Computability and Complexity

Lecture 3, Tuesday August 8, 2023

Ari Feiglin

Definition 3.1:

A decision problem S is called **NP-hard** if for every $S' \in \mathbf{NP}$, there exists a Karp reduction from S' to S. The set of all **NP**-hard problems is denoted **NPH**. An **NP**-hard decision problem isn't necessarily in **NP**, but if it is, it is called **NP-complete**. The set of all **NP**-complete problems is denoted **NPC**.

So an **NP**-hard problem is a problem which is at least as hard as every **NP** hard (hence why it is called **NP**-*hard*).

Suppose that if S_1 is **NP**-hard, and g is a Karp reduction from S_1 to S_2. Then for every $S \in \mathbf{NP}$, there exists a Karp reduction f from S to S_1. Then $g \circ f$ is a Karp reduction from S to S_2, since

$$x \in S \iff f(x) \in S_1 \iff g(f(x)) \in S_2$$

And so S_2 is also **NP**-hard. Thus we can think of the set of **NPH** as being *upward closed*, meaning if S is **NP**-hard, and S' is harder than S then S' is also **NP**-hard.

Proposition 3.2:

There exists an **NP**-complete problem.

Proof:

Let U be a universal Turing machine, ie. a Turing machine where given an input (M, ω) of a Turing machine M and an input ω, it runs M on ω. We can assume that for each transition of M, U takes linear (or worst case, polynomial, but this will not affect our proof) time, with respect to the length of (M, ω).

Let us define the decision problem

$$S_U = \left\{ (M, \omega, t) \;\middle|\; \begin{array}{l} M \text{ is a Turing machine, } \omega \text{ is some binary input string, and } t \text{ is a number in unary} \\ \text{such that there exists some } y \text{ whose length is at most } t, \text{ such that } M \text{ accepts } (\omega, y) \\ \text{within } t \text{ steps} \end{array} \right\}$$

We will show that S_U is **NP**-complete.

Firstly, S_U is in **NP**. We can define a verifier $V((M, \omega, t), y)$ which simply runs $U(M, (\omega, y))$ (ie. we run M on the input (ω, y)) for t transitions. If M accepts (ω, y) within t transitions, V should accept, and otherwise reject. This is polynomial time, as the time for U to run each transition $M(\omega, y)$ is polynomial with respect to the length of (M, ω, y), and since we run this t times, and $|y| \leq t$ (which is in unary), we see that the runtime of V is polynomial with respect to (M, ω, t). Thus V is a polynomial proof system for S_U, and so $S_U \in \mathbf{NP}$.

The purpose of having t in unary is since otherwise V would run in polynomial time with respect to the length of (M, ω, t, y), but if t is binary then $|y| \leq t$ means that $|y| \leq 2^{|t|}$, and so V would not run in polynomial time with respect to (M, ω, t).

Now we must show that S_U is **NP**-hard. Suppose S is in **NP**, and so it has a polynomial proof system $V(x, y)$ and its polynomial p. Since V runs in polynomial time, its time complexity is bound by a polynomial q. Let us define

$$t(n) = q(n + p(n))$$

So t is a polynomial bound for V's runtime with respect to n, the length of x. That is $V(x, y)$ runs in $t(|x|)$ time.

All that remains is for us define a Karp reduction f from S to S_U. If $x \in S$ then there exists a y such that $|y| \leq p(|x|)$ and $V(x, y) = 1$. And if $x \notin S$, then no matter what y we have $V(x, y) = 0$. So let us define

$$f(x) = (V, x, t(|x|))$$

If $x \in S$, then there exists a y such that $|y| \leq t(|x|)$ and $V(x, y)$ accepts within $t(|x|)$ steps, so $f(x) \in S_U$. Otherwise, there does not exist a y such that $V(x, y) = 1$, and so $f(x) \notin S_U$. So we have that

$$x \in S \iff f(x) \in S_U$$

Computing $t(|x|)$ takes polynomial time, and so f takes polynomial time, and therefore f is a Karp reduction, as required. ■

Definition 3.3:

A **boolean circuit** is a directed acyclic graph where each vertex is one of the following types:

(1) Input: has no incoming edges, and at least one outgoing edge.

(2) Output: has no outgoing edges, and at exactly one incoming edge.

(3) Gate: one or two incoming edges, and at least one outgoing edge.

Each gate also contains a boolean or unary operation (depending on the number of incoming edges).

If C is a boolean circuit with n input edges and m output edges, we say that the output of C on $\tau \in \{0,1\}^n$ is the value obtained by computing the logical operations in the gates on the values in τ, which becomes a boolean vector in $\{0,1\}^m$.

If a boolean circuit has one output edge, we say that a boolean vector τ of length n satisfies it, if by inputting τ into the circuit, the output is 1.

Lemma 3.4:

Every boolean function $f\colon \{0,1\}^n \longrightarrow \{0,1\}^m$ has a boolean circuit which computes it.

Proof:

Suppose that $m = 1$, then this is simply a boolean function $f\colon \{0,1\}^n \longrightarrow \{0,1\}$, and we know this can be written as a composition of ands, ors, and nots. Composing these together creates a boolean circuit which computs f. And if $m > 1$, then $f = (f_1, \ldots, f_m)$ and so we can create boolean circuits for $f_1, \ldots, f_m$ and simply place them together to share input nodes. ■

Theorem 3.5 (The Cook-Levin Theorem):

The decision problem

$$\text{SAT} = \{\varphi \mid \varphi \text{ is a satisfiable boolean formula in CNF}\}$$

is **NP**-complete.

Proof:

Let us define the following decision problem

$$\text{CSAT} = \left\{ C \;\middle|\; \begin{array}{l} C \text{ is a boolean circuit with one output gate, and there exists an boolean vector as an} \\ \text{input which satisfies the circit} \end{array} \right\}$$

CSAT is in **NP**, as we can create a verifier $V(C, \tau)$ which simply valuates τ in C and checks if it is 1. This obviously is a verifier for CSAT, and it runs in polynomial time since all it must do is compute a boolean operation for each node in C. And if $C \in \text{CSAT}$ then the input which satisfies it must have a length less than $|C|$, as its length is the number of input nodes in C, so $p(x) = x$ satisfies the condition for V to be a polynomial proof system. Thus $\text{CSAT} \in \mathbf{NP}$ as required.

Let $S \in \mathbf{NP}$, then suppose V and p form a polynomial proof system for S. Suppose V's runtime is bound by the polynomial q. Let us define, as before,

$$t(n) = q(n + p(n))$$

And so $V(x, y)$'s runtime is bound by $q(|x|)$. We must define a Karp reduction from S to CSAT.

If $x \in S$, let $n = |x|$, and so we can define A, a $t(n) \times t(n)$ matrix where each row corresponds to the state of $V(x, y)$ (as a Turing machine). So the first row corresponds to the the initial state:

$$(x_1, q_0) \quad (x_2, \sqcup) \quad \cdots \quad (x_n, \sqcup) \quad (y_1, \sqcup) \quad \cdots \quad (y_m, \sqcup) \quad (\sqcup, \sqcup) \quad \cdots$$

(ie. the current state is q_0, and the pointer is pointing at x_1. The content of the tape is $x_1 \cdots x_2 y_1 \cdots y_m$, ie. xy.) Since the runtime of $V(x, y)$ is $t(n)$, the number of cells used is bound by $t(n)$ as well, hence why the matrix is $t(n) \times t(n)$.

Notice that the contents of a cell in A is dependent only on the three cells above it, that is if we know the contents of the three cells above it, then we know the content of that cell. This is since the value in that cell changes from its value in the row above only if the pointer is on it on the row above, and it writes to the cell, or if the pointer is on a neighbor of the cell and the pointer moves to the cell. So for example if the three cells are $(a, \sqcup)$, $(b, \sqcup)$, (c, q_n), we can determine based on this state what V will write to c, where the pointer will move, and what the next state will be. Suppose it writes d, moves to the left, and goes to the state q_{n+1}, then we know that the states in the next row will be $(a, \sqcup)$, (b, q_{n+1}), $(d, \sqcup)$.

So if we denote the length of a cell in A by N, there exists a function $\{0,1\}^{3N} \to \{0,1\}$ where given three neighboring cells in A, it computes the middle cell in the next row. Since each cell in A contains a character x_i, and a state q_i (or a blank), and given an encoding of V, we can keep N constant (take the maximum length of the encoding of some state, and add one for the character x_i).

So if for each cell in A we create a boolean circuit which corresponds to that function $\{0,1\}^{3N} \to \{0,1\}$, there are $t(n)^2$ such circuits. We compose these together and we get a boolean circuit which creates a circuit which computes $V(x, y)$, and let us evaluate the input nodes corresponding to x, and this gives us a boolean circuit C_x whose input nodes correspond to y. And so $C_x(y)$ computes $V(x, y)$.

Computing C_x takes $O(t(n)^2)$ time to define and compose each of the smaller circuits together, and then simplifying the circuit by evaluating it on the values of x is negligible compared to this. And so C_x takes polynomial time to compute with respect to x. And so the function $x \mapsto C_x$ is a polynomial time mapping, and it is a Karp reduction since if $x \in S$ then there exists a y such that $V(x, y) = C_x(y) = 1$, so $C_x \in$ CSAT. And if $x \notin S$, then no y satisfies $V(x, y)$ and thus $C_x(y) = 0$, and so $C_x \notin$ CSAT.

Our next step is to define a Karp reduction from CSAT to SAT. Since **NPH** is upward-closed, this means that SAT is in **NPH**, and since SAT is in **NP**, this means that SAT is **NP**-complete.

Let the input nodes of C be $x_1, \dots, x_n$ and the gates be $g_1, \dots, g_m$. Suppose g is a gate which accepts inputs from a_1 and a_2, then we want the CNF of the formula $g \equiv a_1 * a_2$ where $*$ is g's operation. This can be constructed naively (it is the conversion of a three variable formula to CNF, which takes constant time). If $g = g_i$, let us denote this by φ_i.

Now, we have one more formula which we need, as we have not considered the output node. If a is the node which goes into the output edge, we define φ_{m+1} to be a (since it is a gate or input, so it is a variable). Then we define

$$\varphi = \varphi_1 \wedge \cdots \wedge \varphi_{m+1}$$

Since each φ_i is in CNF, so is φ. This takes polynomial time, as we must compute $m + 1$ formulas in CNF, but each one takes constant time.

Now, if $C \in$ CSAT then there exists $x_1, \dots, x_n$ which satisfies C. If we run $x_1, \dots, x_n$ we get values in the gates $g_1, \dots, g_n$, and these values for $x_1, \dots, x_n, g_1, \dots, g_m$ satisfies φ, since φ_i is satisfied for $i \leq m$ as the values for g_i are obtained by running C on $x_1, \dots, x_n$, and φ_{m+1} is satisfied since the final gate (which connects to the output node), must be true (as C is satisfied), and so φ_{m+1} is satisfied as well. So all φ_i are satisfied, and therefore φ is, and so $\varphi \in$ SAT.

And if $C \notin$ CSAT, then for any valuations of $x_1, \dots, x_n, g_1, \dots, g_m$, $C(x_1, \dots, x_n)$ will not be satisfied. Let us assume that φ is satisfied. Then each φ_i is satisfied for $i \leq m$, so the values of g_i are the same as what they'd be when running C on $x_1, \dots, x_n$, but since $C(x_1, \dots, x_n)$ is not satisfied, the final gate (which connects to the output) cannot be satisfied and so φ_{m+1} is unsatisfied, in contradiction.

So the mapping from C to φ is a Karp reduction, and so CSAT $\leq$ SAT, meaning SAT is **NP**-complete, as required. ■

Corollary 3.6:

The decision problems

$$\text{IS} = \{(G, k) \mid G \text{ is a graph with an independent set of size} \geq k\}$$
$$\text{doubleIS} = \{(G, k) \mid G \text{ has two distinct independent sets whose size is at least } k\}$$
$$\text{almostIS} = \{(G, k) \mid G \text{ has an almost independent set whose size is at least } k\}$$

is **NP**-complete.

Last recitation we defined a Karp reduction from SAT to IS, and so IS is **NP**-complete. And we also defined Karp

reductions from IS to doubleIS and almostIS, so they too are **NP**-complete.