NP-completeness III

P vs NP

One of the biggest open questions in Computer Science is the following:

Open Question: Is P equal to NP ?

(This literally is a million dollar question; for details check `www.claymath.org/index.htm`)

Unfortunately, we are currently unable to prove whether or not P is equal to NP. However, it seems very unlikely that these classes are equal. If they were equal, all of the search problems mentioned so far in these notes would be solvable in polynomial time, because they are reducible (in the sense of $\xrightarrow{p}$) to a language in NP. More generally, most combinatorial optimization problems would be solvable in polynomial time, for essentially the same reason.

Related to this is the following lemma. This lemma says that if P = NP, then if $L \in$ NP, then for every string x, not only would we be able to compute in polynomial-time whether or not $x \in L$, but in the case that $x \in L$, we would also be able to actually find a certificate y that demonstrates this fact. That is, for every language in NP, the associated search problem would be polynomial-time computable. That is, whenever we are interested in whether a short string y exists satisfying a particular (easy to test) property, we would automatically be able to efficiently find such a string if one exists.

Lemma 1 Assume that P = NP. Let $R(x, y)$ be a polytime predicate, and let p be a polynomial, and define $L = \{x | \exists y(|y| \leq p(|x|) \wedge R(x, y))\}$. Then, there is a polytime Turing machine M with the following property: for every $x \in L$, if M is given x, then M outputs a string y such that $|y| \leq p(|x|)$ and $R(x, y)$.

PROOF: Define the language $L' = \{\langle x, w\rangle | \exists z(|wz| \leq c|x|^k \wedge R(x, wz))\}$. Here $\langle x, y\rangle$ denotes the encoding of the string x, y as a single string, and wz denotes the sting w appended by the string z.

It is easy to see that $L' \in$ NP (exercise), and so, since we assumed that P = NP, $L' \in$ P.

We construct the machine M to work as follows. Let $x \in L$. Using a polytime algorithm for L', we will construct a certificate $y = b_1 b_2 \dots$ for x, one bit b_i at a time. We begin by checking if $\langle x, 0\rangle \in L'$; if so, we let $b_1 = 0$, if not, we let $b_1 = 1$. We now check if $R(x, b_1)$, holds, and if it does we let $y = b_1$; if it does not hold we check if $\langle x, b_1 0\rangle \in L'$, and we continue this way for at most $p(|x|)$ steps, until we obtain the certificate y. $\square$

This lemma has some powerful consequences. It implies that if P = NP, then virtually all cryptography could be easily broken. The reason for this is that for most cryptography (except for a special case called "one-time pads"), if we are lucky enough to guess the secret key, then we can verify that we have the right key; thus, if P = NP, then we can actually find the secret key, and break the cryptosystem.

For the above reason, most computer scientists conjecture that P $\neq$ NP.

Conjecture: P $\neq$ NP

For any language L_0, if L_0 is NP-complete and $L_0 \in$ P, then P=NP. For example, since SKD is NP-complete, if SKD is in P, then P=NP.

Lemma 2 Let L be an NP-complete language. Then, $L \in$ P iff P=NP.

PROOF: [$\Longrightarrow$] Suppose that L is NP-complete, and $L \in$ P. Let L' be any language in NP. Then, by definition of NP-completeness, $L' \leq_p L$. But, since $L \in$ P, so is L', and therefore NP $\subseteq$ P. Since we have already shown that P $\subseteq$ NP, we have P = NP.

[$\Longleftarrow$] Easy: if P = NP, then any language which is in NP (and by definition all NP-complete languages are in NP) is also in P, so $L \in$ P $\square$

Given that we can't prove that P $\neq$ NP, is there any way we can gain confidence that a particular language $L \in$ NP is not in P? If we can prove that L is NP-complete, this will imply that if $L \in$ P, then P = NP; we will take this as being very strong evidence that $L \notin$ P.

Thus, proving that a decision problem in NP-complete is our way of gaining confidence about the lack of efficient algorithms to solve this problem (or its corresponding search version).

Showing that problems are NP-complete

Lemma 3 (Composition) Suppose $f : \Sigma^* \longrightarrow \Sigma^*$ and $g : \Sigma^* \longrightarrow \Sigma^*$ are two polytime functions and $h = f \circ g$, i.e., $h(x) = f(g(x))$. Then $h : \Sigma^* \longrightarrow \Sigma^*$ is also polytime.

PROOF: Since f, g are polytime functions, by definition they can be computed by polytime Turing machines M_f and M_g. This means that there exist polynomials p_f and p_g, such that given any y and x, the number of steps taken by M_f and M_g to compute $f(y)$ and $g(x)$, respectively, is bounded by $p_f(|y|)$ and $p_g(|x|)$, respectively.

Also, since the number of steps taken by M_g to compute $g(x)$ is bounded by $p_g(|x|)$, it follows that $|g(x)| \leq p_g(|x|)$, since in $p_g(|x|)$ steps M_g cannot get beyond $p_g(|x|)$ squares. Note that we can assume that p_f, p_g have positive coefficients (they are upper bounds, so if we switch the signs of all coefficients to be positive, the result is still an upper bound).

Taking all this into account, we define a polytime Turing machine M_h that computes h. M_h first simulates M_g on input x (this takes $p_g(|x|)$ steps), and then M_h switches to simulate M_f on $g(x)$ (this takes $p_f(|g(x)|) \leq p_f(p_g(|x|))$ steps—here we use the fact that p_f and p_g have only positive coefficients (how?)).

Thus, all-together M_h takes at most $p_h(|x|) = p_g(|x|) + p_f(p_g(|x|))$ steps (p_h is a polynomial, as $p_g + (p_f \circ p_g)$ is a polynomial). $\square$

Lemma 4 (Transitivity) If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

PROOF: Assume that $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$. Then:

$$\begin{aligned} x \in L_1 &\iff g(x) \in L_2 \qquad \text{for all } x \in \Sigma^* \\ y \in L_2 &\iff f(y) \in L_3 \qquad \text{for all } y \in \Sigma^* \end{aligned}$$

where f and g are polytime functions. We want to show $L_1 \leq_p L_3$, i.e., we need to find a polytime function $h(x)$ such that $x \in L_1 \iff h(x) \in L_3$ for all $x \in \Sigma^*$. This property holds if we define $h(x) = f(g(x))$, and by the Composition Lemma above $h = f \circ g$ is polytime. $\square$

Lemma 5 (NP-completeness) If L_0 is NP-complete, and $L_1 \in$ NP, and $L_0 \leq_p L_1$, then L_1 is NP-complete.

PROOF: Assume that $L_0 \leq_p L_1$ and L_0 is NP-complete, and $L_1 \in$ NP. To show that L_1 is NP-complete, we must show that given any language $L \in$ NP, $L \leq_p L_1$. Since L_0 is NP-complete, for any language $L \in$ NP, $L \leq_p L_0$, and since $L_0 \leq_p L_1$, the result follows by transitivity. $\square$

Thus, to show that a problem is NP-complete, we need to show that a previously known NP-complete problem is reducible to it. *Important*: Make sure that you get the reduction in the right direction!

SAT

SAT (for SATisfiability) is the decision problem for boolean formulas in Conjunctive Normal Form (CNF). The problem is: given a formula ϕ in CNF, is this formula satisfiable by some truth assignment to its variables? Often, instead of giving a formula in CNF, the problem consists in deciding whether a given set of clauses is satisfiable; the two version of the problem are equivalent, and we use them both, mixing notation as we find convenient.

We now define this problem more precisely:

Start with a set of *Boolean variables* $x, y, z, w, \ldots$

A *literal* is either a variable x or a negated variable $\bar{x}$.

A *clause* is a finite disjunction $l_1 \vee l_2 \vee \ldots \vee l_k$ of distinct literals, $k \geq 0$. If $k = 0$, then the clause is assumed to be empty.

A *truth assignment* to a set of clauses is a map $t : U \longrightarrow \{T, F\}$, where U is the set of variables appearing in the set of clauses, and $T =$ True and $F =$ False.

$$t(\bar{x}) = \begin{cases} T & \text{if } t(x) = F \\ F & \text{if } t(x) = T \end{cases}$$

If C is a clause, $C = l_1 \vee l_2 \vee \ldots \vee l_k$, then t *satisfies* C if it makes it true, i.e., $t(l_i) = T$ for some $1 \leq i \leq k$.

Let $S = \{C_1, C_2, \ldots, C_m\}$ be set of clauses. Then t satisfies S iff t satisfies every clause in S, i.e. t *satisfies* $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ (which is our CNF formula).

Decision problem SAT:

Input: Finite set $\{C_1, \ldots, C_m\}$ of clauses.

Question: Is S satisfiable? That is, is there a truth assignment t which satisfies S?

For example, $S = \{x \vee \bar{y} \vee z, \bar{x}, \bar{y} \vee \bar{z}, y \vee z\}$. S is satisfiable and a satisfying assignment is $t(x) = F$, $t(y) = F$, $t(z) = T$.

Theorem 1 (Cook, Levin 1971) SAT is NP-complete.

Thus SAT is our first NP-complete problem.

It is easy to show that SAT is in NP: Given $S = \{C_1, \ldots, C_m\}$, let U be the set of variables in S. Guess a truth assignment t. Check that t satisfies S. It is possible to decide in polynomial time whether t satisfies S, given both S and t.

Note: If a set S has u different variables, then there are 2^u different truth assignments.

The difficult part is demonstrating that for every language $L \in$ NP $L \leq_p$ SAT. We are *not* going to prove this — check Garey and Johnson if you are interested.

3SAT:

Input: Set $S = \{C_1, \ldots, C_m\}$ of clauses, such that every clause has exactly 3 literals.

Question: Is S satisfiable?

Theorem 2 3SAT is NPC

Proof: 3SAT $\in$ NP. Use the same guess and check algorithm as for SAT (and check that input has the right format).

To show 3SAT is NP-complete, use the NP-completeness lemma and the fact that SAT is NP-complete. We must show that SAT $\leq_p$ 3SAT.

Given an input instance $S = \{C_1, \ldots, C_m\}$ to SAT, we must compute (in polytime) an input $S' = \{C'_1, \ldots, C'_{m'}\}$ to 3SAT, such that S is satisfiable iff S' is satisfiable. We need to "shrink" the existing clauses and introduce more clauses. For that, we introduce new variables.

Suppose $C_1 = l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_k$, where $k \geq 4$. Let p be a new variable (p does *not* occur in

S). Make p equivalent to $l_1 \vee l_2$ by introducing clauses, i.e. introduce 3 new clauses:

$$\begin{aligned}
&\text{clause } D_1 \text{ given by } \bar{p} \vee l_1 \vee l_2 \\
&\text{clause } D_2 \text{ given by } p \vee \bar{l_1} \\
&\text{clause } D_3 \text{ given by } p \vee \bar{l_2}
\end{aligned}$$

These three clauses assert that for any truth assignment t, $t(p)$ is equal to $t(l_1 \vee l_2)$, provided that t satisfies $D_1 \wedge D_2 \wedge D_3$. Let $C_1' = p \vee l_3 \vee \ldots \vee l_k$ (C_1 shrunk by 1).

Claim 1 The new set of clauses $S' = \{D_1, D_2, D_3, C_1', C_2, \ldots, C_m\}$ is satisfiable iff S is satisfiable.

Proof of claim: If t satisfies S', then t also satisfies S, because $t(p) = t(l_1 \vee l_2)$, since t satisfies $D_1 \wedge D_2 \wedge D_3$.

Conversely, suppose t satisfies S. Is S' satisfiable? Let t' be the same as t, except let $t'(p) = t(l_1 \vee l_2)$. Then t' satisfies $C_2, \ldots, C_m$ because p does not occur in these clauses. Also, t' satisfies D_1, D_2 and D_3, and t' satisfies $C_1' = p \vee l_3 \vee \ldots \vee l_k$, because t satisfies $C_1 = l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_k$.

We can continue doing this $k - 3$ times, until C_1 is reduced to 3 literals. Now proceed to reduce C_2 in the same way.

This can be computed in polytime. The resulting S' will be satisfiable iff the original S is satisfiable.

Given $l_1 \vee l_2$, expand it to $\{l_1 \vee l_2 \vee p, l_1 \vee l_2 \vee \bar{p}\}$, where p is any variable, not necessarily new. Then t satisfies $l_1 \vee l_2$ iff t satisfies both ($l_1 \vee l_2 \vee p$ and $l_1 \vee l_2 \vee \bar{p}$).

Exercise: What do you do in the case that $C = l_1$, i.e. C is a clause consisting of a single literal?