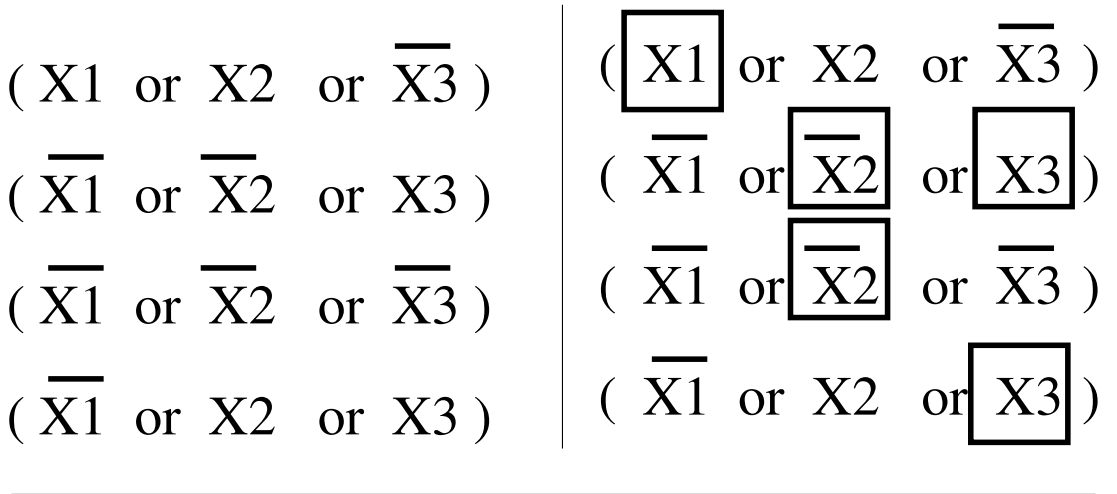

14.10 Satisfiability

Input description: A set of clauses in conjunctive normal form.

Problem description: Is there a truth assignment to the Boolean variables such that every clause is simultaneously satisfied?

Discussion: Satisfiability arises whenever we seek a configuration or object that must be consistent with (i.e. , satisfy) a set of logical constraints. A primary application is in verifying that a hardware or software system design works correctly on all inputs. Suppose a given logical formula $S(\bar{X})$ denotes the specified result on input variables $\bar{X} = X_1, \ldots, X_n$, while a different formula $C(\bar{X})$ denotes the Boolean logic of a proposed circuit for computing $S(\bar{X})$. This circuit is correct unless there exists an $\bar{X}$ such that $S(\bar{X}) \neq C(\bar{X})$.

Satisfiability (or SAT) is *the* original NP-complete problem. Despite its applications to constraint satisfaction, logic, and automatic theorem proving, it is perhaps most important theoretically as the root problem from which all other NP-completeness proofs originate. So much engineering has gone into today's best SAT solvers that they represent a reasonable starting point if one really needs to solve an NP-complete problem *exactly*. That said, employing heuristics that give good but nonoptimal solutions is usually the better approach in practice.

Issues in satisfiability testing include:

- *Is your formula the AND of ORs (CNF) or the OR of ANDs (DNF)?* – In satisfiability, constraints are specified as a logical formula. There are two

primary ways of expressing logical formulas—conjunctive normal form (CNF) and disjunctive normal form (DNF). In CNF formulas, we must satisfy all clauses, where each clause is constructed by and-ing or's of literals together, such as

$$(v_1 \text{ or } \bar{v_2}) \text{ and } (v_2 \text{ or } v_3)$$

In DNF formulas, we must satisfy any one clause, where each clause is constructed by or-ing ands of literals together. The formula above can be written in DNF as

$$(\bar{v_1} \text{ and } \bar{v_2} \text{ and } v_3) \text{ or } (\bar{v_1} \text{ and } v_2 \text{ and } \bar{v_3}) \text{ or } (\bar{v_1} \text{ and } v_2 \text{ and } v_3) \text{ or } (v_1 \text{ and } \bar{v_2} \text{ and } v_3)$$

Solving DNF-satisfiability is trivial, since any DNF formula can be satisfied unless *every* clause contains both a literal and its complement (negation). However, CNF-satisfiability is NP-complete. This seems paradoxical, since we can use De Morgan's laws to convert CNF-formulae into equivalent DNF-formulae, and vice versa. The catch is that an exponential number of terms might be constructed in the course of translation, so that the translation itself does not run in polynomial time.

- *How big are your clauses?* – k-SAT is a special case of satisfiability when each clause contains at most k literals. The problem of 1-SAT is trivial, since we must set true any literal appearing in any clause. The problem of 2-SAT is not trivial, but can still be solved in linear time. This is interesting, because certain problems can be modeled as 2-SAT using a little cleverness. The good times end as soon as clauses contain three literals each (i.e. , 3-SAT) for 3-SAT is NP-complete.

- *Does it suffice to satisfy* most *of the clauses?* – If you must solve it exactly, there is not much you can do to solve satisfiability except by backtracking algorithms such as the Davis-Putnam procedure. In the worst case, there are 2^m truth assignments to be tested, but fortunately there are many ways to prune the search. Although satisfiability is NP-complete, how hard it is in practice depends on how the instances are generated. Naturally defined "random" instances are often surprisingly easy to solve, and in fact it is nontrivial to generate instances of the problem that are truly hard.

  Still, we can benefit by relaxing the problem so that the goal becomes satisfying as many clauses as possible. Optimization techniques such as simulated annealing can then be put to work to refine random or heuristic solutions. Indeed, any random truth assignment to the variables will satisfy each k-SAT clause with probability $1 - (1/2)^k$, so our first attempt is likely to satisfy most of the clauses. Finishing off the job is the hard part. Finding an assignment that satisfies the maximum number of clauses is NP-complete even for nonsatisfiable instances.

When faced with a problem of unknown complexity, proving it NP-complete can be an important first step. If you think your problem might be hard, skim through Garey and Johnson [GJ79] looking for your problem. If you don't find it, I recommend that you put the book away and try to prove hardness from first principles, using the basic problems of 3-SAT, vertex cover, independent set, integer partition, clique, and Hamiltonian cycle. I find it much easier to start from these than some complicated problem in the book, and more insightful too, since the reason for the hardness is not obscured by the hidden hardness proof for the complicated problem. Chapter 9 focuses on strategies for proving hardness.

Implementations: Recent years have seen tremendous progress in the performance of SAT solvers. An annual SAT competition identifies the top performing solvers in each of three categories of instances (drawn from industrial, handmade, and random problem instances, respectively).

The top three programs of the SAT 2007 industrial competition were Rsat (*http://reasoning.cs.ucla.edu/rsat/*), PicoSAT (*http://fmv.jku.at/picosat/*) and MiniSAT (*http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/*). The source code for all these solvers and more are available from the competition webpage (*http://www.satcompetition.org/*).

SAT *Live!* (*http://www.satlive.org/*) is the most up-to-date source for papers, programs, and test sets for satisfiability and related logic optimization problems.

Notes: The most comprehensive overview of satisfiability testing is Kautz, et al. [KSBD07]. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a backtracking algorithm introduced in 1962 for solving satisfiability problems. Local search techniques work better on certain classes of problems that are difficult for DPLL solvers. *Chaff* [MMZ$^+$01] is a particularly influential solver, available at *http://www.princeton.edu/∼chaff/*. See [KS07] for a survey of recent progress in the field of satisfiability testing.

An algorithm for solving 3-SAT in worst-case $O^*(1.4802^n)$ appears in [DGH$^+$02]. Efficient (but nonpolynomial) algorithms for NP-complete problems are surveyed in [Woe03].

The primary reference on NP-completeness is [GJ79], featuring a list of roughly 400 NP-complete problems. The book remains an extremely useful reference; it is perhaps the book I reach for most often. An occasional column by David Johnson in the *Journal of Algorithms* and (now) *ACM Transactions on Algorithms* provides updates.

Good expositions of Cook's theorem [Coo71], where satisfiability is proven hard, include [CLRS01, GJ79, KT06]. The importance of Cook's result became clear in Karp's paper [Kar72], showing the hardness of over 20 different combinatorial problems.

A linear-time algorithm for 2-SAT appears in [APT79]. See [WW95] for an interesting application of 2-SAT to map labeling. The best heuristic known approximates maximum 2-SAT to within a factor of 1.0741 [FG95].

Related Problems: Constrained optimization (see page 407), traveling salesman problem (see page 533).