Math 416

Worksheet 23. Satisfiability

Satisfiability. Suppose that we have a supply of **boolean variables** $x_1, \ldots, x_n$, each of which can take the value 0 (i.e., 'false') or 1 (i.e., 'true').

- A **term** is either one of these variables or its negation: x_k or $\neg x_k$.
- A **clause** is a disjunction of terms: e.g. $x_1 \vee x_3 \vee \neg x_5$. (The symbol $\vee$ means OR, and the symbol $\wedge$ means AND.)
- A **truth assignment** is a function $v\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$.
- A truth assignment v **satisfies** a clause C iff it causes C to evaluate to true ($= 1$) under the rules of Boolean logic.

Problem 1. Consider $n = 3$ and the truth assignment

$$v\colon \begin{cases} x_1 & \longmapsto & 0 \\ x_2 & \longmapsto & 1 \\ x_3 & \longmapsto & 0 \end{cases}$$

Find a clause that v satisfies, and find one that it does not satisfy.

Definition. A truth assignment v satisfies a collection $C_1, \ldots, C_k$ of clauses iff it satisfies *all* of $C_1, \ldots, C_k$, i.e., iff v causes the conjunction

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

to evaluate to true ($= 1$) under the rules of Boolean logic.

- We say that the set $\{C_1, \ldots, C_k\}$ of clauses is **satisfiable** iff there is a truth assignment v that satisfies it.
- The algorithmic problem SAT is this: given a set of clauses $C_1, \ldots, C_k$, determine whether it's satisfiable.
- An important special case is 3-SAT: given a set of clauses $C_1, \ldots, C_k$, each C_l of *length* 3, determine whether it's satisfiable.

3-SAT (and hence SAT) is computationally hard: there is no known polynomial-time algorithm, and other computationally hard problems reduce to 3-SAT.

Note! 3-SAT exhibits one-sided difficulty: if you are given a satisfying assignment, it is easy to check that it works.

An approximation algorithm for 3-SAT. What if we turn 3-SAT into an optimization problem? We could try to find a truth assignment satisfying as many clauses among $C_1, \ldots, C_n$ as possible.

The simplest thing to do would be to simply assign truth values independently at random. Define the following random variables:

$$Z_j = \begin{cases} 1 & \text{if } C_j \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

And let Z be the number of satisfied clauses.

Problem 2.

(a) How are Z and $Z_1, \ldots, Z_n$ related?
(b) Fix j. What is the probability that C_j is not satisfied? (Remember that each C_j is a clause of size 3.)
(c) Find $E[Z]$.

Problem 3. Explain why the following Theorem is true. Make sure that you understand its significance; notice that it does not mention probability anywhere!

Theorem. For every instance of 3-SAT there is a truth assignment satisfying at least 7/8 of all clauses.

Corollary. Every instance of 3-SAT with ≤ 7 clauses is satisfiable.

Problem 4. Explain why the Corollary follows from the Theorem.

Problem 5 (For fun, if you want)**.** Write down 7 clauses (each of 3 terms) in, say, 5 boolean variables, and find a truth assignment that satisfies all of them.

Problem 6 (2-SAT)**.** Describe and analyze a polynomial-time algorithm for solving 2-SAT. (So you must determine satisfiability of all sets of clauses each of size 2.)

(*Hint:* This is hard if you haven't seen it before. You can think of a length two clause as an implication (why??), and then try to construct a directed graph and consider the strongly connected components. This is a fun exercise, but if you get frustrated just move on!)

Poly-time reductions. Remember that we defined an *efficient* algorithm as one that runs in polynomial time in the size of the input.

(To make this precise, we really need a real *model of computation* — a standard one is the *Turing Machine* — but we will continue to pretend with our informal model of computation.)

Definition. If X and Y are two *computational problems*,[1] and there is an algorithm solving X that runs in polynomial time and is allowed to make 'black-box' calls to an oracle for Y, then we write $X \leq_P Y$ and say that X **is polynomial-time-reducible to** Y.

For example, suppose that Y itself can be solved in polynomial time. Then we can replace each black-box call to an oracle for Y to a call to an (efficient) subroutine solving Y, and so:

Lemma. If $X \leq_P Y$ and Y can be solved in polynomial time, then so can X.

Problem 7. Suppose that $X \leq_P Y$. Explain why, if X cannot be solved in polynomial time, then neither can Y.

Definition. We write P for the set of problems that can be solved in polynomial time (without use of any oracle).

Problem 8. Explain why $X \leq_P Y$ and $Y \leq_P Z$ imply $X \leq_P Z$.

Definition. A **decision problem** is a set X of finite binary strings (or a set of strings over any finite alphabet). An algorithm A for a decision problem receives an input string s and returns the value 0 (for *false*) or 1 (for *true*). We say that A **solves** the problem X if for all strings s, we have $A(s) = 1$ iff $s \in X$.

We say that A has **polynomial running time** if there is a polynomial function p so that for every input string s, the algorithm A terminates on s in at most $O(p(\mathrm{lh}(s)))$ computation steps. We write P for the set of decision problems X for which there exists an algorithm with polynomial running time that solves X.

Problem 9. Explain how SAT and 3-SAT are (can be coded as) decision problems.

Definition. We say that B is an **efficient certifier** for a problem X if it has the following properties.

(i) B is a polynomial-time algorithm that takes two input arguments s and t.

(ii) There is a polynomial p so that for every string s we have $s \in X$ iff there is a string t for which $\mathrm{lh}(t) \leq p(\mathrm{lh}(s))$ and $B(s,t) = 1$.

[1] You can take *computational problem* to mean subset X of $\mathbb{N}$; and an algorithm solves X iff the algorithm returns 1 on input x if $x \in X$ and 0 otherwise. But it takes some thought to convince yourself that this definition captures all the computational problems we want.

We think of the t as a *certificate* or *proof* that $s \in X$, and so $s \in X$ iff the certifier can certify $s \in X$ efficiently.

Problem 10. Briefly describe an efficient certifier for SAT.

Problem 11. Explain how an efficient certifier can be used as the core component of a brute-force algorithm for a problem X. What is its running time?

(*Hint:* Try all proofs.)

Definition. The class NP is the class of decision problems for which there exists an efficient certifier.

One of the most important open questions in math/theoretical computer science is...

Question. Does $\mathsf{P} = \mathsf{NP}$?

Most researchers believe the answer is *No*, but we don't have a proof. It might interest you to know that all of digital cryptography depends on the assumption that some problems in NP are not in P.

Problem 12. Prove that $\mathsf{P} \subseteq \mathsf{NP}$.

(*Hint:* Assume that A is a poly-time algorithm that solves X; show that $(s, t) \mapsto A(s)$ is an efficient certifier for X.)

In the absence of a proof of $\mathsf{P} \neq \mathsf{NP}$, we analyze how far from being in P some hard problems in NP are.

Definition. A decision problem X is **NP-complete** iff $X \in \mathsf{NP}$ and $Y \leq_P X$ for all $Y \in \mathsf{NP}$.

Problem 13. Suppose that X is NP-complete. Prove that $X \in \mathsf{P}$ iff $\mathsf{P} = \mathsf{NP}$.

Conclude that, in order to prove $\mathsf{P} \neq \mathsf{NP}$, it is enough to find an NP-complete problem and show that it isn't in P.

Problem 14. Suppose that X is NP-complete and that $X \leq_P Y$. Prove that if $Y \in \mathsf{NP}$ then Y is NP-complete too.

Theorem. SAT is NP-complete. In fact, $3\text{-}\mathsf{SAT}$ is NP-complete.