Chapter 5

DPLL

Clause 8: No Title of Nobility shall be granted by the United States: And no Person holding any Office of Profit or Trust under them, shall, without the Consent of the Congress, accept of any present, Emolument, Office, or Title, of any kind whatever, from any King, Prince, or foreign State.
The United States Constitution

Contents

In this chapter we describe DPLL: a method of satisfiability checking. The name of this method is due to its authors Davis, Logemann and Loveland [1962]. DPLL is the most popular method of satisfiability checking, implemented in a number of systems. This methods works not on arbitrary formulas, but on formulas in *conjunctive normal forms*, or, equivalently, sets of *clauses*. The notions of conjunctive normal form and clause are introduced in Section 5.1. In Section 5.2 we show how one can transform an arbitrary formula to an equivalent formula in conjunctive normal form. In the worst case, this transformation can give a formula exponential in the size of the input. To cope with the problem of exponential increase in size, we in Section 5.3 we define an alternative *definitional transformation* of formulas to sets of clauses.

In Section 5.4 we introduce a special case of the satisfiability problem for clauses, called k-*SAT*, in which the length of clauses is restricted by k.

Sections 5.5–**??** deal with DPLL. In Section 5.5 we introduce a key simplification rule used by DPLL called *unit propagation*. Section **??** describes a DPLL algorithm. Finally, in Section **??** we discuss two simple optimizations of clausal satisfiability checking methods: *tautology elimination* and the *pure literal rule*.

5.1 Clauses

The splitting method has one disadvantage: arbitrary propositional formulas are difficult to manipulate with efficiently. Nearly all methods of automated reasoning, both in propositional and first-order logics, use a special kind of formulas, called *clauses*. The reasons for using clauses are the following:

(1) The satisfiability problem for arbitrary formulas can be reduced to the satisfiability problem for sets of clauses.

(2) Compared to arbitrary formulas, clauses are very simple. They are easier to implement.

In clause-based satisfiability-checking procedures, the input formula or set of formulas is first translated into a set of clauses, and then a clausal satisfiability-checking method is applied to this set. We will first define another special kind of formulas, called *literals*, and then clauses.

DEFINITION 5.1 (Literal) A *literal* is either an atom A or a negation $\neg A$ of an atom. A literal is called *positive* if it is an atom, and *negative* otherwise. The *complementary literal to a literal* L, denoted $\tilde{L}$, is defined by:

$$\tilde{L} \overset{\text{def}}{=} \begin{cases} \neg L, & \text{if } L \text{ is positive;} \\ A, & \text{if } L \text{ has the form } \neg A. \end{cases}$$ ❑

Note that literals p and $\neg p$ are complementary to each other. Evidently, for every literal L we have $\tilde{\tilde{L}} = L$.

DEFINITION 5.2 (Clause) A *clause* is a disjunction $L_1 \vee \ldots \vee L_n$, $n \geq 0$ of literals. When $n = 0$, the clause is called *empty* and denoted by $\square$. When $n = 1$ we say that the clause is *unit*. The clause is *Horn* if at most one of the L_i's is a positive literal. The clause is *positive* if all of the L_i's are positive, and *negative* if all of the L_i's are negative. ❑

Figure 5.1 illustrates this definition. Note the following

(1) every unit clause is Horn;

clause	unit	Horn	positive	negative
$\square$		✓	✓	✓
p	✓	✓	✓	
$\neg p$	✓	✓		✓
$p \vee q$			✓	
$\neg p \vee q$		✓		
$\neg p \vee \neg q$		✓		✓

Figure 5.1: Clauses

(2) a positive clause is Horn if and only if it is either unit or empty;

(3) every negative clause is Horn.

Let us now consider how the previously introduced notions, such as satisfiability, behave in the case of literals and clauses. Obviously, an interpretation I satisfies a positive literal p if and only $I(p) = 1$. Likewise, I satisfies a negative literal $\neg p$ if and only $I(p) = 0$. An interpretation I satisfies a clause C if and only if it satisfies at least one literal in C. For this reason we assume that *the empty clause is equivalent to* $\bot$, i.e., it is false in every interpretation. We will use the empty clause instead of $\bot$ in situations when we deal with clauses instead of arbitrary formulas.

The empty clause is unsatisfiable. Evidently, every non-empty clause is satisfiable. It is easy to see that a clause is a tautology if and and only if it contains a pair of complementary literals p and $\neg p$. Indeed, every clause $p \vee \neg p \vee C$ is, obviously, a tautology. Conversely, if a clause C contains no complimentary literals, one can define an interpretation I such that $I \not\models C$ as follows: for every positive literal p of C define $I(p) \stackrel{\text{def}}{=} 0$ and for every negative literal $\neg q$ of C define $I(q) \stackrel{\text{def}}{=} 1$.

The problems of checking satisfiability or validity of a single clause are, therefore, trivial. This is not the case for the problem of satisfiability of sets of clauses. We will now show that the satisfiability problem for arbitrary formulas is equivalent to the satisfiability problem for sets of clauses. To this end, we introduce the notions of conjunctive and disjunctive normal forms.

DEFINITION 5.3 (CNF, DNF) A formula A is in *conjunctive normal form*, or simply *CNF*, if it is either $\top$, or $\bot$, or a conjunction of disjunctions of literals:

$$A = \bigwedge_i \bigvee_j L_{i,j}.$$

Likewise, A is in *disjunctive normal form*, or simply *DNF*, if it is either $\top$, or $\bot$, or a disjunction of conjunctions of literals:

$$A = \bigvee_i \bigwedge_j L_{i,j}.$$

A formula B is called a conjunctive (respectively, disjunctive) normal form of a formula A if B is equivalent to A and B is in conjunctive (respectively, disjunctive) normal form. ❏

Let us show that the satisfiability problem for formulas in CNF can be regarded as a satisfiability problem for sets of clauses. Indeed, consider any formula in CNF:

$$A = \bigwedge_{i=1\ldots n} \bigvee_j L_{i,j}.$$

Denote by C_i the formula $\bigvee_j L_{i,j}$. Note that each C_i is a clause, so A is a conjunction of clauses: $A = \bigwedge_i C_i$. But then A is satisfiable if and only if so is the set of clauses $\{C_1, \ldots, C_n\}$. Thus, if we can effectively build a CNF of a formula, we can also reduce the satisfiability problem for arbitrary formulas to the satisfiability problem for clauses.

5.2 CNF Transformation

We will now give an algorithm for building a CNF of a formula. This algorithm is defined by a system of rewrite rules repeatedly applied to a formula until a CNF of this formula has been obtained.

ALGORITHM 5.4 (Standard CNF Transformation) An algorithm transforming formulas to their CNFs is given by the rewrite rule system on formulas of Figure 5.2. As usual, these rewrite rules are applied modulo permutation of arguments of $\land$ and $\lor$, for example, we do not make a difference between a formula $B_1 \land \ldots \land B_n$ and any formula $B_{i_1} \land \ldots \land B_{i_n}$, where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$. In addition, we use the following convention: any formula $(A_1 \land \ldots \land A_n) \land B_1 \land \ldots \land B_m$ is identified with $A_1 \land \ldots \land A_n \land B_1 \land \ldots \land B_m$, and similar for $\lor$ instead of $\land$.

The algorithm works as follows: given an input formula G, we apply the rewrite rules to it until we obtain a normal form. ❏

We claim that the rewrite rule system is terminating and that the computed normal form is, indeed, a CNF of G.

THEOREM 5.5 (CNF) *(i) The rewrite rule system of Figure 5.2 is terminating. (ii) Let G be a formula and G' be a normal form of G w.r.t. this rewrite rule system. Then G' is a CNF of G.* ❏

PROOF. We leave the termination proof to the reader. It is not hard to argue that every rewrite rule replaces a formula by an equivalent one, so by Equivalent Replacement Theorem 3.9, G' is equivalent to G. It remain to prove that G' is in CNF.

Since G' is irreducible, observe the following.

$$\begin{aligned}
A \leftrightarrow B &\Rightarrow (\neg A \vee B) \wedge (\neg B \vee A), && (5.1)\\
A \rightarrow B &\Rightarrow \neg A \vee B, && (5.2)\\
\neg(A \wedge B) &\Rightarrow \neg A \vee \neg B, && (5.3)\\
\neg(A \vee B) &\Rightarrow \neg A \wedge \neg B, && (5.4)\\
\neg\neg A &\Rightarrow A, && (5.5)\\
(A_1 \wedge \ldots \wedge A_m) \vee B_1 \vee \ldots \vee B_n &\Rightarrow \quad (A_1 \vee B_1 \vee \ldots \vee B_n) \quad \wedge && (5.6)\\
& \qquad\qquad \cdots \qquad\qquad\quad \wedge &&\\
& \quad (A_m \vee B_1 \vee \ldots \vee B_n). &&
\end{aligned}$$

Figure 5.2: Rewrite rules for the CNF transformation

- Neither (5.1) nor (5.2) are applicable to G', so G' contains no occurrences of $\leftrightarrow, \rightarrow$.
- Since G' does not contain $\leftrightarrow, \rightarrow$ and (5.3)–(5.5) are not applicable to G', negation in G' is only applied to atomic subformulas;
- Since (5.6) is not applicable, G' has no subformulas of the form $(A_1 \wedge \ldots \wedge A_m) \vee B_1 \vee \ldots \vee B_n$ for $m \geq 2$ and $n \geq 1$.

It is easy to see that these properties guarantee that G' is in conjunctive normal form. ❑

EXAMPLE 5.6 Consider the formula of Example 4.11: $\neg((p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r))$. We can rewrite it into CNF using the rewrite rules of Figure 5.2 as shown in Figure 5.3. The main connective of the rewritten subformula in every rewrite step is marked by shading it, for example $\rightarrow$. Therefore, to check this formula for unsatisfiability, one should check for unsatisfiability the set of clauses

$$\{\neg p \vee q,\ \neg p \vee \neg q \vee r,\ p,\ \neg r\}. \qquad ❑$$

If we have to check for satisfiability a *set* S of formulas rather than a single formula, we can transform into a CNF every formula in S and take the union of the resulting sets of clauses.

5.3 Clausal Form and Definitional Transformation

Consider the formula

$$p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6)))). \tag{5.7}$$

If we try to compute a CNF of this formula using the rewrite rule system of Figure 5.3, the first rewriting step yields

$$\begin{array}{ll}
\neg((p \rightarrow q) \land (p \land q \rightarrow r) \rightarrow (p \rightarrow r)) & \Rightarrow \\
\neg(\neg((p \rightarrow q) \land (p \land q \rightarrow r)) \lor p \rightarrow r)) & \Rightarrow \\
\neg(\neg((p \rightarrow q) \land (p \land q \rightarrow r)) \lor \neg p \lor r) & \Rightarrow \\
\neg\neg((p \rightarrow q) \land (p \land q \rightarrow r)) \land \neg\neg p \land \neg r & \Rightarrow \\
(p \rightarrow q) \land (p \land q \rightarrow r) \land \neg\neg p \land \neg r & \Rightarrow \\
(p \rightarrow q) \land (p \land q \rightarrow r) \land p \land \neg r & \Rightarrow \\
(\neg p \lor q) \land (p \land q \rightarrow r) \land p \land \neg r & \Rightarrow \\
(\neg p \lor q) \land (\neg(p \land q) \lor r) \land p \land \neg r & \Rightarrow \\
(\neg p \lor q) \land (\neg p \lor \neg q \lor r) \land p \land \neg r. &
\end{array}$$

Figure 5.3: Rewriting into CNF

$$\begin{array}{l}
p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6)))) \Rightarrow \\
\qquad (\neg p_1 \lor (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))) \\
\qquad \land (\neg(p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6)))) \lor p_1).
\end{array}$$

The resulting formula contains two occurrences of the subformula $p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6)))$. If we rewrite each of these subformulas using the rules for $\leftrightarrow$, we will obtain four copies of the subformula $p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))$. In general, rewriting a formula with many occurrences of $\leftrightarrow$ may cause an exponential growth of the formula.

The question arises if there exists a more efficient CNF transformation algorithm. The bad news is that for some formulas their shortest CNF is exponential in the size of the formulas. The good news is that the notion of CNF is too strong and can be replaced by a weaker notion.

One can obtain a shorter set of clauses than that obtained by the rewrite rules of Figure 5.3 using the following idea of introduction of *definitions*. Consider formula (5.7). Let us introduce a new propositional variable n and "define" it to be equivalent to $p_5 \leftrightarrow p_6$ by adding the formula $n \leftrightarrow (p_5 \leftrightarrow p_6)$. As soon as we have done so, we can replace the subformula $p_5 \leftrightarrow p_6$ of (5.7) by the "equivalent" formula n. As a result, instead of formula (5.7), we obtain the following set consisting of two formulas

$$\begin{array}{l}
\{p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow n))), \\
\ n \leftrightarrow (p_5 \leftrightarrow p_6)\}.
\end{array}$$

It is not hard to argue that (5.7) is satisfiable if and only if so is this set of two formulas (see Lemma 5.8 below). However, note that the first formula of this set is simpler than (5.7) since it contains fewer occurrences of $\leftrightarrow$.

The new variable n in this example acts as an *name* or an *abbreviation* for the formula $p_5 \leftrightarrow p_6$. The formula $n \leftrightarrow (p_5 \leftrightarrow p_6)$ can be considered as a *definition* of n.

Note that by adding definitions we do not obtain a set of formulas equivalent to the original one. To demonstrate this, consider a simple example. Let S be the set consisting of a single formula p. Then the models of S are all interpretations I in which p is true. If we add a definition of p to S, we obtain the set $S' = \{p, n \leftrightarrow p\}$. The models of S are all interpretations I in which *both* p and n are true. Thus, S and S' are not equivalent: for example, the interpretation $\{p \mapsto 1, n \mapsto 0\}$ is a model of S but not of S'. Hence, we cannot use introduction of definitions for finding conjunctive normal forms.

The standard CNF transformation algorithm can give a formula of an exponential size even for formulas without equivalences, see Exercise 5.1. To justify the introduction of definitions, we introduce a more general notion which can be used to transform formulas into sets of clauses.

DEFINITION 5.7 (Clausal Form) Let A be a formula. Its *clausal form* is a set of clauses which is satisfiable if and only if A is satisfiable. Likewise, a clausal form of a set S of formulas is a set of clauses which is satisfiable if and only if so is S. ❏

Note that the notion of clausal form is fundamentally different from that of CNF. In the definition of CNF we require that CNF of a formula A be equivalent to A, while in the definition of clausal form we only require that A and its clausal form are equally satisfiable. This implies that every formula or set of formulas has a polynomially short clausal form. Indeed, if a formula A is satisfiable, then the empty set of clauses $\{\}$ is a clausal form of A. If A is unsatisfiable, then $\{\square\}$ is a clausal form of A. However, to find one of these clausal forms is computationally hard, since we have to check whether A is satisfiable. Therefore, we will be interested in clausal forms which are not hard to build. In addition, if a formula turns out to be satisfiable, then we also want to find a model or models of this formula, so in practice we are interested in clausal forms whose models can be used to find models of the original formula.

The use of names and definitions is justified by the following lemma.

LEMMA 5.8 *Suppose that S is a set of formulas and B be a formula. Let n be a boolean variable not occurring in S or B. Then S is satisfiable if and only if so is the set of formulas $S \cup \{n \leftrightarrow B\}$.*

PROOF. The "if" direction is obvious: every model of $S \cup \{n \leftrightarrow B\}$ is also a model of S.

Let us prove the "only if" direction. Suppose that S is satisfiable. Then it has a model I. Define the interpretation I' by setting the value of n to be the same as of B as follows:

$$I'(q) \overset{\text{def}}{=} \begin{cases} I(B), & \text{if } q = n; \\ I(q), & \text{otherwise.} \end{cases}$$

Since I' agrees with I on all atoms different from n and n does not occur in S, then I' is also a model of S. For the same reason we have $I(B) = I'(B)$. But by the definition of I' we have $I'(n) = I(B)$, so $I'(n) = I'(B)$. Therefore, I' also satisfies $n \leftrightarrow B$. ❏

Note that the condition that the boolean variable p is fresh for S is essentially used in the proof, see Exercise 5.3.

The proof of this lemma also gives us a correspondence between models of S and those of $S' \overset{\text{def}}{=} S \cup \{n \leftrightarrow B\}$. Indeed, every model of S' is also a model of S. Likewise, every model I of S can be turned into a model of S' by defining $I(n) \overset{\text{def}}{=} I(B)$.

The use of this lemma is immediate. If we have $p \leftrightarrow B$, then by Equivalent Replacement Lemma 3.8 every occurrence of B can be replaced by p. In particular, if B is a complex formula having many occurrences in the set S, we can obtain a set of formulas of a smaller size. We will say that the new boolean variable p is a *name* for the subformula B, hence the technique based on the introduction of such symbols is sometimes called the *naming technique*. We will also call the new formula $n \leftrightarrow B$ a *definition* of n.

The most obvious use of the naming technique is the following: for every "non-trivial" subformula occurring in the set of formulas S, introduce a name for this subformula, replace the subformula by its name, and a add the corresponding definition to S.

ALGORITHM 5.9 (Definitional Transformation) This algorithm converts a formula A into a set of clauses S such that S is a clausal normal form of A.

If A has the form $C_1 \wedge \ldots \wedge C_n$, where $n \geq 1$ and each C_i is a clause, then $S \overset{\text{def}}{=} \{C_1, \ldots, C_n\}$.

Otherwise, for each subformula B of A such that B is not a literal, introduce a new boolean variable p_B. Define a function n (the name function) on subformulas of A as follows:

$$n(B) \overset{\text{def}}{=} \begin{cases} B, & \text{if } B \text{ is a literal;} \\ p_B, & \text{otherwise.} \end{cases}$$

Note that $n(B)$ is always a literal. Introduce also the function $\tilde{n}(B)$ which returns the literal complementary to $n(B)$. Again, note that $\tilde{n}(B)$ is always a literal.

Now, for each newly introduced symbol p_B do the following.

(1) If B has the form $B_1 \wedge \ldots \wedge B_m$, then add to S the clauses obtained by transforming into CNF the formula $p_B \leftrightarrow n(B_1) \wedge \ldots \wedge n(B_m)$, i.e.,

$$\begin{array}{l} \neg p_B \vee n(B_1), \\ \ldots \\ \neg p_B \vee n(B_m), \\ \tilde{n}(B_1) \vee \ldots \vee \tilde{n}(B_m) \vee p_B. \end{array}$$

(2) If B has the form $B_1 \vee \ldots \vee B_m$, then add to S the clauses obtained by transforming into CNF the formula $p_B \leftrightarrow n(B_1) \vee \ldots \vee n(B_m)$, i.e.,

$$\begin{array}{l} \neg p_B \vee n(B_1) \vee \ldots \vee n(B_m), \\ \tilde{n}(B_1) \vee p_B, \\ \ldots \\ \tilde{n}(B_m) \vee p_B. \end{array}$$

(3) If B has the form $B_1 \to B_2$, then add to S the clauses obtained by transforming into CNF the formula $p_B \leftrightarrow (n(B_1) \to n(B_2))$, i.e.,

$$\begin{array}{l} \neg p_B \lor \tilde{n}(B_1) \lor n(B_2), \\ n(B_1) \lor p_B, \\ \tilde{n}(B_2) \lor p_B. \end{array}$$

(4) If B has the form $\neg B_1$, then add to S the clauses obtained by transforming into CNF the formula $p_B \leftrightarrow \neg n(B_1)$, i.e.,

$$\begin{array}{l} \neg p_B \lor \tilde{n}(B_1), \\ n(B_1) \lor p_B. \end{array}$$

(5) If B has the form $B_1 \leftrightarrow B_2$, then add to S the clauses obtained by transforming into CNF the formula $p_B \leftrightarrow (n(B_1) \leftrightarrow n(B_2))$, i.e.,

$$\begin{array}{l} \neg p_B \lor \tilde{n}(B_1) \lor n(B_2), \\ \neg p_B \lor \tilde{n}(B_2) \lor n(B_1), \\ n(B_1) \lor n(B_2) \lor p_B, \\ \tilde{n}(B_1) \lor \tilde{n}(B_2) \lor p_B. \end{array}$$

Strictly speaking, the standard CNF transformation of $p_B \leftrightarrow (n(B_1) \leftrightarrow n(B_2))$ introduces also two tautologies, $\tilde{n}(B_1) \lor n(B_1) \lor p_B$ and $n(B_2) \lor \tilde{n}(B_2) \lor p_B$, not included in this set.

Finally, add to S the unit clause p_A. ❑

Consider an example.

EXAMPLE 5.10 Take the (unsatisfiable) formula of Example 4.11: $\neg((p \to q) \land (p \land q \to r) \to (p \to r))$. We will use the new boolean variables $p_1, p_2, \ldots$ to name the non-literal subformulas of this formula. The subformulas, their names, and the corresponding formulas to be transformed to CNF are shown in Figure 5.4. ❑

The definitional transformation can be improved by one observation related to the polarities of subformulas: if a subformula B has only positive or only negative occurrences in a set of formulas S, then it suffices to use only "one half" of the definition $p \leftrightarrow B$, i.e., either $p \to B$ or $B \to p$.

LEMMA 5.11 *Let S be a set formulas, B a formula, and p a boolean variable not occurring in S, B. Let S' be a set of formulas obtained from S by replacing one or more positive (respectively, negative) occurrences of B in S by p. Then S is satisfiable if and only if so is $S' \cup \{p \to B\}$ (respectively, $S' \cup \{B \to p\}$).* ❑

name	subformula	formula to be transformed to CNF	clauses
			p_1
p_1	$\neg((p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r))$	$p_1 \leftrightarrow \neg p_2$	$\neg p_1 \vee \neg p_2$
			$p_1 \vee p_2$
p_2	$(p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r)$	$p_2 \leftrightarrow (p_3 \rightarrow p_7)$	$\neg p_2 \vee \neg p_3 \vee p_7$
			$p_3 \vee p_2$
			$\neg p_7 \vee p_2$
p_3	$(p \rightarrow q) \wedge (p \wedge q \rightarrow r)$	$p_3 \leftrightarrow (p_4 \wedge p_5)$	$\neg p_3 \vee p_4$
			$\neg p_3 \vee p_5$
			$\neg p_4 \vee \neg p_5 \vee p_3$
p_4	$p \rightarrow q$	$p_4 \leftrightarrow (p \rightarrow q)$	$\neg p_4 \vee \neg p \vee q$
			$p \vee p_4$
			$\neg q \vee p_4$
p_5	$p \wedge q \rightarrow r$	$p_5 \leftrightarrow (p_6 \rightarrow r)$	$\neg p_5 \vee \neg p_6 \vee r$
			$p_6 \vee p_5$
			$\neg r \vee p_5$
p_6	$p \wedge q$	$p_6 \leftrightarrow (p \wedge q)$	$\neg p_6 \vee p$
			$\neg p_6 \vee q$
			$\neg p \vee \neg q \vee p_6$
p_7	$p \rightarrow r$	$p_7 \leftrightarrow (p \rightarrow r)$	$\neg p_7 \vee \neg p \vee r$
			$p \vee p_7$
			$\neg r \vee p_7$

Figure 5.4: Definitional transformation applied to the formula $\neg((p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r))$

PROOF. We will prove this lemma only for the case of the positive occurrences of B, the other case is similar. Suppose that S is satisfiable. Then by Lemma 5.8 the set $S' \cup \{p \leftrightarrow B\}$ is satisfiable too. But every model of $S' \cup \{p \leftrightarrow B\}$ is obviously a model of $S' \cup \{p \rightarrow B\}$, so $S' \cup \{p \rightarrow B\}$ is satisfiable too.

Suppose now that $S' \cup \{p \rightarrow B\}$ is satisfiable and show that S is satisfiable too. Take any model I of $S' \cup \{p \rightarrow B\}$. We claim that I is also a model of S. To this end we have to show that I satisfies each formula $A \in S$. Take $A \in S$, then by the definition of S' there exists a formula $A' \in S'$ such that A is obtained from A' by replacing some occurrences of p by B. But all occurrences of p are positive, and we have $I \models A'$ and $I \models p \rightarrow B$, so by Lemma 4.12 we have $I \models A$. ❑

Using this lemma, we can optimize the definitional transformation so that it introduces fewer clauses when some subformula has only positive or only negative occurrences.

ALGORITHM 5.12 (Optimized Definitional Transformation) This algorithm converts a formula A into a set of clauses S such that S is a clausal normal form of A. It is defined in the same way as Definitional Transformation Algorithm 5.9 except that the clauses introduced for every symbol p_B are defined as follows. If B has occurrences in A of polarity 0 or has both positive and negative occurrences, then the clauses are defined exactly as in Algorithm 5.9. Otherwise, they are defined depending on the form of B, as explained below.

(1) If B has the form $B_1 \wedge \ldots \wedge B_m$, then add to S the clauses as in the following table:

polarity $+1$	polarity -1
$p_B \rightarrow n(B_1) \wedge \ldots \wedge n(B_m)$	$n(B_1) \wedge \ldots \wedge n(B_m) \rightarrow p_B$
$\neg p_B \vee n(B_1)$, $\ldots$ $\neg p_B \vee n(B_m)$.	$\tilde{n}(B_1) \vee \ldots \vee \tilde{n}(B_m) \vee p_B$.

The first row of this table shows the polarity of all occurrences of B. The second row shows, for each polarity, the formula that is to be transformed to the set of clauses. The third row shows the clauses obtained by applying the standard CNF transformation to this formula.

(2) If B has the form $B_1 \vee \ldots \vee B_m$, then add to S the clauses as in the following table:

polarity $+1$	polarity -1
$p_B \rightarrow n(B_1) \vee \ldots \vee n(B_m)$	$n(B_1) \vee \ldots \vee n(B_m) \rightarrow p_B$
$\neg p_B \vee n(B_1) \vee \ldots \vee n(B_m)$.	$\tilde{n}(B_1) \vee p_B$, $\ldots$ $\tilde{n}(B_m) \vee p_B$.

(3) If B has the form $B_1 \rightarrow B_2$, then add to S the clauses as in the following table:

polarity $+1$	polarity -1
$p_B \rightarrow (n(B_1) \rightarrow n(B_2))$	$(n(B_1) \rightarrow n(B_2)) \rightarrow p_B$
$\neg p_B \vee \tilde{n}(B_1) \vee n(B_2)$.	$n(B_1) \vee p_B$, $\tilde{n}(B_2) \vee p_B$.

(4) If B has the form $\neg B_1$, then add to S the clauses as in the following table:

polarity $+1$	polarity -1
$p_B \rightarrow \neg n(B_1)$	$\neg n(B_1) \rightarrow p_B$
$\neg p_B \vee \tilde{n}(B_1)$.	$n(B_1) \vee p_B$.

name	subformula	polarity	formula to be transformed to CNF	clauses
				p_1
p_1	$\neg((p \to q) \wedge (p \wedge q \to r) \to (p \to r))$	$+1$	$p_1 \to \neg p_2$	$\neg p_1 \vee \neg p_2$
p_2	$(p \to q) \wedge (p \wedge q \to r) \to (p \to r)$	-1	$(p_3 \to p_7) \to p_2$	$p_3 \vee p_2$ $\neg p_7 \vee p_2$
p_3	$(p \to q) \wedge (p \wedge q \to r)$	$+1$	$p_3 \to (p_4 \wedge p_5)$	$\neg p_3 \vee p_4$ $\neg p_3 \vee p_5$
p_4	$p \to q$	$+1$	$p_4 \to (p \to q)$	$\neg p_4 \vee \neg p \vee q$
p_5	$p \wedge q \to r$	$+1$	$p_5 \to (p_6 \to r)$	$\neg p_5 \vee \neg p_6 \vee r$
p_6	$p \wedge q$	-1	$(p \wedge q) \to p_6$	$\neg p \vee \neg q \vee p_6$
p_7	$p \to r$	-1	$(p \to r) \to p_7$	$p \vee p_7$ $\neg r \vee p_7$

Figure 5.5: Optimized definitional transformation applied to the formula $\neg((p \to q) \wedge (p \wedge q \to r) \to (p \to r))$

(5) If B has the form $B_1 \leftrightarrow B_2$, then add to S the clauses as in the following table:

polarity $+1$	polarity -1
$p_B \to (n(B_1) \leftrightarrow n(B_2))$	$(n(B_1) \leftrightarrow n(B_2)) \to p_B$
$\neg p_B \vee \tilde{n}(B_1) \vee n(B_2)$, $\neg p_B \vee \tilde{n}(B_2) \vee n(B_1)$.	$n(B_1) \vee n(B_2) \vee p_B$, $\tilde{n}(B_1) \vee \tilde{n}(B_2) \vee p_B$.

Finally, add to S the unit clause p_A. ❑

EXAMPLE 5.13 Consider the formula of Example 5.10: $\neg((p \to q) \wedge (p \wedge q \to r) \to (p \to r))$. The optimized transformation applied to this formula is illustrated in Figure 5.5. Compare the resulting set of clauses with the one obtained by the standard transformation, see Figure 5.4. ❑

In both examples of this section, the definitional transformation generated a set of clauses larger than the one obtained by the standard clausal form transformation. In general this is not true. It is not hard to prove that the size of the clausal form obtained by the definitional transformation is polynomial in the size of the input formula, see Exercise 5.2. The size of a clausal form generated by the standard CNF transformation algorithm is, in the worst case, exponential in the size of the input formula.

5.4 SAT and k-SAT

By using definitional clausal form transformation one can reduce, in polynomial time, the satisfiability problem for propositional formulas to the satisfiability problem for clauses. The satisfiability problem for clauses has its own name.

DEFINITION 5.14 (SAT) *SAT* is the following decision problem. An instance is a finite set of clauses. The answer is “yes” if this set is satisfiable. ❏

In the rest of this chapter and in the following chapters we will be interested in algorithms for solving SAT. Let us introduce special cases of SAT in which the number of literals in a clause is restricted.

DEFINITION 5.15 (k-SAT) Let k be a positive integer. By a k*-clause* we mean a clause with exactly k literals. Any k-clause for $k \geq 1$ can also be considered as $k+1$-clause by duplicating a literal in this clause. For example, the 2-clause $p \vee \neg q$ can be equivalently represented as the 3-clause $p \vee p \vee \neg q$. By k*-SAT* we denote the satisfiability problem for set of clauses having at most k literals. ❏

With respect to the complexity, the following is known about k-SAT:

(1) 2-SAT can be solved in polynomial time.

(2) k-SAT for $k \geq 3$ is NP-complete.

We will show a simple reduction of the satisfiability problem for propositional formulas to 3-SAT. This reduction can be used to establish that 3-SAT is NP-complete. We have already shown that, using definitional transformation, one can reduce[1] the satisfiability problem for arbitrary formulas to the satisfiability problem for clauses. Suppose now that we have a set S of clauses and would like to find a set S' of 3-clauses such that S is satisfiable if and only if so is S'. This can be done using the naming technique Suppose that S contains a clause $C = L_1 \vee L_2 \vee L_3 \vee \ldots \vee L_n$ having more than three literals. Take a boolean variable p not occurring in S and replace the clause $L_1 \vee L_2 \vee L_3 \vee \ldots \vee L_n$ by two clauses $\neg p \vee L_1 \vee L_2$ and $p \vee L_3 \vee \ldots \vee L_n$. The former of these clauses has exactly three literals, the latter one literal less than C. Therefore, after a number of such transformations we will obtain a set of clauses with at most three literals. Using the same arguments as in the proof of Lemma 5.11, we can show that the transformation preserves satisfiability.

5.5 Unit Propagation

The use of sets of clauses instead of arbitrary formulas allows us to use a new kind of simplification, called *unit propagation*. Consider, for example, the set S of clauses obtained from the unsatisfiable formula $\neg((p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r))$ in Example 5.6.

[1]More precisely, reduce in polynomial time.

$$\{\neg p \lor q,\ \neg p \lor \neg q \lor r,\ p,\ \neg r\}.$$

We can make an immediate observation about models of this set of clauses: in every such model I the atom p must be true. Indeed, if an interpretation I satisfies this set of clauses, then it satisfies every clause in the set, so it satisfies p, hence $I \vDash p$. But then $I \nvDash \neg p$, so in the clause $\neg p \lor q$ the literal $\neg p$ cannot be satisfied. If $\neg p$ cannot be satisfied, we can delete this literal from the clause $\neg p \lor q$ obtaining the clause q. Likewise, when we observe that p must be true, we can remove all clauses containing p as a literal, since all of these clauses are satisfied in I. Let us illustrate this by the following picture.

$$\begin{array}{l|ll} \neg p \lor q & & q \\ \neg p \lor \neg q \lor r & & \neg q \lor r \\ \cancel{p} & & \\ \neg r & \neg r & \end{array}$$

In this picture we show the set S of clauses on the left of the bar. The removed clause is shown by putting a diagonal line through it: $\cancel{p}$. The literals that can be deleted from their clauses are shaded, for example $\neg p \lor q$. On the right of the bar we show the simplified set of clauses obtained from S by removed the clauses containing p as a literal and deleting the literal $\neg p$ from the remaining clauses. This simplification is called *unit propagation* and defined below.

DEFINITION 5.16 (Unit Propagation) Let S be a set of clauses. We say that a set of clauses S' is obtained from S by *unit propagation* if S' is obtained from S by repeatedly performing the following transformation: if S contains a unit clause, i.e. a clause consisting of one literal L, then

(1) remove from S every clause of the form $L \lor C'$;

(2) replace in S every clause of the form $\tilde{L} \lor C'$ by the clause C'. ❏

Unit propagation is a very powerful simplification. Let us show how we can show unsatisfiability of a non-trivial set of clauses by using only unit propagation. We take the set of clauses obtained by the definitional translation from the formula $\neg((p \to q) \land (p \land q \to r) \to (p \to r))$, see Figure 5.4. The first step of unit propagation eliminates the variable p_1:

~~p_1~~	$\neg q \lor p_4$		$\neg q \lor p_4$
$\neg p_1 \lor \neg p_2$	$\neg p_5 \lor \neg p_6 \lor r$	$\neg p_2$	$\neg p_5 \lor \neg p_6 \lor r$
~~$p_1 \lor p_2$~~	$p_6 \lor p_5$		$p_6 \lor p_5$
$\neg p_2 \lor \neg p_3 \lor p_7$	$\neg r \lor p_5$	$\neg p_2 \lor \neg p_3 \lor p_7$	$\neg r \lor p_5$
$p_3 \lor p_2$	$\neg p_6 \lor p$	$p_3 \lor p_2$	$\neg p_6 \lor p$
$\neg p_7 \lor p_2$	$\neg p_6 \lor q$	$\neg p_7 \lor p_2$	$\neg p_6 \lor q$
$\neg p_3 \lor p_4$	$\neg p \lor \neg q \lor p_6$	$\neg p_3 \lor p_4$	$\neg p \lor \neg q \lor p_6$
$\neg p_3 \lor p_5$	$\neg p_7 \lor \neg p \lor r$	$\neg p_3 \lor p_5$	$\neg p_7 \lor \neg p \lor r$
$\neg p_4 \lor \neg p_5 \lor p_3$	$p \lor p_7$	$\neg p_4 \lor \neg p_5 \lor p_3$	$p \lor p_7$
$\neg p_4 \lor \neg p \lor q$	$\neg r \lor p_7$	$\neg p_4 \lor \neg p \lor q$	$\neg r \lor p_7$
$p \lor p_4$		$p \lor p_4$	

The resulting set of clauses contains the unit clause $\neg p_2$, so we can apply unit propagation again:

~~$\neg p_2$~~	$\neg q \lor p_4$		$\neg q \lor p_4$
~~$\neg p_2 \lor \neg p_3 \lor p_7$~~	$\neg p_5 \lor \neg p_6 \lor r$		$\neg p_5 \lor \neg p_6 \lor r$
$p_3 \lor p_2$	$p_6 \lor p_5$	p_3	$p_6 \lor p_5$
$\neg p_7 \lor p_2$	$\neg r \lor p_5$	$\neg p_7$	$\neg r \lor p_5$
$\neg p_3 \lor p_4$	$\neg p_6 \lor p$	$\neg p_3 \lor p_4$	$\neg p_6 \lor p$
$\neg p_3 \lor p_5$	$\neg p_6 \lor q$	$\neg p_3 \lor p_5$	$\neg p_6 \lor q$
$\neg p_4 \lor \neg p_5 \lor p_3$	$\neg p \lor \neg q \lor p_6$	$\neg p_4 \lor \neg p_5 \lor p_3$	$\neg p \lor \neg q \lor p_6$
$\neg p_4 \lor \neg p \lor q$	$\neg p_7 \lor \neg p \lor r$	$\neg p_4 \lor \neg p \lor q$	$\neg p_7 \lor \neg p \lor r$
$p \lor p_4$	$p \lor p_7$	$p \lor p_4$	$p \lor p_7$
	$\neg r \lor p_7$		$\neg r \lor p_7$

Now we have two units: p_3 and $\neg p_7$. We can apply unit propagation to them simultaneously:

~~p_3~~	$\neg q \lor p_4$		$\neg q \lor p_4$
~~$\neg p_7$~~	$\neg p_5 \lor \neg p_6 \lor r$		$\neg p_5 \lor \neg p_6 \lor r$
$\neg p_3 \lor p_4$	$p_6 \lor p_5$	p_4	$p_6 \lor p_5$
$\neg p_3 \lor p_5$	$\neg r \lor p_5$	p_5	$\neg r \lor p_5$
~~$\neg p_4 \lor \neg p_5 \lor p_3$~~	$\neg p_6 \lor p$		$\neg p_6 \lor p$
$\neg p_4 \lor \neg p \lor q$	$\neg p_6 \lor q$	$\neg p_4 \lor \neg p \lor q$	$\neg p_6 \lor q$
$p \lor p_4$	$\neg p \lor \neg q \lor p_6$	$p \lor p_4$	$\neg p \lor \neg q \lor p_6$
	~~$\neg p_7 \lor \neg p \lor r$~~		
	$p \lor p_7$		p
	$\neg r \lor p_7$		$\neg r$

We obtained four new units: p_4, p_5, p and $\neg r$. The next unit propagation step gives

~~p_4~~	~~$\neg q \lor p_4$~~		
~~p_5~~	$\neg p_5 \lor \neg p_6 \lor r$		$\neg p_6$
$\neg p_4 \lor \neg p \lor q$	~~$p_6 \lor p_5$~~	q	
~~$p \lor p_4$~~	~~$\neg r \lor p_5$~~		
	~~$\neg p_6 \lor p$~~		
	$\neg p_6 \lor q$		$\neg p_6 \lor q$
	$\neg p \lor \neg q \lor p_6$		$\neg q \lor p_6$
	~~p~~		
	~~$\neg r$~~		

We obtained two units q and $\neg p_6$. The next unit propagation step gives

$$\begin{array}{ll|c} \cancel{q} & \cancel{\neg p_6} & \\ & \cancel{\neg p_6 \vee q} & \\ & \neg q \vee p_6 & \square \end{array}$$

We obtained a set of clauses containing the empty clause. Since the empty clause is unsatisfiable, the original set of clauses S is unsatisfiable too.

5.6 Horn SAT

In this section we show that unit propagation alone is sufficient to solve the satisfiability problem for sets of Horn clauses. First, let us note a general property of unsatisfiable sets of clauses.

LEMMA 5.17 *Let S be an unsatisfiable set of clauses. Then S contains at least one positive clause and at least one negative clause.*

PROOF. We prove that S contains a positive clause, the proof of existence of a negative clause is similar. Suppose, by contradiction, that S contains no positive clauses. Then every clause in S contains a negative literal. Consider the interpretation I in which all variables are false, this interpretation satisfies all clauses in S, which contradicts to the unsatisfiability of S. ❏

THEOREM 5.18 *Let S be a set of Horn clauses and S' be obtained from S by unit propagation. Then S is satisfiable if and only if $\square \notin S'$.*

PROOF. Since unit propagation preserves satisfiability, S is satisfiable if and only if so is S'. Therefore, it is enough to prove that S' is satisfiable if and only if $\square \notin S'$. Obviously a satisfiable set of clauses cannot contain the empty clause. Hence, it remains to show that if S' is unsatisfiable, then it contains $\square$.

Note that unit propagation applied to a set of Horn clauses gives a set of Horn clauses, so S' consists of Horn clauses only. By Lemma 5.17 S' must contain a positive clause C. But a positive Horn clause is either a unit clause or $\square$. Since S' is a result of unit propagation, it contains no unit clauses, so $C = \square$. ❏

This theorem gives a simple algorithm for checking satisfiability of a set S of Horn clauses. First, apply unit propagation to S. If the result contain the empty clause, then S is unsatisfiable, otherwise S is satisfiable. Obviously, unit propagation can be done in polynomial time, so satisfiability for sets of Horn clauses can be checked in polynomial time. Using appropriate data structures, one can prove a stronger result given below.

THEOREM 5.19 *Satisfiability of sets of Horn clauses can be checked in linear time.* ❏

```
procedure $DPLL(S)$
input: set of clauses $S$
output: $satisfiable$ or $unsatisfiable$
parameters: function $select\_literal$
begin
  $S$ := $propagate(S)$
  if $S$ is empty then return $satisfiable$
  if $S$ contains $\square$ then return $unsatisfiable$
  $L$ := $select\_literal(S)$
  if $DPLL(S \cup \{L\}) = satisfiable$
    then return $satisfiable$
    else return $DPLL(S \cup \{\tilde{L}\})$
end
```

Figure 5.6: The DPLL algorithm

5.7 DPLL

Unit propagation is a simplification rule. Unit propagation can only complement satisfiability-checking methods but cannot replace them, except for special cases, such as sets of Horn clauses. Indeed, when a set of clauses S contains no units, unit propagation will not simplify S. The best satisfiability checkers are based on a method that combines unit propagation with splitting. This method is called DPLL, after the names of its authors [Davis et al. 1962].

ALGORITHM 5.20 (DPLL) The DPLL algorithm is shown in Figure **??**. The algorithm is parametrized by a function $select\_literal$ returning a literal. The function $propagate$ applies unit propagation to S. ❏

Note that the input to this algorithm is a set of clauses. If we have to check for satisfiability a set of formulas using this algorithm, we first have to transform the set of formulas into a set of clauses using any clausal form transformation algorithm.

EXAMPLE 5.21 Let us apply the DPLL algorithm to the following set S consisting of four clauses

$$\{\neg p \lor \neg q,\ \neg p \lor q,\ p \lor \neg q,\ p \lor q\}.$$

It contains no unit clauses, so unit propagation does not change this set. Select now a literal in this set, for example $\neg p$. According to the DPLL algorithm, we first have to consider the set S augmented with the unit clause $\neg p$.

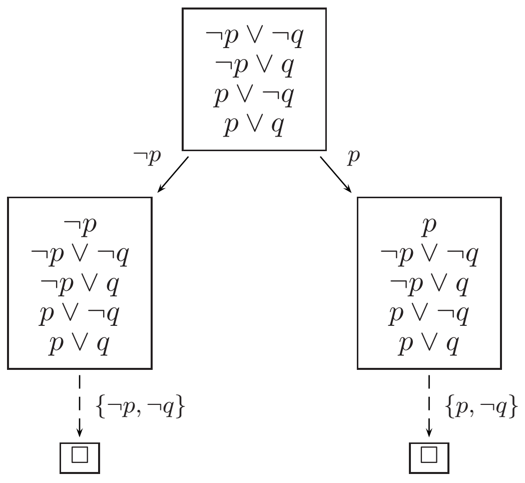

Figure 5.7: A tree obtained by DPLL

By adding $\neg p$ to S and applying unit propagation we obtain the following sequence of sets:

$$\begin{array}{ll} \{\cancel{\neg p \lor \neg q},\ \cancel{\neg p \lor q},\ p \lor \neg q,\ p \lor q,\ \cancel{\neg p}\} & \Rightarrow \\ \{\cancel{\neg q},\ q\} & \Rightarrow \\ \{\Box\}. & \end{array}$$

Therefore $S \cup \{\neg p\}$ is unsatisfiable. According to the DPLL algorithm, we now consider the set S augmented with the unit clause p. Unit propagation yields

$$\begin{array}{ll} \{\neg p \lor \neg q,\ \neg p \lor q,\ \cancel{p \lor \neg q},\ \cancel{p \lor q},\ \cancel{p}\} & \Rightarrow \\ \{\cancel{\neg q},\ q\} & \Rightarrow \\ \{\Box\}. & \end{array}$$

So $S \cup \{\neg p\}$ is unsatisfiable too, and the algorithm returns $\mathit{unsatisfiable}$. ❏

This algorithm can be illustrated by drawing a tree consisting of all clause sets considered by the algorithm, similarly to trees illustrating the splitting algorithm. A DPLL corresponding to Example 5.21 is given in Figure **??**. It contains two branches. The steps where the algorithm chooses a literal are denoted by solid lines labeled by the chosen literal. The steps where unit propagation has been applied are denoted by dashed lines labeled by the set of propagated literals, for example $\{\neg p, \neg q\}$.

When DPLL establishes that the set of clauses is satisfiable, we are normally interested in finding an interpretation satisfying the set. This can be done using the following theorem.

THEOREM 5.22 (Model Building) *Suppose that the algorithm returned "satisfiable" on a set S of clauses. Let $\mathcal{L}$ be the set of all literals used for unit propagation on the branch leading to the empty set of clauses. There exists at least one interpretation I such that*

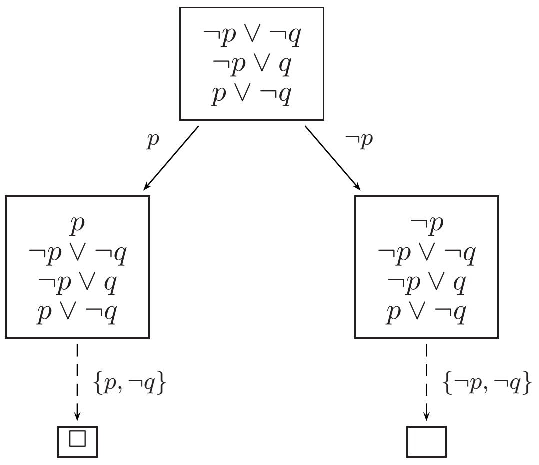

Figure 5.8: A tree obtained by DPLL for a satisfiable set of clauses

$$I(p) = \begin{cases} 1, & \textit{if } p \in \mathcal{L}; \\ 0, & \textit{if } \neg p \in \mathcal{L}. \end{cases}$$

Moreover, for every interpretation satisfying this property we have $I \vDash S$.

PROOF. Suppose that $L \in \mathcal{L}$, that is, L was used for unit propagation. Since unit propagation removes all occurrences of $\tilde{L}$, we have $\tilde{L} \notin \mathcal{L}$. Hence, there exists at least one interpretation I satisfying the conditions of the theorem. Let us show that for every such interpretation I we have $I \vDash \mathcal{L}$. To this end, notice that every clause C removed from S at some step of the algorithm contains a literal in $\mathcal{L}$, therefore $I \vDash C$. Since all clauses in S are eventually removed by the algorithm, we have $I \vDash S$. ❏

Consider an example illustrating the Model Building Theorem.

EXAMPLE 5.23 Let us remove one clause from the set of clauses of Example 5.21. We obtain the following set of three clauses: $\{\neg p \vee \neg q,\ \neg p \vee q,\ p \vee \neg q\}$. A DPLL tree for this set of clauses is shown in Figure **??**. The rightmost branch of it terminates at the empty set of clauses. The set of literals used for unit propagation on this branch is $\{\neg p, \neg q\}$. Therefore, the interpretation $\{p \mapsto 0,\ q \mapsto 0\}$ satisfies this set of clauses. ❏

5.8 Optimizations and Implementation

The DPLL method is very simple, so it is difficult to find logical optimizations for it. As a consequence, the state-of-the-art systems usually try to optimize the algorithm rather than the logic of DPLL. Nevertheless, DPLL allows for a few logical optimizations: tautology elimination and pure literal rule.

Since tautologies are valid formulas, they can obviously be removed from any set of clauses without changing the satisfiability of this set. This *tautology elimination* rule is applied to the clause set once, after the clausal form has been constructed, since the DPLL algorithm itself does not introduce new tautologies.

In Section 4.3 we introduced an optimization of the splitting algorithm based on the notion of pure atom. Let us consider this optimization in the context of the DPLL method. Clauses are very simple compared to arbitrary formulas, and polarities of occurrences of atoms are easy to calculate. Namely, an atom p has only positive occurrences in a set S of clauses if no clause in S has the form $\neg p \vee C$. Likewise, an atom p has only negative occurrences in a set S of clauses if no clause in S has the form $p \vee C$.

DEFINITION 5.24 (Pure Literal) A literal L occurring in a set of clauses S is said to be *pure* in S if S contains no clauses of the form $\tilde{L} \vee C$. *Pure literal rule* removes from a set of clauses all clauses containing a pure literal. ❏

The intuition behind the pure literal rule is simple. Assume, for example, that $\neg p$ is pure in a set of clauses S. Then by setting p to 0 we can satisfy all clauses containing the variable p. After removing these clauses from S, we obtain a set of clauses not containing p.

We will illustrate pure literal rule on the formula $\neg((p \to q) \wedge (p \wedge q \to r) \to (\neg p \to r))$ of Example 4.17. The standard CNF transformation of this formula into clausal form yields the following set of clauses

$$\{\neg p \vee q,\ \neg p \vee \neg q \vee r,\ \neg p,\ \neg r\}.$$

The literal $\neg p$ in this set of clauses is pure, so all clauses containing this literal can be removed. After this we are left with the set $\{\neg r\}$ consisting of a single clause. This clause can also be eliminated since the literal $\neg r$ is pure in it. Thus, we established satisfiability of this set of clauses using only the pure literal rule. The pure literals $\neg p, \neg r$ used in the analysis of this set of clauses give us a model of this set $\{p \mapsto 0, r \mapsto 0\}$.

Pure literal rule can be implemented in DPLL as follows. With every literal L, we keep a counter which stores the number of occurrences of this literal in the set. When the counter becomes 0, the literal $\tilde{L}$ is pure.

Implementations of DPLL use backtracking algorithms. When the function $select\_literal$ selects the next literal L, the current state is memorized, and the search continues with the set $S \cup \{L\}$. When the search is over, the memorized state is restored and the search continues with the set $S \cup \{\tilde{L}\}$. It is the backtracking that causes exponential behavior of the search procedure. A lot of efforts in the satisfiability checking community is aimed at reducing the amount of backtracking in search procedures.

It is now widely recognized that good heuristics for selecting the next literal, i.e., implementing the function $select\_literal$, are crucial for the efficient implementation of DPLL.

In addition, efficient implementation of unit propagation are important too. One very popular heuristics for selecting the next literal is called *MOMS*, which stands for *Maximum Occurrences in clauses of Minimal Size*. This heuristics is believed to cause more clauses to become unit, and thus increase the efficiency of unit propagation.

5.9 Bibliography

The DPLL procedure is sometimes mistakenly called the *Davis-Putnam procedure*, due to a similar procedure described in [Davis and Putnam 1958]. It is also sometimes called *DPLL* (Davis-Putnam-Logemann-Loveland), but in fact the procedure was described in [Davis et al. 1962].

Some techniques for obtaining small clausal forms are considered in [Nonnengart and Weidenbach 2001]. The definitional transformation was introduced in [Plaisted and Greenbaum 1986] as *structure-preserving clausal form transformation*. The naming technique is considered in a number of papers, see, e.g., [Degtyarev and Voronkov 2001].

Linear algorithms for Horn sets (see Theorem 5.19) are described in [Dowling and Gallier 1984, Itai and Makowsky 1987].

Some state-of-the-art propositional satisfiability checkers are described in [Zhang and Malik 2002, Goldberg and Novikov 2002].

Exercises

EXERCISE 5.1 Show that Algorithm 5.4 for building CNF of a formula based on rewrite rules of Figure 5.2 gives, in general, a set of clauses exponential in the size of the input formula. To this end, show a sequence of formulas $A_1, A_2, \ldots$ and a constant $c > 0$ independent of i such that the size of the CNF of A_i constructed by this algorithm is at least $|A_i|^{ci}$, where $|A_i|$ is the size of A_i. Moreover, show that the this result also holds if we require the formulas A_i contain no occurrences of $\leftrightarrow$. ❏

EXERCISE 5.2 Show that the definitional transformation gives a clause set polynomial in the size of the input formula. (In fact, it can be shown to be $O(n \log n)$.) ❏

EXERCISE 5.3 Show that in Lemma 5.8 the condition on p to be fresh is essential. ❏

EXERCISE 5.4 Apply the standard CNF transformation algorithm and the definitional transformation algorithm (both the non-optimized and optimized versions) to each of the following formulas:

$$\neg((p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)),$$
$$(p \leftrightarrow q) \leftrightarrow r.$$
❏

EXERCISE 5.5 Apply the definitional transformation to the formula

$$\neg(((p \leftrightarrow q) \leftrightarrow r) \leftrightarrow (p \leftrightarrow (q \leftrightarrow r))).$$

Apply the DPLL algorithm to the resulting set of clauses.[2] ❑

EXERCISE 5.6 (★) Show that if a polynomial-time algorithm for finding a CNF exists, then P = NP.[3] ❑

EXERCISE 5.7 Apply the DPLL algorithm to the following sets of clauses:

$$\begin{array}{llll} p \lor q \lor r, & \neg p \lor \neg q \lor \neg r, & p \lor \neg q \lor \neg r, & \neg p \lor q, \\ \neg p \lor r, & p \lor q \lor \neg r, & p \lor \neg q \lor r. & \end{array}$$

Is this set satisfiable? If yes, find a model of this set. ❑

EXERCISE 5.8 Apply the DPLL algorithm to the following sets of clauses:

$$\begin{array}{lll} p_1 \lor p_3, & \neg p_2 \lor \neg p_3, & p_1 \lor \neg p_3, \\ \neg p_1 \lor p_2, & p_1 \lor p_2 \lor \neg p_3, & p_1 \lor p_2 \lor p_3. \end{array}$$

Is this set satisfiable? If yes, find a model of this set. ❑

EXERCISE 5.9 Apply the DPLL algorithm to the following sets of clauses:

$$\begin{array}{lll} p_1 \lor \neg p_2 \lor p_3, & \neg p_1 \lor \neg p_2, & p_2 \lor p_3, \\ p_1 \lor p_3, & \neg p_1 \lor p_2 \lor \neg p_3, & p_1 \lor \neg p_2 \lor \neg p_3. \end{array}$$

Is this set satisfiable? If yes, find a model of this set. ❑

EXERCISE 5.10 The following set of clauses formalizes an instance of the pigeonhole problem for 3 pigeons and 2 holes, see Section 6.6 for more detail: it is impossible to put 3 pigeons in 2 holes so that every hole contains at most one pigeon. We will use the symbols $\{p_{ij} \mid i = 1 \dots 3, j = 1, 2\}$ to denote that pigeon i is put in hole j. The following set of 3 clauses describes that each of the pigeons is put in some hole.

$$p_{11} \lor p_{12}, \quad p_{21} \lor p_{22}, \quad p_{31} \lor p_{32}.$$

We have to formalize that every hole contains at most one pigeon. This can be done as follows: for every pair of different pigeons i_1, i_2 and every hole j, either pigeon i_1 is not put in hole j, or pigeon i_2 is not put in hole j. This gives us three the following six clauses:

$$\begin{array}{lll} \neg p_{11} \lor \neg p_{21}, & \neg p_{11} \lor \neg p_{31}, & \neg p_{21} \lor \neg p_{31}, \\ \neg p_{12} \lor \neg p_{22}, & \neg p_{12} \lor \neg p_{32}, & \neg p_{22} \lor \neg p_{32}. \end{array}$$

Show unsatisfiability of the resulting set of nine clauses using the DPLL procedure. ❑

EXERCISE 5.11 Find an algorithm for computing DNF. ❑

[2] The reader is advised not to try the standard CNF transformation on this formula, unless she has a computer program for this readily available.

[3] Our proof for this statement essentially relies upon the fact that we are looking for a CNF, not a clausal form. Polynomial-time transformations into clausal form obviously exist, the definitional transformation is one of them.

EXERCISE 5.12 Prove that satisfiability of formulas in DNF can be checked in polynomial time. ❑

EXERCISE 5.13 Suppose that a formula $\neg A$ has the following CNF:

$$\bigwedge_i \bigvee_j L_{i,j}.$$

Show that the following formula is a DNF of A:

$$\bigvee_i \bigwedge_j \tilde{L}_{i,j}.$$ ❑

EXERCISE 5.14 Find a model of the following set of clauses using the pure literal rule:

$$\begin{array}{lll} \neg p_2 \lor \neg p_3, & \neg p_1 \lor \neg p_2, & p_1 \lor p_2 \lor \neg p_3 \\ p_1 \lor \neg p_3, & \neg p_1 \lor p_2, & p_1 \lor \neg p_2 \lor \neg p_3. \end{array}$$ ❑

EXERCISE 5.15 Let A be a formula, a set of clauses S be obtained from A by applying the standard CNF transformation and p be an atom occurring in A. Then p is pure in A if and only if one of the literals $p, \neg p$ if pure in A. ❑

EXERCISE 5.16 Prove that Exercise 5.15 also holds for the optimized definitional transformation but, in general, does not hold for the definitional transformation. ❑

EXERCISE 5.17 Show that the following rewrite rule system computes DNF of a formula.

$$\begin{array}{rcl} A \leftrightarrow B & \Rightarrow & (A \land B) \lor (\neg A \land \neg B), \\ A \to B & \Rightarrow & \neg A \lor B, \\ \neg(A \land B) & \Rightarrow & \neg A \lor \neg B, \\ \neg(A \lor B) & \Rightarrow & \neg A \land \neg B, \\ \neg\neg A & \Rightarrow & A, \\ (A_1 \lor \ldots \lor A_m) \land B_1 \land \ldots \land B_n & \Rightarrow & \begin{array}{cc} (A_1 \land B_1 \land \ldots \land B_n) & \lor \\ \cdots & \lor \\ (A_m \land B_1 \land \ldots \land B_n). & \end{array} \end{array}$$

❑