The algorithms we describe are for checking satisfiability: the SAT problem. (As noted earlier, testing entailment, \( \models \beta \), can be done by testing unsatisfiability of \( \alpha \land \neg \beta \).) We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

7.6.1 A complete backtracking algorithm

The first algorithm we consider is often called the **Davis–Putnam algorithm**, after the seminal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses like \( (A \lor B) \land (A \lor C) \) and \( (C \lor A) \). It is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of \( \text{TTEntails?} \):

- **Early termination:** The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence \( (A \lor B) \land (A \lor C) \) is true if \( A \) is true, regardless of the values of \( B \) and \( C \). Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

- **Pure symbol heuristic:** A *pure* symbol is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses \( (A \lor \neg B) \), \( (B \lor \neg C) \), and \( (C \lor A) \), the symbol \( A \) is pure because only the positive literal appears, \( B \) is pure because only the negative literal appears, and \( C \) is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains \( B = \text{false} \), then the clause \( \neg B \lor \neg C \) is already true, and in the remaining clauses \( C \) appears only as a positive literal: therefore \( C \) becomes pure.

- **Unit clause heuristic:** A *unit* clause was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned *false* by the model. For example, if the model contains \( B = \text{true} \), then \( \neg B \lor \neg C \) simplifies to \( \neg C \), which is a unit clause. Obviously, for this clause to be true, \( C \) must be set to *false*. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that
Section 7.6. Effective Propositional Model Checking 261

function \textsc{DPLL-SATISFIABLE}(s) returns true or false
inputs: s, a sentence in propositional logic

\texttt{clauses} ← the set of clauses in the CNF representation of
\texttt{symbols} ← a list of the proposition symbols in \texttt{s}
return \textsc{DPLL}(\texttt{clauses}, \texttt{symbols}, \{\})

function \textsc{DPLL}(\texttt{clauses}, \texttt{symbols}, \texttt{model}) returns true or false

if every clause in \texttt{clauses} is true in \texttt{model} then return true
if some clause in \texttt{clauses} is false in \texttt{model} then return false
\texttt{P \_ value \_ FIND-PURE-SYMBOL(\texttt{symbols}, \texttt{clauses}, \texttt{model})}
if \texttt{P} is non-null then return \textsc{DPLL}(\texttt{clauses}, \texttt{symbols} — \texttt{P}, \texttt{model} U \{\texttt{P=\text{true}}\})
\texttt{P \_ value \_ FIND-UNIT-CLAUSE(\texttt{clauses}, \texttt{model})}
if \texttt{P} is non-null then return \textsc{DPLL}(\texttt{clauses}, \texttt{symbols} — \texttt{P}, \texttt{model} U \{\texttt{P=\text{true}}\})
\texttt{P \_ FIRST(\texttt{symbols}); rest 4— REST(\texttt{symbols})}
return \textsc{DPLL}(\texttt{clauses}, \texttt{rest}, \texttt{model} U \{\texttt{P=\text{true}}\}) or \textsc{DPLL}(\texttt{clauses}, \texttt{rest}, \texttt{model} U \{\texttt{P=\text{false}}\})

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like \textsc{TT-ENTAILS}, DPLL operates over partial models.

any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.23). Notice also that assigning one unit clause can create another unit clause—for example, when \texttt{C} is set to \texttt{false}, \texttt{(C V A)} becomes a unit clause, causing \texttt{true} to be assigned to \texttt{A}. This "cascade" of forced assignments is called unit propagation. It resembles the process of forward chaining with definite clauses, and indeed, if the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining. (See Exercise 7.24.)

The DPLL algorithm is shown in Figure 7.17, which gives the the essential skeleton of the search process.

What Figure 7.17 does not show are the tricks that enable SAT solvers to scale up to large problems. It is interesting that most of these tricks are in fact rather general, and we have seen them before in other guises:

1. Component analysis (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called components that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.

2. Variable and value ordering (as seen in Section 6.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value \texttt{true} before \texttt{false}. The degree heuristic (see page 216) suggests choosing the variable that appears most frequently over all remaining clauses.