Chapter 1

Boolean Satisfiability Solvers: Techniques and Extensions

Sharad Malik, Georg Weissenbacher
Princeton University

1.1 Introduction

Boolean Satisfibility (SAT) is the problem of checking if a propositional logic formula can ever evaluate to true. This problem has long enjoyed a special status in computer science. On the theoretical side, it was the first problem to be classified as being NP-complete. NP-complete problems are notorious for being hard to solve; in particular, in the worst case, the computation time of any known solution for a problem in this class increases exponentially with the size of the problem instance. On the practical side, SAT manifests itself in several important application domains such as the design and verification of hardware and software systems, as well as applications in artificial intelligence. Thus, there is strong motivation to develop practically useful SAT solvers.

However, the NP-completeness is cause for pessimism, since it is unlikely that we will be able to scale the solutions to large practical instances. While attempts to develop practically useful SAT solvers have persisted for almost half a century, for the longest time it was a largely academic exercise with little hope of seeing practical use. Fortunately, several relatively recent research developments have enabled us to tackle instances with millions pff variables and constraints – enabling SAT solvers to be effectively deployed in practical applications including in the analysis and verification of software systems.

The first part of this chapter (Section 1.3) covers the techniques used in modern SAT solvers. Moreover, it covers basic extensions such as the constructions of unsatisfiability proofs. For instances that are unsatisfiable, the proofs of unsatisfiability have been used to derive an unsatisfiable subset of constraints of the formula, referred to as the UNSAT core. The UNSAT core has seen successful applications in model checking.

The second part (Section 1.4) considers extensions of these solvers that have proved to be useful in analysis and verification. Related to the UNSAT core are the concepts of minimal correction sets and maximally satisfiable subsets. A maximally satisfiable subset of an unsatisfiable instance is a maximal subset of constraints that is satisfiable, and a minimal correction set is a minimal subset of constraints that needs to be dropped to make the formula satisfiable. Section 1.4 discusses how these concepts are related and covers algorithms to derive them.

1.2 Preliminaries

1.2.1 Propositional Logic

Notation

Let $\mathcal{V}$ be a set of n propositional logic variables and let $\mathtt{0}$ and $\mathtt{1}$ denote the elements of the Boolean domain $\mathbb{B}$. Every Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ can be expressed as a propositional logic formula F in n variables $x_1, \ldots, x_n \in \mathcal{V}$. The syntax of propositional logic formulae is provided in Figure 1.1a.

The interpretation of the logical connectives $\{-, +, \cdot, \rightarrow, \leftrightarrow, \oplus\}$ is provided in Table 1.1. We use $\equiv$ to denote logical equivalence. For brevity, we may omit $\cdot$ in conjunctions (e.g., $x_1\overline{x}_3$). An assignment $\mathcal{A}$ is a total mapping from $\mathcal{V}$ to $\mathbb{B}$, and $\mathcal{A}(x)$ refers to the value $\mathcal{A}$ assigns to x. A partial assignment $\mathcal{A}$ is a total mapping from $\mathcal{V}$ to $(\mathbb{B} \cup \mathcal{V})$ such that $\mathcal{A}(x) \in \mathbb{B}$ for all assigned variables $x \in \mathcal{V}$ and $\mathcal{A}$ is the identity function $\mathcal{A}(y) = y$ for all unassigned variables $y \in \mathcal{V}$. $\mathcal{A}$ *satisfies* a formula $F(x_1, \ldots x_n)$ iff $F(\mathcal{A}(x_1), \ldots, \mathcal{A}(x_n))$ evaluates to $\mathtt{1}$ (denoted by $\mathcal{A} \models F$). A formula F is satisfiable iff $\exists \mathcal{A} \,.\, \mathcal{A} \models F$, and unsatisfiable (inconsistent, respectively) otherwise. We use $\#\mathcal{A}_F$ to denote the number of satisfying assignments of a formula F and drop the subscript if F is clear from the context. A formula F *holds* iff $\forall \mathcal{A} \,.\, \mathcal{A} \models F$.

We use $\mathtt{Lit}_{\mathcal{V}} = \{x, \overline{x} \,|\, x \in \mathcal{V}\}$ to denote the set of literals over $\mathcal{V}$, where $\overline{x}$ is the negation of x. Given a literal $\ell \in \mathtt{Lit}_{\mathcal{V}}$, we write $\mathrm{var}(\ell)$ to denote the variable occuring in ℓ. A *cube* over $\mathcal{V}$ is a product of literals $\ell_1 \ldots \ell_m$ such that $\ell_i \in \mathtt{Lit}_{\mathcal{V}}$ and $\mathrm{var}(\ell_i) \neq \mathrm{var}(\ell_j)$ for all $i, j \in \{1..m\}$ with $i \neq j$. We write $\ell \in C$ to indicate that the literal ℓ occurs in a cube C. Given an assignment $\mathcal{A}$, we use $C_{\mathcal{A}}$ to denote the cube $\prod_{i=1}^{n} \ell_i$ where $\ell_i = x_i$ if $\mathcal{A}(x_i) = \mathtt{1}$ and $\ell_i = \overline{x}_i$ otherwise.

Conjunctive Normal Form

Figure 1.1b shows the syntax of propositional logic formulae in Conjunctive Normal Form (CNF).

x	y	$\overline{x}$	$x \cdot y$	$x + y$	$x \rightarrow y$	$x \leftrightarrow y$	$x \oplus y$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	0	0	1
1	1	0	1	1	1	1	0

Table 1.1: Definition of Propositional Logic Operators

$$\begin{array}{rcl}
\textit{formula} & ::= & \textit{formula} \cdot \textit{formula} \mid \textit{formula} + \textit{formula} \mid \\
 & & \textit{formula} \rightarrow \textit{formula} \mid \textit{formula} \leftrightarrow \textit{formula} \mid \\
 & & \textit{formula} \oplus \textit{formula} \mid \overline{\textit{formula}} \mid (\textit{formula}) \mid \textit{atom} \\
\textit{atom} & ::= & \textit{propositional identifier} \mid \textit{constant} \\
\textit{constant} & ::= & \mathsf{1} \mid \mathsf{0}
\end{array}$$

(a) Propositional Logic

$$\begin{array}{rcl}
\textit{formula} & ::= & \textit{formula} \cdot (\textit{clause}) \mid (\textit{clause}) \\
\textit{clause} & ::= & \textit{clause} + \textit{literal} \mid \textit{literal} \\
\textit{literal} & ::= & \textit{atom} \mid \overline{\textit{atom}} \\
\textit{atom} & ::= & \textit{propositional identifier}
\end{array}$$

(b) Propositional Logic in Conjunctive Normal Form

Figure 1.1: Syntax of Propositional Logic

- CNF formula: A product of sums (conjunction of clauses)

$$\prod_i \sum_j \ell_{i,j}\,, \qquad \ell_{i,j} \in \{x, \overline{x} \mid x \in \mathcal{V}\}$$

  e.g.,

$$\overline{x}_1 \cdot (x_1 + \overline{x}_2) \cdot (\overline{x}_1 + x_2) \cdot x_1$$

- Note:
  - $\sum_{\ell \in \emptyset} \ell \;\equiv\; \mathsf{0}$ (we use $\Box$ to denote the empty clause)
  - $\prod_{\ell \in \emptyset} \ell \;\equiv\; \mathsf{1}$

- Alternative (more compact) notation:

$$(\overline{x}_1)\;(x_1\,\overline{x}_2)\;(\overline{x}_1\,x_2)\;(x_1)$$

- Clauses are represented as *sets of literals*. Accordingly, $(\overline{x}_1\,x_2\,x_2)$ and $(x_1\,x_2)$ are indistinguishable from their logically equivalent counterparts $(\overline{x}_1\,x_2\,x_2)$ and $(x_2\,x_1)$, respectively. Note that this representation implicitly incorporates *factoring* (i.e., merging of unifiable literals). A formula in CNF is a set of sets of literals.

- Obtained through Tseitin transformation [Tse83]. Given a formula F in propositional logic (c.f. Figure 1.1a):

  1. Recursively replace each sub-formula $(F_1 \rhd F_2)$ of F (where $\rhd \in \{-, +, \cdot, \rightarrow, \leftrightarrow, \oplus\}$) with a fresh propositional identifier x and add the constraint $x \leftrightarrow (F_1 \rhd F_2)$.
  2. Rewrite all constraints into CNF (see Table 1.2).

  The resulting formula G in CNF is *equi-satisfiable* (i.e., $(\exists \mathcal{A} \,.\, \mathcal{A} \models F) \leftrightarrow (\exists \mathcal{A} \,.\, \mathcal{A} \models G)$) and its size is polynomial in the size of the original formula.

Example 1.2.1 *Consider the formula $y \oplus z$.*

1. *By definition of $\oplus$, $(y \oplus z) \equiv \overline{(y \leftrightarrow z)}$.*

2. *Replace $(y \leftrightarrow z)$ with x_1:*
   $\overline{x}_1 \cdot (x_1 \leftrightarrow (y \leftrightarrow z))$

3. *Replace $\overline{x}_1$ with x_2 (this step is optional, since $(\overline{x}_1)$ is already in clausal form):*
   $x_2 \cdot (x_2 \leftrightarrow \overline{x}_1) \cdot (x_1 \leftrightarrow (y \leftrightarrow z))$

4. *Rewrite according to Table 1.2:*
   $$x_2 \cdot \underbrace{(x_2 = \overline{x}_1)}_{(\overline{x_1}+\overline{x}_2)\cdot(x_1+x_2)} \cdot \underbrace{(x_1 \leftrightarrow (y \leftrightarrow z))}_{(\overline{x}_1+\overline{y}+z)\cdot(\overline{x}_1+\overline{z}+y)\cdot(\overline{y}+\overline{z}+x_1)\cdot(y+z+x_1)}$$

5. *We obtain an* equi-satisfiable *formula in CNF:*
   $x_2 \cdot (\overline{x_1} + \overline{x}_2) \cdot (x_1 + x_2) \cdot (\overline{x}_1 + \overline{y} + z) \cdot (\overline{x}_1 + \overline{z} + y) \cdot (\overline{y} + \overline{z} + x_1) \cdot (y + z + x_1)$

1.3 Boolean Satisfiability Checking: Techniques

1.3.1 Problem Definition

Definition 1.3.1 (Boolean Satisfiability Problem) *Given a propositional logic formula F, determine whether F is satisfiable.*

The Boolean Satisfiability Problem, usually referred to as SAT, is a prototypical NP-complete problem [Coo71], i.e., there is no known algorithm that efficiently solves all instances of SAT.

Using Tseitin's transformation (c.f. Section 1.2.1) any arbitrary propositional formula can be transformed into an equi-satisfiable formula in clausal form. It is therefore sufficient to focus on formulae in CNF.

There are two important sub-classes of SAT:

- *2-SAT*. Each clause of the formula contains at most 2 literals. The satisfiability of such 2-CNF formulae can be decided in polynomial time [Kro67]: Each clause (ℓ_1, ℓ_2) can be rewritten as an implication $\overline{\ell}_1 \rightarrow \ell_2$ (or $1 \rightarrow \ell_1$

Negation:

$$\begin{aligned} x \leftrightarrow \overline{y} &\equiv (x \rightarrow \overline{y}) \cdot (\overline{y} \rightarrow x) \\ &\equiv (\overline{x} + \overline{y}) \cdot (y + x) \end{aligned}$$

Disjunction:

$$\begin{aligned} x \leftrightarrow (y + z) &\equiv (y \rightarrow x) \cdot (z \rightarrow x) \cdot (x \rightarrow (y + z)) \\ &\equiv (\overline{y} + x) \cdot (\overline{z} + x) \cdot (\overline{x} + y + z) \end{aligned}$$

Conjunction:

$$\begin{aligned} x \leftrightarrow (y \cdot z) &\equiv (x \rightarrow y) \cdot (x \rightarrow z) \cdot ((y \cdot z) \rightarrow x) \\ &\equiv (\overline{x} + y) \cdot (\overline{x} + z) \cdot (\overline{(y \cdot z)} + x) \\ &\equiv (\overline{x} + y) \cdot (\overline{x} + z) \cdot (\overline{y} + \overline{z} + x) \end{aligned}$$

Equivalence:

$$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\equiv (x \rightarrow (y \leftrightarrow z)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (x \rightarrow ((y \rightarrow z) \cdot (z \rightarrow y)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (x \rightarrow (y \rightarrow z)) \cdot (x \rightarrow (z \rightarrow y)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot (((y \cdot z) + (\overline{y} \cdot \overline{z})) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot ((y \cdot z) \rightarrow x) \cdot ((\overline{y} \cdot \overline{z}) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot (\overline{y} + \overline{z} + x) \cdot (y + z + x) \end{aligned}$$

Table 1.2: Tseitin transformation [Tse83] for standard Boolean connectives

and $\overline{\ell}_1 \rightarrow \mathsf{0}$ in case of a clause (ℓ_1) with only one literal). The formula is satisfiable if the transitive closure of the implications does not yield $\mathsf{0}$. This approach effectively amounts to resolution (see Section 1.3.2).

- *3-SAT.* Each clause of the formula contains at most 3 literals. This form is relevant because any arbitrary formula in CNF can be reduced to an equi-satisfiable 3-CNF formula by means of Tseitin's transformation (Section 1.2.1).

1.3.2 Resolution Proofs

The *resolution principle* states that an assignment satisfying the clauses $C + x$ and $D + \overline{x}$ also satisfies $C \vee D$. The clauses $C + x$ and $D + \overline{x}$ are the *antecedents*, x is the *pivot*, and $C \vee D$ is the *resolvent*. Let $\mathrm{Res}(C, D, x)$ denote the resolvent of the clauses C and D with the pivot x. It is formally described below.

$$\frac{C + x \qquad D + \overline{x}}{C + D} \;\; [\mathsf{Res}]$$

Resolution corresponds to existential quantification of the pivot and subsequent quantifier elimination:

$$\begin{aligned}
&\exists x \,.\, (C + x) \;\cdot\; (D + \overline{x}) \\
\equiv\, &((C + x) \cdot (D + \overline{x}))\,[x/\mathsf{1}] + ((C + x) \cdot (D + \overline{x}))\,[x/\mathsf{0}] \\
\equiv\, &\underbrace{(C + \mathsf{1})}_{1} \cdot \underbrace{(D + \overline{\mathsf{1}})}_{D} + \underbrace{(C + \mathsf{0})}_{C} \cdot \underbrace{(D + \overline{\mathsf{0}})}_{1} \\
\equiv\, &C + D
\end{aligned}$$

($F[x/e]$ denotes the substitution of all free occurrences of x in F with the expression e.) Repeated application of the resolution rule results in a resolution proof.

Definition 1.3.2 *A* resolution proof *R is a DAG $(V_R, E_R, piv_R, \ell_R, \mathsf{s}_R)$, where V_R is a set of vertices, E_R is a set of edges, piv_R is a pivot function, ℓ_R is the clause function, and $\mathsf{s}_R \in V_R$ is the sink vertex. An* initial vertex *has in-degree 0. All other vertices are* internal *and have in-degree 2. The sink has out-degree 0. The pivot function maps internal vertices to variables. For an internal vertex v and $(v_1, v), (v_2, v) \in E_R$, $\ell_R(v) = \mathrm{Res}(\ell_R(v_1), \ell_R(v_2), piv_R(v))$.*

A resolution proof R is a refutation if $\ell_R(\mathsf{s}_R) = \square$. A refutation R is a refutation for a formula F (in CNF) if the label of each initial vertex of R is a clause of F.

Example 1.3.1 (Unit Propagation and Resolution) *Figure 1.2 shows an example of a resolution proof for the formula*

$$(\overline{x}_1) \cdot (x_1 + \overline{x}_2) \cdot (\overline{x}_1 + x_2) \cdot (x_1)\,. \tag{1.1}$$

Note that this formula is a 2-CNF formula and can therefore be solved by means of transitive closure of the corresponding implications. Equivalently, the unsatisfiability of Formula (1.1) can be established by repeated application of the unit-resolution rule*:*

$$\frac{\ell \qquad D \vee \overline{\ell}}{D} \;\; [\mathsf{URes}]$$

Here, ℓ denotes a literal over the pivot variable.

1.3.3 The Davis-Putnam Procedure

The resolution rule is sufficient to devise a complete algorithm for deciding the satisfiability of a CNF formula [Rob65].

Theorem 1.3.1 (Completeness of Propositional Resolution) *If F is an inconsistent formula in CNF, then there is a resolution refutation for F.*

$$\begin{tikzcd}
x_1\overline{x}_2 \arrow[ddrr] & & \overline{x}_1x_2 \arrow[dr] & & x_1 \arrow[dl] \\
 & & & x_2 \arrow[dl] & \\
\overline{x}_1 \arrow[dr] & & x_1 \arrow[dl] & & \\
 & \Box & & &
\end{tikzcd}$$

Figure 1.2: Resolution proof

Proof sketch. (c.f. [Bus98]) By induction over the number of variables in F. In the base case, where no variables appear in F, the formula must contain the empty clause $\Box$. For the induction step, let x be a fixed variable in F, and let F' to be the formula defined as follows:

1. For all clauses $(C + x)$ and $(D + \overline{x})$ in F, the resolvent $\text{Res}((C + x), (D + \overline{x}), x)$ is in F'.

2. Every clause C in F which contains neither x nor $\overline{x}$ is in F'.

It is clear that x does not occur in F' unless F contains trivial clauses C for which $\{x, \overline{x}\} \subseteq C$. W.l.o.g., such tautological clauses can be dropped. Then, F' is satisfiable if and only if F is, from whence the theorem follows by the induction hypothesis.

Remark Resolution is merely *refutation-complete*, i.e., while it is always possible to derive $\Box$ from an inconsistent formula, it does not enable us to derive all valid implications (we cannot deduce $(x + y)$ from (x) by means of resolution, for instance, even though the latter obviously entails the former).

The constructive proof sketch above is interesting for two reasons:

- It demonstrates that propositional resolution is complete even if we fix the order of pivots along each path in the proof, and

- it outlines a decision procedure which is known as Davis-Putnam procedure [DP60].

We refer to the algorithm presented in [DP60] as "Davis-Putnam" procedure or simply DP. The Davis-Putnam procedure comprises three rules:

1. *1-literal rule.* Whenever one of the clauses in F is a *unit clause*, i.e., contains only a single literal ℓ, then we obtain a new formula F' by

   (a) removing any instances of $\overline{\ell}$ from the other clauses, and

   (b) removing any clause containing ℓ, including the unit clause itself.

   This rule obviously subsumes unit-resolution (see Example 1.3.1).

2. *The affirmative-negative rule.* If any literal ℓ occurs *only positively* or *only negatively* in F, then remove all clauses containing ℓ. This transformation obviously preserves satisfiability.

3. *The rule for eliminating atomic formulae.* For all clauses $(C + x)$ and $(D+\overline{x})$ in F, where neither C nor D contain x or $\overline{x}$, the resolvent $\mathrm{Res}((C+x),(D+\overline{x}),x)$ is in F'. Moreover, every clause C in F which contains neither x nor $\overline{x}$ is in F'.

The last rule can make the formula increase in size significantly. However, it completely eliminates all occurrences of the atom x. The correctness of the transformation is justified by the resolution principle (see Section 1.3.2).

In practice, the resolution rule should only be applied *after* the 1-literal rule and affirmative-negative rule. The 1-literal rule is also known as *unit propagation* and lends itself to efficient implementations.

Once this option is exhausted, we face a choice of which pivot variable x to resolve on. While there is no "wrong" choice that forfeits completeness (as established in the proof of Theorem 1.3.1), a "bad" choice of a pivot may result in a significant blowup of the formula, and therefore retard the performance of the solver. We postpone the discussion of selection strategies to Section 1.3.7.

1.3.4 The Davis-Putnam-Logemann-Loveland Procedure

For realistic problems, the number of clauses generated by the DP procedure grows quickly. To avoid this explosion, Davis, Logemann, and Loveland [DLL62] suggested to replace the resolution rule with a case split. This modified algorithm is commonly referred to as DPLL procedure. It is based on the identity known as Shannon's expansion:

$$F \equiv x \cdot F[x/1] + \overline{x} \cdot F[x/0] \tag{1.2}$$

Accordingly, checking the satisfiability of a formula F can be reduced to testing $F \cdot x$ and $F \cdot \overline{x}$ separately. The subsequent application of unit propagation (the 1-literal rule, respectively) can reduce the size of these formulae significantly. This transformation, applied recursively, yields a complete decision procedure.

In practice, this split is not implemented by means of recursion but in an iterative manner (using tail recursion, respectively). We keep track of the recursive case-splits and their implications using an explicit trail. Each entry in this trail represents an assignment to a variable of F imposed by either a case split or by unit propagation. We refer to the former as *guessed* and to the latter as *implied* assignments.

Definition 1.3.3 (Clauses under Partial Assignments) *A trail represents a* partial *assignment $\mathcal{A}$ to the variables $\mathcal{V}$ of F.*

- *A clause C is* satisfied *if one or more of its literals evaluates to* 1 *under the partial assignment $\mathcal{A}$.*

Level	**Partial Assignment**	**Clauses**	**Trail**
0	–	$(\overline{x}_1\,\overline{x}_4\,x_3)(\overline{x}_3\overline{x}_2)$	
1	$\{x_1 \mapsto 1\}$	$(\overline{x}_1\,\overline{x}_4\,x_3)(\overline{x}_3\overline{x}_2)$	x_1, guessed
2	$\{x_1 \mapsto 1, x_4 \mapsto 1\}$	$(\overline{x}_1\,\overline{x}_4\,x_3)(\overline{x}_3\overline{x}_2)$	x_4, guessed
	$\{x_1 \mapsto 1, x_4 \mapsto 1, x_3 \mapsto 1\}$	$(x_3)(\overline{x}_3\overline{x}_2)$	x_3, implied
	$\{x_1 \mapsto 1, x_4 \mapsto 1, x_3 \mapsto 1, x_2 \mapsto 0\}$	$(\overline{x}_2)$	$\overline{x}_2$, implied

Table 1.3: Assignment trail for Example 1.3.2

- *A clause C is* conflicting *if all of its literals are assigned and C evaluates to* $\mathsf{0}$ *under $\mathcal{A}$.*
- *A clause C becomes* unit *under a partial assignment if all but one of its literals are assigned but C is not satisfied. As such, C gives rise to an implied assignment. In this case, we say that C is the* antecedent *of the implied assignment.*
- *In all other cases, we say that the clause C is* unresolved*.*

Example 1.3.2 *Consider the clauses*

$$C_1 \equiv (\overline{x}_1\,\overline{x}_4\,x_3) \qquad \textit{and} \qquad C_2 \equiv (\overline{x}_3\,\overline{x}_2)\,.$$

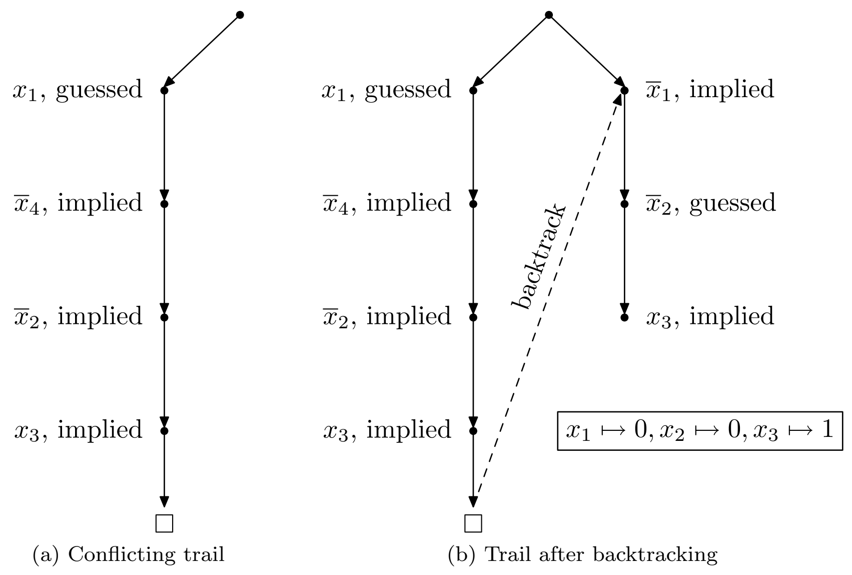
Table 1.3 shows a possible trail for this instance. Initially, neither of the clauses is unit, forcing us to guess *an assignment for one of the variables and thus to introduce a new decision level. We choose to explore the branch in which x_1 is assigned* $\mathsf{1}$ *first. the first entry in the trail, the literal x_1, represents this decision. Neither of the clauses is unit under this assignment; we decide to assign x_4. The clause C_1 is unit under the partial assignment $\{x_1 \mapsto \mathsf{1}, x_4 \mapsto \mathsf{1}\}$ and implies the assignment $x_3 \mapsto \mathsf{1}$ (note that we mark the assignment as "implied" in the trail). This assignment, in turn, makes C_2 unit, imposing the assignment $x_2 \mapsto \mathsf{0}$. The resulting assignment satisfies C_1 as well as C_2.*

A trail may lead to a dead end, i.e., result in a conflicting clause, in which case we have to explore the alternative branch of one of the case splits previously made. This corresponds to reverting one of the decisions or *backtracking*, respectively.

Example 1.3.3 (Backtracking) *Consider the set of clauses*

$$C_1 \equiv (\mathtt{x}_2\,x_3) \quad C_2 \equiv (\overline{x}_1\overline{x}_4) \quad C_3 \equiv (\overline{x}_2 x_4) \quad C_4 \equiv (\overline{x}_1 x_2 \overline{\mathtt{x}}_3)\,.$$

Figure 1.3a shows a trail that leads to a conflict (assignments are represented as literals, c.f. Section 1.2.1). Clause C_4 is conflicting under the given assignment. The last (and only) guessed assignment on the given trail is $x_1 \mapsto \mathsf{1}$. Accordingly, we backtrack *to this most recent decision (dropping all implications up to this point) and revert it to $x_1 \mapsto \mathsf{0}$ (see Figure 1.3b). We tag the*



x_1, guessed
$\overline{x}_4$, implied
$\overline{x}_2$, implied
x_3, implied
(a) Conflicting trail
x_1, guessed
$\overline{x}_4$, implied
$\overline{x}_2$, implied
x_3, implied
backtrack
$\overline{x}_1$, implied
$\overline{x}_2$, guessed
x_3, implied
$x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 1$
(b) Trail after backtracking

Figure 1.3: Backtracking

assignment $x_1 \mapsto 0$ *as* implied, *since* $x_1 \mapsto 1$ *led to a conflict. Thus, we prevent that this assignment is reverted back to* $x_1 \mapsto 1$ *at a later point in time, which would lead to a loop.*

When backtracking enough times, the search algorithm always yields a conflicting clause or a satisfying assignment and eventually exhausts all branches. However, always reverting the last decision made is not necessarily the best strategy, as the following example from [Har09] shows.

Example 1.3.4 *Consider the clauses* $C_1 \equiv (\overline{x}_1\, \overline{x}_n\, x_{n+1})$ *and* $C_2 \equiv (\overline{x}_1\, \overline{x}_n\, \overline{x}_{n+1})$ *as part of an unsatisfiable formula* F. *Exploring the trail* $x_1\, x_2 \cdots x_{n-1}\, x_n$ *leads to a conflict forcing us to backtrack and explore the trail* $x_1\, x_2 \cdots x_{n-1}\overline{x}_n$. *Since* F *is unsatisfiable, we are eventually (perhaps after further case-splits) forced to backtrack. Unfortunately, each time we change one of the assignments to* $x_1, \ldots, x_{n-1}$, *we will unnecessarily explore the case in which* x_n *is* 1 *again.*

The next section introduces *conflict clauses* as a means to prevent the repeated exploration of infeasible assignments.

1.3.5 Conflict-Driven Clause Learning

In their solver Grasp, João Marques-Silva and Karen Sakallah [MSS96] introduced a novel mechanism to analyse the conflicts encountered during the search for a satisfying assignment.

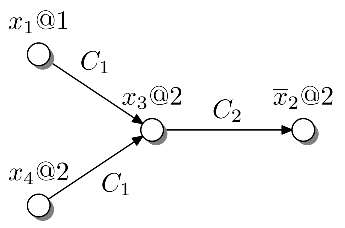


Figure 1.4: An implication graph for the trail in Table 1.3

First, they partition trails into decision levels according to recursion depth of the case-splits performed.

Definition 1.3.4 (Decision Levels) *Each recursive application of the splitting rule gives rise to a new* decision level*. If a variable x is assigned* 1 *(owing to either a case split or unit propagation) at decision level n, we write x@n. Conversely, $\overline{x}$@n denotes an assignment of* 0 *to x at decision level n.*

Secondly, the implications in a trail are represented using an *implication graph.*

Definition 1.3.5 (Implication Graph) *An implication graph is a labelled directed acyclic graph $G(V, E)$.*

- *The nodes V represent assignments to variables. Each $v \in V$ is labelled with a literal and its corresponding decision level.*
- *Each edge in an implication graph represents an implication deriving from a clause that is unit under the current partial assignment. Accordingly, edges are labelled with the respective antecedent clauses of the assignment the edge points to.*
- *An implication graph may contain a single* conflict node*, whose incoming edges are labelled with the corresponding conflicting clause.*

Example 1.3.5 (Implication Graph for Example 1.3.2) *Figure 1.4 shows the implication graph for the trail presented in Example 1.3.2.*

If the implication graph contains a conflict, we can use it to determine the decisions that led to this conflict. Moreover, it enables us to derive a *conflict clause*, which, if added to the original formula, prevents the algorithm from repeating the decision(s) that led to the conflict.

Example 1.3.6 (Implication Graph with Conflict) *Figure 1.5 shows the implication graph for a trail emanating from the decision $x_1 \mapsto 1$ for the clauses*

$$C_1 \equiv (x_2\, x_3),\ C_2 \equiv (\overline{x}_1 \overline{x}_4),\ C_3 \equiv (\overline{x}_2 x_4),\ C_4 \equiv (\overline{x}_1 x_2 \overline{x}_3)\,.$$

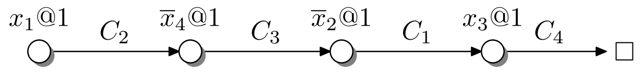


Figure 1.5: An implication graph with a conflict

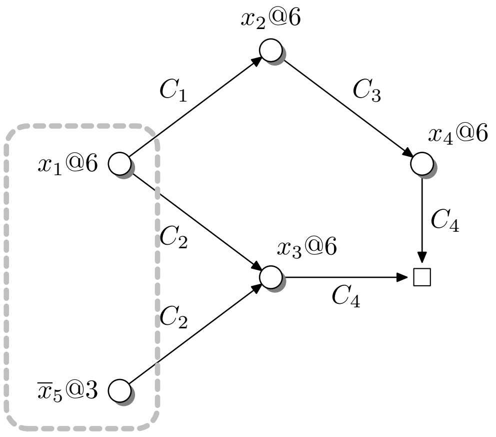


Figure 1.6: An implication graph for Example 1.3.7 (presented in [KS08])

The final node in the graph represents a conflict. The initial node of the graph is labelled with the decision that causes the conflict. Adding the unit clause $(\overline{x}_1)$ *to the original clauses guarantees that the decision* x_1 *will never be repeated.*

Example 1.3.7 *Figure 1.6 shows a partial implication graph for the clauses*

$$C_1 \equiv (\overline{x}_1 x_2), \quad C_2 \equiv (\overline{x}_1 x_3 x_5), \quad C_3 \equiv (\overline{x}_2 x_4), \quad \textit{and} \quad C_4 \equiv (\overline{x}_3 \overline{x}_4)$$

and the decisions x_1@6 *and* $\overline{x}_5$@3*. Using the implication graph, the decisions responsible for the conflict can be easily determined. Adding the conflict clause* $(\overline{x}_1 + x_5)$ *to the original formula rules out that this very combination of assignments is explored again.*

The advantage of *conflict clauses* over simple backtracking becomes clear when we revisit Example 1.3.4. Using an implication graph, we can quickly determine the assignments x_1@1 and x_n@m which caused a conflict for either $C_1 \equiv (\overline{x}_1\, \overline{x}_n\, x_{n+1})$ or $C_2 \equiv (\overline{x}_1\, \overline{x}_n\, \overline{x}_{n+1})$. The conflict clause $(\overline{x}_1 + \overline{x}_n)$ eliminates this combination, pruning a large fraction of the search space which simple backtracking would have otherwise explored.

After adding a conflict clause, at least some of the decisions involved in the conflict need to be reverted (otherwise, the trail remains inconsistent with the clauses). Changing an assignment in the trail might invalidate all subsequently made decisions. Therefore, if we *backtrack* to a certain decision level n, we discard all decisions made at a level higher than n. It is clear that, of all decisions contributing to the conflict clause, we have to at least revert the

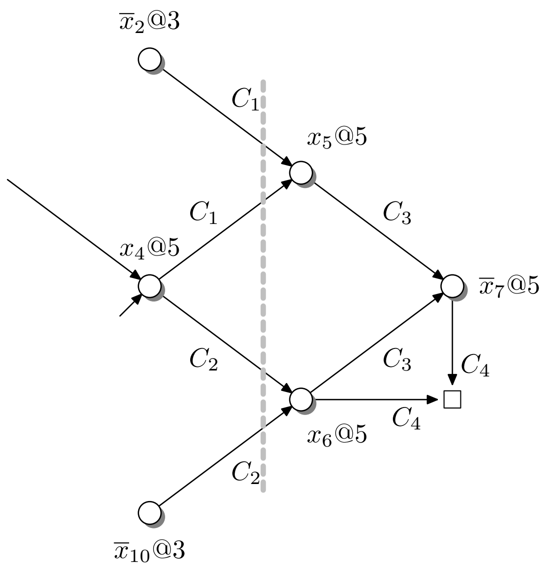


Figure 1.7: Conflict analysis and resolution

one associated with the *current* decision level (x_1@6 in Example 1.3.7, for instance). The *conflict-driven backtracking strategy* suggests to backtrack to the *second most recent decision level in the conflict clause* [MZM+01] (level 3 in Example 1.3.7). This strategy has a compelling advantage: The blocking clause is *unit* (or *assertive*) under the resulting partial assignment. For instance, $(\overline{x}_1+x_5)$ in Example 1.3.7 immediately implies $\overline{x}_1$ in this scenario.

1.3.6 Conflict Clauses and Resolution

Clause learning with conflict analysis does not impair the completeness of the search algorithm: even if the learnt clauses are dropped at a later point during the search, the trail guarantees that the solver never repeatedly enters a decision level with the same partial assignment.

We show the correctness of clause learning by demonstrating that each conflict clause is implied by the original formula.

Example 1.3.8 (Conflict Clauses and Resolution) *Figure 1.7 shows a partial implication graph for the clauses*

$$C_1 \equiv (\overline{x}_4\, x_2\, x_5) \quad C_2 \equiv (\overline{x}_4\, x_{10}\, x_6) \quad C_3 \equiv (\overline{x}_5\, \overline{x}_6\, \overline{x}_7) \quad C_4 \equiv (\overline{x}_6\, x_7)\ .$$

The conflicting clause in this example is C_4. The immediate cause for the conflict are assignments x_6@5 and $\overline{x}_7$@5 to the literals $\overline{x}_6$ and x_7 of the clause C_4. These literals are implied by the clauses C_3 and C_2, respectively. Clearly, C_3 and C_4 (and C_2 and C_4) do not agree on the assignment of x_7 (and x_6, *respectively). Accordingly, if we construct the* resolvent *of C_3 and C_4 for the pivot x_7, we obtain a clause C_5:*

$$C_5 \quad \equiv \quad \text{Res}(C_4, C_3, x_7) \quad \equiv \quad (\overline{x}_5\, \overline{x}_6)$$

While C_5 is certainly conflicting under the current partial assignment, we cannot use it as a conflict clause: both x_5 and x_6 are assigned at decision level 5 and therefore C_5 is not assertive after backtracking.

As previously mentioned, C_2 is the antecedent of x_6, and by a similar resolution step as before we obtain

$$C_6 \quad \equiv \quad \text{Res}(C_5, C_2, x_6) \quad \equiv \quad (\overline{x}_4\, \overline{x}_5\, x_{10})\ .$$

Again, x_4 as well as x_5 are assigned at decision level 5. The clause C_1 is the antecedent of x_5, and we execute a final resolution step:

$$C_7 \quad \equiv \quad \text{Res}(C_6, C_1, x_5) \quad \equiv \quad (x_2\, \overline{x}_4\, x_{10})$$

The resulting clause $(x_2\, \overline{x}_4\, x_{10})$ has the virtue of containing only one literal which is assigned at decision level 5 while still conflicting with the current partial assignment. Accordingly, if we backtrack to a decision level below 5, C_7 becomes assertive, forcing the solver to flip x_4. Therefore, it is admissible to choose C_7 as conflict clause.

We observe in Example 1.3.8 that it is possible to derive a conflict clause from the antecedents in the implication graph by means of resolution. These antecedents might in turn be conflict clauses. However, by induction, each conflict clause is implied by the original formula. Formal arguments establishing the completeness and correctness of clause learning and conflict analysis are provided in [MS95, MS99, Zha03].

The following example demonstrates that, in general, there is a choice of assertive conflict clauses.

Example 1.3.9 *Consider the partial implication graph in Figure 1.8. Figure 1.9 shows three possible cuts that separate the decisions causing the conflict from the conflicting node. This results in three candidates for conflict clauses:*

1. $C_7 \equiv (x_{10}\, \overline{x}_1\, x_9\, x_{11})$
2. $C_8 \equiv (x_{10}\, \overline{x}_4\, x_{11})$
3. $C_9 \equiv (x_{10}\, \overline{x}_2\, \overline{x}_3\, x_{11})$

We can dismiss the last clause, since it fails to be assertive after backtracking. The clauses $(x_{10}\, \overline{x}_1\, x_9\, x_{11})$ and $(x_{10}\, \overline{x}_4\, x_{11})$, however, are viable candidates for a conflict clause.

The distinguishing property of clauses C_7 and C_8 when compared to clause C_9 in Example 1.3.9 is that the former two clauses contain only one literal assigned at the current decision level. This literal corresponds to a *unique implication point* (UIP).

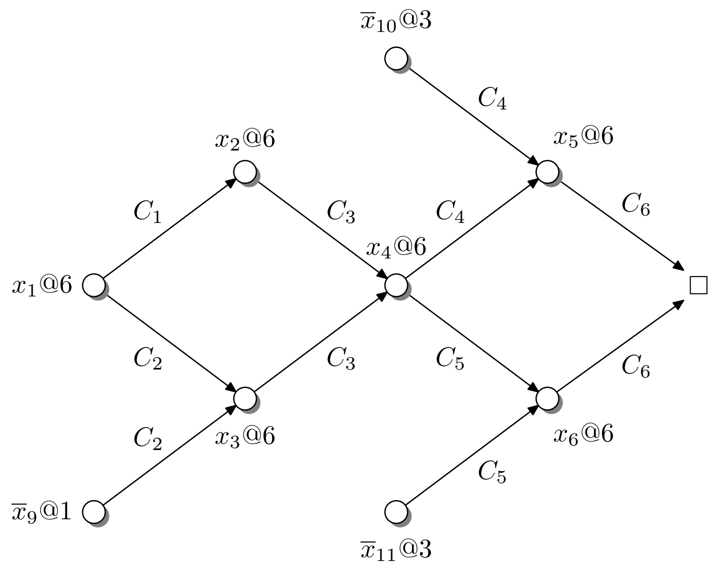


Figure 1.8: An implication graph with two *unique implication points*

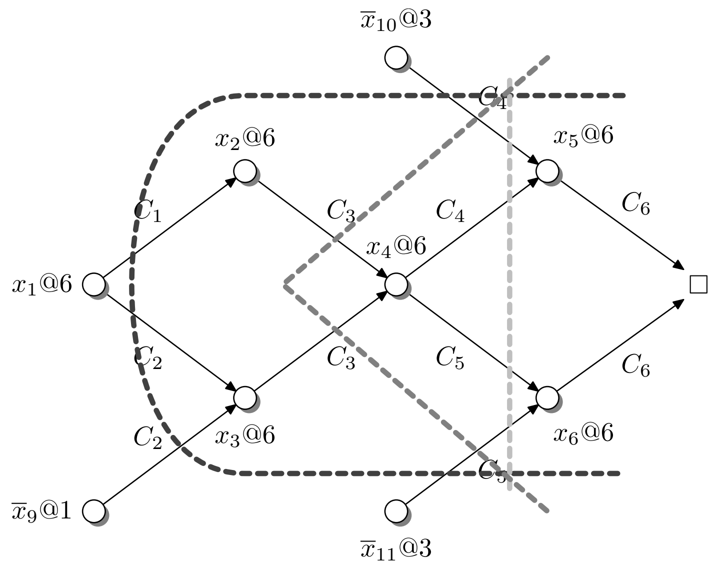


Figure 1.9: Possible cuts separating decision variables from the conflicting clause

```
① If conflict at decision level 0 → Unsat
② Repeat:
    ❶ if all variables assigned return Sat
    ❷ Make decision
    ❸ Propagate constraints
    ❹ No conflict? Go to ❶
    ❺ If decision level = 0 return Unsat
    ❻ Analyse conflict
    ❼ Add conflict clause
    ❽ Backtrack and go to ❸
```

Figure 1.10: The DPLL algorithm with clause learning

Definition 1.3.6 (Unique Implication Point) *A unique implication point is any node (other than the conflict node) in the partial conflict graph which is on* all paths *from the decision node*[1] *to the conflict node of the current decision level.*

Accordingly, we can stop searching for a conflict clause (which is done by means of resolution) once we reach a unique implication point. But which UIP should we choose? We will base our choice on the following property of the conflict clause corresponding to the UIP *closest* to the conflict (referred to as the *first* UIP): by construction, the conflict clause induced by the first UIP *subsumes* any other conflict clause except for the asserting literal. For instance, in Example 1.3.9, $C_7 \equiv (x_{10}\,\overline{x}_1\,x_9\,x_{11})$ contains all literals that occur in $C_8 \equiv (x_{10}\,\overline{x}_4\,x_{11})$, except for the literal $\overline{x}_4$ which was assigned at decision level 6. Therefore, choosing C_8 as conflict clause has the following advantages:

1. The conflict clause C_8 is smaller than C_7, making it a more likely candidate for unit implications at a later point in the search algorithm.

2. Stopping at the first UIP has the lowest computational cost.

3. The second most recent decision level in the clause C_8 is at least as low as in any other conflict clause, which forces the solver to backtrack to a lower decision level.

The "first UIP" strategy is implemented in Chaff [MZM+01], whereas Grasp [MSS96], in contrast, learns clauses at all UIPs.

Figure 1.10 shows the complete DPLL algorithm with clause learning.

[1]The decision node of the current decision level is a unique implication point by definition.

1.3.7 Decision Heuristics

Step ②.❷ in Figure 1.10 leaves the question of which variable to assign open. As we know from Section 1.3.3, this choice has no impact on the completeness of the search algorithm. It has, however, a significant impact on the performance of the solver, since this choice is instrumental in pruning the search space. [MS99] provides an overview over heuristics for choosing decision variables.

Dynamic Largest Individual Sum

Choose assignment s.t. number of satisfied clauses is maximised

- $p_{\mathtt{x}}$ . . . # of unresolved clauses containing $\mathtt{x}$
- $n_{\mathtt{x}}$ . . . # of unresolved clauses containing $\overline{\mathtt{x}}$
- Let $\mathtt{x}$ be variable for which $p_{\mathtt{x}}$ is maximal
- Let $\mathtt{y}$ be variable for which $n_{\mathtt{y}}$ is maximal
- If $p_{\mathtt{x}} > n_{\mathtt{y}}$ choose $\mathtt{x} \mapsto 1$
- Otherwise, choose $\mathtt{y} \mapsto 0$

Disadvantage: High overhead

Jeroslow-Wang

Assign high weight to variables occurring in short clauses [JW90]

- For each literal ℓ in F:

$$J(\ell) = \sum_{c \in F \text{ } s.t. \text{ } \ell \in c} 2^{-|c|}$$

- Exponentially higher weight to literals in short clauses
- Choose (unassigned) literal ℓ that maximises $J(\ell)$ and $\ell \mapsto 0$
- Weight updated dynamically (whenever conflict clause added)

Variable State Independent Decaying Sum (VSIDS)

Favour literals in recently added conflict clauses

- Each literal has counter initialised to 0
- When clause is added, literals in clause are *boosted*
- Periodically, all counters divided by constant
- Choose unassigned literal with highest counter

- Implemented in CHAFF [MZM+01]
  - Maintain list of unassigned literals sorted by counter
  - Update list when adding conflict clauses
  - Decision in $O(1)$
- Improved performance by order of magnitude
- Enforces *local* search

Representing the counter using integer variables leads to a large number of ties. MINISAT avoids this problem by using a floating point number to represent the weight [ES04]. Another possible (but significantly more complex) strategy is to concentrate *only* on unresolved conflicts by maintaining a stack of conflict clauses [GN02].

1.3.8 Unsatisfiable Cores

Given an unsatisfiable instance F, we can use the techniques described in Section 1.3.6 to construct a resolution refutation (see Definition 1.3.2 in Section 1.3.2). Intuitively, such a refutation identifies a *reason* for the inconsistency of the clauses in F. The clauses at the leaves of a resolution refutation are a subset of the clauses of F. By construction, the conjunction of these clauses is unsatisfiable.

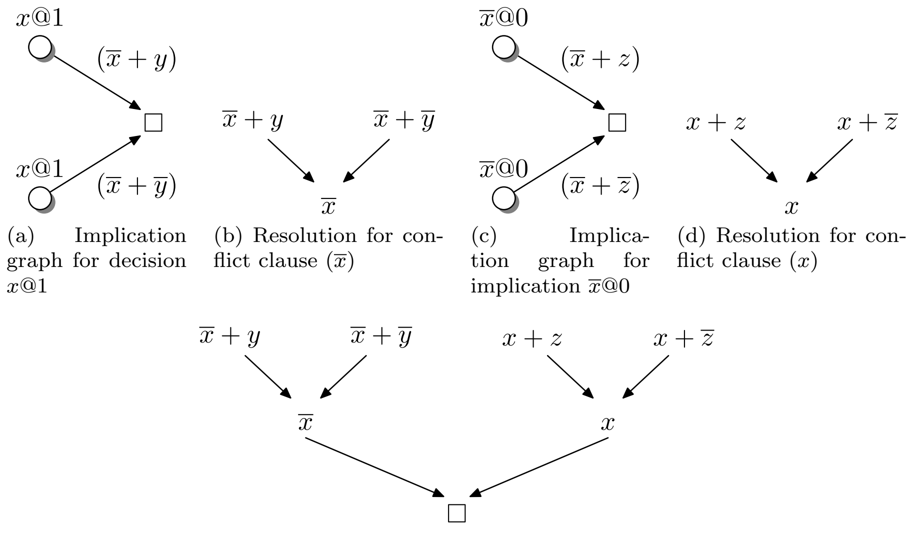
Definition 1.3.7 (Unsatisfiable Core) *Given an unsatisfiable formula $F \equiv \prod_{i=1}^{n} C_i$, any unsatisfiable subset of the set of clauses of F is an unsatisfiable core.*

- An unsatisfiable core of a formula is not necessarily unique.
- An unsatisfiable core is minimal if removing any clause from the core makes the remaining set of clauses satisfiable.

Example 1.3.10 (Constructing Unsatisfiable Cores) *Consider the following formula in conjunctive normal form:*

$$(\overline{x} + y) \cdot (\overline{x} + \overline{y}) \cdot (x + z) \cdot (x + \overline{z}) \cdot (z + y + \overline{x})$$

The problem instance does not contain unit literals, so the satisfiability solver is forced to make a decision. The VSIDS heuristic (see Section 1.3.7) assigns the highest priority to the literal $\overline{x}$. Accordingly, the solver assigns $x \mapsto 1$. This decision immediately yields a conflict, as depicted in Figure 1.11a. Accordingly, the solver derives a conflict clause (x) – the justifying resolution step is shown in Figure 1.11b. The conflict clause (x) forces the solver to assign $x \mapsto 1$ at decision level zero (x@0). Again, this leads to a conflict (see Figure 1.11c). The corresponding conflict clause is (x). This time, however, the conflict occurs at decision level zero and the satisfiability solver determines that the instance is



(a) Implication graph for decision $x@1$

(b) Resolution for conflict clause $(\overline{x})$

(c) Implication graph for implication $\overline{x}@0$

(d) Resolution for conflict clause (x)

(e) Final resolution proof

Figure 1.11: Construction of a resolution proof

unsatisfiable. The SAT solver finalises the resolution proof by resolving $(\overline{x})$ *and* (x) *(see Figure 1.11e).*

The unsatisfiable core

$$\{\quad (\overline{x}+y),\ (\overline{x}+\overline{y}),\ (x+z),\ (x+\overline{z})\quad\}$$

can be easily extracted from the resolution proof in Figure 1.11e. The clause $(z+y+\overline{x})$ *did not contribute to the contradiction and is therefore not contained in the core.*

Resolution proofs and unsatisfiable cores have applications in hardware verification [McM03].

Definition 1.3.8 (Minimal and Minimum Unsatisfiable Cores) *Let* UC *be an unsatisfiable core of the formula* F *(i.e., a set of clauses* $UC \subseteq F$ *such that* $\bigwedge\{C|C \in UC\} \rightarrow \mathsf{0}$*). The unsatisfiable core* UC *is* minimal *if removing any one of its clauses* C_i *leaves the conjunction of the remaining clauses* $UC \setminus C_i$ *satisfiable. An unsatisfiable core is* minimum *if the original formula does not contain an unsatisfiable core* UC_2 *such that* $|UC_2| < |UC|$.

1.3.9 Incremental Satisfiability Solving

Many applications of SAT solvers require solving a sequence of similar instances which share a large number of clauses. Incremental satisfiability solvers [Str01, KSW01] support the reuse of learnt clauses in subsequent calls to the SAT solver when only a fraction of the clauses of the original problem have changed. To this end, an incremental solver drops all learnt clauses and reverts all decisions that derive from clauses that are part of the original instance but not of the subsequent related problem.

1.3.10 Pre-processing Formulae

This section covers pre-processing techniques presented in [EB05] which enable us to reduce the size of the formula either before passing it to a satisfiability checker or during the search process.

Subsumption

A clause C_1 is said to subsume a clause C_2 if $C_1 \subseteq C_2$, i.e., all literals in C_1 also occur in C_2. If formula in CNF contains two clauses C_1 and C_2 such that C_1 subsumes C_2, then C_2 can be discarded. This is justified by the fact that, given a resolution proof, we can replace any occurrence of a clause C_2 by a clause C_1 which subsumes C_2 without invalidating the correctness of the proof. In fact, such a modification typically enables a reduction of the size of the proof [BIFH$^+$11].

Self-subsuming Resolution

Even though initial instance does not necessarily contain clauses subsuming others, such clauses may materialise during the search process. Eén and Biere [EB05] observes that formulae in CNF often contain clauses $(x+C_1)$ which *almost* subsume clauses $(\overline{x}+C_2)$ (where $C_1 \subseteq C_2$). After one resolution step we obtain the clause $\mathrm{Res}((x+C_1),(\overline{x}+C_2)) = C_2$, which subsumes $(\overline{x}+C_2)$. Accordingly, the clause $(\overline{x}+C_2)$ can be dropped after resolution. Eén and Biere dubbed this simplification rule *self-subsuming resolution*.

Efficient data structures for implementing (self-)subsumption are presented in [EB05].

Variable Elimination by Substitution

Formulae that are encoded in CNF using the transformation introduced in Section 1.2.1 (or a similar approach) typically contain a large number of *functionally dependent* variables, namely the fresh variables introduced to represent terms (or gate outputs, respectively). In the formula in Example 1.2.1, for instance, the value of the variable x_1 is completely determined by the values of y and z:

$$\underbrace{(x_1 \leftrightarrow (y \leftrightarrow z))}_{(\overline{x}_1+\overline{y}+z)\cdot(\overline{x}_1+\overline{z}+y)\cdot(\overline{y}+\overline{z}+x_1)\cdot(y+z+x_1)}$$

The algorithms previously presented are oblivious to this structural property and therefore fail to exploit it to restrict the set of decision variables to the functionally *independent* variables.

Eén and Biere [EB05] presents an approach that eliminates dependent variables by substitution. First, note that the auxiliary variable x_1 can be eliminated using the *rule for eliminating atomic formulae* introduced in Section 1.3.3. The application of this rule amounts to variable elimination by means of resolution. Given a set S of clauses all of which contain x, we can partition S into clauses containing x and clauses containing $\overline{x}$. Let $S_x \overset{\text{def}}{=} \{C \,|\, C \in S, x \in C\}$ and $S_{\overline{x}} \overset{\text{def}}{=} \{C \,|\, C \in S, \overline{x} \in C\}$. Abusing the notation we introduced in Section 1.3.2, we define

$$\text{Res}(S_x, S_{\overline{x}}, x) \overset{\text{def}}{=} \{\text{Res}(C_x, C_{\overline{x}}, x) \,|\, C_x \in S_x, C_{\overline{x}} \in S_{\overline{x}}\}\,.$$

A clause is trivial if it contains a literal and its negation. We observe that the pairwise resolution of the clauses corresponding to a definition of x introduced by the Tseitin transformation (see Table 1.2) yields *only* trivial clauses. We demonstrate this for the definition $x_1 \leftrightarrow (y \leftrightarrow z)$ introduced in Example 1.2.1. Let

$$G \overset{\text{def}}{=} \{(\overline{x}_1 + \overline{y} + z), (\overline{x}_1 + \overline{z} + y), (\overline{y} + \overline{z} + x_1), (y + z + x_1)\}$$

denote the set of clauses introduced by the transformation. Splitting G as suggested above yields

$$G_{\overline{x}} = \{(\overline{x}_1 + \overline{y} + z), (\overline{x}_1 + \overline{z} + y)\} \qquad G_x = \{(\overline{y} + \overline{z} + x_1), (y + z + x_1)\}\,,$$

and we obtain

$$\text{Res}(G_x, G_{\overline{x}}, x) = \{(\overline{y} + z + \overline{z}), (\overline{y} + z + y), (\overline{z} + y + \overline{y}), (\overline{z} + y + z)\}\,.$$

The reader may verify that this holds for all transformations presented in Table 1.2. Accordingly, given a set of clauses S (all of which contain x) and the definition $G \subseteq S$ of x, we can safely omit the resolution steps $\text{Res}(G_x, G_{\overline{x}}, x)$. Let $R = S \setminus G$ be the *remaining* clauses that are not part of the definition of x. Then one can partition $\text{Res}(S_x, S_{\overline{x}}, x)$ into

$$\underbrace{\text{Res}(R_x, G_{\overline{x}}, x) \;\cdot\; \text{Res}(G_x, R_{\overline{x}}, x)}_{S''} \;\cdot\; \underbrace{\text{Res}(G_x, G_{\overline{x}}, x)}_{G'} \;\cdot\; \underbrace{\text{Res}(R_x, R_{\overline{x}}, x)}_{R'}\,.$$

In our example, G_x and $G_{\overline{x}}$ encode $x + \overline{(y \leftrightarrow z)}$ (i.e., $\overline{x} \rightarrow \overline{(y \leftrightarrow z)}$) and $\overline{x} + (y \leftrightarrow z)$, respectively. Accordingly, $\text{Res}(R_x, G_{\overline{x}}, x)$ can be interpreted as *substitution* of $(y \leftrightarrow z)$ for x in R_x (and similarly for $\text{Res}(G_x, R_{\overline{x}}, x)$). As a consequence, R' can be derived from S'' in a single hyper-resolution step (or a sequence of resolution steps) [GOMS04]. It is therefore admissible to replace S with S''.

Example 1.3.11 *Consider the CNF instance*

$$\underbrace{(x_1 + u)}_{R_{x_1}} \cdot \underbrace{(\overline{x}_1 + v)}_{R_{\overline{x}_1}} \cdot \underbrace{(\overline{x}_1 + \overline{y} + z) \;\cdot\; (\overline{x}_1 + \overline{z} + y)}_{G_{\overline{x}_1}} \cdot \underbrace{(\overline{y} + \overline{z} + x_1) \;\cdot\; (y + z + x_1)}_{G_{x_1}}\,.$$

We obtain

$$S'' \quad \equiv \quad (u + \overline{y} + z) \;\cdot\; (u + \overline{z} + y) \;\cdot\; (v + \overline{y} + \overline{z}) \;\cdot\; (v + y + z)\,,$$

allowing us to reduce the size of the original formula by two clauses.

1.4 Boolean Satisfiability Checking: Extensions

1.4.1 All-SAT

Given a satisfiable formula, the algorithms presented in Section 1.3 provide a single satisfying assignment. Some applications, however, require us to enumerate *all* satisfiable assignments of a formula [McM02]. It is easy to see that solving this problem is at least as hard as the Boolean satisfiability problem (Definition 1.3.1). In fact, the problem of determining the number of satisfying assignments of a formula is a prominent representative of the complexity class #P (see, for instance, [AB09]).

In practice, the problem can be tractable for certain instances, even though no polynomial algorithm is known. We can force the SAT solver to enumerate all satisfying assignments by subsequently *blocking* all assignments previously found. As explained in Section 1.2.1, any satisfying assignment $\mathcal{A}$ of a formula F can be represented as a *cube* over the variables of F. The negation of such a cube is a clause (by De Morgan's theorem). Adding this clause C to F effectively *blocks* the assignment, since C is clearly in conflict with the current assignment. C is therefore called a *blocking clause*.

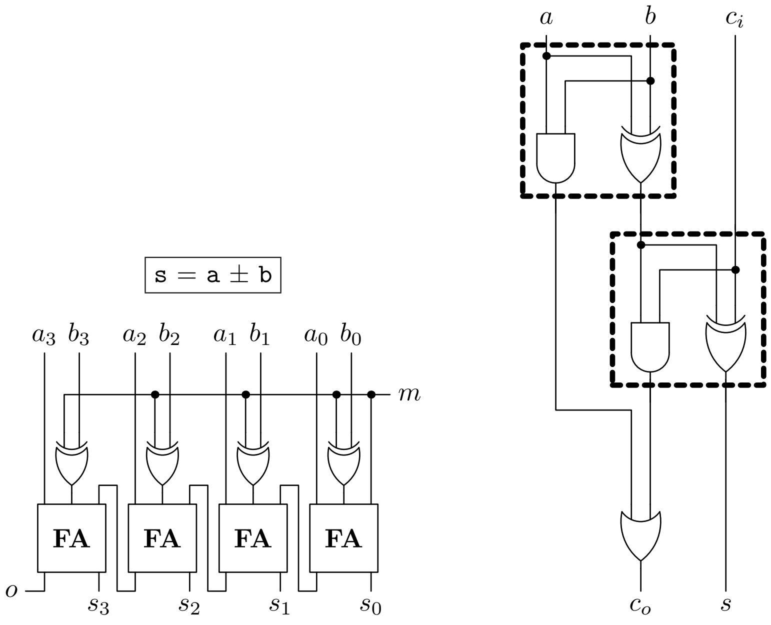
While we can take advantage of incremental satisfiability checking algorithms (c.f. Section 1.3.9), the size of the formula augmented with blocking clauses grows quickly. Moreover, blocking clauses which contain *all* variables of the original instance are less likely to become unit. Therefore, it is desirable to reduce the size of the blocking clause [McM02], i.e., to construct a smaller clause which still blocks the assignment $\mathcal{A}$. One possibility is to block the *decisions* that led to the current assignment (this information can be extracted from the trail described in Section 1.3.4). Let D be the cube representing these decisions. Clearly, $F \cdot D \leftrightarrow C_{\mathcal{A}}$, i.e., the decisions, in conjunction with the original formula, imply the single and unique assignment $\mathcal{A}$ (and *vice versa*). Moreover, $\overline{D} \rightarrow \overline{C_{\mathcal{A}}}$. Therefore $\overline{D}$ is a viable candidate for blocking $\mathcal{A}$.

An example application of All-SAT is presented in Example 1.4.7.

1.4.2 Cardinality Constraints

Satisfiability solvers are designed to work in the Boolean domain and do not support numeric reasoning *per se*. There is a number of applications for which it is desirable, however, to have at least rudimentary support for arithmetic over bounded domains. A common approach is to represent binary numbers using the two's complement system and to encode arithmetic operations using their corresponding circuit representation. Figure 1.12 shows the encoding of addition/subtraction as a ripple-carry-adder (Figure 1.12a), implemented as a chain of full adders (Figure 1.12b). This technique is known as eager bit-flattening. We refer the reader to [KS08] for a more detailed treatment of this topic.

Cardinality constraints are a common application of numerical constraints. Given a set $\{\ell_1, \ldots, \ell_n\}$ of literals, a cardinality constraint $((\sum_i \ell_i) \leq k)$ rules



(a) Ripple carry adder (b) Full adder

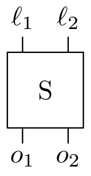
Figure 1.12: Encoding addition and subtraction in propositional logic

out all assignments in which more than k of these literals evaluate to 1 (here, $\sum$ denotes the arithmetic sum and not the logical or operator). This constraint can technically be encoded by constructing a circuit that computes $k-(\sum_i \ell_i)$ (using a chain of adder-subtractors as shown in Figure 1.12) and checking for arithmetic underflow. Such an encoding, however, introduces chains of exclusive-or gates, which pose a challenge to contemporary satisfiability checkers.

Figure 1.13 shows a *sorting network* for two literals, an alternative way of encoding the constraint $((\sum_i \ell_i) \leq k)$ (where $i = 2$ in Figure 1.13). Intuitively, a sorting network shuffles all input values that are 1 "to the left", i.e., if m of the inputs of an n-bit sorting network (where $m \leq n$) are 1, then the output is a sequence of m ones followed by $n - m$ trailing zeroes. To encode an "at most k" constraint it is therefore sufficient to constrain the $(k+1)^{\text{th}}$ output signal to 0. The advantage of this construction over the previously discussed encoding is that sorting networks can be built entirely from and-gates and or-gates (by cascading the circuit shown in Figure 1.13). Sorting networks for n bits can be implemented using $O(n \cdot (\log n)^2)$ (see, for instance, [Par92]) or even $O(n \cdot \log n)$ gates [AKS83].

1.4.3 Maximum Satisfiability Problem (MAX-SAT)

Definition 1.4.1 (Maximum Satisfiability Problem) *Given a formula F in conjunctive normal form, determine the maximum number of clauses of F*



ℓ_1	ℓ_2	o_1	o_2
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

$$o_1 \overset{\text{def}}{=} \ell_1 + \ell_2$$
$$o_2 \overset{\text{def}}{=} \ell_1 \cdot \ell_2$$

Figure 1.13: A sorting network for two literals

that can be satisfied by some assignment.

If (and only if) the formula F is satisfiable, then there is an assignment that satisfies *all* of its clauses. Accordingly, the MAX-SAT problem is NP-hard. If, however, F is unsatisfiable, one needs to determine the largest subset of the clauses of F which, if conjoined, are still satisfiable. Equivalently, one can compute the smallest set of clauses that need to be dropped from the original instance to make it satisfiable.

Example 1.4.1 *Consider the unsatisfiable formula*

$$(\overline{r} + \overline{s} + t) \ \cdot \ (\overline{r} + s) \ \cdot \ (r) \ \cdot \ (\overline{t}) \ \cdot \ (s)\,. \tag{1.3}$$

Dropping the clause $(\overline{t})$ makes the instance satisfiable. Note that the largest set of satisfiable clauses is not unique: dropping the clause (r) also results in a satisfiable formula with four clauses as well.

The *partial* MAX-SAT problem is a generalisation of the MAX-SAT problem, in which some of the clauses are tagged as *hard* and must not be dropped.

Definition 1.4.2 (Partial Maximum Satisfiability Problem) *Given a formula F and a set $\{C_1, \ldots, C_m\} \subseteq F$ of* hard *clauses, determine the maximum number of clauses of F that can be satisfied by some assignment $\mathcal{A} \models \prod_{i=1}^{m} C_i$.*

We refer to clauses of a partial MAX-SAT instance that are not hard as *soft* clauses.

Example 1.4.2 *We revisit Example 1.4.1, but require that the clauses (r) and $(\overline{t})$ of Formula (1.3) must not be dropped this time. In this scenario, dropping $(\overline{r} + \overline{s} + t)$ makes the formula satisfiable. Note that dropping either $(\overline{r} + t)$ or (s) does not yield a satisfiable instance.*

Relaxation Literals

The satisfiability checking techniques covered in Section 1.3 lack the ability to drop clauses. Contemporary satisfiability solvers such as MiniSAT [ES04], however, do at least provide the option to a specify a partial assignment, which can be reverted in a subsequent call to the solver without sacrificing the learnt clauses that do not depend on this assignment.

As it turns out, such a mechanism is not required to exclude clauses from the search process if we augment these clauses with so called *relaxation literals*. A relaxation literal is a variable v that does not occur in the original formula. If we replace a clause C_i that is part of the original formula with the *relaxed* clause $(v_i + C_i)$, the variable v_i acts as a switch which enables us to activate the clause C_i by setting v_i to 0. Conversely, the solver will ignore $(v_i + C_i)$ if v_i is set to 1 (by virtue of the *affirmative-negative rule* introduced in Section 1.3.3).

Example 1.4.3 *We continue working in the setting of Example 1.4.2. The following formula resembles Formula 1.3, except for the fact that the* soft *clauses have been augmented with the relaxation literals u, v, and w, respectively:*

$$(u + \overline{r} + \overline{s} + t) \cdot (v + \overline{r} + s) \cdot (r) \cdot (\overline{t}) \cdot (w + s) . \tag{1.4}$$

Now, any satisfiability solver can be used to determine that Formula 1.4 is satisfiable. The resulting satisfying assignment to u, v, and w determines which clauses were "dropped" by the solver.

Unfortunately, the technique outlined in Example 1.4.3 gives us no control over *which*, and more importantly, *how many* clauses the solver drops. Unless we modify the decision procedure, minimality is not guaranteed.

We can, however, restrict the number of dropped clauses by adding *cardinality constraints* (Section 1.4.2) to the relaxed formula. The corresponding constraint for the formula in Example 1.4.3, $(u + v + w) \leq 1$, instructs the SAT solver to drop *at most one* clause. Moreover, we already know that the solver has to drop *at least one* clause, since the original formula is unsatisfiable [MPLMS08]. In the case of Example 1.4.3, the SAT solver will find a satisfying (both literally and figuratively) solution. The rather restrictive cardinality constraint, however, does not account for (partial) MAX-SAT solutions that require the relaxation of more than one clause.

Example 1.4.4 *Consider the unsatisfiable formula*

$$(s) \cdot (\overline{s}) \cdot (t) \cdot (\overline{t}) .$$

Note that this formula has two disjoint *unsatisfiable cores (c.f. Section 1.3.8). Accordingly, the formula*

$$(u + s) \cdot (v + \overline{s}) \cdot (w + t) \cdot (x + \overline{t}) \cdot (\textstyle\sum\{u, v, w, x\} \leq 1)$$

is still *unsatisfiable.*

The formula in Example 1.4.4 requires the solver drop at least two clauses. This can be achieved by *replacing* the cardinality constraint with the slightly modified constraint $\sum\{u, v, w, x\} \leq 2$. As outlined in Section 1.4.2, this can be easily achieved by modifying a *single unit clause* as long as we use sorting networks to encode the constraint. Moreover, such a modification does not necessarily require us to restart the search from scratch, as mentioned in the first paragraph of Section 1.4.3. Incremental satisfiability solvers (see Section 1.3.9) are able to retain at least some of the clauses learnt from the first instance.

Accordingly, it is possible to successively relax the cardinality constraint in an efficient manner. If we follow this scheme, we obtain an algorithm to solve the partial MAX-SAT problem. If we successively increase the numeric parameter of the cardinality constraint (by forcing one single assignment of a literal of the sorting network), starting with one, we have a solution of the partial MAX-SAT problem readily at hand as soon as the SAT solver finds a satisfying assignment.

Core-Guided MAX-SAT

Example 1.4.5 *Consider the unsatisfiable formula*

$$(r+t)\ \cdot\ (r+s)\ \cdot\ (s)\ \cdot\ (\overline{s})\ \cdot\ (t)\ \cdot\ (\overline{t})\ ,$$

which resembles the formula in Example 1.4.4, except for the two clauses $(r+t)$ and $(r+s)$. Neither of these clauses influences the satisfiability of the formula. Accordingly, instrumenting these clauses with relaxation literals unnecessarily introduces additional variables and increases the size of the sorting network.

It is possible to avoid this unnecessary overhead in Example 1.4.5 by excluding the clauses $(r+t)$ and $(r+s)$ from the set of clauses the solver considers for removal. However, how can we know that this is sound? The exclusion of a random clause may result in an invalid answer to the MAX-SAT problem.

The answer lies in the *minimal* unsatisfiable cores (Definition 1.3.8 in Section 1.3.8) of the formula. A clause C that is not contained in *any* (minimal) unsatisfiable core of F has no impact on the satisfiability of F. Accordingly, it is not necessary to instrument C with a relaxation literal.

Accordingly, it is possible to use cores to *guide* the selection of clauses to be relaxed can be [FM06] as demonstrated in the following example.

Example 1.4.6 *We continue working in the setting of Example 1.4.5. Following the method presented in Example 1.3.10, we obtain an initial core $\{(s),(\overline{s})\}$. Similar to Example 1.4.4, we instrument the clauses occurring this core with fresh relaxation literals and impose a cardinality constraint on these literals:*

$$(r+t)\ \cdot\ (r+s)\ \cdot\ (u+s)\ \cdot\ (v+\overline{s})\ \cdot\ (t)\ \cdot\ (\overline{t})\ \cdot\ (\textstyle\sum\{u,v\}\leq 1) \tag{1.5}$$

This relaxation "deactivates" the core (and also overlapping non-minimal cores, which demonstrates that core guiding our instrumentation is not required to be minimal). The modified formula (1.5), however, is still not satisfiable. It contains a second core, namely $\{(t),(\overline{t})\}$. Defusing this core in a similar manner as the previous one yields

$$(r+t)\cdot(r+s)\cdot(u+s)\cdot(v+\overline{s})\cdot(w+t)\cdot(x+\overline{t})\cdot(\textstyle\sum\{u,v\}\leq 1)\cdot(\textstyle\sum\{w,x\}\leq 1)$$

```
① While instance unsatisfiable, repeat:
    ❶ Obtain unsatisfiable core $UC$
    ❷ If $UC$ contains no soft clauses, return UNSAT
    ❸ For all soft clauses $\{C_1, \ldots, C_n\} \subseteq$ UC
        ∗ introduce fresh relaxation variable $v_i$
        ∗ $C_i := C_i \cup \{v_i\}$
    ❹ Add constraint $(\sum_{i=1}^{n} v_i) \leq 1$
② Obtain satisfying assignment $\mathcal{A}$
③ Return number of relaxation literals $v_i$ with $\mathcal{A}(v_i) = 1$
```

Figure 1.14: A core-guided MAX-SAT algorithm

A final run of the satisfiability solver yields a satisfying assignment which indicates that we need to relax two clauses. Note that it was not necessary to instrument the clauses $(r + t)$ and $(r + s)$ – this is a crucial advantage large when it comes to large problem instances.

Figure 1.14 shows the pseudo-code of the core-guided MAX-SAT algorithm outlined in Example 1.4.6. Note that the introduction of relaxation literals complicates the use of incremental SAT algorithms (c.f. Section 1.3.9). At least the clauses learnt from hard constraints, however, can be retained across all instances.

1.4.4 Minimal Correction Sets (MCS)

In the previous section, the focus gradually shifted from clauses that can be satisfied simultaneously to clauses that need to be dropped to obtain a satisfiable formula. A set of clauses that has the latter property is also known as *minimal correction set* (MCS). The complement of each maxim*al* set of satisfiable clauses is an MCS. Accordingly, minimal correction sets are a generalisation of the MAX-SAT problem [LS09] – as the name indicates, we merely require minimality, i.e., in general, an MCS is not a minim*um*.

Given this close relationship between the MAX-SAT problem and MCSes, it seems natural to extend the algorithm from Figure 1.14 to compute correction sets. Indeed, the algorithm readily provides one MCS (whose size, in fact, is a minimum). But what if we desire to compute more than one, or even *all* MCSes? The technique presented in [LS09] is based on the algorithm in Section 1.4.3 and relies on blocking clauses (see Section 1.4.1) to exhaustively enumerate *all* minimal correction sets.

The algorithm in Figure 1.15 uses several auxiliary helper functions which implement techniques we have encountered in the previous sections.

- The procedure INSTRUMENT adds relaxation literals to clauses of the formula provided as parameter. If no second parameter is provided, the procedure instruments *all* clauses. Otherwise, the procedure only instruments clauses contained in the set of clauses provided as second parameter. This process is outlined in Example 1.4.3.
- The procedure BLOCK adds blocking clauses that rule out the minimal correction sets provided as parameter. To this end, BLOCK adds one blocking clause for each MCS and assures thus that at least one clause of each MCS provided as parameter is *not* dropped.
- ATMOST generates a cardinality constraint which states that at most k clauses are dropped from the set of clauses provided as second parameter. Cardinality constraints are discussed in Section 1.4.2.
- ALLSAT enumerates all satisfying assignments to the relaxation literals contained in the formula provided as parameter. In our context, each of these assignments represents a minimal correction set. The respective techniques are covered in Section 1.4.1.

At the core of the algorithm in Figure 1.15 lies the MAX-SAT algorithm from Figure 1.14. In particular, the first intermediate result of the algorithm in Figure 1.15 is the set of all minim*um* correction sets, obtained by means of computing all solutions to the MAX-SAT problem. Subsequently, the algorithm gradually relaxes the cardinality constraint, allowing for correction sets of a larger cardinality while blocking MCSes found in previous iterations. In each iteration, the algorithm enumerates *all* correction sets of cardinality k. By induction, this guarantees the completeness of the algorithm; a formal argument is given in [LS09].

Example 1.4.7 *We recall the Formula 1.3 presented in Example 1.4.1:*

$$(\overline{r}+\overline{s}+t)\ \cdot\ (\overline{r}+s)\ \cdot\ (r)\ \cdot\ (\overline{t})\ \cdot\ (s)$$

We simulate the algorithm in Figure 1.15 on this example. Since MCSes = ∅ in the initial iteration of the algorithm, the relaxed formula in line ④ is satisfiable. If we follow the algorithm presented in Section 1.3.8, the satisfiability solver returns the unsatisfiable core $\big\{(\overline{r}+\overline{s}+t), (r), (\overline{t}), (s)\big\}$. *Accordingly, the algorithm constructs the formula*

$$(u+\overline{r}+\overline{s}+t)\ \cdot\ (\overline{r}+s)\ \cdot\ (v+r)\ \cdot\ (w+\overline{t})\ \cdot\ (x+s)\ \cdot(\sum\{u,v,w,x\}\leq 1)$$

Then, it incrementally *constructs all satisfying assignments to* $\{u,v,w,x\}$ *that are consistent with this formula. We obtain the partial assignments*

$$\{u\mapsto 1, v\mapsto 0, w\mapsto 0, x\mapsto 0\},$$
$$\{u\mapsto 0, v\mapsto 1, w\mapsto 0, x\mapsto 0\},\ \textit{and}$$
$$\{u\mapsto 0, v\mapsto 0, w\mapsto 1, x\mapsto 0\}\,.$$

```
① $k := 1$
② MCSes $:= \emptyset$
③ $\mathrm{UC}_k :=$ unsatisfiable core of $F$
④ While ($\textsc{Instrument}(F) \cdot (\textsc{Block}(\mathrm{MCSes}))$ is satisfiable
    ❶ Instrument clauses in $\mathrm{UC}_k$ with relaxation literal:
      $F_k := \textsc{Instrument}(F,\ \mathrm{UC}_k) \cdot \textsc{AtMost}(k, \mathrm{UC}_k)$
    ❷ Enumerate satisfying assignments to relaxation variables:
      $\mathrm{MCSes} := \mathrm{MCSes} \cup \textsc{AllSAT}(F_k)$
    ❸ $\mathrm{UC}_{k+1} := \mathrm{UC}_k \cup$ core of $F_k \cdot \textsc{Block}(MCSes)$
      (projected to clauses of $F$)
    ❹ $k := k + 1$
⑤ return MCSes
```

Figure 1.15: A core-guided algorithm to compute MCSes

and the corresponding blocking clauses $(\overline{u})$, $(\overline{v})$, *and* $(\overline{w})$. *The respective MCSes of cardinality one are* $\{(\overline{r}+\overline{s}+t)\}$, $\{r\}$, *and* $\{\overline{t}\}$. *Note that the partial assignment* $\{u \mapsto 0, v \mapsto 0, w \mapsto 0, x \mapsto 1\}$ *is not a satisfying assignment, since dropping the clause* (s) *does not make the formula satisfiable – the unit clause* (s) *can be inferred from* $(\overline{r} + s)$ *and* (r). *After blocking all MCSes (controlled by the variables* $\{u, v, w, x\}$*), we end up with the formula*

$$(u + \overline{r} + \overline{s} + t) \cdot (\overline{r} + s) \cdot (v + r) \cdot (w + \overline{t}) \cdot (x + s) \cdot (\textstyle\sum\{u, v, w, x\} \leq 1) \cdot (\overline{u}) \cdot (\overline{v}) \cdot (\overline{w})\,.$$

In step ❸, *the algorithm constructs the core of this formula, replaces the instrumented clauses with their original counterparts, and drops the cardinality constraint and the blocking clauses from the core. We obtain the new core*

$$\{(\overline{r} + \overline{s} + t)\,, (\overline{r} + s)\,, (r), (\overline{t})\}\,.$$

Note that, since the blocking clauses do not prevent (s) *from being dropped, the clause* $(\overline{r} + s)$ *must be contained in this core.*

In the next step, the algorithm increases k. *Now, all clauses have to be instrumented (since the union of both cores computed so far happens to be the set of all clauses of the original formula), and all MCSes computed so far need to be blocked. In combination with the new cardinality constraint, we obtain*

$$(u + \overline{r} + \overline{s} + t) \cdot (y + \overline{r} + s) \cdot (v + r) \cdot (w + \overline{t}) \cdot (x + s) \cdot (\overline{u}) \cdot (\overline{v}) \cdot (\overline{w}) \cdot (\textstyle\sum\{u, v, w, x, y\} \leq 2)\,.$$

Since neither dropping (s) nor dropping $(\overline{r}+s)$ from the original instance makes the formula satisfiable, the algorithm determines the satisfying assignment $\{u \mapsto 0, y \mapsto 1, v \mapsto 0, w \mapsto 0, x \mapsto 1\}$. This assignment is in fact the only *satisfying partial assignment to the variables $\{u, v, w, x, y\}$ for the given formula. The corresponding blocking clause is $(\overline{x}+\overline{y})$.*

We leave it to the reader to verify that INSTRUMENT$(F)\cdot(\overline{u})\cdot(\overline{v})\cdot(\overline{w})\cdot(\overline{x}+\overline{y})$ *in line ④ is now unsatisfiable, and that the algorithm therefore terminates reporting the MCSes*

$$\{(\overline{r}+\overline{s}+t)\}, \quad \{(r)\}, \quad \{(\overline{t})\}, \quad \textit{and} \quad \{(s), (\overline{r}+s)\}\,.$$

1.4.5 Minimal Unsatisfiable Cores

Definition 1.4.3 (Hitting Set) *Given a set of sets $\mathcal{S}$, a hitting set of $\mathcal{S}$ is a set H such that*

$$\forall S \in \mathcal{S} \,.\, H \cap S \neq \emptyset$$

- Let $\mathcal{S}$ be the set of all MCSes of an unsatisfiable formula F. Then each (minimal) hitting set of $\mathcal{S}$ is a (minimal) unsatisfiable core (see Section 1.3.8).
- Let $\mathcal{S}$ be the set of all minimal unsatisfiable cores of an unsatisfiable formula F. Then each (minimal) hitting set of $\mathcal{S}$ is a (minimal) correction set for F.

Example 1.4.8 *The leftmost column in Figure 1.16 shows the set of all minimal correction sets $\{\{(\overline{s})\}, \{(r), (s)\}, \{(s), (\overline{r}+s)\}\}$ for the unsatisfiable formula*

$$F \quad \equiv \quad (\overline{s}) \cdot (\overline{r}+s) \cdot (r) \cdot (s)\,.$$

The check-marks in the table indicate the occurrences of the clauses of F in the respective MCS. By choosing a subset of clauses of F which "hit" all *MCSes, we obtain a minimal unsatisfiable core. The formula F has two minimal unsatisfiable cores, namely $\{(s), (\overline{s})\}$ and $\{(r), (\overline{s}), (\overline{r}+s)\}$. The choice of appropriate "hitting" clauses is indicated in Figure 1.16 by oval and rectangular boxes, respectively.*

The problem of deciding whether a given set of sets has a hitting set of size k (or smaller) is NP-complete ([Kar72] in [LS08]). An algorithm optimised for the purpose of extracting cores from sets of MCSes can be found in [LS08].

Instead of presenting the algorithm suggested in [LS08], we draw the readers attention to the fact that after the final iteration of the algorithm in Figure 1.15, the set of clauses (BLOCK(MCSes)) in step ③ is a *symbolic* representation of all minimal correction sets. Essentially, we are looking for assignments that satisfy the CNF formula (BLOCK(MCSes)). Note that the phase of all literals in (BLOCK(MCSes)) is negative, since the respective clauses block assignments of 1 to relaxation variables. Accordingly, in order to find minimal unsatisfiable cores,

MCS	$(\overline{s})$	$(\overline{r}+s)$	(r)	(s)
$\{(\overline{s})\}$	$\boxed{\boxed{\checkmark}}$			
$\{(r),(s)\}$			$\boxed{\checkmark}$	$\boxed{\checkmark}$
$\{(s),(\overline{r}+s)\}$		$\boxed{\checkmark}$		$\boxed{\checkmark}$

MUS: $\boxed{\{(s),(\overline{s})\}}$ $\boxed{\{(r),(\overline{s}),(\overline{r}+s)\}}$

Figure 1.16: MCSes are hitting sets of MUSes, and vice versa

we need to minimise the number of variables set to 0 in the satisfying assignment to (Block(MCSes)). Again, this can be achieved by means of gradually relaxed cardinality constraints.

Example 1.4.9 *In Example 1.4.7, we ended up with the blocking clause*

$$(\overline{u})\cdot(\overline{v})\cdot(\overline{w})\cdot(\overline{x}+\overline{y})\,,$$

where the relaxation literals u, v, w, x, and y correspond to the clauses $(\overline{r}+\overline{s}+t)$, (r), $(\overline{t})$, (s), and $(\overline{r}+s)$, respectively. Each of the clauses is satisfied if at least one of its literals evaluates to 1 *(and the corresponding variable evaluates to* 0*, respectively). In order to find a* minimal *hitting set, we constrain the literals using a cardinality constraint:*

$$(\overline{u})\cdot(\overline{v})\cdot(\overline{w})\cdot(\overline{x}+\overline{y})\cdot\left(\sum\{\overline{u},\overline{v},\overline{w},\overline{x},\overline{y}\}\leq k\right)$$

Note that k has to be at least four, since there are four clauses which do not share any literals. This threshold can be obtained using a syntactical analysis of the formula or simply by incrementally increasing k until it is sufficiently large.

If we generate all *minimal satisfying assignments to the constrained formula (using blocking clauses in a way similar to Example 1.4.7) we obtain the following assignments for $k=4$:*

$$\{\overline{u}\mapsto 1,\overline{v}\mapsto 1,\overline{w}\mapsto 1,\overline{x}\mapsto 1,\overline{y}\mapsto 0\}\ \textit{and}$$
$$\{\overline{u}\mapsto 1,\overline{v}\mapsto 1,\overline{w}\mapsto 1,\overline{x}\mapsto 0,\overline{y}\mapsto 1\}$$

These assignments correspond to the minimal unsatisfiable cores

$$\{(\overline{r}+\overline{s}+t),(r),(\overline{t}),(s)\}$$
$$\{(\overline{r}+\overline{s}+t),(r),(\overline{t}),(\overline{r}+s)\}\,.$$

The hitting set problem is equivalent to the set cover problem, an NP-complete problem that has been extensively studied in complexity theory. We do not claim that the technique in Example 1.4.9 is competitive compared to other algorithms such as the one presented in [LS08] – the purpose of the example is to gain a deeper understanding of hitting sets.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 1st edition, 2009.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1983.

[BIFH$^+$11] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Reducing the size of resolution proofs in linear time. *Software Tools for Technology Transfer (STTT)*, 13(3):263–272, 2011.

[Bus98] Samuel R. Buss. *Handbook of proof theory.* Studies in logic and the foundations of mathematics. Elsevier, 1998.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–214, July 1960.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pages 102–104. Springer, 2005.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919, pages 502–518. Springer, 2004.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149. IEEE, 2002.

[GOMS04] Éric Grégoire, Richard Ostrowski, Bertrand Mazure, and Lakhdar Saïs. Automatic extraction of functional dependencies. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2004.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[Kro67] M. R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly*, 13(1-2):15–20, 1967.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: An algorithmic point of view*. Texts in Theoretical Computer Science (EATCS). Springer, 2008.

[KSW01] Joonyoung Kim, Karem Sakallah, and Jesse Whittemore. SATIRE: A new incremental satisfiability engine. In *Design Automation Conference (DAC)*, pages 542–545. IEEE, 2001.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[LS09] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided MAX-SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *Lecture Notes in Computer Science*, pages 481–494. Springer, 2009.

[McM02] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[MPLMS08] Paulo J. Matos, Jordi Planes, Florian Letombe, and João Marques-Silva. A MAX-SAT algorithm portfolio. In *European Conference on Artificial Intelligence*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 911–912. IOS Press, 2008.

[MS95] João Paulo Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits.* PhD thesis, University of Michigan, 1995.

[MS99] João P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence, (EPIA)*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[MSS96] João Paulo Marques-Silva and Karem A. Sakallah. GRASP – a new search algorithm for satisfiability. In *International Conference on Computer-aided Design (ICCAD)*, pages 220–227. IEEE, 1996.

[MZM$^+$01] Sharad Malik, Ying Zhao, Conor F. Madigan, Lintao Zhang, and Matthew W. Moskewicz. Chaff: Engineering an efficient SAT solver. *Design Automation Conference (DAC)*, pages 530–535, 2001.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, January 1965.

[Str01] Ofer Strichman. Pruning techniques for the sat-based bounded model checking problem. In *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001.

[Tse83] G. Tseitin. On the complexity of proofs in poropositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer, 1983. Originally published 1970.

[Zha03] Lintao Zhang. *Searching the Truth: Techniques for Satisfiability of Boolean Formulas.* PhD thesis, Princeton University, 2003.