function \textsc{WalkSat}(\text{clauses}, p, \text{max}_\text{flips}) \text{ returns} \text{ a satisfying model or failure}

\textbf{inputs:} \text{clauses}, \text{a set of clauses in propositional logic}
\hspace{1cm} p, \text{the probability of choosing to do a “random walk” move, typically around 0.5}
\hspace{1cm} \text{max}_\text{flips}, \text{number of value flips allowed before giving up}

model \leftarrow \text{a random assignment of true/false to the symbols in clauses}

\textbf{for each} i = 1 \text{ to } \text{max}_\text{flips} \text{ \textbf{do}}
\hspace{1cm} \text{if model satisfies clauses \textbf{then return model}}
\hspace{1cm} \text{clause} \leftarrow \text{a randomly selected clause from clauses that is false in model}
\hspace{1cm} \text{if } \text{RANDOM}(0, 1) \leq p \text{ \textbf{then}}
\hspace{1.5cm} \text{flip the value in model of a randomly selected symbol from clause}
\hspace{1cm} \text{else} \text{ flip whichever symbol in clause maximizes the number of satisfied clauses}

\textbf{return} \text{ failure}

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

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.

\subsection*{7.6.2 Local search algorithms}

We have seen several local search algorithms so far in this book, including \textsc{Hill-Climbing} (page 111) and \textsc{Simulated-Annealing} (page 115). 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 \textsc{Mini-Conflicts} algorithm for CSPs (page 198). 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 \textsc{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 \textsc{WalkSat} returns a model, the input sentence is indeed satisfiable, but when it returns \textit{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}_\text{flips} = \infty \text{ and } p > 0, \textsc{WalkSat} will eventually return a model (if one exists), because the random-walk steps will eventually hit upon the