**Section 7.6. Effective Propositional Model Checking**

**function DPLL-SATISFIABLE??(s) returns true or false**
*inputs: s, a sentence in propositional logic*

-clauses ← the set of clauses in the CNF representation of s
-symbols ← a list of the proposition symbols in s

-return DPLL(clauses, symbols, { })

**function DPLL(clauses, symbols, model) returns true or false**

-if every clause in clauses is true in model then return true
-if some clause in clauses is false in model then return false

-P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
-if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
-P, value ← FIND-UNIT-CLAUSE(clauses, model)
-if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})

-P ← FIRST(symbols); rest ← REST(symbols)

-return DPLL(clauses, rest, model ∪ {P=true}) or DPLL(clauses, rest, model ∪ {P=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 TI-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 C is set to false, (C ∨ A) becomes a unit clause, causing true to be assigned to 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 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 true before false. The degree heuristic (see page 216) suggests choosing the variable that appears most frequently over all remaining clauses.
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
inputs: clauses, a set of clauses in propositional logic
        p, the probability of choosing to do a “random walk” move, typically around 0.5
        max_flips, number of flips allowed before giving up

model ← a random assignment of true/false to the symbols in clauses
for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause ← 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 WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

upon the solution. Alas, if max_flips is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

For this reason, WALKSAT is most useful when we expect a solution to exist—for example, the problems discussed in Chapters 3 and 6 usually have solutions. On the other hand, WALKSAT cannot always detect unsatisfiability, which is required for deciding entailment. For example, an agent cannot reliably use WALKSAT to prove that a square is 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 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. Easy problems can be solved by any old algorithm, 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 backtracking 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 assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, n-queens is easy because it is underconstrained.

When we look at satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively few clauses constraining the variables. For example, here is a randomly generated 3-CNF sentence with five symbols and five clauses:

\((\neg D \lor \neg B \lor C) \land (B \lor \neg A \lor \neg C) \land (\neg C \lor \neg B \lor E)\)
\(\land (E \lor \neg D \lor B) \land (B \lor E \lor \neg C)\).

Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are
most such underconstrained problems. On the other hand, an overconstrained problem has many clauses relative to the number of variables and is likely to have no solutions.

To go beyond these basic intuitions, we must define exactly how random sentences are generated. The notation $CNF_k(m, n)$ denotes a $k$-CNF sentence with $m$ clauses and $n$ symbols, where the clauses are chosen uniformly, independently, and without replacement from among all clauses with $k$ different literals, which are positive or negative at random. (A symbol may not appear twice in a clause, nor may a clause appear twice in a sentence.)

Given a source of random sentences, we can measure the probability of satisfiability. Figure 7.19(a) plots the probability for $CNF_3(m, 50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of the clause/symbol ratio, $m/n$. As we expect, for small $m/n$ the probability of satisfiability is close to 1, and at large $m/n$ the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. Empirically, we find that the “cliff” stays in roughly the same place (for $k=3$) and gets sharper and sharper as $n$ increases. Theoretically, the satisfiability threshold conjecture says that for every $k \geq 3$, there is a threshold ratio $r_k$ such that, as $n$ goes to infinity, the probability that $CNF_k(n, rn)$ is satisfiable becomes 1 for all values of $r$ below the threshold, and 0 for all values above. The conjecture remains unproven.

![Figure 7.19](image)

**Figure 7.19** (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio $m/n$. (b) Graph of the median run time (measured in number of recursive calls to DPLL, a good proxy) on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

Now that we have a good idea where the satisfiable and unsatisfiable problems are, the next question is, where are the hard problems? It turns out that they are also often at the threshold value. Figure 7.19(b) shows that 50-symbol problems at the threshold value of 4.3 are about 20 times more difficult to solve than those at a ratio of 3.3. The underconstrained problems are easiest to solve (because it is so easy to guess a solution); the overconstrained problems are not as easy as the underconstrained, but still are much easier than the ones right at the threshold.