3

From Propositional to Quantifier-Free Theories

3.1 Introduction

In the previous chapter we studied CDCL-based procedures for deciding propositional formulas. Suppose, now, that instead of propositional variables we have other predicates, such as equalities and disequalities over the reals, e.g.,

$$(x_1 = x_2 \vee x_1 = x_3) \wedge (x_1 = x_2 \vee x_1 = x_4) \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \ . \quad (3.1)$$

Or, perhaps we would like to decide a Boolean combination of linear-arithmetic predicates:

$$((x_1 + 2x_3 < 5) \vee \neg(x_3 \leq 1) \wedge (x_1 \geq 3)) \ , \quad (3.2)$$

or a formula over arrays:

$$(i = j \wedge a[j] = 1) \wedge \neg(a[i] = 1) \ . \quad (3.3)$$

Equalities, linear predicates, arrays... is there a general framework to define them? Of course there is, and it is called first-order logic. Each of the above examples is a formula in some quantifier-free fragment of a first-order *theory*. Let us recall some basic terminology that was introduced already in Sect. 1.4. Generally such formulas can use propositional connectives and a set Σ of additional function and predicate symbols that uniquely define the theory T—indeed Σ is called the *signature* of T.[1] A decision procedure for T can decide the validity of T-formulas. Accordingly such formulas can be characterized as being T-valid, T-satisfiable (also called T-consistent), etc. We refer the reader to Sect. 1.4 for a more elaborate discussion of these matters.

[1] In this book we only consider signatures with commonly used symbols (e.g., "+", "*", "<") and assume that they are interpreted in the standard way (e.g., the "+" symbol corresponds to addition). Hence, the interpretation of the symbols in Σ is fixed.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_3

In this chapter we study a general method—a framework, really—that generalizes CDCL to a decision procedure for decidable quantifier-free first-order theories, such as those above.[2] The method is commonly referred to as DPLL(T), emphasizing that it is parameterized by a theory T. The fact that it is called DPLL(T) and not CDCL(T) is attributed to historical reasons only: it is based on modern CDCL solvers (see Sect. 2.4 for a discussion on the differences). It is implemented in most **Satisfiability Modulo Theories** (SMT) solvers. In the case of (3.1), for example, T is simply the theory of equality (see Chap. 4). DPLL(T) is based on an interplay between a SAT solver and a decision procedure DP_T for the conjunctive fragment of T, i.e., formulas which are a *conjunction* of T-literals.

DP_T

The following example demonstrates the existence of a decision procedure DP_T for the case of a conjunction of equalities:

Example 3.1. In the case where T is the theory of equality, a simple procedure DP_T is easy to design. The T-literals are either equalities or inequality predicates over some set of variables V. Given a conjunction of T-literals φ, build an undirected graph $G(N, E_=, E_\neq)$ where the nodes N correspond to the variables V, and there are two kinds of edges, $E_=$ and $E_\neq$, corresponding respectively to the equality and inequality predicates in φ. This is called an *equality graph*. It is not hard to see that the formula φ is unsatisfiable if and only if there exists an edge $(v_1, v_2) \in E_\neq$ such that v_2 is reachable from v_1 through a sequence of $E_=$ edges. The equality graph in Fig. 3.1, for example, corresponds to $x_1 \neq x_2 \wedge x_2 = x_3 \wedge x_1 = x_3$. This procedure can be implemented with $|E_\neq|$ depth-first search (DFS) calls over G, and is hence polynomial in the size of the input formula. More efficient procedures exist, and will be discussed in Chap. 4. ⌟

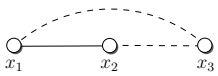

Fig. 3.1. An equality graph corresponding to $x_1 \neq x_2 \wedge x_2 = x_3 \wedge x_1 = x_3$

To reason about formulas with arbitrary propositional structure rather than just conjunctions, we can simply perform case-splitting (see Sect. 1.3), and decide each case with DP_T. If any of the cases is satisfiable, then so is the original formula. For example, there are four cases to consider in order to decide (3.1):

[2] Extending the framework for solving quantified formulas (which is not necessarily decidable) will be discussed later on in the book, namely in Sect. 9.5.

$$\begin{array}{l}(x_1 = x_2 \wedge x_1 = x_2 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ , \\ (x_1 = x_2 \wedge x_1 = x_4 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ , \\ (x_1 = x_3 \wedge x_1 = x_2 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ , \\ (x_1 = x_3 \wedge x_1 = x_4 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ ,\end{array} \tag{3.4}$$

all of which are unsatisfiable. Hence, we can conclude that (3.1) is unsatisfiable.

The primary problem with this approach is the fact that the number of cases is in general exponential in the size of the original formula—think how many cases we would have if the original formula had n clauses—and indeed, when attempting to solve formulas that have a nontrivial propositional structure with this method, the number of cases is typically a major bottleneck (see Example 1.18). Furthermore, this method misses any opportunity for learning, as each case is solved independently. In the example above, the contradiction $x_1 = x_2 \wedge x_1 \neq x_2$ appears in two separate cases, but we still have to infer inconsistency for each one of them separately.

A better approach is to leverage the learning capabilities of SAT (see p. 35) and other means of efficiency, and combine it with DP_T in order to solve such formulas. The two main engines in this framework work in tight collaboration: the SAT solver chooses those literals that need to be satisfied in order to satisfy the Boolean structure of the formula, and DP_T checks whether this choice is T-satisfiable.

The advantage of this approach is that any reasoning about the propositional part of φ is performed by the propositional SAT solver; any explicit case splitting of disjunctions in φ is avoided. This has strong practical advantages, as the resulting algorithms are both very modular and very efficient.

We will now present in detail how this combination can be implemented, while continuing to use the theory of equality as an example.

3.2 An Overview of DPLL(T)

Recall that every theory T is defined with respect to a signature Σ, which is the set of allowed symbols. In the case of the theory of equality, for example, $\Sigma = \{\text{`='}\}$. When we write Σ-literals (or, similarly, Σ-atoms and Σ-formulas), it means that the literal only uses symbols from Σ.

Let $at(\varphi)$ denote the set of Σ-atoms in a given NNF formula φ. Assuming some predefined order on the atoms, we denote the i-th distinct atom in φ by $at_i(\varphi)$.

Given atom a, we associate with it a unique Boolean variable $e(a)$, which we call the Boolean **encoder** of this atom. Extending this idea to formulas, given a Σ-formula t, $e(t)$ denotes the Boolean formula resulting from substituting each Σ-atom in t with its Boolean encoder.

For example, if $x = y$ is a Σ-atom, then $e(x = y)$, a Boolean variable, denotes its encoder. And if

$at(\varphi)$

$at_i(\varphi)$

$e(a)$

$e(t)$

$$\varphi := \ x = y \vee x = z \tag{3.5}$$

is a Σ-formula, then

$$e(\varphi) := \ e(x = y) \vee e(x = z) \ . \tag{3.6}$$

For a Σ-formula φ, the resulting Boolean formula $e(\varphi)$ is called the **propositional skeleton** of φ.

Using this notation, we can now begin to give an overview of the method studied in this chapter, while following a simple example. Some of the notation that we shall use in this example will be defined more formally later on.

As before we will use the theory of equality for the example. Let

$$\varphi := \ x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z) \ . \tag{3.7}$$

The propositional skeleton of φ is

$$e(\varphi) := \ e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z)) \ . \tag{3.8}$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \ e(\varphi) \ . \tag{3.9}$$

As the next step, we pass $\mathcal{B}$ to a SAT solver. Assume that the formula is satisfiable and that the SAT solver returns the satisfying assignment

$$\alpha := \ \{e(x = y) \mapsto \text{TRUE}, \ e(y = z) \mapsto \text{TRUE}, \ e(x = z) \mapsto \text{FALSE}\} \ .$$

The decision procedure DP_T now has to decide whether the conjunction of the literals corresponding to this assignment is satisfiable. We denote this conjunction by $\hat{Th}(\alpha)$ (the "*Th*" is intended to remind the reader that the result of this function is a *Th*eory, and the "hat" that it is a conjunction of symbols). For the assignment above,

$$\hat{Th}(\alpha) := \ x = y \wedge y = z \wedge \neg(x = z) \ . \tag{3.10}$$

This formula is not satisfiable, which means that the negation of this formula is a tautology. Thus $\mathcal{B}$ is conjoined with $e(\neg\hat{Th}(\alpha))$, the Boolean encoding of this tautology:

$$e(\neg\hat{Th}(\alpha)) := \ (\neg e(x = y) \vee \neg e(y = z) \vee e(x = z)) \ . \tag{3.11}$$

This clause contradicts the current assignment, and hence *blocks* it from being repeated. Such clauses are called **blocking clauses**. In general, we denote by t the formula—also called the **lemma**—returned by DP_T. In this example, $t := \neg\hat{Th}(\alpha)$, that is, the lemma is the negation of the full assignment α and hence it is a clause, but generally t can be multiple clauses, depending on the implementation of DP_T.

After the blocking clause has been added, the SAT solver is invoked again and suggests another assignment, for example,

$$\alpha' := \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE},\ e(x = z) \mapsto \text{TRUE}\} .$$

The corresponding Σ-formula

$$\hat{Th}(\alpha') := x = y \wedge y = z \wedge x = z \tag{3.12}$$

is satisfiable, which proves that φ, the original formula, is satisfiable. Indeed, any assignment that satisfies $\hat{Th}(\alpha')$ also satisfies φ.

Figure 3.2 illustrates the information flow between the two components of the decision procedure.

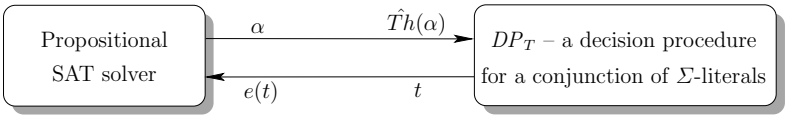

Fig. 3.2. The information exchanged between the SAT solver and a decision procedure DP_T for a conjunction of Σ-literals

There are many improvements to this basic procedure, some of which we shall cover later in this chapter, and some of which are left as exercises in Sect. 3.5. One such improvement, for example, is to invoke the decision procedure DP_T after some or all *partial assignments*, rather than waiting for a full assignment. A contradicting partial assignment leads to a more powerful lemma t, as it blocks all assignments that extend it. Further, when the partial assignment is not contradictory, it can be used to derive implications that are propagated back to the SAT solver. Continuing the example above, consider the partial assignment

$$\alpha := \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE}\} , \tag{3.13}$$

and the corresponding formula that is transferred to DP_T,

$$\hat{Th}(\alpha) := x = y \wedge y = z . \tag{3.14}$$

This leads DP_T to conclude that $x = z$ is implied, and hence accordingly to inform the SAT solver that $e(x = z) \mapsto \text{TRUE}$ is implied by the current partial assignment α. Thus, in addition to the normal Boolean constraint propagation (BCP) performed by the SAT solver, there is now also **theory propagation**. Such propagation may lead to further BCP, which means that this process may iterate several times before the next decision is made by the SAT solver.

In the next few sections, we describe variations of the process demonstrated above.

3.3 Formalization

α For a given encoding $e(\varphi)$, we denote by α an assignment, either full or partial, to the encoders in $e(\varphi)$. Then for an encoder $e(at_i)$ that is assigned a truth value by α, we define the corresponding literal, denoted $Th(at_i, \alpha)$, as follows:

$$Th(at_i, \alpha) \doteq \begin{cases} at_i & \alpha(at_i) = \text{TRUE} \\ \neg at_i & \alpha(at_i) = \text{FALSE} . \end{cases} \tag{3.15}$$

$Th(\alpha)$ Somewhat overloading the notation, we write $Th(\alpha)$ to denote the set of literals such that their encoders are assigned by α:

$$Th(\alpha) \doteq \{Th(at_i, \alpha) \mid e(at_i) \text{ is assigned by } \alpha\} . \tag{3.16}$$

$\hat{Th}(\alpha)$ We denote by $\hat{Th}(\alpha)$ the conjunction of the elements of the set $Th(\alpha)$.

Example 3.2. To demonstrate the use of the above notation, let

$$at_1 = (x = y),\ at_2 = (y = z),\ at_3 = (z = w) , \tag{3.17}$$

and let α be a partial assignment such that

$$\alpha :=\ \{e(at_1) \mapsto \text{FALSE},\ e(at_2) \mapsto \text{TRUE}\} . \tag{3.18}$$

Then

$$Th(at_1, \alpha) :=\ \neg(x = y),\ Th(at_2, \alpha) :=\ (y = z) , \tag{3.19}$$

and

$$Th(\alpha) :=\ \{\neg(x = y), (y = z)\} . \tag{3.20}$$

Conjoining these terms gives us

$$\hat{Th}(\alpha) :=\ \neg(x = y) \wedge (y = z) . \tag{3.21}$$

▟

Recall that DP_T is a decision procedure for a conjunction of T-literals, where T is a theory defined over the symbols in Σ. Let DEDUCTION be a procedure based on DP_T, which receives a conjunction of T-literals as input, decides whether it is satisfiable, and, if the answer is negative, returns constraints over these literals, as explained below. On the basis of such a procedure, we now examine variations of the method that is demonstrated in the introduction to this chapter.

Algorithm 3.3.1 (LAZY-BASIC) decides whether a given T-formula φ is
$\mathcal{B}$ satisfiable. It does so by iteratively solving a propositional formula $\mathcal{B}$, starting from $\mathcal{B} = e(\varphi)$, and gradually strengthening it with encodings of constraints that are computed by DEDUCTION.

In each iteration, SAT-SOLVER returns a tuple ⟨assignment, result⟩ in line 4. If $\mathcal{B}$ is unsatisfiable, then so is φ: LAZY-BASIC returns "Unsatisfiable"

in line 5. Otherwise, in line 7 DEDUCTION checks whether $\hat{Th}(\alpha)$ is satisfiable. It returns a tuple of the form ⟨constraint, result⟩ where the constraint is a clause over Σ-literals, and the result is one of {"Satisfiable", "Unsatisfiable"}. If it is "Satisfiable", LAZY-BASIC returns "Satisfiable" in line 8 (recall that α is a full assignment). Otherwise, the formula t returned by DEDUCTION (typically one or more clauses) corresponds to a lemma about φ. In line 9, LAZY-BASIC continues by conjoining the $e(t)$ with $\mathcal{B}$ and reiterating.

Algorithm 3.3.1: LAZY-BASIC

Input: A formula φ
Output: "Satisfiable" if φ is satisfiable, and "Unsatisfiable" otherwise

```
1. function LAZY-BASIC(φ)
2.     B := e(φ);
3.     while (TRUE) do
4.         ⟨α, res⟩ := SAT-SOLVER(B);
5.         if res ="Unsatisfiable" then return "Unsatisfiable";
6.         else
7.             ⟨t, res⟩ := DEDUCTION(Th^(α));
8.             if res ="Satisfiable" then return "Satisfiable";
9.             B := B ∧ e(t);
```

Consider the following three requirements on the formula t that is returned by DEDUCTION:

1. The formula t is T-valid, i.e., t is a tautology in T. For example, if T is the theory of equality, then $x = y \wedge y = z \implies x = z$ is T-valid.
2. The atoms in t are restricted to those appearing in φ.
3. The encoding of t contradicts α, i.e., $e(t)$ is a blocking clause.

The first requirement is sufficient for guaranteeing soundness. The second and third requirements are sufficient for guaranteeing termination. Specifically, the third requirement guarantees that α is not repeated.

Two of the three requirements above can be weakened, however:

- Requirement 1: t can be any formula that is implied by φ, and not just a T-valid formula. However, finding formulas that on the one hand are implied by φ and on the other hand fulfill the other two requirements may be as hard as deciding φ itself, which misses the point. In practice, the amount of effort dedicated to computing t needs to be tuned separately for each theory and decision procedure, in order to maximize the overall performance.

- Requirement 2: t may refer to atoms that do not appear in φ, as long as the number of such new atoms is finite. For example, in equality logic, we may allow t to refer to all atoms of the form $x_i = x_j$ where x_i, x_j are variables in $var(\varphi)$, even if only some of these equality predicates appear in φ.

Integration into CDCL

$\mathcal{B}^i$

Let $\mathcal{B}^i$ be the formula $\mathcal{B}$ in the i-th iteration of the loop in Algorithm 3.3.1. The constraint $\mathcal{B}^{i+1}$ is strictly stronger than $\mathcal{B}^i$ for all $i \geq 1$, because blocking clauses are added but not removed between iterations. It is not hard to see that this implies that any conflict clause that is learned while solving $\mathcal{B}^i$ can be reused when solving $\mathcal{B}^j$ for $i < j$. This, in fact, is a special case of **incremental satisfiability**, which is supported by most modern SAT solvers.[3] Hence, invoking an incremental SAT solver in line 4 can increase the efficiency of the algorithm.

A better option is to integrate DEDUCTION into the CDCL-SAT algorithm, as shown in Algorithm 3.3.2. This algorithm uses a procedure ADDCLAUSES, which adds new clauses to the current set of clauses at run time. We leave the question of why this is a better option than using an incremental SAT solver to the reader (see Problem 3.1). We note that α, which is referred to in line 9, is the current assignment to the propositional variables.

3.4 Theory Propagation and the DPLL(T) Framework

3.4.1 Propagating Theory Implications

Algorithm 3.3.2 can be optimized further. Consider, for example, a formula φ that contains an integer variable x_1 and, among others, the literals $x_1 \geq 10$ and $x_1 < 0$.

Assume that the DECIDE procedure assigns $e(x_1 \geq 10) \mapsto$ TRUE and $e(x_1 < 0) \mapsto$ TRUE. Inevitably, any call to DEDUCTION results in a contradiction between these two facts, independently of any other decisions that are made. However, Algorithm 3.3.2 does not call DEDUCTION until a full satisfying assignment is found. Thus, the time taken to complete the assignment is wasted. Moreover, as was mentioned in the introduction to this chapter, the refutation of this full assignment may be due to other reasons (i.e., a proof that a different subset of the assignment is contradictory), and, hence, additional assignments that include the same wrong assignment to $e(x_1 \geq 10)$ and $e(x_1 < 0)$ are not ruled out.

[3] Incremental satisfiability was described in Sect. 2.2.7. It is concerned with the more general case in which clauses can also be *removed*. The question in that case is which conflict clauses can be reused safely. See also Problem 2.16.

Algorithm 3.3.2: LAZY-CDCL

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
 1. function LAZY-CDCL
 2.     ADDCLAUSES(cnf(e(φ)));
 3.     while (TRUE) do
 4.         while (BCP() = "conflict") do
 5.             backtrack-level := ANALYZE-CONFLICT();
 6.             if backtrack-level < 0 then return "Unsatisfiable";
 7.             else BackTrack(backtrack-level);
 8.         if ¬DECIDE() then                              ▷ Full assignment
 9.             ⟨t, res⟩:=DEDUCTION(T̂h(α));                ▷ α is the assignment
10.             if res="Satisfiable" then return "Satisfiable";
11.             ADDCLAUSES(e(t));
```

Algorithm 3.3.2 can therefore be improved by running DEDUCTION even before a full assignment to the encoders is available. This early call to DEDUCTION can serve two purposes:

1. Contradictory partial assignments are ruled out early.
2. Implications of literals that are still unassigned can be communicated back to the SAT solver, as demonstrated in Sect. 3.2. Continuing our example, once $e(x_1 \geq 10)$ has been assigned TRUE, we can infer that $e(x_1 < 0)$ must be FALSE and avoid the conflict altogether.

This brings us to the next version of the algorithm, called DPLL(T), which was first introduced in an abstract form by Tinelli [274]. As in Algorithms 3.3.1 and 3.3.2, the components of the algorithm are those of CDCL and a decision procedure for a conjunctive fragment of a theory T. The name DPLL(T) (as mentioned above, it can also be called CDCL(T)) emphasizes that this is a framework that can be instantiated with different theories and corresponding decision procedures.

In the version of DPLL(T) presented in Algorithm 3.4.1 (see also Fig. 3.3), DEDUCTION is invoked in line 9 after no further implications can be made by BCP. It then finds T-implied literals and communicates them to the CDCL part of the solver in the form of a constraint t.[4] Hence, in addition to implications in the Boolean domain, there are now also implications due to the

[4] DEDUCTION also returns the result *res* (whether $\hat{Th}(\alpha)$ is satisfiable), but *res* is not used. We have kept it in the pseudocode in order for the algorithm to stay similar to the previous algorithms.

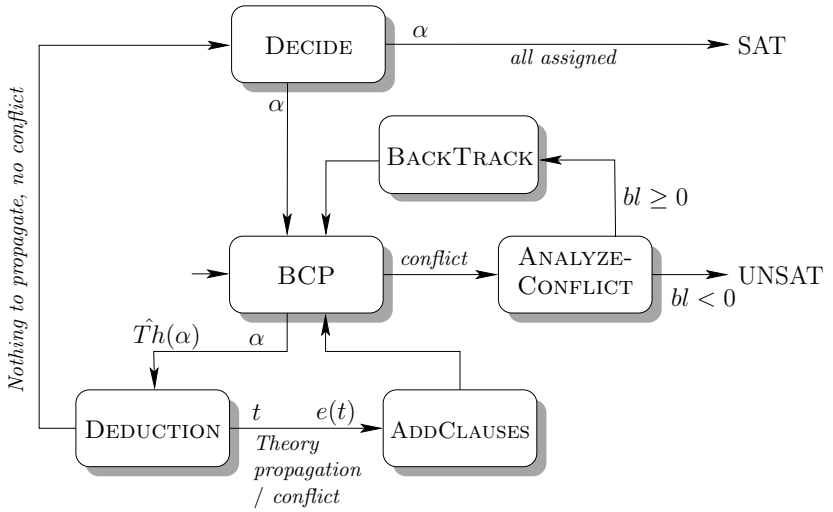

Fig. 3.3. The main components of DPLL(T). Theory propagation is implemented in Deduction

theory T. Accordingly, this technique is known by the name **theory propagation**.

What are the requirements on these new clauses? As before, they have to be implied by φ and are restricted to a finite set of atoms—typically to φ's atoms. It is desirable that, when $\hat{Th}(\alpha)$ is unsatisfiable, $e(t)$ blocks α; it is not mandatory, because whether it blocks α or not does not affect correctness—Deduction only needs to be complete when α is a full assignment. Certain SMT solvers exploit this fact to perform cheap checks on partial assignments, e.g., bound the time dedicated to them. What if $\hat{Th}(\alpha)$ is satisfiable? Then we require t to fulfill one of the following two conditions in order to guarantee termination:

1. The clause $e(t)$ is an asserting clause under α (asserting clauses are defined in Sect. 2.2.3). This implies that the addition of $e(t)$ to $\mathcal{B}$ and a call to BCP leads to an assignment to the encoder of some literal.
2. When Deduction cannot find an asserting clause t as defined above, t and $e(t)$ are equivalent to true.

The second case occurs, for example, when all the Boolean variables are already assigned, and thus the formula is found to be satisfiable. In this case, the condition in line 11 is met and the procedure continues from line 13, where Decide is called again. Since all variables are already assigned, the procedure returns "Satisfiable".

Example 3.3. Consider once again the example of the two encoders $e(x_1 \geq 10)$ and $e(x_1 < 0)$. After the first of these has been set to true, the procedure

DEDUCTION detects that $\neg(x_1 < 0)$ is implied, or, in other words, that

$$t := \neg(x_1 \geq 10) \vee \neg(x_1 < 0) \tag{3.22}$$

is T-valid. The corresponding encoded (asserting) clause

$$e(t) := (\neg e(x_1 \geq 10) \vee \neg e(x_1 < 0)) \tag{3.23}$$

is added to $\mathcal{B}$, which leads to an immediate implication of $\neg e(x_1 < 0)$, and possibly further implications. ⌟

Algorithm 3.4.1: DPLL(T)

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
 1. function DPLL(T)
 2.     AddClauses(cnf(e(φ)));
 3.     while (true) do
 4.         repeat
 5.             while (BCP() = "conflict") do
 6.                 backtrack-level := Analyze-Conflict();
 7.                 if backtrack-level < 0 then return "Unsatisfiable";
 8.                 else BackTrack(backtrack-level);
 9.             ⟨t, res⟩:=Deduction(Th(α));
10.             AddClauses(e(t));
11.         until t ≡ true;
12.         if α is a full assignment then return "Satisfiable";
13.         Decide();
```

3.4.2 Performance, Performance...

Recall that, when α is partial, DEDUCTION checks if there is a contradiction on the theory side, and if not, it performs theory propagation.

For performance, it is frequently better to run an approximation in this step for finding contradictions. Indeed, as long as α is partial, there is no need for a *complete* procedure for deciding satisfiability. This is not changing the completeness of the overall algorithm, since a complete check *is* performed when α is full. A good example of this is what competitive solvers do when the theory is *integer linear arithmetic* (to be covered in Sect. 5.3). Deciding the conjunctive fragment of this theory is NP-complete, and therefore they only consider the real relaxation of the problem (this means that they refer

to the variables as being in $\mathbb{R}$ rather than in $\mathbb{Z}$—reals instead of integers), which can be solved in polynomial time. This means that DEDUCTION will occasionally miss a contradiction and hence not return a blocking clause.

Another performance consideration is related to theory propagation. It is important to note that theory propagation is required not for *correctness*, but only for *efficiency*. Hence, the amount of effort invested in computing new implications needs to be well tuned in order to achieve the best overall performance.

The term **exhaustive theory propagation** refers to a procedure that finds and propagates *all* literals that are implied in T by $\hat{Th}(\alpha)$. A simple, generic way (called "plunging") to perform theory propagation is the following: Given an unassigned theory atom at_i, check whether $\hat{Th}(\alpha)$ implies either at_i or $\neg at_i$. The set of unassigned atoms that are checked in this way depends on how exhaustive we want the theory propagation to be.

Example 3.4. Consider equality logic, and the notation we used in Example 3.1. A simple way to perform exhaustive theory propagation in equality logic is the following: For each unassigned atom of the form $x_i = x_j$, check if the current partial assignment forms a path in $E_=$ between x_i and x_j. If yes, then this atom is implied by the literals in the path. If the partial assignment forms a disequality path (a path in which exactly one edge is from $E_{\neq}$), the negation of this atom is implied.

This generic method is typically not the most efficient, however. In many cases a better strategy is to perform only simple, inexpensive propagations, while ignoring more expensive ones. In the case of linear arithmetic, for example, experiments have shown that exhaustive theory propagation has a negative effect on overall performance. It is more efficient in this case to search for simple-to-find implications, such as "if $x > c$ holds, where x is a variable and c a constant, then any literal of the form $x > d$ is implied if $d < c$".

3.4.3 Returning Implied Assignments Instead of Clauses

Another optimization of theory propagation is concerned with the way in which the information discovered by DEDUCTION is propagated to the Boolean part of the solver. So far, we have required that the clause t returned by DEDUCTION be T-valid. For example, if α is such that $\hat{Th}(\alpha)$ implies a literal lit, then

$$t := \ (lit \vee \neg \hat{Th}(\alpha)) \ . \tag{3.24}$$

The encoded clause $e(t)$ is of the form

$$\big(e(lit) \vee \bigvee_{lit' \in Th(\alpha)} \neg e(lit')\big) \ . \tag{3.25}$$

Nieuwenhuis, Oliveras, and Tinelli concluded that this was an inefficient method, however [211]. Their experiments on various sets of benchmarks

showed that on average fewer than 0.5% of these clauses were ever used again, and that they burden the process. They suggested a better alternative, in which DEDUCTION returns a list of implied assignments (containing $e(lit)$ in this case) that the SAT solver then performs.

These implied assignments have no antecedent clauses in $\mathcal{B}$, in contrast to the standard implications due to BCP. This causes a problem in ANALYZE-CONFLICT (see Algorithm 2.2.2), which relies on antecedent clauses for deriving **conflict clauses**. As a solution, when ANALYZE-CONFLICT needs an antecedent for such an implied literal, it queries the decision procedure for an **explanation**, i.e., a clause implied by φ that implies this literal given the partial assignment at the time the assignment was created.

The explanation of an assignment might be the same clause that could have been delivered in the first place, but not necessarily: for efficiency reasons, typical implementations of DEDUCTION do not retain such clauses, and hence need to generate a new explanation. As an example, to explain an implied literal $x = y$ in equality logic, one needs to search for an equality path in the equality graph between x and y, in which all the edges were present in the graph at the time that this implication was identified and propagated.

3.4.4 Generating Strong Lemmas

If $\hat{Th}(\alpha)$ is unsatisfiable, DEDUCTION returns a blocking clause t to eliminate the assignment α. The stronger t is, the greater the number of inconsistent assignments it eliminates. One way of obtaining a stronger formula is to construct a clause consisting of the negation of those literals that participate in the proof of unsatisfiability of $\hat{Th}(\alpha)$. In other words, if S is the set of literals that serve as the premises in the proof of unsatisfiability, then the blocking clause is

$$t := \left(\bigvee_{l \in S} \neg l \right) . \tag{3.26}$$

Computing the set S corresponds to computing an *unsatisfiable core* of the formula.[5] Given a deductive proof of unsatisfiability, a core is easy to find. For this purpose, one may represent such a proof as a directed acyclic graph, as demonstrated in Fig. 3.4 (in this case for T being equality logic and uninterpreted functions). In this graph the nodes are labeled with literals and an edge (n_1, n_2) denotes the fact that the literal labeling node n_1 was used in the inference of the literal labeling node n_2. In such a graph, there is a single sink node labeled with FALSE, and the roots are labeled with the premises (and possibly axioms) of the proof. The set of roots that can be reached by a backward traversal from the FALSE node correspond to an unsatisfiable core.

[5] *Unsatisfiable cores* are defined for the case of propositional CNF formulas in Sect. 2.2.6. The brief discussion here generalizes this earlier definition to inference rules other than BINARY RESOLUTION.

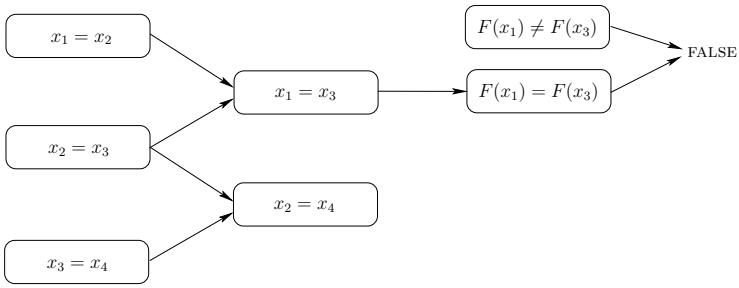

Fig. 3.4. The premises of a proof of unsatisfiability correspond to roots in the graph that can be reached by backward traversal from the FALSE node (in this case all roots other than $x_3 = x_4$). Whereas lemmas correspond to all roots, this subset of the roots can be used for generating *strong* lemmas

3.4.5 Immediate Propagation

Now consider a variation of this algorithm that calls DEDUCTION after every new assignment to an encoding variable—which may be due to either a decision or a BCP implication—rather than letting BCP finish first. Furthermore, assume that we are implementing exhaustive theory propagation as described above. In this variant, a call to DEDUCTION *cannot lead to a conflict*, which means that it never has to return a blocking clause. A formal proof of this observation is left as an exercise (Problem 3.5). An informal justification is that, if an assignment to a single encoder makes $\hat{Th}(\alpha)$ unsatisfiable, then the negation of that assignment would have been implied and propagated in the previous step by DEDUCTION. For example, if an encoder $e(x = y)$ is implied and communicated to DEDUCTION, this literal can cause a conflict only if there is a disequality path (such paths were discussed in Example 3.4) between x and y according to the previous partial assignment. This means that, in the previous step, $\neg e(x = y)$ should have been propagated to the Boolean part of the solver.

3.5 Problems

Problem 3.1 (incrementality in LAZY-CDCL). Recall that an incremental SAT solver is one that knows which conflict clauses can be reused when given a problem similar to the previous one (i.e., some clauses are added and others are erased). Is there a difference between Algorithm 3.3.2 (LAZY-CDCL) and replacing line 4 in Algorithm 3.3.1 with a call to an incremental SAT solver?

Problem 3.2 (an optimization for Algorithms 3.3.1–3.4.1?).

1. Consider the following variation of Algorithms 3.3.1–3.4.1 for an input formula φ given in NNF (negations are pushed all the way into the

atoms, e.g., $\neg(x = y)$ appears as $x \neq y$). Rather than sending $\hat{Th}(\alpha)$ to DEDUCTION, send $\bigwedge Th_i$ for all i such that $\alpha(e_i) = \text{TRUE}$. For example, given an assignment

$$\alpha := \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{FALSE},\ e(x = z) \mapsto \text{TRUE}\}\ , \quad (3.27)$$

check

$$x = y \wedge x = z\ . \quad (3.28)$$

Is this variation correct? Prove that it is correct or give a counterexample.

2. Show an example in which the above variation reduces the number of iterations between DEDUCTION and the SAT solver.

Problem 3.3 (theory propagation). Let DP_T be a decision procedure for a conjunction of Σ-literals. Suggest a procedure for performing exhaustive theory propagation with DP_T.

Problem 3.4 (pseudocode for a variant of DPLL(T)). Recall the variant of DPLL(T) suggested at the end of Sect. 3.4.5, where the partial assignment is sent to the theory solver after every assignment to an encoder, rather than only after BCP. Write pseudocode for this algorithm, and a corresponding drawing in the style of Fig. 3.3.

Problem 3.5 (exhaustive theory propagation). In Sect. 3.4.5, it was claimed that with exhaustive theory propagation, conflicts cannot occur in DEDUCTION and that, consequently, DEDUCTION never returns blocking clauses. Prove this claim.

3.6 Bibliographic Notes

The following are some bibliographic details about the development of the lazy encoding frameworks and DPLL(T). In 1999, Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia in [4] proposed a solver based on an interplay between a SAT solver and a theory solver, in a fashion similar to the simple lazy approach introduced at the beginning of this chapter. Their solver was tailored to a single theory called disjunctive temporal constraints, which is a restricted version of difference logic. In fact, they combined lazy with eager reasoning: they used a preprocessing step that adds a large set of constraints to the propositional skeleton (constraints of the form $(\neg e_1 \vee \neg e_2)$ if a preliminary check discovers that the theory literals corresponding to these encoders contradict each other). This saves a lot of work later for the lazy-style engine. In the same year LPSAT was introduced [286], which also includes many of the features described in this chapter, including a process of learning strong lemmas.

The basic idea of integrating DPLL with a decision procedure for some (single) theory was suggested even earlier than that; the focus of these efforts are modal and description logics [5, 129, 149, 219].

The step-change in performance of SAT solving due to the CHAFF SAT solver in 2001 [202] led several groups, a year later, to (independently) propose decision procedures that leverage this progress. All of these algorithms correspond to some variation of the lazy approach described in Sect. 3.3: CVC [19, 268] by Aaron Stump, Clark Barrett, and David Dill; ICS-SAT [113] by Jean-Christophe Filliatre, Sam Owre, Harald Ruess, and Natarajan Shankar; MATHSAT [7] by Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani; DLSAT [186] by Moez Mahfoudh, Peter Niebert, Eugene Asarin, and Oded Maler; and VERIFUN [115] by Cormac Flanagan, Rajeev Joshi, Xinming Ou, and Jim Saxe. Most of these tools were built as generic engines that can be extended with different decision procedures. Since the introduction of these tools, this approach has become mainstream, and at least twenty other solvers based on the same principles have been developed and published.

DPLL(T) was originally described in abstract terms, in the form of a calculus, by Cesare Tinelli in [274]. Theory propagation had already appeared under various names in the papers by Armando et al. [4] and Audemard et al. [7] mentioned above. Efficient theory propagation tailored to the underlying theory T (T being equalities with uninterpreted functions (EUF) in that case) appeared first in a paper by Ganzinger et al. [120]. These authors also introduced the idea of propagating theory implications by maintaining a stack of such implied assignments, coupled with the ability to explain them a posteriori, rather than sending asserting clauses to the DPLL part of the solver. The idea of minimizing the lemmas (blocking clauses) can be attributed to Leonardo de Moura and Harald Ruess [93], although, as we mentioned earlier, finding small lemmas already appeared in the description of LPSAT.

Various details of how a DPLL-based SAT solver could be transformed into a DPLL(T) solver were described for the case of EUF in [120] and for difference logic in [209]. A good description of DPLL(T), starting from an abstract DPLL procedure and ending with fine details of implementation, was given in [211]. A very comprehensive survey on lazy SMT was given by Roberto Sebastiani [253]. There has been quite a lot of research on how to design T-solvers that can give *explanations*, which, as pointed out in Sect. 3.4.5, is a necessary component for efficient implementation of this framework—see, for example, [95, 210, 270].

Let us mention some SMT solvers that are, at the time of writing this (2015), leading at various categories according to the annual competition:

1. Z3 from Microsoft Research, developed by Leonardo de Moura and Nikolaj Bjørner [92]. In addition to its superior performance in many categories, it also offers a convenient application-programming interface (API) in several languages, and infrastructure for add-ons.

2. CVC-4 [17, 15], the development of which is led by Clark Barrett and Cesare Tinelli. CVC enables each theory to produce a proof that can be checked independently with an external tool.
3. YICES-2 [90], which was originally developed by Leonardo de Moura and later by Bruno Dutertre and Dejan Jovanovic.
4. MATHSAT-5 [71], by Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani.
5. BOOLECTOR [52], by Armin Biere and Robert Brummayer, which specializes in solving bit-vector formulas.

A procedure based on Binary Decision Diagrams (BDDs) [55], where the predicates label the nodes, appeared in [132] and [199]. In the context of hardware verification there have been a number of publications on multiway decision graphs [81], a generalization of BDDs to various first-order theories.

3.7 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
DP_T	A decision procedure for a conjunction of T-atoms	60
$e(a)$	The propositional encoder of a Σ-atom a	61
$\alpha(t)$	A truth assignment (either full or partial) to the variables of a formula t	61
$at(\varphi)$	The atoms of φ	61
$at_i(\varphi)$	Assuming some predefined order on the atoms, this denotes the i-th distinct atom in φ	61
α	An assignment (either full or partial) to the atoms	64
$Th(at_i, \alpha)$	See (3.15)	64
$Th(\alpha)$	See (3.16)	64
$\hat{Th}(\alpha)$	The conjunction over the elements in $Th(\alpha)$	64
$\mathcal{B}$	A Boolean formula. In this chapter, initially set to $e(\varphi)$, and then strengthened with constraints	64

continued on next page

continued from previous page

Symbol	**Refers to ...**	**First used on page ...**
t	For a Σ-theory T, t represents a Σ-formula (typically a clause) returned by DEDUCTION	65
$\mathcal{B}^i$	The formula $\mathcal{B}$ in the i-th iteration of the loop in Algorithm 3.3.1	66