show that for both Nullstellensatz and polynomial calculus upper bounds on degree imply upper bounds on size, in the sense that if a CNF formula over $n$ variables can be refuted in degree $d$, then such a refutation can be carried out in size $O(n^d)$. This is qualitatively similar to the bound for resolution, although the arguments are a bit more involved. Just as for resolution, this upper bound has been proven to be tight up to constant factors in the exponent for polynomial calculus [ALN16], and it follows from [LLMO09] that this also holds for Nullstellensatz.

In the other direction, a lower bound on size in terms of degree exactly analogous to the size-width bound (1.7) for resolution [BW01] holds also for polynomial calculus, as shown in [IPS99]. For Nullstellensatz it is not possible to obtain lower bounds on size from degree lower bounds in this way, and pebbling formulas provide a counter-example [BCIP02].

Interestingly, the paper [IPS99] is a precursor to [BW01], and although it was far from obvious at the time it turns out that one can run exactly the same proof for both resolution and polynomial calculus. As for resolution, the ordering
Chapter 1. Proof Complexity

principle formulas in [1.9a]–[1.9d] witness the optimality of this size-degree lower bound, as shown by [GL10]. As for resolution, almost all size lower bounds are derived via degree lower bounds.

The basic tool for proving polynomial calculus degree lower bounds is that of $R$-operators, which are analogues of the $d$-designs used for Nullstellensatz. As proposed in [Raz98], the idea is to give an overapproximation of what polynomials can be derived in degree at most $d$ by defining an operator $R$ on multilinear polynomials such that all degree-$d$ consequences of the axioms are contained in the set $\{p \mid R(p) = 0\}$. The degree lower bound then follows by showing that $R(1) \neq 0$.

Formally, let $d \in \mathbb{N}^+$ be a positive integer. Suppose that there exists a linear operator $R$ on a set $\mathcal{P}$ of (multilinear) polynomials with the following properties:

1. $R(1) \neq 0$.
2. $R(f) = 0$ for all axioms $f \in \mathcal{P}$.
3. For every term $t$ with $\text{Deg}(t) < d$ and every variable $x$ it holds that $R(xt) = R(xR(t))$.

Then any polynomial calculus refutation of $\mathcal{P}$ requires degree strictly greater than $d$. (Note that here we can restrict our attention to PC without twin variables for literals, since the degree measure is the same for PC and PCR.)

The proof of this claim is not hard. The basic idea is that $R$ will map all axioms to 0 by property 2 and further derivation steps in degree at most $d$ will yield polynomials that also map to 0 by the linearity of $R$ and property 3 (where we use that without loss of generality we can implement multiplication by a monomial by multiplying by all variables in it one by one). But then property 1 implies that no derivation in degree at most $d$ can reach 1 and establish contradiction.

However, constructing such operators to obtain degree lower bounds seems much harder than proving resolution width lower bounds, and the technical machinery is much less well developed.

With the exception of Tseitin formulas and onto FPHP formulas, all the formulas discussed in detail in Section 1.4.1 are equally hard also with respect to polynomial calculus size, which can be shown via degree lower bounds arguments:

- Hardness of the standard CNF encoding [1.6a]–[1.6b] of PHP formulas\footnote{Here a twist is needed since these formulas have high initial degree, but we will not go into this. The most elegant solution is to consider so-called graph PHP formulas as discussed in, e.g., [BW01][MN15].} follows from [AR03], with some earlier work on other non-CNF encodings in [Raz98][IPS99]. The proof in [AR03] works also if onto clauses [1.6d] are added, and more recently it was shown in [MN15] that FPHP formulas with clauses [1.6a]–[1.6c] are also hard (whereas with both onto and functionality axioms added the formulas are easy, as noted above).

- Strong degree and size lower bounds on random $k$-CNF formulas were shown by [BH99] for polynomial calculus over fields of characteristic distinct from 2, and lower bounds in any characteristic including 2 were established by different methods in [AR03].

- For the subset cardinality formulas in Figure 1.6 [MN14] also proved poly-
Chapter 1. Proof Complexity

nomial calculus degree and size lower bounds.

• Also, “Tseitin formulas with the wrong modulus” are hard — one can define Tseitin-like formulas encoding counting modulo primes $q$, and such formulas are hard over fields of characteristic $p \neq q$ [BGIP01, AR03].

1.5.5. Polynomial Calculus Space

We next turn to a discussion of space complexity. This measure does not make too much sense for Nullstellensatz, since refutations of the form (1.13) are presented in “one shot” and it is hard to talk about any intermediate results. Thus, our discussion of space will focus on polynomial calculus.

Recall that for resolution we measure space as the number of clauses in memory, and since clauses turn into monomials in polynomial calculus resolution the natural analogue here is monomial space (in our discussion of space we are always focusing on PCR). The first monomial space lower bounds were shown for PHP formulas in [ABRW02]. These formulas have wide axioms, however, and if one applies the 3-CNF conversion from (1.1a) to (1.1b) the lower bound proof breaks down.

Monomial space lower bounds for formulas of bounded width were proven only in [FLN+15] for an obfuscated 4-CNF version of PHP formulas. This was followed by optimal, linear lower bounds for random 4-CNF formulas [BG15], and then for Tseitin formulas over expanders but with the added assumptions that either these graphs are sampled randomly or one adds two copies of every edge to get a multigraph [FLM+13]. Somewhat intriguingly, none of these papers could show any nontrivial lower bounds for any 3-CNF formulas. This barrier was finally overcome by [BBG+15], who proved optimal lower bounds on random 3-CNFs. However, the following open problems show that we still do not understand polynomial calculus space very well.

Open Problem 1.2. Prove polynomial calculus space lower bounds (optimal, linear bounds, or even any bounds) for Tseitin formulas over $d$-regular expander graphs for $d = 3$ or even $d > 3$ using no other assumptions than expansion only.

Open Problem 1.3. Prove that PHP formulas require large monomial space in polynomial calculus even when converted to 3-CNF.

Another intriguing question is whether an analogue of the lower bound (1.10) on space in terms of width in resolution holds for $k$-CNF formulas also for polynomial calculus.

Open Problem 1.4. Is it true that space $\geq$ degree + $O(1)$ in polynomial calculus?

For a long time, essentially nothing was known about this problem, except that the work [FLM+13] made what can be described as some limited progress by showing that if a formula requires large resolution width (which is a necessary, but not sufficient, condition for high degree), then the XOR-substituted version

---

Footnote: It is worth noting that these space lower bounds hold for any characteristic, so although Tseitin formulas have small-size refutations over GF(2), such refutations still require large space.
(as in (1.11a)–(1.11c)) requires large space. When applied to Tseitin-like formulas over expander graphs, this result yields an optimal separation of space and degree. Namely, it follows that these formulas can be refuted in degree $O(1)$ but require space $\Omega(N)$. To obtain such separations we have to commit to a finite characteristic $p$ of the underlying field, however, and the formulas encoding counting mod $p$ will separate space and degree only for fields of this characteristic. It would be nice to get a separation that would work in any characteristic, and the candidate formulas to obtain such a result readily present themselves.

**Open Problem 1.5.** Prove (or disprove) that substituted pebbling formulas as in Figure 1.8 require monomial space lower-bounded by the pebbling space of the underlying DAG (which if true would yield a space-degree separation independent of the field characteristic).

Very recently, some quite exciting news regarding Open Problem 1.4 has been announced [GKT19], namely that the PCR monomial space of refuting a formula is lower-bounded by the square root of the resolution refutation width (which, as mentioned above, is stronger than a lower bound in terms of degree, since resolution width can be much larger than polynomial calculus degree). It is not clear whether this result is tight or not.

### 1.5.6. Polynomial Calculus Trade-offs

When it comes to trade-offs in polynomial calculus we again recognize most of the picture from resolution, but there are also some differences. Here is a summary of what is known (where the upper bounds in all of the results hold for PC whereas the lower bound apply also for PCR):

- For space versus degree in polynomial calculus we know strong, essentially optimal trade-offs from [BNT13], and the formulas exhibiting such trade-offs are the same vanilla pebbling contradictions as for resolution (for which we get exactly the same bounds).
- The paper [BNT13] also showed strong size-space trade-offs, and again the formulas used are pebbling contradictions over appropriate DAGs and Tseitin formulas over long, skinny grids. Here there is some loss in parameters as compared to resolution, however, which seems to be due to limitations of the proof techniques rather than to actual differences in formula behaviour. Also, the Tseitin formula trade-off results do not hold over fields of characteristic 2.
- We do not yet know for sure whether the size blow-up in [IPS99] when degree is decreased is necessary, however, since the analysis in [Tha16] works only for resolution (at least so far). This leads to the final open problem about polynomial calculus that we want to highlight in this section.

**Open Problem 1.6.** Are there size-degree trade-offs in polynomial calculus in the sense that size has to blow up when degree is decreased in [IPS99]?

Interestingly, for the weaker Nullstellensatz proof systems size-degree trade-offs were recently shown in [dRMNR19]. For instance, there is a family of 3-CNF formulas $\{F_n\}_{n=1}^{\infty}$ of size $\Theta(n)$ such that:
Chapter 1. Proof Complexity

1. There is a Nullstellensatz refutation of $F_n$ in degree $O(\sqrt[3]{n} \log n)$.
2. There is a Nullstellensatz refutation of $F_n$ of nearly linear size $O(n^{1+\epsilon})$ and degree $O(\sqrt[3]{n} \log n)$.
3. Any Nullstellensatz refutation of $F_n$ in degree at most $\sqrt[3]{n}$ must have exponential size.

This family of formulas are pebbling formulas over suitably chosen graphs.

1.5.7. Algebraic SAT Solving

We conclude this section with a(n all too) brief discussion of algebraic SAT solving. There seems to have been quite some excitement, at least in the theory community, about the Gröbner basis approach to SAT solving after the paper [CEI96] appeared. However, the hoped for improvement in performance from this method failed to materialize in practice. Instead, the CDCL revolution happened...

Some Gröbner basis SAT solvers have been developed, the most notable example perhaps being PolyBoRi [BD09, BDG+09], but they do not seem competitive with resolution-based solvers (and, sadly, PolyBoRi is no longer maintained). Some SAT solvers such as March and CryptoMiniSat successfully implement Gaussian elimination [HvM05], but this is only very limited form of polynomial calculus reasoning.

Is it harder to build good algebraic SAT solvers than CDCL solvers? Or is it just that too little work has been done? (Witness that it took over 40 years for resolution-based SAT solvers to become really efficient.) Or is it perhaps a little bit of both?

Whatever the answer is to these questions, it seems clear that one needs to find some kind of shortcut in order to use Gröbner bases for efficient SAT solving. A full Gröbner basis computation does too much work, since it allows us not only to decide satisfiability but also to read off the number of satisfying assignments, which is believed to be a strictly harder problem.

It is important to emphasize that this slightly downbeat discussion of algebraic SAT solving should not be taken to mean that algebraic methods cannot be used for successfully solving hard combinatorial problems. In this context, it is relevant to mention the sequence of papers [DLMM08, DLM09, DLM11], a body of work that was recognised with the INFORMS Computing Society Prize 2010. In these papers the authors solve graph colouring problems (with great success) essentially by constructing Nullstellensatz certificates of non-colourability. Hence, for some NP-complete problems it seems that even lowly Nullstellensatz can be a quite powerful approach.

Another very interesting line of work is exemplified by the papers [RBK17, RBKT18, RBK19] using Gröbner bases computations to attack the challenging problem of verifying multiplier circuits. As a part of this work, the authors develop a formal proof logging system to certify correctness, and this proof system is nothing other than polynomial calculus (but with the field $F$ chosen to be the rational numbers $\mathbb{Q}$ rather than a finite field).
1.6. Cutting Planes

The cutting planes [CCT87] proof system, which formalizes the integer linear programming algorithm in [Gom63, Chv73] and underlies so-called pseudo-Boolean (PB) SAT solvers, operates with linear inequalities over the reals with integer coefficients. To reason about CNF formulas in cutting planes, the disjunctive clauses are translated to linear inequalities, which are then manipulated to derive a contradiction. Thus, the question of Boolean satisfiability is reduced to the geometry of polytopes over the real numbers. Just as algebraic proof systems can deal not only with translated CNF formulas but with arbitrary sets of polynomials, cutting planes can operate on arbitrary 0-1 integer linear constraints, which we will also refer to as pseudo-Boolean constraints.

In the standard proof complexity setting, we use only positive literals (unnegated variables) and identify $z$ with $1 - z$ so that, for instance, the clause $x \lor y \lor z$ gets translated to $x + y + (1 - z) \geq 1$, or $x + y - z \geq 0$ after we have moved all integer constants to the right-hand side. However, in order to give a description of cutting planes that is helpful also when we want to reason about pseudo-Boolean solvers, and in order to get compact notation, it is more useful to keep negated literals as variables in their own right, and to insist that all inequalities consist of linear combinations of (positive or negative) literals with only non-negative coefficients. It will also be convenient here to use the notation $x^\sigma$, $\sigma \in \{0, 1\}$, mentioned in Section 1.2, where we recall that $x^1 = x$ and $x^0 = \overline{x}$. With this notation, assuming that our set of variables is $\{x_1, \ldots, x_n\}$ we can write all linear constraints in normalized form

$$\sum_{i \in [n], \sigma \in \{0, 1\}} a^\sigma_i x^\sigma_i \geq A,$$  \hspace{1cm} (1.19)

where for all $i \in [n]$ and $\sigma \in \{0, 1\}$ it holds that $a^\sigma_i \in \mathbb{N}$ and $\min\{a^0_i, a^1_i\} = 0$ (the latter condition specifies that variables occur only with one sign in any given inequality), and where $A \in \mathbb{N}^+$ (this constant term is often referred to as the degree of falsity, or just degree, in an applied pseudo-Boolean solving context). In what follows, all expressions of the type (1.19) are supposed to be in normalized form, and all sums are assumed to be taken over all combinations of $i \in [n]$ and $\sigma \in \{0, 1\}$ except as specified under the summation sign.

If the input is a CNF formula $F$ we just view every clause $C \in F$ of the form

$$C = x_1^{\sigma_1} \lor x_2^{\sigma_2} \lor \cdots \lor x_w^{\sigma_w}$$  \hspace{1cm} (1.20a)

as a linear constraint

$$x_1^{\sigma_1} + x_2^{\sigma_2} + \cdots + x_w^{\sigma_w} \geq 1.$$  \hspace{1cm} (1.20b)

That is, a disjunctive clause is simply a constraint on the form (1.19) where $a^\sigma_i \in \{0, 1\}$ and $A = 1$ (in particular, our example clause $x \lor y \lor z$ now becomes $x^1 + y^1 + z^0 \geq 1$).

Pseudo-Boolean constraints can be exponentially more concise than CNF, as is shown by a comparison of the constraint

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 3$$  \hspace{1cm} (1.21a)
with

\[(x_1 \lor x_2 \lor x_3 \lor x_4) \land (x_1 \lor x_2 \lor x_3 \lor x_5) \land (x_1 \lor x_2 \lor x_3 \lor x_6)\]
\[\land (x_1 \lor x_2 \lor x_4 \lor x_5) \land (x_1 \lor x_2 \lor x_4 \lor x_6) \land (x_1 \lor x_2 \lor x_5 \lor x_6)\]
\[\land (x_1 \lor x_3 \lor x_4 \lor x_5) \land (x_1 \lor x_3 \lor x_4 \lor x_6) \land (x_1 \lor x_3 \lor x_5 \lor x_6)\]
\[\land (x_1 \lor x_4 \lor x_5 \lor x_6) \land (x_2 \lor x_3 \lor x_4 \lor x_5) \land (x_2 \lor x_3 \lor x_4 \lor x_6)\]
\[\land (x_2 \lor x_3 \lor x_5 \lor x_6) \land (x_2 \lor x_4 \lor x_5 \lor x_6) \land (x_3 \lor x_4 \lor x_5 \lor x_6)\]

Constraints of the form (1.21a), i.e., such that \(a_i^c \in \{0, 1\}\) holds for all coefficients, are called \textit{cardinality constraints}, since they encode that at least \(A\) of the literals in the constraint are true. We can also have general pseudo-Boolean constraints such as, say, \(x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 \geq 7\).

\subsection{1.6.1. Pseudo-Boolean Rules of Reasoning}

Not only are pseudo-Boolean constraints much more concise than clauses, but the rules used to manipulate them are also more powerful. Using the normalized form (1.19), the derivation rules in the standard proof complexity definition of cutting planes are

\begin{align*}
\text{Literal axioms} & \quad x_i^c \geq 0 & (1.22a) \\
\text{Multiplication} & \quad \sum a_i^c x_i^c \geq A \quad c \in \mathbb{N}^+ & (1.22b) \\
\text{Addition} & \quad \sum a_i^c x_i^c \geq A \quad \sum b_i^c x_i^c \geq B \quad \sum (a_i^c + b_i^c)x_i^c \geq A + B & (1.22c) \\
\text{Division} & \quad \sum ca_i^c x_i^c \geq A \quad \sum a_i^c x_i^c \geq \lceil A/c \rceil \quad c \in \mathbb{N}^+ & (1.22d)
\end{align*}

where in the addition rule (1.22c) we implicitly assume that the result is rewritten in normalized form. Let us illustrate this by a small example. It is important to note that when we add literals of opposite sign, the result is

\[x_i^1 + x_i^0 = x_i^1 + (1 - x_i^1) = 1\]  \hspace{1cm} (1.23)

(which is just another way of saying that it will always be the case that exactly one of the literals \(x\) and \(\overline{x}\) is true). If we have the two constraints

\[x + 2y + 3z + 4\overline{w} \geq 5\]  \hspace{1cm} (1.24)

and

\[3x + 2\overline{y} + z \geq 3\ ,\]  \hspace{1cm} (1.25)

then by applying the addition rule (1.22c) we get the expression

\[x + (2 - 2)y + (3 - 1)z + 4\overline{w} + 3x \geq 5 + 3 - (2 + 1)\]  \hspace{1cm} (1.26a)
which in the normalized form \((1.19)\) becomes
\[
4x + 2z + 4w \geq 5
\]
(1.26b)
(where we suppress terms with zero coefficients). We note that when adding \((1.24)\) and \((1.25)\) to obtain \((1.26b)\) the coefficients for \(y\) and \(z\) cancel so that this variable disappears. In general, when we add two constraints \(\sum a_i^\sigma x_i^\sigma \geq A\) and \(\sum b_i^\sigma x_i^\sigma \geq B\) such that there is a variable \(x_i\) and a \(\sigma \in \{0,1\}\) for which \(a_i^\sigma = b_i^{1-\sigma} > 0\), we say that this is an instance of \textit{cancelling addition}.

More generally, when two linear constraints \(\sum a_i^\sigma x_i^\sigma \geq A\) and \(\sum b_i^\sigma x_i^\sigma \geq B\) share a variable \(x_j\) for which \(a_j^\sigma > 0\) and \(b_j^{1-\sigma} > 0\) hold, then we can multiply the constraints by the smallest numbers \(c_A\) and \(c_B\) such that \(c_A a_j^\sigma = c_B b_j^{1-\sigma}\), and then apply cancelling addition. Writing \(d = \gcd(a_j^\sigma, b_j^{1-\sigma})\) for the greatest common divisor of \(a_j^\sigma\) and \(b_j^{1-\sigma}\), the formal specification of this rule is
\[
\frac{a_j x_j^\sigma + \sum_{i \neq j, \sigma} a_i^\sigma x_i^\sigma \geq A}{\sum_{i \neq j, \sigma} (b_i a_i^\sigma / d) + (a_j b_j^\sigma / d)) x_i^\sigma \geq b_j A / d + a_j B / d - a_j b_j / d}
\]
(1.27)
(where as before we implicitly assume that the result of the linear combination is put into normalized form). Note that this can be viewed as a kind of generalization of the resolution rule \((1.2)\) from disjunctive clauses to general linear constraints. We therefore refer to \((1.27)\) as \textit{generalized resolution} (or sometimes \textit{cancelling linear combination}), and we say that the two constraints are \textit{resolved over} \(x_j\).

This rule essentially goes back to Hooker [Hoo88, Hoo92] (although Hooker’s definition is slightly different in that the cancelling addition has to be followed by a division step).

Given a set of linear inequalities, one can show that there is no \(\{0,1\}\)-valued solution by using the cutting planes rules to derive the inequality \(0 \geq 1\) from the given linear inequalities. It is clear that such a refutation can exist only if the instance is indeed unsatisfiable. The other direction also holds, but requires more work to establish.

We want to highlight that in the division rule \((1.22d)\) (which is also known as the \textit{Chvátal-Gomory cut rule}) we can divide with the common factor \(c\) on the left and then \textit{divide and round up} the constant term on the right to the closest integer, since we know that we are only interested in \(\{0,1\}\)-valued solutions. This division rule is where the power of cutting planes lies. (And, indeed, this is how it must be, since a moment of thought reveals that the other rules are sound also for real-valued variables, and so without the division rule we would not be able to distinguish sets of linear inequalities that have real-valued solutions but no \(\{0,1\}\)-valued solutions.)

It is not hard to see that we can modify the definitions slightly to obtain a more cleanly stated \textit{general division rule}
\[
\frac{\sum a_i^\sigma x_i^\sigma \geq A}{\sum |a^\sigma | / c|x_i^\sigma \geq |A / c|} \quad c \in \mathbb{N}^\ast
\]
(1.28)
without changing anything essential, since this rule can easily be simulated by using rules \((1.22a)\) and \((1.22d)\). Therefore, although the standard definition of division in the proof complexity literature is as in \((1.22d)\) without loss of generality we will use rule \((1.28)\).
Chapter 1. Proof Complexity

A small example just to illustrate how the rules can be combined is the derivation

\[
\begin{align*}
6x + 2y + 3z & \geq 5 \\
2x + 4y + w & \geq 1 \\
\frac{8x + 6y + 3z + 2w}{3x + 2y + z + w} & \geq 7
\end{align*}
\]

where we note that we do not lose any information in the final division step. This is not always true when using the general division rule (1.28)—for instance, a further division of \(3x + 2y + z + w \geq 3\) by 3 would yield the clause \(x + y + z + w \geq 1\), which is a strictly weaker constraint.

Pseudo-Boolean SAT solvers such as Sat4j [LP10] do not implement the full set of cutting planes derivation rules as presented above, however. To describe how they work, we need to introduce two other rules, namely

\[
\begin{align*}
\text{Weakening} & : \quad \sum_{i}(a_{i}x_{i}) \geq A \quad \sum_{i, i \neq j}(a_{i}x_{i}) \geq A - (a_{j} + a_{j}) \\
\text{Saturation} & : \quad \sum_{i}(a_{i}x_{i}) \quad \sum_{i}(a_{i}A)x_{i} \geq A \
\end{align*}
\]

The weakening rule is merely a convenient shorthand for one application each of the rules (1.22a), (1.22b), and (1.22c). Just as division, saturation is a special rule in that it is sound only for integral solutions. A second small toy example

\[
\begin{align*}
3\pi + 2y + 2z + u + w & \geq 5 \\
2\pi + 2y + u + w & \geq 3 \\
\frac{7\pi + 3z + 2u + 2w}{6\pi + 3z + 2u + 2w} & \geq 6
\end{align*}
\]

shows how weakening, generalized resolution, and saturation can be combined.

In the proof complexity literature the focus has been on investigating the power of general cutting planes. To understand the reasoning power of pseudo-Boolean solvers, however, it makes sense to study several different variants of the cutting planes proof system as discussed in [VEG+18]:

**General cutting planes:** Rules (1.22a)–(1.22c) and (1.28).

**Cutting planes with resolution:** As general cutting planes above, except that all applications of (1.22b) and (1.22c) have to be combined into applications of the generalized resolution rule (1.27).

**Cutting planes with saturation:** Rules (1.22a)–(1.22c) and (1.30b) (with no restrictions on the linear combinations).

**Cutting planes with saturation and resolution:** As cutting planes with saturation above, except that all applications of (1.22b) and (1.22c) have to be combined instances of generalized resolution (1.27).
Chapter 1. Proof Complexity

1.6.2. Conflict-Driven Pseudo-Boolean Solving

To explain why the different flavours of cutting planes just introduced are interesting from an applied point of view, we next turn to pseudo-Boolean solvers and how they can be used to determine the satisfiability of pseudo-Boolean formulas (i.e., sets of pseudo-Boolean constraints).

One approach to solving pseudo-Boolean formulas is to convert them to CNF, either lazily by learning clauses from PB constraints during conflict analysis, as in one of the version in the Sat4j library [LP10], or eagerly by re-encoding the whole formula to CNF and running a CDCL solver as in, e.g., MiniSat+ [ES06], Open-WBO [MML14], or NaPS [SN15]. In the context of cutting planes we are more interested in solvers doing native pseudo-Boolean reasoning, such as PRS [DG02], Galena [CK05], Pueblo [SS06], Sat4j [LP10], and RoundingSat [EN18], so this is where our focus will be below (we mention, though, that related, but slightly different, ideas were also explored in bsolo [MM06]). Our discussion of pseudo-Boolean solvers cannot come anywhere close to doing full justice to the topic of PB solving or the even richer area of PB optimization—for this, we refer the reader to the chapter [RM09] in this handbook. Another source of much useful information is Dixon’s PhD thesis [Dix04].

In our discussion of PB solving, the standard setting is that the input is a PB formula without 0-1 solutions, and the goal of the solver is to decide that the formula is contradictory. For readers more interested in optimization, this is also the situation when the solver should prove that the (linear) objective function cannot be better than in the current solution.

Just as CDCL solvers can be viewed as searching for resolution proofs, we will see that the pseudo-Boolean solving techniques we will discuss generate proofs in different subsystems of cutting planes. Simplifying somewhat, when building a PB solver on top of cutting planes we have the following choices:

1. Boolean rule: (a) saturation or (b) division.
2. Linear combinations: (a) generalized resolution or (b) no restrictions.

As we will soon see, the choice of generalized resolution seems inherent in a conflict-driven setting, which is what we are focusing on here, but which Boolean rule to prefer is less clear. Saturation was used in the seminal paper [CK05] and has also been the rule of choice in what is arguably the most popular pseudo-Boolean solver to date, namely Sat4j [LP10]. Division appeared only recently in RoundingSat [EN18] (although it was suggested in a more general integer linear programming setting in [MM13]). We will return to a discussion of saturation versus division later, but let us first describe the general set-up.

Naively, when generalizing CDCL to a pseudo-Boolean setting we just want to build a solver that decides on variable values and propagates forced values until conflict, at which point a new linear constraint is learned and the solver backtracks. To decide when constraints are propagating or conflicting it is convenient to use the concept of slack, which we define next.

The slack of a constraint \( \sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A \) under a partial assignment \( \rho \) (which we should think of as the variables currently assigned on the trail) measures how far \( \sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A \) is from being falsified by \( \rho \), and is
Chapter 1. Proof Complexity

**Proof Complexity**

\[ \rho \]  
\[ \text{slack}(C; \rho) \]  
Comment

| \( (\emptyset) \) | 8 | Sum of coefficients minus degree of falsity. |
| \( (\mathbb{T}_1) \) | 3 | Propagates \( \mathbb{T}_1 \), since coefficient > slack. |
| \( (\mathbb{T}_2, \mathbb{T}_4) \) | 3 | Propagation does not change slack. |
| \( (\mathbb{T}_3, \mathbb{T}_4, \mathbb{T}_5, x_2) \) | -2 | Conflict, since slack < 0. |

**Figure 1.10:** Slack of \( C = x_1 + 2\mathbb{T}_2 + 3x_3 + 4\mathbb{T}_4 + 5x_5 \geq 7 \) for different trails \( \rho \).

**Proposition**

\[ \text{slack} \left( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A; \rho \right) = \sum_{\rho(x_{\sigma}^i) \neq 0} a_{\sigma}^i - A, \quad (1.32) \]

i.e., the sum of the coefficients of all literals that are not falsified by \( \rho \) minus the constant term. The constraint \( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A \) is conflicting under \( \rho \) if

\[ \text{slack} \left( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A; \rho \right) < 0 \quad (1.33) \]

and it propagates \( x_{\sigma}^{i'} \) under \( \rho \) if

\[ 0 \leq \text{slack} \left( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A; \rho \right) < a_{\sigma}^{i'} \quad (1.34) \]

(which is just a fancy way of saying that we have to satisfy \( x_{\sigma}^{i'} \), or else there will be no way to satisfy the constraint).

The above definitions might be easier to digest by studying the example in Figure 1.10 of how the slack changes for the constraint \( C = x_1 + 2\mathbb{T}_2 + 3x_3 + 4\mathbb{T}_4 + 5x_5 \geq 7 \) under different partial assignments \( \rho \) on the trail. The initial slack is just the sum of the coefficients minus the degree. If \( x_5 \) is assigned to false, then \( x_4 \) is propagated to false according to (1.34). Assigning a variable to its propagated value does not change the slack. If now the solver for some other reason sets \( x_3 \) to false and \( x_2 \) to true, we have a conflict according to (1.33). Note that in contrast to clauses in CDCL, a pseudo-Boolean constraint can be conflicting even though not all variables have been assigned.

Now we can give a slightly more detailed (though still incomplete) sketch of how we would like our pseudo-Boolean solver to work:

1. While there is no conflict, iteratively propagate all literals \( x_{\sigma}^{i'} \) such that there is a constraint for which \( 0 \leq \text{slack} \left( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A; \rho \right) < a_{\sigma}^{i'} \) (i.e., add the literals to the current assignment \( \rho \)).
2. If there is no conflict, find some unset literal \( x_{\sigma}^i \), decide to set it to true (i.e., add \( x_{\sigma}^i \) to \( \rho \)), and go to step 1.
3. Otherwise, fix some conflicting constraint \( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A \), i.e., such that \( \text{slack} \left( \sum_{i \in [n], \sigma \in \{0, 1\}} a_{\sigma}^i x_{\sigma}^i \geq A; \rho \right) < 0 \), and resolve it with the propagating constraints found in step 1 in reverse chronological order until we derive a constraint that is the analogue of an asserting clause.
4. Add this constraint to the formula, backtrack to the first point in the trail where the constraint is not falsified, and go to step [ ]

From this description it is possible to see why the generalized resolution enters the picture in a natural way—the only time we use linear combinations is in step [ ] where there are two constraints that have opposing opinions about what value the current variable under consideration should take.

The simple approach outlined above will not quite work. In what remains of this subsection, we want first to show an example why it fails, and then discuss what can be done to fix the problem. From the examples in Section [ ] we can see that an important invariant during CDCL conflict analysis is that the assignment that is “left on the trail” always falsifies the currently derived clause. This means that every derived constraint “explains” the conflict by showing what assignments on the trail are inconsistent, and we can continue the conflict analysis until the derived constraint looks “nice,” at which point the solver learns it and backtracks.

The concept of “niceness” in CDCL is that the constraint should be asserting, i.e., that if we remove further literals from the trail in reverse chronological order until the first time when the learned constraint is not falsified, then at this point the constraint propagates some variable that flips an earlier assignment. When we generalized conflict-driven solving to a pseudo-Boolean setting, we would like the conflict analysis to work in the same way.

As a running example, let us consider the PB formula consisting of the two constraints

\[
C_1 = 2x_1 + 2x_2 + 2x_3 + x_4 \geq 4 \quad (1.35a)
\]

\[
C_2 = 2x_1 + 2x_2 + 2x_3 \geq 3 \quad (1.35b)
\]

(which is just a rather artificial obfuscated way of writing that a majority of the variables \( \{x_1, x_2, x_3, \} \) have to be true and false at the same time, but it is a simple example that will help us illustrate our main points). Note that both constraints have slack 3 under the empty assignment, which is larger than all coefficients, so there are no propagations at decision level 0. Suppose that the solver sets \( x_1 \) to false to get the trail

\[
\rho_1 = (x_1 \leftarrow 0) \quad .
\]

Now \( \text{slack}(C_1; \rho_1) = 1 \), which is less than the coefficient 2 of \( x_2 \), so \( C_1 \) propagates \( x_2 \) to true, yielding

\[
\rho_2 = (x_1 \leftarrow 0, x_2 \leftarrow C_1) \quad .
\]

For the same reason, \( x_3 \) is also propagates to true by \( C_1 \) (note that, in contrast to CDCL, the same constraint can propagate several times). For this trail

\[
\rho_3 = (x_1 \leftarrow 0, x_2 \leftarrow C_1, x_3 \leftarrow C_1) \quad (1.36c)
\]

the slack of the other constraint \( C_2 \) is \( \text{slack}(C_2; \rho_3) = -1 \), and we have reached a conflict.

Inspired by CDCL conflict analysis, we take the reason constraint \( C_1 \equiv \text{reason}(x_3; \rho_2) \) propagating the last literal \( x_3 \) under \( \rho_2 \) and resolve it over \( x_3 \) with
the conflicting constraint \( C_2 \), i.e., performing the derivation step
\[
2x_1 + 2x_2 + 2x_3 + x_4 \geq 4 \quad \frac{2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3}{x_4 \geq 1}
\]
(1.37)
to obtain the resolvent \( x_4 \geq 1 \). But now we have a problem: the slack of the resolvent with respect to what remains on the trail is \( \text{slack}(x_4 \geq 1; \rho_2) = 0 \). This is no longer negative, so we have lost the invariant from CDCL that the constraint derived on the conflict side should be conflicting!

Taking a step back to analyse what happened, the reason for this failure is that it is in fact possible to satisfy both constraints \( C_1 \) and \( C_2 \) by extending \( \rho_2 \) with the assignment \( x_3 = \frac{1}{2} \). Of course, this is not a Boolean assignment, but taking linear combinations is not a Boolean rule but is sound also over the reals. For this reason, there is no way we can guarantee that the invariant of a conflicting constraint on the conflict side can be maintained if we use only the generalized resolution rule (1.27).

Thus, we need to get some Boolean derivation rule into play. We will now describe how Chai and Kuehlmann \( [CK05] \) adapt conflict analysis to a pseudo-Boolean setting using the saturation rule. Saturation in itself cannot help fix our problem, because both constraints resolved in (1.37) are already saturated, as is the resolvent. But if we combine saturation with weakening of the reason constraint, then (perhaps somewhat counter-intuitively) we can get the conflict analysis to work. When resolving a propagating constraint on the reason side with the currently derived constraint on the conflict side, we will iterate the following procedure:

1. weaken the reason constraint on some non-falsified literal (other than the last one propagated);
2. saturate the weakened constraint;
3. resolve with the conflicting constraint over the propagated literal;

until we obtain a resolvent that is conflicting. Let us first show how this works for our example, and then discuss why this is a correct approach in general.

If we weaken \( \text{reason}(x_3, \rho_2) \equiv C_1 \) on \( x_2 \), which is the first non-falsified literal that is not the one currently propagated, then we get the following derivation:

\[
\begin{align*}
\text{Weakening on } x_2 & : \quad 2x_1 + 2x_2 + 2x_3 + x_4 \geq 4 \\
\text{Saturation} & : \quad 2x_1 + 2x_3 + x_4 \geq 2 \\
\text{Resolution on } x_3 & : \quad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3 \\
\text{Resolution on } x_3 & : \quad 2\bar{x}_2 + x_4 \geq 1
\end{align*}
\]

Unfortunately, this does not solve the problem, since \( 2\bar{x}_2 + x_4 \geq 1 \) has positive slack 2 with respect to the trail \( \rho_2 \) in (1.36).

We cannot weaken away \( x_3 \), since this is the propagating literal we want to resolve over, but we can weaken \( C_2 \) also on \( x_4 \), which is not falsified. This yields

\[
\begin{align*}
\text{Weakening on } x_2 & : \quad 2x_1 + 2x_2 + 2x_3 + x_4 \geq 4 \\
\text{Weakening on } x_4 & : \quad 2x_1 + 2x_3 + x_4 \geq 2 \\
\text{Saturation} & : \quad x_1 + x_3 \geq 1 \\
\text{Resolution on } x_3 & : \quad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3 \\
\text{Resolution on } x_3 & : \quad 2\bar{x}_2 \geq 1
\end{align*}
\]

(1.39)
while slack(resolve(C_{confl}, C_{reason}, ℓ); ρ) ≥ 0 do
    ℓ’ ← literal in C_{reason} \ {ℓ} not falsified by ρ;
    C_{reason} ← saturate(weaken(C_{reason}, ℓ’));
end
return C_{reason};

Figure 1.11: Chai-Kuehlmann reason reduction reduceSat(C_{confl}, C_{reason}, ℓ, ρ).

and now we have slack(2x_2 ≥ 1; ρ_2) = −1 < 0, i.e., we have derived a new constraint that maintains the invariant of having negative slack with respect to what remains on the trail.

When we continue by saturating this new constraint and resolving it over x_2 with reason(x_2, ρ_1) = C_1 we obtain

\[
\begin{align*}
\text{Resolution on } x_2 & \quad 2x_1 + 2x_2 + 2x_3 + x_4 ≥ 4 \\
& \quad \frac{2x_2 ≥ 1}{2x_1 + 2x_3 + x_4 ≥ 4} \quad \text{Saturation}
\end{align*}
\]

(1.40)

and although we have not formally defined anything like 1UIP PB constraints—and, indeed, doing so requires some care—it should be clear that the constraint now derived is asserting. If we undo all decisions on the trail, then at top level we have slack(2x_1 + 2x_3 + x_4 ≥ 4; ∅) = 1, and since this is smaller than the coefficient 2 of x_1 and x_3 both variables propagate to true. This causes a conflict with C_2, and since no decisions have been made the solver can terminate and report that the formula consisting of the constraints (1.35a) and (1.35b) is indeed unsatisfiable.

The key to the pseudo-Boolean conflict analysis just described is that we apply a reduction algorithm on the reason constraint, combining weakening and saturation, to ensure that when the reduced reason constraint is resolved with the currently derived conflict constraint, then the result will be a new conflicting constraint. The pseudocode for this reduction algorithm from [CK05] is given in Figure 1.11. But how do we know that it is guaranteed to work?

Briefly, the reason is that slack is subadditive, i.e., if we take a linear combination of two constraints C and D, then it is not hard to verify that

\[
\text{slack}(c \cdot C + d \cdot D; ρ) ≤ c \cdot \text{slack}(C; ρ) + d \cdot \text{slack}(D; ρ)
\]

(1.41)

holds. By the invariant, we know for the currently derived constraint C_{confl} on the conflict side that we have slack(C_{confl}; ρ) < 0. It is also easy to see directly from the definition (1.32) that weakening the reason constraint C_{reason} leaves slack(C_{reason}; ρ) unchanged, since we only weaken on non-falsified literals. But saturation can decrease the slack, and if we have not reached non-positive slack before, then at the very latest this will happen when all non-falsified literals except the propagating one have been weakened away—at this stage the only coefficient contributing to the slack is that of the propagating literal, and since the constraint is saturated this coefficient must be equal to the degree of falsity, so that the whole constraint has slack 0. (This is exactly what happened in our
while $C_{\text{conf}}$ not asserting do
\[ \ell \leftarrow \text{literal assigned last on trail } \rho; \]
if $\ell$ occurs in $C_{\text{conf}}$ then
\[ C_{\text{reason}} \leftarrow \text{reason}(\ell, \rho); \]
\[ C_{\text{reason}} \leftarrow \text{reduceSat}(C_{\text{reason}}, C_{\text{conf}}, \ell, \rho); \]
\[ C_{\text{conf}} \leftarrow \text{resolve}(C_{\text{conf}}, C_{\text{reason}}, \ell); \]
\[ C_{\text{conf}} \leftarrow \text{saturate}(C_{\text{conf}}); \]
end
\[ \rho \leftarrow \text{removeLast}(\rho); \]
end
return $C_{\text{conf}}$;

Figure 1.12: Pseudo-Boolean conflict analysis \texttt{analyzePBconflict}($C_{\text{conf}}, \rho$).

example.) Plugging this into (1.41), we see that a positive linear combination of zero and a negative number will be negative, and the invariant is maintained.

Using this reason reduction method, the whole pseudo-Boolean conflict analysis algorithm will be as in Figure 1.12. The highlighted reduction step is new compared to CDCL, but everything else is essentially the same (at least at a high level). So how does our conflict analysis compare to CDCL? Let us just point out three important aspects here, which will motivate some of the discussions later.

1. On difference is how much work needs to be performed at each step. When we resolve a new reason with the current conflict clause in CDCL, then we only have to “tag on” the reason clause to the conflict clause, but we do not have to touch the literals already in the conflict clause. Therefore, the total amount of work during CDCL conflict analysis is linear in the sum of the clause sizes. But in pseudo-Boolean analysis we might have to multiply both the reason and the conflict during the lcm computations for the generalized resolution steps, and this means that we might have to touch all literals in the constraint on the conflict side over and over again. In the worst case, this will incur an extra linear factor in the running time.

2. Because of the lcm computations, it can also be the case that the coefficients in the constraints grow very large. If this happens, then the integer arithmetic can get hugely expensive. This can become a very serious problem in practice.

3. Perhaps the most serious problem, though, is that for inputs in CNF this procedure described above degenerates to resolution. All that will happen during the conflict analysis is that trivial resolution derivations will produce new clauses, and so the whole algorithm just becomes CDCL but with much more expensive data structures. Hence, if we take the pigeonhole principle formula encoded in CNF and feed it into a pseudo-Boolean solver using this conflict analysis, then although the formula is easy for cutting planes as shown in [CCT87] it will be exponentially hard in practice.

The third issue is perhaps the most important one in that it shows how sensitive pseudo-Boolean solvers can be to details of the encoding. We will return to the question of CNF inputs and discuss it in a bit more detail in Section 1.8.
Chapter 1. Proof Complexity

51

$c \leftarrow \text{coeff}(C_{\text{reason}}, \ell)$;

while slack(resolve($C_{\text{conf}}, \text{divide}(C_{\text{reason}}, c), \ell, \rho$); $\rho \geq 0$ do

$\ell_j \leftarrow$ literal in $C_{\text{reason}} \setminus \{\ell\}$ such that $\ell_j \notin \rho$ and $c \nmid \text{coeff}(C, \ell_j)$;

$C_{\text{reason}} \leftarrow \text{weaken}(C_{\text{reason}}, \ell_j)$;

end

return divide($C_{\text{reason}}, c$);

Figure 1.13: Reduction reduceDiv($C_{\text{conf}}, C_{\text{reason}}, \ell, \rho$) using division.

In this context, it is interesting to consider whether the division rule could be a competitive alternative to saturation. It is known that general cutting planes (consisting of the rules (1.22a)–(1.22c) and (1.28)) is implicationally complete, meaning that if a PB formula implies a certain constraint, there is a way to derive this constraint. This is not true for cutting planes with saturation [VEG+18], i.e., when the saturation rule (1.30b) is substituted for the division rule (1.28). For instance, one can encode a cardinality constraint in CNF in such a way that cutting planes with saturation cannot recover a constraint of the form (1.21a). In view of this, it is natural to ask whether the use of division could perhaps yield a stronger conflict analysis algorithm. As mentioned above, using division was proposed in the context of general integer linear programming in CutSat [JdM13], although it appears that this approach does not work so well in practice. What we will discuss below is a fairly recent variant of pseudo-Boolean conflict analysis that uses division instead of saturation, and that does seem to work very well in practice.

In the conflict analysis method using division, for each resolution step we will iterate the steps

1. weaken the reason constraint on some non-falsified literal with coefficient not divisible by the coefficient of the propagating literal;
2. divide the weakened constraint by the coefficient of the propagating literal;
3. resolve with the conflicting constraint over the propagated literal

until the resolvent obtained is conflicting. Before arguing about correctness, let us do as we did for the saturation-based method and illustrate how this approach works for the constraints $C_1$ and $C_2$ from (1.35a)–(1.35b) with the trail $\rho_3 = (x_1 \overset{\text{sec}}{\leftarrow} 0, x_2 \overset{\text{sec}}{\leftarrow} 1, x_3 \overset{\text{sec}}{\leftarrow} 1)$ in (1.36c), under which $C_2$ is conflicting. The first attempt to resolve the reason $C_1$ for $x_3$ with the conflict constraint $C_2$ now yields the following derivation:

$$2x_1 + 2x_2 + 2x_3 + x_4 \geq 4$$

Weakening on $x_4$

$$2x_1 + 2x_2 + 2x_3 \geq 3$$

Division by 2

$$x_1 + x_2 + x_3 \geq 2$$

Resolution on $x_3$

$$2x_1 + 2x_2 + 2x_3 \geq 3$$

That is, for this particular example the solver immediately derives contradiction and can terminate with a report that the instance is unsatisfiable.

The pseudocode for the reason reduction algorithm from [EN18] using division is given in Figure 1.13. Let us sketch the argument why this algorithm is...
Chapter 1. Proof Complexity

\[ c \leftarrow \text{coeff}(C, \ell); \]
\[
\text{foreach literal } \ell_j \text{ in } C \text{ do}
\]
\[
\text{if } \ell_j \notin \rho \text{ and } c \mid \text{coeff}(C, \ell_j) \text{ then}
\]
\[
C \leftarrow \text{weaken}(C, \ell_j);
\]
\end{verbatim}
\text{return } \text{divide}(C, c);

Figure 1.14: Reason reduction roundToOne\((C, \ell, \rho)\) used in RoundingSat.

guaranteed to return a reduced reason constraint that when resolved with the currently derived conflict constraint will maintain the invariant that the constraint on the conflict side has negative slack with respect to the current trail. Just in the analysis of the reason reduction algorithm reduceSat using saturation in Figure 1.11 it is sufficient to prove that at some point the constraint on the reason side is guaranteed to become non-positive. This is sufficient to maintain the conflict analysis invariant of negative slack, since the constraint \(C_{\text{conf}}\) on the conflict side has negative slack by assumption and slack is subadditive (1.41), meaning that the resolvent of the reason and conflict constraints also has to have negative slack.

Following the notation in Figure 1.13 let \(c\) be the coefficient of the literal \(\ell\) propagated by \(C_{\text{reason}}\). By the definition of propagation in (1.34) we have

\[ 0 \leq \text{slack}(C_{\text{reason}}; \rho) < c, \quad (1.43) \]

and since weakening on non-falsified literals does not change the slack these inequalities holds at all times during the execution of reduceDiv. Suppose we have reached the point in the algorithm when all coefficients of non-falsified literals not divisible by \(c\) have been weakened away. Consider what contribution the literals in divide\((C_{\text{reason}}, c)\) give to the slack. Falsified literals in \(C_{\text{reason}}\) do not contribute at all, and all remaining non-falsified literals have coefficients divisible by \(c\). Therefore, the slack of the reason constraint is divisible by \(c\), i.e., we have

\[ \text{slack(divide}(C_{\text{reason}}, c); \rho) = \frac{\text{slack}(C_{\text{reason}}; \rho)}{c}, \quad (1.44) \]

and it follows from this and (1.43) that

\[ 0 \leq \text{slack(divide}(C_{\text{reason}}, c); \rho) < 1, \quad (1.45) \]

i.e., \(\text{slack(divide}(C_{\text{reason}}, c); \rho) = 0\). This proves the correctness of reduceDiv.

We remark that the reason reduction method roundToOne actually used in RoundingSat, presented in Figure 1.14 does not weaken literals one by one, but does the maximal amount of weakening right away. This is guaranteed to maintain the invariant by the proof just outlined above. Also, this method is used not only for reason reduction but is applied more aggressively during the conflict analysis. The pseudocode for the conflict analysis in RoundingSat is presented in Figure 1.15.
while $C_{conf}$ contains no or multiple falsified literals on last level do
  if no current solver decisions then
    output UNSAT and terminate
  end
  $\ell \leftarrow$ literal assigned last on trail $\rho$;
  if $\ell$ occurs in $C_{conf}$ then
    $C_{conf} \leftarrow \text{roundToOne}(C_{conf}, \ell, \rho)$;
    $C_{reason} \leftarrow \text{roundToOne}(C_{reason}(\ell, \rho), \ell, \rho)$;
    $C_{conf} \leftarrow \text{resolve}(C_{conf}, C_{reason}, \ell)$;
  end
  $\rho \leftarrow \text{removeLast}(\rho)$;
end
$\ell \leftarrow$ literal in $C_{conf}$ last falsified by $\rho$;
return $\text{roundToOne}(C_{conf}, \ell, \rho)$;

Figure 1.15: Pseudo-Boolean conflict analysis in RoundingSat using roundToOne.

It is an interesting question how saturation and division compare when used for pseudo-Boolean solving, but this is currently not very well understood. It is clear from our example (1.42) that division can sometimes be more efficient, but one can also construct crafted benchmarks where it seems that saturation can be better [GNY19]. Nevertheless, some preliminary conclusions are that for instances where pseudo-Boolean reasoning does not help, so that a competitive approach would have been to translate to CNF and run a CDCL solver, then the conflict speed, and hence the search speed, is orders of magnitude faster in RoundingSat than in Sat4j [EN18]. For crafted benchmarks where pseudo-Boolean reasoning appears to be crucial the conflict speed goes down significantly, but the performance is still much better than for pseudo-Boolean solvers using saturation [EGNV18]. One extra bonus is that the frequent use of division helps keep the coefficients small, so that one can use fixed-precision arithmetic (this of course also needs that one has to handle overflow, which is an issue we ignore in the pseudocode presented here). However, the main problem we identified with saturation-based solvers still remains: for CNF inputs, the algorithm still degenerates to a CDCL search for resolution proofs.

Before wrapping up our discussion of pseudo-Boolean solving, we wish to mention some other reasoning rules that are relevant to consider in this context. A natural question to ask is whether general linear constraints are needed to harness the full power of pseudo-Boolean solvers, or whether they could equally well work with a more limited set of constraints. One particularly interesting class of linear inequalities are cardinality constraints (1.21a). Given a PB constraint

$$3x_1 + 2x_2 + x_3 + x_4 \geq 4,$$  \hspace{1cm} (1.46a)

one can compute the least number of literals that have to be true, which results in the constraint

$$x_1 + x_2 + x_3 + x_4 \geq 2.$$  \hspace{1cm} (1.46b)

This is used in the solver Galena [CK05], which only learns cardinality constraints. The fact that all learned constraints will be of a particular form can also
make other aspects of the algorithm easier. Formally, this cardinality constraint reduction rule can be written as

$$\sum_{i \in [n], \sigma \in \{0,1\}} a^\sigma_i x^\sigma_i \geq A$$

$$T = \min\{|I| : I \subseteq [n], \sum_{i \in I, \sigma} a^\sigma_i \geq A\}.$$  \hfill (1.47)

Another interesting rule is strengthening, which we also introduce by giving an example. Suppose we have the PB formula

$$(x + y \geq 1) \land (x + z \geq 1) \land (y + z \geq 1)$$ \hfill (1.48a)

and that we set $x \leftarrow 0$ and propagate. This yields $y \leftarrow 1$ and $z \leftarrow 1$, meaning that the final constraint $y + z \geq 1$ is "oversatisfied" by a margin of 1. A moment of thought reveals that from this we can deduce

$$x + y + z \geq 2$$ \hfill (1.48b)

since either $x$ is true, in which the constraint certainly holds, or else $y + z \geq 1$ is oversatisfied. Slightly more formally, the strengthening rule, which seems to have been imported by [DG02] from operations research, can be described as follows:

- Suppose that assigning $x^\sigma_j = 0$ implies that the constraint

$$\sum_{i \neq j, \sigma} a_i x^\sigma_i \geq A$$

has to be oversatisfied by an amount of $K$.

- Then it is sound to deduce the constraint

$$K x^\sigma_j + \sum_{i \neq j, \sigma} a_i x^\sigma_i \geq A + K$$ \hfill (1.49)

In theory, using strengthening can allow the solver to recover from bad encodings such as CNF (in our example, we recovered the cardinality constraint (1.48a) from the CNF encoding (1.48a)). In practice, however, it seems hard to get this to work in an efficient way.

A final interesting scenario that we want to discuss is the following. Suppose we have a PB formula

$$(2x + 3y + 2z + w \geq 3) \land (2\overline{x} + 3y + 2z + w \geq 3)$$ \hfill (1.50a)

Then by eyeballing we can conclude that

$$3y + 2z + w \geq 3$$ \hfill (1.50b)

must hold, since $x$ is either true or false. But an application of the generalized resolution rule instead results in the constraint

$$6y + 4z + 2w \geq 4,$$ \hfill (1.50c)

reflecting that this rules also takes the possibility that $x = \frac{1}{2}$ into consideration. Applying saturation to (1.50c) yields

$$4y + 4z + 2w \geq 4$$ \hfill (1.50d)
and division does not help either since it yields the equivalent constraint

\[ 2y + 2z + w \geq 2, \]

which is clearly weaker than (1.50b). As observed by [Gec17], it would be quite convenient to have an implementation of what we can call a "fusion resolution" rule

\[
\frac{a_j x_j + \sum_{i \neq j, \sigma} b_i^\sigma x_i^\sigma \geq B}{\sum_{i \neq j, \sigma} b_i^\sigma x_i^\sigma \geq \min\{B, B'\}}.
\]

The need for such a rule shows up in some tricky benchmarks in [EGNV18], but there is no obvious way for cutting planes to perform such reasoning in an efficient way.

We conclude our review of conflict-driven pseudo-Boolean solving with a discussion of some of the challenges that lie ahead. On the theory side, one challenge is that there seem to be many more degrees of freedom in PB solving compared to CDCL. As we have seen above, there are several different ways of generalizing CDCL to a pseudo-Boolean context, and it seems far from obvious what is the best way to do so.

One interesting question in the algorithms we have discussed so far is how much the reason should be weakened in the reduction step. Is it better to weaken iteratively, literal by literal, until the first point in time when the resolvent is conflicting? Or is it better to do as in RoundingSat and do all the weakening right away? Another intriguing difference from CDCL is that sometimes the slack on the conflict side can be so negative that it is possible to just skip a resolution step and still maintain that the conflict-side constraint when the last propagated literal is removed from the trail (so that its status changes from falsified to non-falsified and it starts contributing to the slack). In such a scenario, is it better to skip the resolution step, in order to get fewer applications of the resolution rule over all, and get a more "compact explanation" of the conflict, or is it preferable to always resolve? It seems that one can cook up crafted benchmarks supporting both approaches.

This leads to the more general question of whether there is a better approach for conflict analysis than the generating (the analogue of) trivial resolution derivations. Note that this question also makes sense for CDCL. The main reason in favour of trivial resolution seems to be that it is simple and runs very fast. But perhaps it could sometimes pay off to be slower and do something smarter? Or, in the opposite direction, could it be that one should not try to learn general pseudo-Boolean constraints, as described above, but instead focus on a more limited form such as cardinality constraints, as done in Galena?

One reason that we do not have any good answer to these questions is that we do not even know much about how the different subsystems of cutting planes described towards the end of Section 1.6.1 relate to each other in terms of theoretical strength. Some limited progress has been made in recent papers such as [VEG18, GNY19], but many open problems still remain. A particularly intriguing question is whether cutting planes with division is stronger than cutting planes with saturation when the linear combination rule is limited to generalized resolution. We will discuss this further in Section 1.8.3.
Chapter 1. Proof Complexity

Among the implementation challenges, one of the most important ones is how to achieve efficient propagation detection for PB constraints. For CDCL solvers, a simple but crucial observation is that as long as a disjunctive clause contains two non-falsified literals it cannot propagate, and the famous “watched literals” scheme implementing this approach in state-of-the-art solvers is an important part of the explanation how such solvers can run so fast. This is not true in pseudo-Boolean solving—for a constraint like $\sum_{i=1}^{n} x_i \geq n - 1$ one has to watch all variables in order to detect propagation, since as soon as any one variable is set to false all others should propagate to true.

A second major challenge is if and how natural and efficient methods can be designed to recover from bad encodings (such as CNF). Although it is hard to make anything like this into a formal claim, it seems that PB solvers are more sensitive to the exact encoding of a problem, and also to the presence or absence of extra, potentially superfluous constraints, than CDCL solvers.

Another interesting question is how far the solver should backjump once an asserting constraint has been learned. In contrast to CDCL, a learned constraint can be asserting at several levels, and there are even different options in how to define what an asserting constraint should be depending on whether one wants to apply syntactic or (potentially harder to check) semantic criteria. A somewhat related concern is how to assess the “quality” of different constraints, for instance, when the solver has the choice of either performing or skipping a resolution step, and one would like to know which of the two options seems to yield the better constraint.

An intriguing observation, made in, e.g., [EGNV18], is that sometimes pseudo-Boolean solvers perform extremely poorly on instances which are infeasible even viewed as linear programs without any integrality constraints. Such instances can be trivial for mixed integer linear programming solvers such as Gurobi [Gur], CPLEX [CPL], and SCIP [SCI], which will detect infeasibility after solving the LP at the root node, while the pseudo-Boolean solvers get completely lost looking for satisfying Boolean assignments in a search space where there are not even any real-valued solutions. On the other hand, it seems that sometimes MIP solvers can fail badly on instances where learning from failed partial assignments PB constraints appears to be crucial (and where, for this reason, conflict-driven PB solvers can shine). It seems like a very tempting proposition to borrow techniques from, or merge techniques with, MIP solvers to obtain pseudo-Boolean solvers that could provide the best of both worlds.

In order to further develop pseudo-Boolean solving techniques, however, a final challenge that we wish to highlight is that it would be very desirable to develop efficient and concise PB proof logging techniques. Although DRAT could be used to log pseudo-Boolean proofs in theory, in practice the masssive overhead makes this virtually impossible (at least with current techniques), and it seems that the focus on logging only disjunctive clauses mixes very poorly with what is arguably the main advantage of pseudo-Boolean solvers, namely the stronger format of their constraints.
1.7. Brief Detour: Lifted CNF Formulas

We now want to proceed to discussing what is known about cutting planes from a proof complexity point of view. Before doing so in Section 1.8, however, it will be helpful to describe a particular way of constructing crafted benchmarks that has played a crucial role in many of the recent advances, namely lifting. This is the purpose of this quick interlude.

Let us start by discussing how to lift functions, and then explain how this idea extends to CNF formulas. The formal definition of lifted formulas seems to have appeared first in [BHP10], though our discussion below relies heavily on the later paper [HN12].

The idea behind lifting of functions is that we can take a base function \( f \) over some domain and extend it to a function over tuples from the same domain by combining it with an indexing or selector function that determines on which coordinates from the tuples \( f \) should be evaluated. More formally, given a positive integer \( \ell \geq 2 \) and a function \( f : \{0,1\}^m \rightarrow \mathbb{Q} \) for some range \( \mathbb{Q} \), the lift of length \( \ell \) of \( f \) is the function \( \text{Lift}_\ell(f) : \{0,1\}^m \times [\ell]^m \rightarrow \mathbb{Q} \) such that for any bit-vector \( \{x_{i,j}\}_{i \in [m], j \in [\ell]} \) and any \( y \in [\ell]^m \) the value of the lifted function is

\[
\text{Lift}_\ell(f)(x,y) = f(x_{y_1,1}, x_{y_2,2}, \ldots, x_{y_m,m}).
\]

In words, the \( y \)-vector selects which coordinates of the \( x \)-vector should be fed to \( f \) (this is illustrated in Figure 1.16). We refer to the coordinates of the \( y \)-vector as selector variables and the coordinates of the \( x \)-vector as main variables, and we write

\[
\text{select}_y(x) = (x_{y_1,1}, x_{y_2,2}, \ldots, x_{y_m,m})
\]

to denote the substring of \( x \) selected by \( y \).

We next extend this definition from functions to relations, or search problems. To this end, let \( S \) be any search problem defined as a subset of \( \{0,1\}^m \times Q \); that is, on any input \( a \in \{0,1\}^m \), the problem is to find some \( q \in Q \) such that \((a,q) \in S\). Then we define the lift of length \( \ell \) of \( S \) to be a new search problem

![Figure 1.16: Illustration of lifted function (for \( \ell = 3, m = 4 \), and \( y = (3,1,2,2) \)).](image-url)
Lift_ℓ(S) ⊆ {0,1}^{m×ℓ} × [ℓ]^m × Q with input domain {0,1}^{m×ℓ} × [ℓ]^m and output range Q such that for any bit-vector \{x_{i,j}\}_{i,j} ∈ [n], any y ∈ [ℓ]^m, and any q ∈ Q, it holds that

\[(x, y, q) ∈ Lift_ℓ(S) \text{ if and only if } \left(\text{select}_y(x), q\right) ∈ S\]  \hspace{1cm} (1.54)

The key behind several recent results establishing lower bounds for the cutting plane proof system has been to study lifted search problems defined in terms of CNF formulas using tools from the area of communication complexity [KN97]. Such lifted search problems are not themselves CNF formulas syntactically speaking, however, and therefore an additional step is needed where these lifted search problems are encoded back into CNF. Such lifted CNF formulas, as first introduced in [BHP10], can be constructed in the following way.

**Definition 1.7.1 (Lift of CNF formula [BHP10]).** Given any CNF formula \(F\) with clauses \(C_1, \ldots, C_m\) over variables \(u_1, \ldots, u_n\), and any positive integer \(ℓ ≥ 2\), the lift of length \(ℓ\) of \(F\) is a CNF formula Lift_ℓ(F) over \(2m\ell\) variables denoted by \(\{x_{i,j}\}_{i,j} ∈ [n], j ∈ [ℓ] \) (main variables) and \(\{y_{i,j}\}_{i,j} ∈ [n], j ∈ [ℓ] \) (selector variables), consisting of the following clauses:

- For every \(i ∈ [n]\), an auxiliary clause

  \[y_{i,1} ∨ y_{i,2} ∨ \cdots ∨ y_{i,ℓ} \hspace{1cm} (1.55)\]

  (where we will refer to \(y_{i,1}, y_{i,2}, \ldots, y_{i,ℓ}\) as a block of selector variables).

- For every clause \(C_i ∈ F\), where \(C_i = u_{i_1} ∨ \cdots ∨ u_{i_s} ∨ \overline{y_{i_{s+1}}} ∨ \cdots ∨ \overline{y_{i_{s+t}}}\) for some \(i_1, \ldots, i_{s+t} ∈ [n]\), and for every tuple \((j_1, \ldots, j_{s+t}) ∈ [ℓ]^{s+t}\), a main clause

  \[\overline{y}_{i_{1,j_1}} ∨ x_{i_{1,j_1}} ∨ \cdots ∨ \overline{y}_{i_{s,j_s}} ∨ x_{i_{s,j_s}} ∨ \overline{y}_{i_{s+1,j_{s+1}}} ∨ \overline{y}_{i_{s+1,j_{s+1}}} ∨ x_{i_{s+1,j_{s+1}}} ∨ \cdots ∨ x_{i_{s+t,j_{s+t}}} \hspace{1cm} (1.56)\]

  (where we will refer to \(C_i\) as the original clause corresponding to the lifted clause in (1.56)).

Let us try to decipher what the notation in Definition 1.7.1 means. The purpose of the auxiliary clauses in (1.55) is to make sure that in every variable subset \(\{y_{i,j} \mid 1 ≤ j ≤ ℓ\}\) at least one of the variables is true. We can think of the selector variables as encoding the vector \(y ∈ [ℓ]^m\) in the lifted search problem above. Since every pair \(y_{i,j} ∨ x_{i,j}\) in a main clause (1.56) is equivalent to an implication \(y_{i,j} → x_{i,j}\), we can rewrite (1.56) as

\[(y_{i_{1,j_1}} → x_{i_{1,j_1}}) ∨ \cdots ∨ (y_{i_{s+t,j_{s+t}}} → x_{i_{s+t,j_{s+t}}}) \hspace{1cm} (1.57)\]

Now we can see that for every clause \(C_i\), the auxiliary clauses encode that there is some choice of selector variables \(y_{i,j}\) which are all true, and for this choice of selector variables the \(x_{i,j}\)-variables in the lifted clause give us back the original clause \(C_i\). It is easily verified that \(F\) is unsatisfiable if and only if \(G = Lift_ℓ(F)\) is unsatisfiable, and that if \(F\) is a \(k\)-CNF formula with \(m\) clauses, then \(G\) is a max(\(2k, ℓ\))-CNF formula with at most \(mℓ^k + n\) clauses. In Figure 1.17 we show the
Chapter 1. Proof Complexity

1.8. Cutting Planes and Proof Complexity

In Section 1.6 we defined different flavours of the cutting planes proof system, and then showed how the conflict-driven SAT solving framework could be lifted from clauses to general 0-1 integer linear constraints. We now discuss what is known about cutting planes from the point of view of proof complexity.

1.8.1. Cutting Planes Size, Length, and Space

The length of a cutting planes refutation is the total number of lines/inequalities in it, and the size also sums the sizes of all coefficients (i.e., the bit size of representing them). The natural generalization of clause space in resolution is to...
define cutting planes (line) space as the maximal number of linear inequalities needed in memory during a refutation, since every clause is translated into a linear inequality. There is no useful analogue of width/degree known for cutting planes\(^{24}\).

Cutting planes can simulate resolution efficiently with respect to length/size and space simultaneously by mimicking the resolution steps one by one, and hence just as was the case for polynomial calculus we get the same worst-case upper bounds.

Cutting planes is strictly stronger than resolution with respect to length and size, since it can refute PHP formulas (1.6a)–(1.6b) efficiently [CCT87]. The reason for this is that in contrast to resolution (and polynomial calculus), cutting planes can count. PHP formulas are refuted simply by summing up the number of pigeons and holes, after which the observation can immediately be made that there are too many pigeons to fit into the holes. Cutting planes and polynomial calculus are incomparable with respect to size, i.e., for both proof systems one can find hard formulas that are easy for the other system. PHP formulas are an example of formulas that are hard for polynomial calculus but easy for cutting planes. In the other direction, very recently it was shown in [GKRS19] that there are formulas that are easy for polynomial calculus (and even Nullstellensatz) over any field, but are hard for cutting planes. These formulas are obtained by lifting as in Section 1.7 but the construction is a bit too involved to describe here.

The length measure in cutting planes does not consider the size of the coefficients. It is natural to ask if, and if so how, the power of cutting planes changes when coefficients are required to be of limited size. In [BC96] it was shown that the size of the coefficients need not be larger than of exponential magnitude if one is willing to tolerate a possible polynomial blow-up in proof length. One can define a subsystem of cutting planes where all inequalities in the proofs have to have coefficients of at most polynomial magnitude measure in the input size (i.e., coefficients should be representable with a logarithmic number of bits), and this subsystem is sometimes denoted \(\text{CP}^*\) in the literature. Understanding the power of \(\text{CP}^*\) remains wide open.

**Open Problem 1.7.** Decide whether cutting planes with coefficients of polynomial magnitude can simulate general cutting planes with at most a polynomial blow-up in proof length, or whether there are formulas for which cutting planes with unbounded coefficients is superpolynomially stronger.

When it comes to space, cutting planes is very much stronger than both resolution and polynomial calculus — it was shown in [GPT15] that any unsatisfiable CNF formula (and, in fact, any set of inconsistent 0-1 linear inequalities) can be refuted in constant line space 5 by cutting planes\(^{25}\). This proof works by starting with a linear inequality/hyperplane that cuts away the all-zero point of the Boolean hypercube \(\{0, 1\}^n\) from the candidate list of satisfying assignments (there has to exist a clause falsified by this assignment, from which the hyperplane

\(^{24}\)That is, one can certainly define width measures, but no such measure is known to have any interesting relation to other complexity measures for cutting planes.

\(^{25}\)Recall that this means that the formula is kept on a read-only input tape, and the working memory never contain more than 5 inequalities at any given time.
can be obtained), and then uses 4 auxiliary hyperplanes to remove further points α ∈ {0, 1}^n one by one in lexicographical order until all possible assignments have been eliminated, showing that the formulas is unsatisfiable. During the course of this refutation the size of the coefficients of the hyperplanes become exponentially large, however, which the line space measure does not charge for. A very interesting question is what happens if coefficients are limited to be of polynomial magnitude, i.e., if we consider the space complexity of CP* refutations.

**Open Problem 1.8.** Determine whether CP* is as strong as general cutting planes with respect to space, or whether there exist families of formulas that require superconstant space in CP*.

This question is completely open, and it cannot currently be ruled out that line space 5 would be sufficient. All that is known is that if we restrict the cutting planes proofs to have coefficients of at most constant size, then there are formulas that require Ω(\log \log \log n) space [GPT15].

If also coefficient sizes are counted, i.e., if one measures the total space of cutting planes refutations, then it is not hard to show a linear lower bound (for instance by combining [BW01] and [BNT13]) and a quadratic worst-case upper bound is immediately implied by resolution. For resolution this quadratic upper bound is known to be tight by [BGT14], but to the best of our knowledge no superlinear lower bounds are known on total space in cutting planes.

Proving space lower bounds, if they exist, seems challenging, however. It might be worth noting in this context that already cutting planes with coefficients of absolute size 2 (which is the minimum needed to simulate resolution) is quite powerful — this is sufficient to construct space-efficient refutations of PHP formulas [GPT15] (when space is measured as the number of inequalities in memory).

For a long time, essentially the only formulas that were known to be hard for the cutting planes proof system with respect to length/size were the **clique-coclique formulas** claiming (the negation of) that “a graph with an m-clique cannot be (m − 1)-colourable.” The formula consists of clauses:

\[
\begin{align*}
q_{k,1} \lor q_{k,2} \lor \cdots \lor q_{k,n} & \quad \text{[some vertex is the kth member of the clique]} \quad (1.58a) \\
\overline{q}_{k,i} \lor q_{k',i} & \quad \text{[clique members are uniquely defined]} \quad (1.58b) \\
p_{i,j} \lor \overline{q}_{k,j} \lor \overline{q}_{k',j} & \quad \text{[clique members are neighbours]} \quad (1.58c) \\
r_{i,1} \lor r_{i,2} \lor \cdots \lor r_{i,m-1} & \quad \text{[every vertex i has a colour]} \quad (1.58d) \\
p_{i,j} \lor r_{i,\ell} \lor \overline{r}_{j,\ell} & \quad \text{[neighbours have distinct colours]} \quad (1.58e)
\end{align*}
\]

where variables \(p_{i,j}\) are indicators of the edges in an n-vertex graph, variables \(q_{k,i}\) identify the members of an m-clique in the graph, and variables \(r_{i,\ell}\) specify a colouring of the vertices, for indices ranging over \(1 \leq i \neq j \leq n, i < j, 1 \leq k \neq k' \leq m,\) and \(1 \leq \ell \leq m - 1.\)

Pudlák [Pud97] proved that these formulas are hard by using a so-called **interpolation** argument, specifically tailored to work for formulas with the right structure. He showed that from any short cutting planes refutation of the formula one can extract a small monotone circuit for clique, which reduces to problem to
a question about size lower bounds for monotone circuits. (For completeness, we mention that essentially the same techniques were used in [HC99] to obtain exponential lower bounds for so-called broken mosquito screen formulas, but due to space constraints we will not discuss these formulas further here.)

It seems plausible that the Tseitin formulas in Figure 1.5 should require long cutting planes refutations, since it should be hard to count mod 2 using linear inequalities. It also seems very likely that random \(k\)-CNF formulas should be exponentially hard, both both of these problems are longstanding open questions in proof complexity.

**Open Problem 1.9.** Prove length lower bounds for cutting planes refutations of Tseitin formulas or random \(k\)-CNF formulas for constant \(k \geq 3\).

The last couple of years have seen some very exciting progress on cutting planes lower bounds. In a breakthrough result, exponential length lower bounds for random CNF formulas of logarithmic width were obtained in [HP17, FPPR17]. Unfortunately, the techniques used in these papers currently seem impossible to apply to formulas of constant width. Another intriguing result established in [GGKS18] is that if one starts with \(k\)-CNF formulas (for \(k = O(1)\)) that require large resolution with, and then apply lifting as described in Section 1.7, then this yields formulas which are weakly exponentially hard for cutting planes.

### 1.8.2. Size-Space Trade-offs for Cutting Planes

Given our very limited understanding of cutting planes, it is perhaps not so surprising that not very much has known about size-space trade-offs for this proof system until quite recently.

[GP18] showed that short cutting planes refutations of Tseitin formulas on expanders must have large space, but this does not provide a real trade-off since it seems likely that such short refutations do not exist at all, regardless of their space complexity. Earlier, [HN12] proved that short cutting planes refutations of one particular version of pebbling contradictions (slightly different from the substituted pebbling contradictions discussed in Section 1.4.3) over one particular family of DAGs require large space — a result that was strengthened and generalized by [GP18] — and for pebbling contradictions such short refutations do exist. Interestingly, and somewhat unexpectedly, all of these results follow from reductions to communication complexity. The state of knowledge regarding pebbling contradictions is much worse here than for resolution and polynomial calculus, however — for the latter two proof systems we know of general methods to translate pebbling trade-offs for (essentially) arbitrary graphs into proof complexity size-space trade-offs.

Since [GPT15] established that any unsatisfiable CNF formula has a constant-space refutation, the lower bounds for pebbling contradictions in [HN12, GP18] yield true size-space trade-off results for cutting planes, with formulas that can be refuted in both small size and small space, but where optimizing both measures simultaneously is impossible. However, the “space-efficient” refutations have coefficients of exponential size. It would be more convincing to obtain trade-offs where the small-space refutations also have small coefficients (which would follow
if known resolution and polynomial calculus results for pebbling contradictions or Tseitin formulas over long, skinny grids could be lifted also to cutting planes).

The rest of Section 1.8 from here onwards is in need of a major update and re-write, and is therefore not quite ready for reviewing.

Open Problem 1.10. Are there trade-offs where the space-efficient CP refutations have small coefficients (say, of polynomial or even constant size)?

1.8.3. Subsystems of Cutting Planes

General cutting planes is implicationally complete, meaning that if a set of linear inequalities imply some other linear inequality, then the latter inequality can also be derived syntactically. This is not true for the subsystems of cutting planes that we consider.

Probably already since \cite{Hoo88, Hoo92} it was known that CP with saturation and resolution collapses to resolution when the input is presented in CNF. It is not hard to show that this holds also if saturation is replaced by division (while restricting linear combinations to general resolution). Even cutting planes with saturation and unrestricted linear combinations is polynomially equivalent to resolution if coefficients are restricted to polynomial magnitude \cite{VEG18}.

In a pseudo-Boolean solving context it is natural that linear combinations are restricted to generalized resolution (such derivation steps arise naturally during conflict analysis, and it is hard to see how to devise good heuristics for non-cancelling linear combinations, although it would be very nice if this could be done).

An obvious question is how the division and saturation rules compare to each other if CP is restricted to generalized resolution. In recent work, we have the following two results indicating that using division or saturation leads to incomparable proof systems.

In one direction, there are pseudo-Boolean formulas which have linear-length refutations in CP with division and resolution but require exponential-length refutations in CP with saturation and unrestricted linear combinations.

In the other direction, simulating one generalized resolution step followed by a saturation step can take an unbounded number of steps in general cutting planes with division and unrestricted linear combinations. This does not show that CP with saturation can be superpolynomially stronger than CP with division, but it does show that if CP with division and resolution polynomially simulates CP with saturation and resolution, then such a simulation is unlikely to proceed line-by-line and instead has to work by some kind of global argument.

In more detail, let $R$ be a positive integer and consider the two constraints

$$Rx + Ry + \sum_{i=1}^{R} z_i \geq R$$

(1.59)
Chapter 1. Proof Complexity

and

\[ Rx + R\bar{y} + \sum_{i=R+1}^{2R} z_i \geq R . \] (1.60)

Resolving these two constraints yields

\[ 2Rx + \sum_{i=1}^{2R} z_i \geq R , \] (1.61)

which after saturation becomes

\[ Rx + \sum_{i=1}^{2R} z_i \geq R . \] (1.62)

However, deriving (1.62) from (1.59) and (1.60) in general cutting planes (with division and arbitrary linear combinations) requires \( \Omega(\sqrt{R}) \) applications of the division rule.

The precise relation between general cutting planes and \( CP^* \) remains open, but [dRMN+19] showed what to the best of my knowledge is the first separation between \( CP \) and \( CP^* \) in the sense of a computational task which \( CP \) provably performs more efficiently than \( CP^* \). It is shown in [dRMN+19] that there are formulas which exhibit strong length-space trade-offs for \( CP^* \) but not for general \( CP \). In more detail, there is a family of CNF formulas \( \{ F_n \}_{n=1}^{\infty} \) of size \( \Theta(n) \) such that:

- \( CP^* \) can refute \( F_n \) in length \( O(n) \) (in fact, even resolution can do this).
- \( CP \) can refute \( F_n \) in length \( O(n \log \log n) \) and line space \( O(1) \) simultaneously.
- For any \( CP^* \) refutation in length \( L \) and line space \( s \) it must hold that

\[ s \cdot \log L = \Omega(n / \log^2 n) . \] (1.63)

In order to make this into a “true trade-off” in the sense of [HN12, dRNV16], we would also like to show that \( CP^* \) can refute these formulas in small space. This is not known, however, and in fact it is conceivable, or even seems likely, that any \( CP^* \) refutation, regardless of the length, would require line space \( \Omega(n / \log^2 n) \). Proving such a space lower bound would seem to require entirely new techniques, however.

As it turns out, one problem with current pseudo-Boolean solvers is that if they get their input in CNF, they cannot even go beyond resolution. Solvers such as Sat4j [LP10] solve PHP formulas very efficiently, but they crucially depend on the input being given as linear inequalities:

\[ p_{i,1} + p_{i,2} + \cdots + p_{i,n} \geq 1 \quad \text{[every pigeon } i \text{ gets a hole]} \] (1.64a)

\[ \overline{p}_{1,j} + \overline{p}_{2,j} + \cdots + \overline{p}_{n+1,j} \geq n \quad \text{[no hole } j \text{ gets two pigeons]} \] (1.64b)

If the input is instead presented in CNF, with the cardinality constraints in Equation (1.64b) encoded as the clauses in Equation (1.6b), then Sat4j runs in exponential time. The same holds for subset cardinality formulas — if a pseudo-Boolean
Chapter 1. Proof Complexity

(a) Eulerian graph. (b) CNF instance. (c) PB instance.

Figure 1.18: Example of even colouring (EC) formula (satisfiable instance).

solver is fed the formula encoded as cardinality constraints, then it runs fast, but on the CNF version in Figure 1.6b it cannot possibly do better than the exponential lower bound on resolution length in [MN14].

Thus, an algorithmic challenge is to make pseudo-Boolean solvers reason more efficiently with CNF inputs, so that they could, e.g., detect and use the cardinality constraints hidden in (1.6a)–(1.6b) to get performance comparable to when the input is given as (1.64a)–(1.64b). It is possible to do a preprocessing step to recover cardinality constraints encoded in CNF, and for PHP formulas and subset cardinality formulas this works well [BLLM14], but full preprocessing of the input to try to detect cardinality constraints is probably not an efficient approach in general.

This is not the only challenge for pseudo-Boolean solvers, however. Another quite intriguing family of benchmark formulas in this context are the even colouring (EC) formulas constructed by [Mar06] and shown in Figure 1.18. Here one starts with a connected graph $G$ having an Eulerian cycle, i.e., with all vertex degrees even, and writes down constraints that edges should be labelled 0/1 in such a way that for every vertex $v$ in $G$ the number of 0-edges and 1-edges incident to $v$ is equal. If the total number of edges in the graph is even, then this formula is satisfiable — just fix any Eulerian cycle and label every second edge 0 and 1, respectively. If the number of edges is odd, however, then cutting planes can sum the at-least-2 constraints in Figure 1.18c (the ones with positive literals) over all vertices to derive $2 \cdot \sum_{e \in E(G)} e \geq |E(G)|$ and then divide by 2 and round up to obtain $\sum_{e \in E(G)} e \geq (|E(G)| + 1)/2$. By instead summing up all at-most-2 constraints (the ones with negated literals) and dividing by 2 one obtains $\sum_{e \in E(G)} e \leq (|E(G)| - 1)/2$, and subtracting these two inequalities yields $0 \geq 1$.

One interesting aspect to observe here is that in contrast to PHP and subset cardinality formulas, the above argument uses crucially that variables are integer-valued. To see the difference, suppose that we are given a PHP or subset cardinality formula encoded as linear constraints. Then for cutting planes
it is sufficient to simply add up the inequalities to derive a contradiction. No integer-based reasoning is needed. Even if we allow putting fractional pigeons into fractional holes, there is no way one can make a pigeon mass of $n + 1$ fit into holes of total capacity $n$. This set of linear inequalities is unsatisfiable even over the rationals, i.e., the polytope defined by the constraints is empty. Similarly, for subset cardinality formulas there is no way $4n + 1$ variables could have a total “true mass” of at least $2n + 1$ and a total “false mass” of $2n + 1$ simultaneously.

But for collections of linear constraints as in Figure 1.18c assigning all edges value $\frac{1}{2}$ is a satisfying fractional solution. The polytope defined by the linear inequalities is not empty, but it does not contain any integer points. Hence, refuting EC formulas in cutting planes crucially requires the division rule (1.22d), and pseudo-Boolean solvers need to implement this rule (or some other form of integer-based reasoning) to decide these formulas efficiently. Experiments in [EGNV18] indicate that EC formulas and some other crafted formulas are much harder for pseudo-Boolean solvers than the cutting planes upper bound would suggest, which seem to indicate that the solvers are still quite far from using the full power of cutting planes reasoning.

It is tempting to conjecture that EC formulas on, e.g., random 6-regular graphs with an odd number of vertices (so that the formulas are unsatisfiable) are exponentially hard for cutting planes with saturation, but proving such a lower bound is beyond current techniques.

1.9. Extended Resolution and DRAT proofs

Extended resolution (ER) was originally introduced by [Tse68a] to allow resolution proofs to work with formulas other than CNFs. The intuition is that an extended resolution refutation is allowed to infer a formula as follows:

$$x \leftrightarrow \varphi,$$

where $x$ is a new variable, and $\varphi$ is an arbitrary formula. The proviso that $x$ is new means that $x$ does not appear in any axiom, does not appear in $\varphi$, and does not appear earlier in the derivation. The extension rule can also be used in derivations, but then $x$ also must not appear in the formula being derived.

The extension rule (1.65) is stated in the form it will be used in extended Frege systems, which are defined in Section 1.10. Since resolution systems are constrained to work with clauses, extended resolution uses a restricted, clausal form of the extension rule. If $a$ and $b$ are literals, and $x$ is a new variable, then the extension rule allows inferring the three clauses

$$\overline{\varphi} \lor a,$n $$\overline{\varphi} \lor b,$n $$a \lor b \lor x$$

These clauses express that $x \leftrightarrow (a \land b)$. An extended resolution derivation is a derivation in which both resolution and extension rules are allowed.

Using (1.66) multiple times, with multiple new variables, allows effectively simulating the action of the full extension inference of (1.65). Consequently, extended resolution simulates (and is simulated by) the extended Frege proof system.
Chapter 1. Proof Complexity

It is almost immediate that adding the extension rule preserves soundness, since if there is a satisfying assignment to a set of clauses, the satisfying assignment can be extended to give the new variable $x$ the (unique) truth value which will satisfy (1.65) or (1.66). Therefore, extended resolution is sound and complete. However, resolution does not polynomially simulate extended resolution. Indeed there is an exponential separation between extended resolution. The classical example of this is the pigeonhole principle which was shown to have polynomial size extended resolution proofs by [CR79], but to require exponential size, $2^{\Omega(n)}$, resolution proofs by [Hak85].

The fact that extended resolution can be so much more powerful than resolution raises the question of whether practical, CDCL-based SAT solvers can incorporate the extension rule. If this could be done well, the gains could be enormous, as this would in principle give CDCL the power to refute CNFs using the full power of resolution. There have been a number of attempts to incorporate the extension rule into CDCL, but the published literature on this is fairly sparse, principally [SB06, AKS10, Hua10, MHB13] So far, the extension rule has been successful in limited situations; however, it has not been successful enough to be generally included in SAT solvers. The bottleneck appears to be that we have no good heuristics for how to choose extension formulas $\varphi$ for use in the extension rule.

In recent years, a new application for the extension rule has appeared, driven by the desire to have SAT solvers output proof traces. Section 1.3.1 discussed RUP proofs as proof traces. RUP proofs, however, are not powerful enough to handle all the preprocessing and inprocessing techniques used by modern SAT solvers. A more powerful “RAT” inference rule has been developed to supplant RUP. Since Deletion is also allowed in the proof trace, the system is called the DRAT proof system.

A RAT inference is defined as follows (see [HJW13b, HJW13a]). Let $\Gamma$ be the current set of clauses in the clause database. (Recall that typically the clause database is repeatedly updated by clause learning and clause deletion.) Let $C$ be a clause $a_1 \lor \cdots \lor a_k \lor b$: here it is permitted, but not required, that the literal $b$ is a new variable. Then $\Gamma$ may infer $C$ with a RAT inference provided that for every clause $D \lor b$ in $\Gamma$, the clause

$$a_1 \lor \cdots \lor a_k \lor D \lor b$$

is a RUP clause with respect to $\Gamma$.

The RAT inference is sound for refutations, but not for derivations. Thus the DRAT system is sound and complete as a refutation system. The critical property is that if $\Gamma$ can derive the clause $C$ with a RAT inference, then $\Gamma$ and $\Gamma \cup \{C\}$ are equisatisfiable; i.e., they are either both satisfiable or both unsatisfiable.

It is not hard to see that DRAT simulates the extension rule; indeed, the three clauses of the extension rule (1.66) can be added one at a time by RAT inferences. Conversely, extended resolution can simulate the RAT inference [KRPH18]. Therefore, the DRAT proof system and extended resolution polynomially simulate each other.

These simulations between DRAT and extended resolution work because the systems allow introducing arbitrary new variables. Thus, using DRAT for proof
search can suffer from the same difficulties as extended resolution; namely, we lack effective methods of deciding what properties the new variables should represent. A different, and very interesting, approach proposed by [HKB17, HKB19] is to consider DRAT without allowing new variables to be introduced. In recent work, it has been shown that many of the variants of DRAT are equivalent even when new variables are not allowed to be introduced (see [HB18, KRPH18, BT18].)

DRAT under the restriction of not introducing new variables is still an area of very active development. It is too new to be properly surveyed here, but recent developments include [HHJW15, HB18, HKB17, BT18] and works cited in those papers. An exciting aspect of DRAT (and related systems) when no new variables are permitted is that it is still a very strong system (see [HKB19, BT18]). [BT18] show that, even without allowing new variables or clause deletion, a modest extension of the RAT rule called subset propagation redundancy (SPR, for short, first introduced by [HKB19]) is powerful enough to give polynomial size proofs of almost all the known hard examples for constant depth Frege proof systems. [BT18] further show, when new variables are not allowed, there is an exponential separation in refutation size between the proof system RAT (the RAT rule without allowing clause deletion) and the proof system DRAT (the RAT rule combined with clause deletion). It is surprising that clause deletion would give this extra power, but this is because removing clauses permits additional RAT inferences.

Of particular note for practical applications of DRAT reasoning in proof search, is the work of [HKSB17] who showed how to use Satisfaction Driven Clause Learning (SDSL) to search for short DRAT refutations without new variables: remarkably they were able to automatically efficiently generate short refutations of the pigeonhole (PHP) clauses. This is noteworthy since it was accomplished in a very general way without explicit checking for cardinality constraints.

1.10. Frege proofs

Frege proofs are the standard “textbook” propositional proof systems. Often they are formulated with connectives \( \neg, \land, \lor \) and \( \rightarrow \), with a finite set of axiom schemes, and with modus ponens as the only rule of inference. As first proved by [CR79] (see also [Rec76]), there are a great many ways to define Frege proof systems, all of which are sound and complete and which polynomially simulate each other.

We will briefly present here a version of Frege proof systems based on the sequent calculus, with its language chosen to correspond closely to the CNF formulas used for resolution. An sequent is an expression of the form

\[
\phi_1, \phi_2, \ldots, \phi_k \Rightarrow \psi_1, \psi_2, \ldots, \psi_l,
\]

where the \( \phi_i \)'s and \( \psi_j \)'s are propositional formulas. The intended meaning of this sequent is that the conjunction of the formulas \( \phi_i \) on the left imply the disjunction of the formulas \( \psi_j \) on the right. Equivalently, the sequent \( 1.67 \) expresses that the disjunction of the \( \phi_i \)'s and the \( \neg \phi_i \)'s is true.

In the version of the sequent calculus, LK, used in this chapter, we not allow negations on arbitrary formulas. Instead, we define formulas inductively
as follows. First, for any variable, the variable and the negated literal are formulas. Secondly, if and are formulas, then their conjunction and their disjunction are formulas. Thus, formulas are formed starting with literals, and forming complex formulas with (binary) connectives “and” and “or”.

Our system has the following axioms and rules of inference. We use to denote multisets of formulas. More generally, in a sequent such as , the righthand and lefthand sides denote multisets of formulas. Letting denote the set of formulas in the multiset , the weakening rule below requires that and .

Initial sequents For any variable , there are three initial sequents (logical axioms):

(1.68)

Structural rules:

\[
\text{Weakening: } \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta^*} \quad \text{Cut: } \frac{\Gamma \Rightarrow \Delta, \varphi, \Gamma \Rightarrow \Delta^*}{\Gamma \Rightarrow \Delta^*}
\]

Logical rules:

\[
\text{and-right: } \frac{\Gamma \Rightarrow \Delta, \varphi, \Gamma \Rightarrow \Delta^*}{\Gamma \Rightarrow \Delta, \varphi \land \psi, \Gamma \Rightarrow \Delta^*}
\]

\[
\text{or-right: } \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \varphi \lor \psi, \Gamma \Rightarrow \Delta^*}
\]

Definition 1.10.1. Let be a formula. A Frege proof of is defined to be an LK proof of . An extended Frege proof is an LK proof which can use multiple extension rules as follows. There is a sequence of pairs for so that each is a formula and each extension variable does not appear in the conclusion of and does not appear in any for . Then the extended Frege proof may use the sequents

(1.69)

as initial sequents (axioms).

We can also use LK as a refutation system. For this, let be a set of sequents. An LK refutation may use both the logical axioms of (1.68) and sequents from as initial sequents and ends with the empty sequent . The initial sequents from are called nonlogical axioms. Since the empty sequent is false under any truth assignment, the LK refutation proves the unsatisfiability of the nonlogical axioms. Extended Frege (as formalized in Definition 1.10.1 as an LK system) is equivalent to extended resolution, in that the two systems simulate each other.

1.11. Constant-Depth Frege proofs

The depth of a formula is defined to equal the number of levels of disjunctions and conjunctions in . More formally, any literal, or is a depth 0 formula. And, if , , , , are all depth , and then any formula formed by
combining those \( k \) formulas with only conjunctions, or with only disjunctions, has depth \( \leq d \). For example \((x \lor (y \lor z)) \land u\) is depth 2.

Resolution can be viewed as a depth 0 LK refutation system. For this, if \( C \) is a clause containing the variables \( x_1, \ldots, x_k \) unnegated and the negated literals \( y_1, \ldots, y_k \), then the corresponding sequent \( S_C \) is \( y_1, \ldots, y_k \Rightarrow x_1, \ldots, x_k \).

Under this translation between clauses and sequents of literals, the resolution inference rule corresponds exactly to a cut inference in LK. In this way, resolution refutations are essentially identical to dag-like LK refutations. Furthermore, tree-like resolution refutations are essentially identical to tree-like LK systems.

**Definition 1.11.1.** Let \( d \geq 0 \). A depth \( d \) LK-proof is an LK proof in which every formula has depth \( \leq d \). We use “\( d \)-LK” to refer to the LK proof system restricted to depth \( d \) proofs.

Earlier we discussed exponential lower bounds on the size of resolution refutations and cutting planes proofs. We also have exponential lower bounds on the size of constant depth LK proofs. The first such bound \cite{Hak85} gave exponential lower bound on the size of resolution refutations for the pigeonhole principle (PHP). This was later extended to give exponential lower bounds on the size of constant depth LK proofs of the PHP tautologies by \cite{BIK92, PB93, KPW95} (improving on \cite{Ajt88}).

Recall the pigeonhole principles tautologies were defined in Equations (1.6a)-(1.6d) as a set of unsatisfiable clauses; for fixed numbers of pigeons \( m \) and holes \( n \), the clauses are denoted \( \text{PHP}_m^n \).

**Theorem 1.11.2.** \cite{BP96} Depth \( d \) LK refutations of the \( \text{PHP}^n_{n+1} \) clauses require size \( \Omega(2^{n^{51}/d}) \).

Other known hard tautologies for constant-depth LK include the counting-mod-\( p \) principles. When \( p = 2 \), this is the parity principle. We define here only the parity principle. Fix \( n > 0 \) an odd integer. The parity principle \( \text{PARITY}_n \) uses variables \( x_{i,j} \) for \( 1 \leq i < j \leq n \); for convenience of notation, we define \( x_{j,i} \) to be the same variable as \( x_{i,j} \). The clauses of \( \text{PARITY}_n \) consist of the \( n \) many totality clauses

\[
\bigvee_{j \neq i} x_{i,j} \quad \text{for } 1 \leq i \leq n,
\]

and the \( n \binom{n-1}{2} \) many clauses

\[
\forall_{i,j} \vee \forall_{i',j} \quad \text{for distinct } i, i', j \in [n].
\]

**Theorem 1.11.3.** \cite{BP96} Depth \( d \) LK refutations of the \( \text{PHP}^n_{n+1} \) clauses require size \( \Omega(2^{n^{61}/(d+1)}) \). (This is true even if arbitrary substitution instances of formulas expressing the unsatisfiability of \( \text{PHP}^n_{n+1} \) are permitted as additional hypotheses.)

The parity principle can be generalized to “counting mod \( p \)” principles for \( p > 2 \). An analogue of Theorem 1.11.3 holds and gives exponential lower bounds.
for general "counting mod \( p \)" principles, as proved by \[ \text{BIK}^{+97} \], building on work of \[ \text{BIK}^{+94} \] and \[ \text{Rii}97a, \text{Rii}97b \].

Theorems 1.11.2 and 1.11.3 are not the very strongest results known, but they are representative of the best known lower bounds on the size of constant depth LK proofs. Along with the earlier discussed lower bounds for the polynomial calculus and cutting planes, these are the current state-of-the-art in proving size lower bounds on propositional proof size. There are a large number of open problems remaining about the size of constant depth LK proofs.

**Open Problem 1.11.** Is there an exponential separation between the size of depth \( d \) LK refutations and depth \( d + 1 \) LK refutations of sets of clauses?

So far, only a superpolynomial separation is known for \( d \)-LK versus \((d+1)\)-LK proofs; this obtained using Theorem 1.11.2 and constructing low depth proofs of small instances pigeonhole principles using a technique of Nepomnjascij, see \[ \text{KI02} \] and \[ \text{PW85} \] for uniform version of this based on bounded arithmetic.

**Open Problem 1.12.** Let LK(\( \oplus \)) denote LK extended with a parity (exclusive-or) logical connective. Are there exponential, or even superpolynomial, lower bounds on the size of LK(\( \oplus \)) proofs? Can LK(\( \oplus \)) be simulated by constant-depth LK augmented with all instances of formulas expressing the unsatisfiability of the \( \text{PARITY}_n \) sets of clauses?

Open Problem 1.12 is related to the Nullstellensatz proof system. In fact, the study of counting principles and counting gates was the original impetus the development of the Nullstellensatz system \[ \text{BIK}^{+96} \].

Finally, it is beyond the scope of this survey, but the study of constant depth LK proofs is very closely related to first order theories of bounded arithmetic \( \text{T}_2 \). For more on bounded arithmetic see \[ \text{Bus86} \] or \[ \text{Kra95} \]. Provability in constant depth LK and provability in bounded arithmetic also have close connections to the theory of \( \text{Total NP Functions} \) (TFNP) as initiated by \[ \text{Pap94} \]. For instance, the PHP and parity principles are the basis of the TFNP classes PPP and PPA. There is extensive work in these areas related to propositional proof complexity in Frege and (apparently) stronger systems; recent work includes \[ \text{BB16, GP16} \].

### 1.12. Quantified Frege proofs

We now briefly discuss systems that are stronger than constant depth LK proof systems. This includes LK, which is a Frege proof system. It also includes extended Frege systems. It is trivial that extended Frege simulates Frege, as every Frege proof is already an extended Frege proof. The converse is open, and it is generally believed that Frege systems cannot simulate extended Frege systems. However, we are lacking good candidate examples; indeed, almost everything we know how to formalize in extended Frege, we can also formalize in Frege systems with only a polynomial size, or at least quasipolynomial size, increase in proof size (see \[ \text{BBP95, HT15, Bus15, ABB16} \]). The main exception is that Cook’s tautologies expressing the consistency of extended Frege proofs can be proved with polynomial size extended Frege proofs \[ \text{Coo75} \], but cannot be proved with polynomial size Frege proofs unless Frege proofs simulate extended Frege \[ \text{Bus91} \].
Quantified Frege proofs are conjectured to be even stronger than extended Frege proofs. There has been extensive research into QBF solvers that solve the consistency or validity question for quantified Boolean formulas (QBF’s). Other chapters in this handbook, [KBB09, GMN09], discuss the proof systems and proof search algorithms used by QBF solvers. We shall instead discuss theoretical aspects of quantified Frege proofs within the framework of LK proofs.

The syntax of quantified Boolean formulas builds on the syntax of formulas defined in Section 1.10. There are now two kinds of variables. The first kind is the free variables, denoted $x, y, z$; free variables are the usual propositional variables. The second kind is the bound variables, denoted $u, v$. Free variables may not be quantified, and thus always appear only freely. Bound variables must be quantified and may not appear free (unquantified) in a formula. When $\varphi$ is an expression, we write $\varphi(u/x)$ for the expression obtained by replacing every occurrence of $x$ in $\varphi$ with $u$.

To define quantified Boolean formulas (QBF’s) we first define semiformulas as follows: (a) if $x$ is a free variable then $x$ and $\overline{x}$ are semiformulas; (a) if $u$ is a bound variable then $u$ and $\overline{u}$ are semiformulas; (c) if $\varphi$ and $\psi$ are semiformulas, then so are $(\varphi \land \psi)$ and $(\varphi \lor \psi)$; and (d) if $\varphi$ is a formula, and $u$ is a bound variable, then $\forall u \varphi$ is a semiformula. An occurrence of a variable $u$ in a formula $\varphi$ is a bound occurrence if it is inside the scope of a quantifier $\forall u$ or $\exists u$; otherwise it is a free occurrence. A quantified Boolean formula (QBF) is a semiformula $\varphi$ in which there is no free occurrence of a bound variable.

If $\varphi$ is a formula (respectively a semiformula), $u$ is a bound variable, and $\psi$ is a formula, we write $\varphi(\psi/u)$ to denote the formula (respectively semiformula) obtained by replacing every free occurrence of $u$ in $\varphi$ with $\psi$.

The quantified sequent calculus, which we here denote QBF-LK, the sequents contain formulas (not semiformulas); QBF-LK has all the rules of inferences of LK, plus the following rules of inferences for introducing quantifiers. In the $\forall$-right and $\exists$-left rules, it is required that the eigenvariable $x$ does not appear in the lower sequent (the conclusion).

Quantifier rules:
\[
\frac{\varphi(\psi/u), \Gamma \Rightarrow \Delta}{\forall u \varphi, \Gamma \Rightarrow \Delta}, \quad \frac{\varphi(x/u), \Gamma \Rightarrow \Delta}{\forall \varphi, \Gamma \Rightarrow \Delta}\]
\[
\frac{\exists u \varphi, \Gamma \Rightarrow \Delta}{\exists \varphi(\psi/u), \Gamma \Rightarrow \Delta}, \quad \frac{\exists \varphi, \Gamma \Rightarrow \Delta}{\exists \varphi, \Gamma \Rightarrow \Delta, \exists u \varphi}\]

A QBF is called a $\Sigma_1$ if it contains only existential quantifiers (if any), and thus no universal quantifiers. A $\Sigma_1$-LK proof is a QBF proof in which all formulas are $\Sigma_1$. This system is equivalent in strength to extended Frege (and hence to extended resolution w.r.t. refuting sets of clauses. This theorem is due essentially to [CR79] and [KP90].

Theorem 1.12.1. The proof systems of $\Sigma_1$-LK and extended Frege are equivalent.

We give a quick proof sketch. Suppose an extended Frege $P$ of a sequent

\[\varphi \Rightarrow \psi, \Gamma \Rightarrow \Delta, \exists u \varphi\]

Similarly, if $P$ has an extended Frege derivation

\[\varphi \Rightarrow \psi, \Gamma \Rightarrow \Delta, \forall u \varphi\]


\[\text{The principal reason for having a distinction between free variables and bound variables is to have cleaner formalization of quantified sequent calculi, including a smooth treatment of cut elimination.}\]
Chapter 1. Proof Complexity

Γ ⇒ Δ uses finitely many extension axioms specified by the pairs \((x_i, \psi_i)\) for \(i = 1, \ldots, k\). It is not hard to see that there is then a polynomial size Frege proof of

\[
\{x_i ↔ \psi_i\}_{i=1}^k, \Gamma ⇒ \Delta.
\]  

(1.70)

There is also a short \(\Sigma_1\)-LK proof, using \(\exists\)-right inferences, of

\[
⇒ (\exists u_1 \ldots u_k) \left( \bigwedge_{i=1}^k (u_i ↔ \psi_i(\vec{u}/\vec{x})) \right).
\]

(1.71)

Combining this with (1.70) and \(\exists\)-left inferences gives a polynomial size \(\Sigma_1\)-LK proof of \(\Gamma ⇒ \Delta\).

Conversely, any \(\Sigma_1\)-LK proof of \(\Gamma ⇒ \Delta\) can be transformed into a polynomial size extended Frege proof. One way to prove this is to first show that Cook’s equational theory PV for polynomial time functions, or the conservative extension \(S^1_2\) ([Bus86]), can prove the consistency of \(\Sigma_1\)-LK proofs, and second invoke a fundamental theorem ([Coo75]) about PV that extended Frege simulates any theory with PV can prove consistent. The first part of the argument uses a witnessing proof to show that there is a polynomial time algorithm that maps Boolean values for the (existentially) quantified variables of \(\Gamma\) to Boolean values for the (existentially) quantified variables of \(\Delta\) so that if \(\Gamma\) is satisfied, then \(\Delta\) is satisfied. It is also possible to give a direct construction of an LK proof, by introducing definitions for the existentially quantified variables of \(\Delta\) using polynomially many extension axioms.

Consequently, \(\Sigma_1\)-LK is already very powerful. The full QBF-LK system is conjectured to be yet more powerful. From the just-mentioned theorem of [Coo75], as in the proof of Theorem 1.12.1, extended Frege and \(\Sigma_1\)-LK are closely connected to first-order bounded arithmetic theories for polynomial time. QBF-LK on the other hand is closely connected to bounded arithmetic theories of polynomial space [Dow78].

1.13. Concluding Remarks

This section has not been changed from the SIGLOG survey [Nor15] and so is thoroughly obsolete. It is in need of a complete rewrite.

In this section, we should comment briefly on topics that we do not cover, such as:

- Sherali-Adams, Lasserre/SOS/Positivstellensatz, Lovász-Schrijver, …
- Ideal proof systems
- Resolution over parities
- Proof systems with inferences verifiable by efficient communication protocols?
- Et cetera

In this paper, we have presented an overview of proof complexity with a focus on connections with SAT solving. The discussion has intentionally been kept at a high level, often with only informal statements of results. For many of the proof
complexity results mentioned in this paper it is possible to find exact, formal statements in the survey paper [Nor13]. This is no longer true — [Nor13] is too much out of date. Consider mentioning again the surveys listed in the introduction, and also Krajíček’s book [Kra19].

On the proof complexity side, the main take-away message is that resolution is fairly well understood, although there are still some interesting open questions left (which we mostly did not discuss). For polynomial calculus we also have a fair amount of knowledge, although there are many more open problems than for resolution. For instance, the techniques for proving degree lower bounds (and hence size lower bounds) are not yet very well developed, and the hardness status of several interesting formula families remains open. Also, we do not understand very well the relations between degree and monomial space. For cutting planes much less is known, and any progress on the open problems listed in this survey would be very exciting.

When it comes to applied SAT solving, we still have quite a poor understanding of why different formulas are easy or hard. It would be interesting to investigate further whether there could be any relevant connections here between proof complexity measures and hardness of SAT, or whether tools and techniques from proof complexity could help to shed light on the inner workings on SAT solvers (perhaps going further, and deeper, along the lines of [EGG+18]).

Finally, the main algorithmic challenge we want to highlight is if and how one can build efficient SAT solvers based on stronger proof systems than resolution. Is it really the case that CDCL, originating in the DPLL method from the early 1960s [DP60, DLL62, Rob65], is the best conceivable paradigm? Or could it be possible that is it now time, over 50 years later, to take the next step and build fundamentally different SAT solvers based on algebraic and/or geometric methods? Are there perhaps fundamental limitations why efficient proof search cannot be implemented within these proof systems? Or could it be that a sustained long-term effort would yield powerful new SAT solving paradigms, just as the immense work spent on optimizing CDCL solvers over the years have led to improvements in performance of several orders of magnitude?

References


Chapter 1. Proof Complexity


[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses qual-
Chapter 1. Proof Complexity


Chapter 1. Proof Complexity

2009.


Chapter 1. Proof Complexity


Chapter 1. Proof Complexity


Chapter 1. Proof Complexity


[CS88] Vašek Chvátal and Endre Szemerédi. Many hard examples for res-


[dRMNR19] Susanna F. de Rezende, Or Meir, Jakob Nordström, and Robert


FLM+15 Yuval Filmus, Massimo Lauria, Mladen Mikša, Jakob Nordström, and Marc Vinyals. From small space to small width in resolution. ACM Transactions on Computational Logic, 16(4):28:1–28:15, July
Chapter 1. Proof Complexity


[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs


[IPS99] Russell Impagliazzo, Pavel Pudlák, and Jiří Sgall. Lower bounds for the polynomial calculus and the Gröbner basis algorithm. Comput-
Chapter 1. Proof Complexity


[151x228][KRPH18] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In Automated Reasoning - 6th
Chapter 1. Proof Complexity


Chapter 1. Proof Complexity


(PD11) Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. Artificial Intelligence,
Chapter 1. Proof Complexity


Chapter 1. Proof Complexity


92

Chapter 1. Proof Complexity

Springer, July 2010.