Chapter 2

PRELIMINARIES

Throughout this thesis, we work with propositional or Boolean variables, that is, variables that take value in the set {TRUE, FALSE}. A propositional formula F representing a Boolean function is formed by combining these variables using various Boolean operators. We use the two binary operators conjunction (AND, $\wedge$) and disjunction (OR, $\vee$), and the unary operator negation (NOT, $\neg$). These three operators are sufficient to express all Boolean functions and, at the same time, provide enough expressiveness to encode many interesting functions in a fairly compact way.

Definition 2.1. A propositional formula F is *satisfiable* if there exists an assignment ρ to its variables such that F evaluates to TRUE under ρ. If no such ρ exists, F is *unsatisfiable*. F is a *tautology* if $\neg F$ is unsatisfiable.

We often use the abbreviations SAT and UNSAT for satisfiable and unsatisfiable, respectively. A variable assignment ρ under which F evaluates to TRUE is referred to as a *satisfying assignment* for F.

Definition 2.2. A propositional formula F is in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, where each clause is a disjunction of *literals* and each literal is either a variable or its negation. The *size* of F is the number of clauses in F.

It is natural to think of F as a set of clauses and each clause as a set of literals. We use the symbol Λ to denote the *empty clause* which is always unsatisfiable. A clause with only one literal is referred to as a *unit* clause. A clause that is a subset of another is called its *subclause*. Let ρ be a partial assignment to the variables of F.

Definition 2.3. The *restricted formula* F^ρ is obtained from F by replacing variables in ρ with their assigned values. F is said to be *simplified* if all clauses with at least one TRUE literal are deleted and all occurrences of FALSE literals are removed from clauses. $F|_\rho$ denotes the result of simplifying the restricted formula F^ρ.

The construction of Tseitin [109] can be used to efficiently convert any given propositional formula to one in CNF form by adding new variables corresponding to its subformulas. For instance, given an arbitrary propositional formula G, one would first locally re-write each of its operators in terms of $\wedge, \vee$, and $\neg$ to obtain, say, $G = (((a \wedge b) \vee (\neg a \wedge \neg b)) \wedge \neg c) \vee d$. To convert this to CNF, one would add four auxiliary variables w, x, y, and z, construct clauses that encode the four relations

$w \leftrightarrow (a \wedge b)$, $x \leftrightarrow (\neg a \wedge \neg b)$, $y \leftrightarrow (w \vee x)$, and $z \leftrightarrow (y \wedge \neg c)$, and add to that the clause $(z \vee d)$. Given this efficient conversion mechanism, we restrict ourselves to CNF formulas.

2.1 The Propositional Satisfiability Problem

The combinatorial problem that lies at the heart of this work is the satisfiability problem that asks whether a given propositional formula has a satisfying assignment. More precisely,

Definition 2.4. The *propositional satisfiability problem* is the following: Given a CNF formula F as input, determine whether F is satisfiable or not. If it is satisfiable, output a satisfying assignment for it.

The decision version of this problem, where one is only asked to report SAT or UNSAT, is also referred to as CNF-SAT in the literature. In their well-known work, Cook [38] and Levin [79] proved the problem to be NP-complete, setting the foundation for a vast amount of research in complexity theory.

In this thesis, we will look at this problem from various perspectives. When a formula F is unsatisfiable, we will be interested in analyzing the size of the shortest *proof* of this fact. The formal machinery using which such proofs are presented and verified is discussed in the following section. From the practical perspective, we will also be interested in designing algorithms to find such proofs efficiently. When F is satisfiable, the task will be to design algorithms that efficiently find a satisfying assignment for it. Although some applications may require one to output several satisfying assignments, we will focus on finding one.

An algorithm that solves the propositional satisfiability problem is called a *satisfiability algorithm*. Practical implementations of such algorithms typically involve smart data structures and carefully chosen parameters in addition to an efficient top-level algorithm. These implementations are referred to as *SAT solvers*. Note the use of SAT here as referring to the propositional satisfiability problem in contrast to being an abbreviation of satisfiable. Henceforth, we leave it up to the context to make the meaning of "SAT" unambiguous.

2.2 Proof Systems

The notion of a propositional proof system was first defined in the seminal work of Cook and Reckhow [39]. It is an efficient (in the size of the proof) procedure to check the correctness of proofs presented in a certain format. More formally,

Definition 2.5. A *propositional proof system* is a polynomial time computable predicate S such that a propositional formula F is unsatisfiable iff there exists a *proof* (or *refutation*) π for which $S(F, \pi)$ holds.

We refer to such systems simply as *proof systems* and omit the word propositional. Note that proof systems can alternatively be defined for tautologies because of the fact that F is an unsatisfiable formula iff $\neg F$ is a tautology. In this manuscript, however, we use the phrase proof system in the context of unsatisfiable formulas only.

The strength of a proof system is characterized by the sizes of proofs it admits for various unsatisfiable formulas: a stronger proof system can verify the correctness of shorter proofs presented in a more complex format. This motivates the following definition.

Definition 2.6. The *complexity* of a formula F under a proof system S, denoted $\mathcal{C}_S(F)$, is the length of the shortest refutation of F in S.

Let $\{F_n\}$ be a family of formulas over an increasing number of variables n. The asymptotic complexity of $\{F_n\}$ in S with respect to n is given by the function $f(n) = \mathcal{C}_S(F_n)$ and is denoted $\mathcal{C}_S(F_n)$, with abuse of notation. We will be interested in characterizing families of formulas as having polynomial or exponential asymptotic complexity under specific proof systems.

2.2.1 Resolution

Resolution (RES) is a widely studied simple proof system that can be used to prove unsatisfiability of CNF formulas. It forms the basis of many popular systems for practical theorem proving. Lower bounds on resolution proof sizes thus have a bearing on the running time of these systems.

The *resolution rule* states that given clauses $C_1 = (A \vee x)$ and $C_2 = (B \vee \neg x)$, one can derive the clause $C = (A \vee B)$ by *resolving on x*. C_1 and C_2 are called the *parent* clauses and C is called their *resolvent*. The resolution rule has the property that a derived clause is satisfied by any assignment that satisfies both the parent clauses.

Definition 2.7. A *resolution derivation* of C from a CNF formula F is a sequence $\pi = (C_1, C_2, \ldots, C_s = C)$ with the following property: each clause C_i in π is either a clause of F (an *initial* clause) or is derived by applying the resolution rule to C_j and C_k, $1 \leq j, k < i$ (a *derived* clause). The *size* of π is s, the number of clauses occurring in it.

We assume that the clauses in π are non-redundant, i.e., each $C_j \neq C$ in π is used to derive at least one other clause $C_i, i > j$. Any derivation of the empty clause Λ from F, also called a *refutation* or *proof* of F, shows that F is unsatisfiable.

Definition 2.8. Let F be a CNF formula and π a resolution proof of its unsatisfiability.

(a) The *size* of π, *size*(π), is the number of clauses appearing in π.

(b) The *resolution complexity* of F, $\mathsf{RES}(F)$, is the minimum of $size(\pi)$ over all resolution proofs π of F; if no such proofs exist, $\mathsf{RES}(F) = \infty$.

(c) The *width* of a clause is the number of literals occurring in it. The width $w(F)$ of F and the width $w(\pi)$ of π are the maximum of the widths of all clauses in F and π, respectively.

(d) The *refutation width* of F, $w(F \vdash \Lambda)$, is the minimum of $w(\pi)$ over all proofs π of F.

As we shall see in Section 2.2.3, to prove a lower bound on $\mathsf{RES}(F)$, it is sufficient to prove a lower bound on the refutation width of F. It also typically turns out to be easier to analyze the width rather than the size of the smallest refutation. This makes the concept of width quite useful in proof complexity.

It is often insightful to think of the *structure* of a resolution refutation (or derivation) π in terms of a directed acyclic graph G_π defined as follows. G_π has a vertex for each clause in π, labeled with that clause. If the clause C_k labeling a vertex v in G_π is derived by resolving clauses C_i and C_j upon a variable x, then (a) v has a secondary label x, and (b) v has two outgoing edges directed to the vertices labeled C_i and C_j. All vertices labeled with initial clauses of π do not have a secondary label and have outdegree zero.

2.2.2 Refinements of Resolution

Despite its simplicity, unrestricted resolution as defined above (also called *general resolution*) is hard to implement efficiently due to the difficulty of finding good choices of clauses to resolve; natural choices typically yield huge storage requirements. Various restrictions on the structure of resolution proofs lead to less powerful but easier to implement refinements that have been studied extensively in proof complexity.

Definition 2.9. Let $\pi = (C_1, C_2, \ldots, C_s = C)$ be a resolution derivation, G_π be the graph associated with it, α be an assignment to the variables in π, α_F be the all FALSE assignment, and α_T be the all TRUE assignment.

(a) π is *tree-like* if each vertex in G_π corresponding to a derived clause has indegree 1.

(b) π is *regular* if no secondary vertex label appears twice in any directed path in G_π.

(c) π is *ordered* or *Davis-Putnam* if the sequence of secondary vertex labels along every directed path in G_π respects a fixed total ordering of the variables.

(d) π is *linear* if each C_i in π is either an initial clause or is derived by resolving C_{i-1} with $C_j, j < i-1$.

(e) π is an *α-derivation* if at least one parent clause involved in each resolution step in it is falsified by α.

(f) π is *positive* if it is an α-derivation for $\alpha = \alpha_F$.

(g) π is *negative* if it is an α-derivation for $\alpha = \alpha_T$.

(h) π is *semantic* if it is an α-derivation for some α.

While all these refinements are sound and complete as proof systems, they differ vastly in efficiency. For instance, in a series of results, Bonet et al. [27], Bonet and Galesi [28], and Buresh-Oppenheim and Pitassi [31] have shown that regular, ordered, linear, positive, negative, and semantic resolution are all exponentially stronger than tree-like resolution. On the other hand, Bonet et al. [27] and Alekhnovich et al. [3] have proved that tree-like, regular, and ordered resolution are exponentially weaker than RES.

2.2.3 The Size-Width Relationship

Most known resolution complexity lower bounds, including our results in subsequent chapters, can be proved using a general result of Ben-Sasson and Wigderson [23] that is derived from earlier arguments by Haken [60] and Clegg, Edmonds, and Impagliazzo [36]. It provides a relationship between the size of resolution proofs and their width (recall Definition 2.8), namely, any short proof of unsatisfiability of a CNF formula can be converted to one of small width. Therefore, a lower bound on the width of a resolution proof implies a lower bound on its size.

For a reason that will become clear in Section 2.3.1, we will use $\mathsf{DPLL}(F)$ to denote the tree-like resolution complexity of a formula F.

Proposition 2.1 ([23]). *For any CNF formula F, $\mathsf{DPLL}(F) \geq 2^{w(F \vdash \Lambda) - w(F)}$.*

Proposition 2.2 ([23]). *For any CNF formula F over n variables and $c = 1/(9 \ln 2)$, $\mathsf{RES}(F) \geq 2^{c(w(F \vdash \Lambda) - w(F))^2/n}$.*

For completeness, we sketch the proof of this result for the case of tree-like resolution. Suppose we have a refutation π of F (over n variables) with $size(\pi) \leq 2^b$. The idea is to use induction on n and b to construct a refutation π' of F such that $width(\pi') \leq b$. Let the last variable resolved upon in π be x. Assume without loss of generality that x is derived in π by a tree-like derivation of size at most 2^{b-1}. This gives a refutation of $F|_{\neg x}$ of the same size by simply removing x from all clauses. By induction on b, this can be converted into a refutation of $F|_{\neg x}$ of width at most $b-1$, which immediately gives a derivation π'' of x from F of width at most b by adding x back to the initial clauses from which it was removed and propagating the change.

On the other hand, π contains a derivation of $\neg x$ of size at most 2^b which can be converted to a refutation of $F|_x$ of the same size. By induction on n, this refutation, and hence the original derivation of $\neg x$, can be converted to one of width at most b. Now resolve, wherever possible, each of the initial clauses of this small width derivation of $\neg x$ with the result x of the derivation π'' and propagate the resulting simplification. This gives a refutation π' of F of width at most b.

2.3 The DPLL *Procedure and Clause Learning*

The Davis-Putnam-Logemann-Loveland or DPLL procedure is both a proof system as well as a collection of algorithms for finding proofs of unsatisfiable formulas. It can equally well be thought of as a collection of complete algorithms for finding a satisfying assignment for a given formula; its failure to find such an assignment constitutes a proof of unsatisfiability of the formula. While the former view is more suited to proof complexity theory, the latter is the norm when designing satisfiability algorithms. Davis and Putnam [44] came up with the basic idea behind this procedure. However, it was only a couple of years later that Davis, Logemann, and Loveland [43] presented it in the efficient top-down form in which it is widely used today.

Algorithm 1, `DPLL-recursive(`F, ρ`)`, sketches the basic DPLL procedure on CNF formulas. The idea is to repeatedly select an unassigned literal ℓ in the input formula F and recursively search for a satisfying assignment for $F|_\ell$ and $F_{\neg\ell}$. The step where such an ℓ is chosen is commonly referred to as the *branching* step. Setting ℓ to TRUE or FALSE when making a recursive call is called a *decision*. The end of each recursive call, which takes F back to fewer assigned variables, is called the *backtracking* step.

A partial assignment ρ is maintained during the search and output if the formula turns out to be satisfiable. If $F|_\rho$ contains the empty clause, the corresponding clause of F from which it came is said to be *violated* by ρ. To increase efficiency, unit clauses are immediately set to TRUE as outlined in Algorithm 1. *Pure literals* (those whose negation does not appear) are also set to TRUE as a preprocessing step and, in some implementations, in the simplification process after every branch.

At any point during the execution of the algorithm, a variable that has been assigned a value at a branching step is called a *decision variable* while one that has been assigned a value by unit propagation is called an *implied variable*. The *decision level* of an assigned variable is the recursive depth (starting at 0) of the call to `DPLL-recursive` that assigns it a value.

Variants of this algorithm form the most widely used family of complete algorithms for formula satisfiability. They are frequently implemented in an iterative rather than recursive manner, resulting in significantly reduced memory usage. The key difference

```
Input   : A CNF formula $F$ and an initially empty partial assignment $\rho$
Output : UNSAT, or an assignment satisfying $F$
begin
    $(F, \rho) \leftarrow$ UnitPropagate($F, \rho$)
    if $F$ contains the empty clause then  return UNSAT
    if $F$ has no clauses left then
        Output $\rho$
        return SAT
    $\ell \leftarrow$ a literal not assigned by $\rho$                    // the branching
    step
    if DPLL-recursive($F|_\ell, \rho \cup \{\ell\}$) $= SAT$ then  return SAT
    return DPLL-recursive($F|_{\neg\ell}, \rho \cup \{\neg\ell\}$)
end

UnitPropagate($F$)
begin
    while $F$ contains no empty clause but has a unit clause $x$ do
        $F \leftarrow F|_x$
        $\rho \leftarrow \rho \cup \{x\}$
    return $(F, \rho)$
end
```

Algorithm 2.1: DPLL-recursive(F, ρ)

in the iterative version is the extra step of *unassigning* variables when one backtracks. The naive way of unassigning variables in a CNF formula is computationally expensive, requiring one to examine every clause in which the unassigned variable appears. However, the *watched literals* scheme of Moskewicz et al. [88] provides an excellent way around this and merits a brief digression.

The Watched Literals Scheme

The key idea behind the watched literals scheme, as the name suggests, is to maintain and "watch" two special literals for each active (i.e., not yet satisfied) clause that are not FALSE under the current partial assignment. Recall that empty clauses halt the DPLL process and unit clauses are immediately satisfied. Hence, one can always find such watched literals in all active clauses. Further, as long as a clause has two such literals, it cannot be involved in unit propagation. These literals are maintained as follows. When a literal ℓ is set to FALSE, we must find another watched literal for the clause that had ℓ as a watched literal. We must also let $\neg\ell$ be a watched literal for previously active clauses that are now satisfied because of this assignment to ℓ. By doing this, positive literals are given priority over unassigned literals for being the watched literals.

With this setup, one can test a clause for satisfiability by simply checking whether at least one of its two watched literals is TRUE. Moreover, the relatively small amount of extra book-keeping involved in maintaining watched literals is well paid off when one unassigns a literal ℓ by backtracking – in fact, one needs to do absolutely nothing! The invariant about watched literals is maintained as such, saving a substantial amount of computation that would have been done otherwise.

2.3.1 Relation to Tree-like Resolution

When a formula F is unsatisfiable, the transcript of the execution of DPLL on F forms a proof of its unsatisfiability. This proof is referred to as a DPLL *refutation* of F. The *size* of a DPLL refutation is the number of branching steps in it. As the following Proposition shows, the structure of DPLL refutations is intimately related to the structure of tree-like resolution refutations.

Proposition 2.3. *A CNF formula F has a* DPLL *refutation of size at most s iff it has a tree-like resolution refutation of size at most s.*

Proof. The idea is to associate with every DPLL refutation τ a tree T^τ and show how T^τ can be viewed as or simplified to the graph G_π associated with a tree-like resolution refutation π of F. Given a DPLL refutation τ, the tree T^τ is constructed recursively by invoking the construction for `DPLL-recursive(`F, ϕ`)`. We describe below the construction in general for `DPLL-recursive(`H, ρ`)` for any sub-formula H of F and partial assignment ρ to the variable of F.

Start by creating the root node v for the tree corresponding to the procedure call `DPLL-recursive(`H, ρ`)`. If the procedure terminates because there is an empty clause after unit propagation, label v with an initial clause of F that has become empty and stop. If not, let ℓ be the literal chosen in the branching step of the call. Recursively create the two subtrees T_ℓ and $T_{\neg\ell}$ associated with the recursive calls to `DPLL-recursive(`$H|_\ell, \{\ell\}$`)` and `DPLL-recursive(`$H|_{\neg\ell}, \{\neg\ell\}$`)`, respectively. If either of T_ℓ or $T_{\neg\ell}$ is labeled by a clause that does not contain $\neg\ell$ or ℓ, respectively, then discard v and the other subtree, associate this one with the procedure call `DPLL-recursive(`H, ρ`)`, and stop. Otherwise, add edges from v to the roots of T_ℓ and $T_{\neg\ell}$. Let x be the variable corresponding to ℓ. Label v with the clause obtained by resolving on x the clauses labeling T_ℓ and $T_{\neg\ell}$. Finally, assign x as the secondary label for v.

It can be verified that the label of the root node of the final tree T^τ corresponding to the call to `DPLL-recursive(`F, ϕ`)` is the empty clause Λ and that T^τ is precisely the graph G_π associated with a legal tree-like resolution refutation π of F.

On the other hand, if one starts with a graph $G|_\pi$ associated with a tree-like resolution refutation π of F, the graph can be viewed unchanged as the tree T^τ associated with a DPLL refutation τ of F. This finishes the proof. □

Corollary 2.1. *For a CNF formula F, the size of the smallest* DPLL *refutation of F is equal to the size of the smallest tree-like resolution refutation of F.*

This explains why we used DPLL(F) to denote the tree-like resolution complexity of F in Section 2.2.3.

2.3.2 Clause Learning

The technique of clause learning was first introduced in the context of the DPLL-based SAT solvers by Marques-Silva and Sakallah [84]. It can be thought of as an extension of the DPLL procedure that caches causes of assignment failures in the form of learned clauses. It proceeds by following the normal branching process of DPLL until there is a "conflict," i.e., a variable is implied to be TRUE as well as FALSE by unit propagation. We give here a brief sketch of how conflicts are handled, deferring more precise details to Section 4.2.

If a conflict occurs when no variable is currently branched upon, the formula is declared UNSAT. Otherwise, the algorithm looks at the graphical structure of variable assignment implications (caused by unit propagation). From this, it infers a possible "cause" of the conflict, i.e., a relatively small subset of the currently assigned variables that, by unit propagation, results in the conflict. This cause is learned in the form of a "conflict clause." The idea is to avoid any future conflicts that may result from a careless assignment to the subset of variables already known to potentially cause a conflict. The algorithm now backtracks and continues as in ordinary DPLL, treating the learned clause just like the initial ones. A clause is said to be *known* at a stage if it is either an initial clause or has previously been learned.

Algorithm 2.2 gives the basic structure of the clause learning algorithm by Moskewicz et al. [88] used in the popular SAT solver zChaff. This algorithm forms the basis of our implementations and experiments in subsequent chapters. We present it here as the top-level iterative process that lies at the heart of zChaff.

The procedure `DecideNextBranch` chooses the next variable to branch on. In zChaff, this is done using the Variable State Independent Decaying Sum (VSIDS) heuristic which assigns a slowly decaying weight to each literal that is boosted whenever the literal is involved in a conflict. Note that there is no explicit variable flip in the entire algorithm. The conflict clause learning strategy used by zChaff automatically (by unit propagation) flips the assignment of the current variable before backtracking. The procedure `Deduce` applies unit propagation, keeping track of any clauses that may become empty, causing what is known as a conflict. If all clauses have been satisfied, it declares the formula to be SAT. The procedure `AnalyzeConflict` looks at the structure of implications and computes from it a conflict clause to learn. It also computes and returns the decision level that one needs to backtrack.

In general, the learning process is expected to save us from redoing the same computation when we later have an assignment that causes conflict due in part to

```
Input  : A CNF formula
Output : UNSAT, or SAT along with a satisfying assignment
begin
    while TRUE do
        DecideNextBranch
        while TRUE do
            status ← Deduce
            if $status$ = $CONFLICT$ then
                blevel ← AnalyzeConflict
                if $blevel = 0$ then return UNSAT
                Backtrack($blevel$)
            else if $status$ = $SAT$ then
                Output current assignment stack
                return SAT
            else break
end
```

Algorithm 2.2: DPLL-ClauseLearning

the same reason. Variations of such conflict-driven learning include different ways of choosing the clause to learn (different *learning schemes*) and possibly allowing multiple clauses to be learned from a single conflict. In the last decade, many algorithms based on this idea have been proposed and demonstrated to be empirically successful on large problems that could not be handled using other methodologies. These include Relsat by Bayardo Jr. and Schrag [13], Grasp by Marques-Silva and Sakallah [84], SATO by Zhang [113], and, as mentioned before, zChaff. We leave a more detailed discussion of the concepts involved in clause learning as well as its formulation as a proof system CL to Section 4.2.

Remark 2.1. Throughout this thesis, we will use the term DPLL to denote the basic branching and backtracking procedure given in Algorithm 1, and possibly the iterative version of it. It will not include learning conflict clauses when backtracking, but will allow intelligent branching heuristics as well as common extensions such as fast backtracking and restarts discussed in Section 4.2. Note that this is in contrast with the occasional use of the term DPLL to encompass practically all branching and backtracking approaches to SAT, including those involving learning.