3. Intelligent backtracking (as seen in Section 6.3 for CSPs): Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of conflict clause learning to record conflicts so that they won’t be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.

4. Random restarts (as seen on page 124 for hill-climbing): Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.

5. Clever indexing (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as "the set of clauses in which variable \( X_i \) appears as a positive literal." This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

With these enhancements, modern solvers can handle problems with tens of millions of variables. They have revolutionized areas such as hardware verification and security protocol verification, which previously required laborious, hand-guided proofs.

7.6.2 Local search algorithms

We have seen several local search algorithms so far in this book, including HILL-CLIMBING (page 122) and SIMULATED-ANNEALING (page 126). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs (page 221). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 7.18). On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state and (2) a "random walk" step that picks the symbol randomly.

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns failure, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set \( \text{max flips} = \infty \) and \( p > 0 \), WALK SAT will eventually return a model (if one exists), because the random-walk steps will eventually hit
function \textsc{WALKSAT}(clauses, p, max\_flips) returns a satisfying model or failure
inputs: clauses, a set of clauses in propositional logic
\begin{itemize}
\item p, the probability of choosing to do a "random walk" move, typically around 0.5
\item max\_flips, number of flips allowed before giving up
\end{itemize}
model \leftarrow \text{a random assignment of true/false to the symbols in clauses}
for a, = 1 to max\_flips do
  if model satisfies clauses then return model
  clause \leftarrow \text{a randomly selected clause from clauses that is false in model}
  with probability \( p \) flip the value in model of a randomly selected symbol from clause
  else flip whichever symbol in clause maximizes the number of satisfied clauses
return failure

Figure 7.18 The \textsc{WALKSAT} algorithm for checking satisfiability by randomly flipping
the values of variables. Many versions of the algorithm exist

upon the solution. Alas, if \texttt{max\_flips} is infinity and the sentence is unsatisfiable, then the
algorithm never terminates!

For this reason, \textsc{WALKSAT} is most useful when we expect a solution to exist—for example,
the problems discussed in Chapters 1 and 6 usually have solutions. On the other hand,
\textsc{WALKSAT} cannot always detect \textit{unsatisfiability}, which is required for deciding entailment.
For example, an agent cannot \textit{reliably} use \textsc{WALKSAT} to prove that a square is \textit{safe}
in the wumpus world. Instead, it can say, "I thought about it for an hour and couldn't come up
with a possible world in which the square \textit{isn't} safe" This may be a good empirical indicator that
the square is safe, but it's certainly not a proof.

7.6.3 The landscape of random SAT problems

Some SAT problems are harder than others. \textit{Easy} problems can be solved by any old algo-
rithm, but because we know that SAT is NP-complete, at least some problem instances must
require exponential run time. In Chapter 6, we saw some surprising discoveries about certain
kinds of problems. For example, the n-queens problem—thought to be quite tricky for back-
tracking search algorithms—turned out to be trivially easy for local search methods, such as
min-conflicts. This is because solutions are very densely distributed in the space of assign-
ments, and any initial assignment is guaranteed to have a solution nearby. Thus, n-queens is
easy because it is under\textit{-}constrained.

When we look at satisfiability problems in conjunctive normal form, an
under\textit{constrained} problem is one with relatively \textit{few} clauses constraining the variables. For example,
here is a randomly generated 3-CNF sentence with five symbols and five clauses:
\[
\neg D \lor u \lor C \land (E \lor \neg D \lor B) \land (B \lor E \lor \neg C),
\]
Sixteen of the 32 possible \textit{assignments} are models of this sentence, so, on average, it would
take just two random guesses to find a model. This is an easy \textit{satisfiability} problem, as are