Lecture 23

NP Hardness and Completeness

In this lecture, we will introduce the notion of *hardness* and *completeness* for complexity classes. (You may have encountered these in other courses.) We will then go over the Cook-Levin theorem, which shows that some natural languages are NP-complete.

23.1 Polynomial-time mapping reductions

Recall that a *mapping reduction* from $A \subseteq \Sigma^*$ to $B \subseteq \Gamma^*$ was a computable function $f\colon \Sigma^* \to \Gamma^*$ such that for all $x \in A$, we have $f(x) \in B$, and for all $x \notin A$, we have $f(x) \notin B$. Such a function allows us to decide decide A if we know how to decide B: given input x, we just apply $f(x)$ and then check if the resulting string is in B. We denoted this by $A \leq_m B$, which signalled that A was "less hard" than B: if B is decidable, then A is decidable, but the reverse need not hold.

We now introduce the notion of a *polynomial time* mapping reduction. We will use the fixed alphabet $\{0,1\}$ for simplicity.

Definition 23.1 (Polynomial-time mapping reduction)**.** *A function $f\colon \{0,1\}^* \to \{0,1\}^*$ is called a polynomial-time mapping reduction between languages A and B if it is a mapping reduction between A and B, and if f is computable by a Turing machine whose running time is a polynomial function of the input size.*

If there is a polynomial-time mapping reduction from A to B, we write $A \leq_m^p B$.

Note that if $B \in \mathsf{P}$ and $A \leq_m^p B$, then $A \in \mathsf{P}$. This because to decide if $x \in A$, we can apply $f(x)$ (which takes polynomial time in $|x|$, and gives an output string $f(x)$ that must be polynomial size in $|x|$ because it was produced in polynomial time), and then apply the polynomial-time algorithm for deciding B on the input $f(x)$. If computing f takes $O(n^k)$ time and if deciding B takes $O(n^\ell)$ time, this algorithm for A takes $O(|x|^{k\ell})$ time, because we are running the $O(n^\ell)$ algorithm for B on the input $f(x)$ of size at most $O(|x|^k)$. Since k and ℓ are constants, the running time $O(n^{k\ell})$ is still polynomial, so $A \in \mathsf{P}$.

This same argument also holds for larger complexity classes! Let's put this in a theorem.

Theorem 23.2. *Let A and B be languages, and suppose that $A \leq_m^p B$. Then*

1. *If $B \in \mathsf{P}$, then $A \in \mathsf{P}$.*
2. *If $B \in \mathsf{NP}$, then $A \in \mathsf{NP}$.*
3. *If $B \in \mathsf{coNP}$, then $A \in \mathsf{coNP}$.*

4. *If* $B \in \mathsf{PSPACE}$*, then* $A \in \mathsf{PSPACE}$.

5. *If* $B \in \mathsf{EXP}$*, then* $A \in \mathsf{EXP}$.

6. *If* $B \in \mathsf{NEXP}$*, then* $A \in \mathsf{NEXP}$.

7. *If* $B \in \mathsf{EXPSPACE}$*, then* $A \in \mathsf{EXPSPACE}$.

Proof. Most of these are straight forward; for example, if $B \in \mathsf{PSPACE}$, then we know that we can decide B in polynomial space; to decide if x is in A, we can simply apply the mapping reduction f to x (which takes polynomial time, and hence polynomial space), and then run the algorithm for B on $f(x)$. This strategy also works for P, EXP, and $\mathsf{EXPSPACE}$.

To handle NP, we could use either the verifier definition of NP or the NTM definition. Let's use verifiers. If $B \in \mathsf{NP}$, then we have a verifier (V, k, ℓ, c) for B, and we want such a tuple for A. On input $\langle x, w\rangle$, the verifier V' for A will compute $f(x)$ (in polynomial time) and then run $V(\langle f(x), w\rangle)$, accepting if it accepts and rejecting otherwise. This takes polynomial time. Now, if $x \in A$, then $f(x) \in B$ and $|f(x)|$ is polynomial in $|x|$; hence there exists w causing $V(\langle f(x), w\rangle)$ to accept, and this w is polynomial in the size of $f(x)$, and hence polynomial in the size of x. This w causes $V'(\langle x, w\rangle)$ to accept. Conversely, if $x \notin A$, then $f(x) \notin B$, so no such w will exist. Thus V' is a valid polynomial-time verifier for A, so $A \in \mathsf{NP}$.

We omit the proof for NEXP, though it can be done similarly to NP (we would need exponential-time verifiers). For coNP, we note that if $B \in \mathsf{coNP}$ then $\overline{B} \in \mathsf{NP}$; since $A \leq_m^p B$, it is not hard to see that $\overline{A} \leq_m^p \overline{B}$, and so $\overline{A} \in \mathsf{NP}$, meaning that $A \in \mathsf{coNP}$. □

Several of the properties of mapping reductions also apply to polynomial-time mapping reductions:

- For all A, we have $A \leq_m^p A$.
- If $A \leq_m^p B$, then $\overline{A} \leq_m^p \overline{B}$.
- If $A \leq_m^p B$ and $B \leq_m^p C$, then $A \leq_m^p C$.

We omit the proofs of these, but they are fairly simple and similar to the case of (non-polynomial-time) mapping reductions.

23.2 Hardness and completeness

An important notion in complexity is the notion of *hardness* and *completeness* for a class of languages. Let us defin this now.

Definition 23.3 (Hardness and completeness)**.** *Let A be a language. We say that A is* hard *for a class of languages $\mathcal{C}$ if for all $B \in \mathcal{C}$, we have $B \leq_m^p A$.*

We say that A is complete *for a class of languages $\mathcal{C}$ if A is hard for $\mathcal{C}$ and also $A \in \mathcal{C}$.*

We generally use the notions of hardness and completeness together with some of the complexity classes we've seen. For example, a language might be NP-hard, which means that any language in NP can be reduced to it. If A is NP-hard, then if $A \in \mathsf{P}$ then $\mathsf{NP} \subseteq \mathsf{P}$; that's because if $A \in \mathsf{P}$, then $B \in \mathsf{P}$ for all $B \leq_m^p A$, but all $B \in \mathsf{NP}$ satisfy $B \leq_m^p A$.

Essentially, if a language is NP-hard this means that it is harder (or equal) to decide than all the languages in NP. Similarly, if a language is PSPACE-hard, it means it is harder (or equal) to

decide than all the languages in PSPACE. This is all up to polynomial-time reductions, so it is all ignoring polynomial factors.

If a language is NP-complete, it means it is both harder than everything in NP, but also itself in NP. In other words, it means it is the hardest language in NP (though it may be tied with others, and this is all only up to polynomial factors anyway).

The notion of P-hardness doesn't make as much sense; any language A other than the empty language and the language of all strings is technically P-hard, because for any other $B \in \mathsf{P}$, we can reduce B to A by taking the input x, deciding if $x \in B$, and setting $f(x)$ to be something in A if $x \in B$ and setting it to something not in A if $x \notin B$. This uses only polynomial time because B is decidable in polynomial-time. So all languages except 2 are P-hard, and all langauges in P except 2 are P-complete.

Note that if A is hard for some class such as NP or PSPACE, and if $A \leq_m^p B$, then B is also hard for the same class. This follows by the transitivity property of mapping reductions. Also, if A and B are two languages that are both complete for a class $\mathcal{C}$, then we must have $A \leq_m^p B$ and $B \leq_m^p A$; this is because by the definition of completeness, we know that A and B are both in $\mathcal{C}$ and also that they are both hard for $\mathcal{C}$, meaning that any language in $\mathcal{C}$ reduces to both of them.

23.3 NP-completeness

We will now prove one of the most striking facts in theoretical computer science: there exist NP-complete languages. In fact, once we find one NP-complete language A, we can prove that another language B is also NP-complete using reductions: we just need to show that $A \leq_m^p B$ (and also that $B \in \mathsf{NP}$). Historically, once a single NP-complete problem was found, it opened the floodgates: using reductions, it was quickly shown that many of the most important problems that people didn't know how to solve in polynomial time were actually NP-hard. In 1971, Cook showed the existence of a reasonably natural NP-complete problem; in 1972, Karp gave a list of 21 NP-complete problems, most of which were famous problems at the time; Karp did this using only reductions. (Similar research was done in parallel in the USSR by Levin; at the time, researchers in the West did not communicate with those in the USSR, and many results were proven twice because of this.)

Recall that to be NP-complete, a language A needs to have two properties:

1. We must have $A \in \mathsf{NP}$,
2. For any $B \in \mathsf{NP}$, we must have $B \leq_m^p A$.

The second condition says that A is at least as hard as any other language in NP; that is, A is (tied for) the hardest language in NP, up to polynomial-time mapping reductions.

23.3.1 A simple NP-complete problem

How could we possibly construct such a language A? Actually, there is a trick that lets us define a very simple NP-complete language. We can set

$$A = \{\langle \langle V \rangle, x, 0^k, 0^\ell \rangle : \exists w \text{ with } |w| \leq k \text{ s.t. } V(\langle x, w \rangle) \text{ accepts within } \ell \text{ steps}\}.$$

In this definition, V is a Turing machine, while x and w are strings and k and ℓ are positive integers.

Theorem 23.4. *The language A above is NP-complete.*

Proof. We first note that $A \in \mathsf{NP}$. To see this, we pick the verifier V' such that $V'(\langle y, w\rangle)$ checks if y has the form $\langle\langle V\rangle, x, 0^k, 0^\ell\rangle$ (rejecting if not), then checks if $|w| \leq k$ (rejecting if not), and then simulates $V(\langle x, w\rangle)$ for at most ℓ steps (rejecting if V fails to halt). If $y \in A$, then $y = \langle\langle V\rangle, x, 0^k, 0^\ell\rangle$ and we know there must exist w that makes $V(\langle w, x\rangle)$ accept within ℓ steps, and hence $V'(\langle y, w\rangle)$ accepts. On the other hand, if $y \notin A$, then there are two options. If y is badly formatted, V' will reject it no matter what witness is attached. Otherwise, if y is correctly formatted, $V'(\langle y, w\rangle)$ will only accept if $|w| \leq k$ and if $V(\langle x, w\rangle)$ accepts within ℓ steps, which can only happen if $y \in A$; since $y \notin A$, $V'(\langle y, w\rangle)$ will reject for all w. Finally, observe that the running time of V' is polynomial for the formatting check, and then $O(\ell)$ for simulating V; since the input contains 0^ℓ, the input size is at least ℓ, and hence V' runs in polynomial time in its input size. Moreover, the witness w which must exist for $y \in A$ has size at most k, and since 0^k is part of the input, this witness size is at most the input size. Hence we conclude that $A \in \mathsf{NP}$.

Next, consider any language $B \in \mathsf{NP}$. We claim that $B \leq_m^p A$. To show this, we must provide a polynomial-time computable function f such that $x \in B \Rightarrow f(x) \in A$ and $x \notin B \Rightarrow f(x) \notin A$. Let (V, k, ℓ, c) be the tuple of verifier and polynomial bounds for the NP language B. We set $f(x) = \langle\langle V\rangle, x, 0^{|x|^k+c}, 0^{|x|^{k\ell}+c}\rangle$. Note that f can be computed in polynomial time. To do so, we construct a Turing machine M for this task. M will have the description $\langle V\rangle$ of V hard-coded, so it will always print it out. On input x, M will print out $\langle V\rangle$ and then x. It will also calculate $|x|^k + c$, and write down that many zeroes. Finally, M will calculate $|x|^{k\ell} + c$ and write down that many zeroes (after a separator symbol). It is clear that this can all be done in polynomial time, since all the calculations are simple and only polynomially many zeroes need to be written.

Now, if $x \in B$, then there exists a witness w such that $|w| \leq |x|^k + c$ and $V(\langle x, w\rangle)$ accepts within $|x|^{k\ell} + c$ steps; it follows that $f(x) \in A$. Conversely, if $x \notin B$, then there does not exist a witness w such that $V(\langle x, w\rangle)$ accepts; but then we must have $f(x) \notin A$. We conclude that f is a polynomial time mapping reduction from B to A, so $B \leq_m^p A$. Since B was arbitrary, any language in NP can be reduced to A, and hence A is NP-hard. Since $A \in \mathsf{NP}$, A is by definition NP-complete. □

What happened here? We just defined A to be the language of all (verifier, input) pairs such that there is some witness causing the verifier to accept the input. If we think of NP as the class of all "puzzles" (where the solution to a puzzle can be checked quickly, but may be hard to find), then A is like a universal puzzle: if you can solve this puzzle, then you can clearly solve every other puzzle, since by definition puzzles have a corresponding verifier that allows you to check solutions. And what is the universal puzzle? It is essentially the problem, "given the description of a puzzle, solve that puzzle".

23.4 Satisfiability

Our next task will be to get a more natural NP-complete problem. Since we already have one NP-complete problem A, we can do this via reduction, by showing that a more natural NP language C has $A \leq_m^p C$. Since for each $B \in \mathsf{NP}$ we have $B \leq_m^p A$, if we show that $A \leq_m^p C$ we would also have $B \leq_m^p C$ for each $B \in \mathsf{NP}$, and so C would be NP-hard.

The language we will choose is called SAT. We now describe this language; we will need some terminology for this. We will deal with Boolean variables, that is, variables x_i which take values in $\{0, 1\}$. We apply Boolean operations to them, such as AND, OR, and NOT. We'll denote by $\overline{x}_i$ the negation of the variable x_i. A *literal* is either a variable x_i or the negation of a variable, $\overline{x}_i$. A *clause* is a disjunction of literals (disjunction is a fancy word for OR; that is, a clause is an OR of

n literals). We denote an OR using $\vee$, so a clause has the form $y_1 \vee y_2 \vee \cdots \vee y_k$, where each y_i is a literal, that is, $y_i = x_j$ or $y_i = \overline{x}_j$ for some variable x_j.

We say that a set of clauses over n Boolean variables is *satisfiable* if there is a $\{0,1\}$ assignment to those variables that satisfies every clause in the set (i.e. makes every clause evaluate to true). For example, $\{x_1 \vee x_2, x_3 \vee \overline{x}_2, \overline{x}_3\}$ is satisfiable, because we can pick the satisfying assignment $x_1 = 1, x_2 = 0, x_3 = 0$, and then $\overline{x}_3 = 1$, $x_3 \vee \overline{x}_2 = 0 \vee 1 = 1$, and $x_1 \vee x_2 = 1 \vee 0 = 1$. On the other hand, $\{x_1, \overline{x}_1\}$ is not satisfiable, because no assignment of a Boolean value to the variable x_1 can cause x_1 and $\overline{x}_1$ to evaluate to 1 at the same time.

The satisfiability problem SAT is the set of all strings $\langle S \rangle$, where S is a set of clauses which is satisfiable. In other words, to determine if a string is in SAT, we must determine if it encodes a set of clauses which is satisfiable. Since it's easy to discard poorly-formatted inputs, the task just becomes determining whether there is a satisfying assignment to the given set of clauses.

Note that if the set of clauses S has $|\langle S \rangle| = n$, then it may have a number of variables up to n, e.g. it may have $n/100$ variables. In this case, there will be $2^{n/100}$ different assignments of $\{0,1\}$ values to those variables; this is exponentially many, so trying all of them cannot be done in polynomial time. However, it should be clear that SAT $\in$ NP: a verifier V can require the witness w to contain the satisfying assignment, and then V can simply check that w is indeed satisfying by plugging w into the variables of the clauses of S and simplifying, to check whether all the clauses are indeed satisfied.

23.5 Satisfiability is NP-complete

Theorem 23.5 (Cook-Levin theorem)**.** SAT *is* NP*-complete.*

We will now sketch the proof of the Cook-Levin theorem, which says that SAT is NP-complete. Since we've seen that SAT $\in$ NP, we just need to show that SAT is NP-hard; we do this by showing that $A \leq_m^p$ SAT, where A is the language we've seen to be NP-complete. This requires defining a polynomial-time computable function f such that $y \in A \Rightarrow f(y) \in$ SAT and $y \notin A \Rightarrow f(y) \notin$ SAT.

We now sketch the definition of f. First, if an input y to A is poorly formatted, we can set $f(y)$ to be a fixed string not in SAT, so we can restrict our focus to strings y of the form $\langle \langle V \rangle, x, 0^k, 0^\ell \rangle$. We want to convert this string to a set of clauses such that those clauses can be satisfied if a witness w exists such that $|w| \leq k$ and $V(\langle x, w \rangle)$ accepts within ℓ steps, and such that these clauses cannot be satisfied if such w does not exist.

What we will do is effectively implement ℓ steps of V using clauses. Note that in ℓ steps, V can only access ℓ cells of its tape; hence we can assume that the tape has size only 2ℓ (with ℓ cells to the left and right of the starting position of the head of V). Thus the entire run of $V(\langle x, w \rangle)$ can be described by ℓ configurations, each of which list just 2ℓ positions of the tape. We will have one variable $T_{i,a,t}$ for each position $i = 1, 2, \ldots, 2\ell$ of the tape, each alphabet symbol $a \in \Gamma$ of the tape alphabet, and each step $t = 0, 1, 2, \ldots, \ell$ of the run of V; this is $2\ell|\Gamma|(\ell + 1)$. We will want the Boolean variable $T_{i,a,t}$ to equal 1 if tape position i contains symbol a at step number t of the run of $V(\langle x, w \rangle)$. Additionally, we will have one variable $Q_{p,t}$ for each state p of V and each step $t = 1, 2, \ldots, \ell$, which represents whether the state of V is p at step number t. Further, we will have one variable $H_{i,t}$ for each position i of the tape and each step number t, which represents whether the head of V is at position i at time step t.

So far, we've listed out polynomially many variables (in terms of the input size $|y|$). We now use clauses to place constraints on these variables. We restrict $H_{\ell,0} = 1$ and $H_{i,0} = 0$ for $i \neq \ell$ in order to force the position of the head to be ℓ at the beginning (at time 0). This can be done by adding $H_{\ell,0}$ and $\overline{H}_{i,0}$ as clauses. We similarly force the starting state to be q_0. For time step 0, we

also force the first $|x|$ tape positions to the right of the head to be x, and then a separator symbol; we then leave k tape variables unconstrained (allowing the witness w to be any string), but force everything afterwards to be blank tape symbols. This restricts the size of w to be at most k.

Next, we add a bunch of clauses that force all the variables to be what we want them to be. We add the clause $\overline{T}_{i,a,k} \vee \overline{T}_{i,b,k}$ for all i, k and all $a \neq b$ in order to prevent a tape position from having more than one symbol; we also add the clause $\bigvee_{a \in \Gamma} T_{i,a,k}$ to force each tape position to have at least one symbol. We do something similar with the H and Q variables to force each time period to have exactly one state and exactly one head position.

Next, we add clauses corresponding to the allowed transitions of V, to make sure that the variables only change from one time period to the next in accordance with the transition function δ. We won't go through the details of how to do this, but it's not too complicated; we just need to say that for every time step, every possible head position, every internal state, and every symbol at that head position, if the head is really in that position reading that symbol at that time step with that internal state (i.e. if the corresponding variables are 1), then in the next time step we must have moved according to δ (i.e. the corresponding variables in the subsequent time step are 1). This if-then type of condition can be translated to an AND of ORs, which means we can implement it by adding a bunch of clauses to our set.

If the accept or reject state has been reached, we force all variables to be the same in the subsequent time periods, to represent the machine halting. Finally, we add one clause which is just the single variable $Q_{q_{\text{acc}},\ell}$, which forces the state at time period ℓ to be the accept state q_{acc}.

This set of clauses can be satisfied if and only if there is a witness w that causes V to reach an accept state. Hence the string $\langle S \rangle$ encoding this set of clauses is in SAT if and only if the original string y was in A. Finally, we claim that this reduction can be implemented in polynomial time; the number of clauses we introduced was polynomial in $|x|$, k, and ℓ (all of which are smaller than the input size $|y|$), and each clause only contained polynomially many variables, as there were only polynomially many variables in total. We won't go through the details of these calculations.

The upshot of this is that the reasonably simple problem SAT, which asks whether a set of clauses is satisfiable, is powerful enough to encode the question of whether a witness exists which causes a Turing machine to accept in ℓ steps. That is, Boolean formulas are powerful enough to encode time-bounded Turing machines. For this reason, SAT is NP-complete.

23.6 3-SAT

We conclude by showing an even simpler NP-complete language, called 3-SAT. The language 3-SAT is the set of all satisfiable sets of clauses for which all the clauses have at most 3 literals. This is a subset of SAT, and seems simpler. It should be clear that 3-SAT is in NP. To show that 3-SAT is NP-complete, all we need to do is show that SAT $\leq_m^p$ 3-SAT. For the latter, we just need to give a polynomial-time computable function f such that $x \in \text{SAT} \Rightarrow f(x) \in \text{3-SAT}$ and $x \notin \text{SAT} \Rightarrow f(x) \notin \text{3-SAT}$.

As usual, it is easy to deal with poorly-formatted inputs, so we restrict our attention to inputs x which look like a set of clauses. What we want to do is to convert each clause with many literals, like $y_1 \vee y_2 \vee \cdots \vee y_k$, into a set of clauses with 3 literals each which is equivalent to the original clause. To do this, we will just show how to make the clause $y_1 \vee y_2 \vee \cdots \vee y_k$ shorter by 1. We do that by replacing two of the variables with a single new variable z: that is, we write $y_1 \vee y_2 \vee \cdots \vee y_{k-2} \vee z$. We also add additional an additional clauses which force z to equal $y_{k-1} \vee y_k$: the clauses $\overline{z} \vee y_{k-1} \vee y_k$, $z \vee \overline{y}_{k-1}$, and $z \vee \overline{y}_k$. It's not hard to see that in order for these three clauses to be satisfied, we must have $z = y_{k-1} \vee y_k$; hence these three together with $y_1 \vee y_2 \vee \cdots \vee y_{k-2} \vee z$ are equivalent to

$y_1 \vee y_2 \vee \cdots \vee y_k$. Thus we created a set of clauses with one clause of size $k-1$ and three of size at most 3; by repeating this operation, we can replace a clause of size k with $O(k)$ clauses of size at most 3 each.

Doing this to every clause in the original set converts a SAT instance into an equivalent 3-SAT instance, where the former is satisfiable if and only if the latter is satisfiable. Moreover, this conversion can be done in polynomial time. Hence 3-SAT is an NP-complete problem.

23.7 Further remarks

3-SAT is the starting point for many additional reductions; many natural problems can be shown to be NP-complete by showing that they can encode arbitrary 3-SAT instances (and also that they are in NP). This is what makes the P vs. NP problem so important: NP-complete problems turn out to be very common, and we would really like to know whether there is a polynomial-time algorithm for solving them.

Generally, if you want to argue that a task is computationally difficult, the best way of doing so is to show that the task is NP-hard. This can be done by showing that an NP-hard language reduces to your task. Once you've shown your task to be NP-hard, it means that no polynomial-time algorithm exists for solving your task unless $\mathsf{P} = \mathsf{NP}$. Most people believe that $\mathsf{P} \neq \mathsf{NP}$, but even if they are wrong about that, at least your boss can't expect you to find a polynomial-time algorithm if the language is NP-hard, since that would mean you've shown $\mathsf{P} = \mathsf{NP}$ and solved one of the most famous open problems in mathematics.

Do there exist PSPACE-complete languages? The answer is yes: it turns out that many game-like problems are PSPACE-complete. For example, if we modified the game Go so that stones cannot be removed from the board, then the problem of "given a Go position on an $n \times n$ board, determine who will eventually win from that position" turns out to be PSPACE-complete. This is true not just for the game Go, but for many other games in which you have some polynomial bound on the number of moves (in this modification of Go, the bound comes from the fact that stones do not get removed and there are only n^2 places to put them on a $n \times n$ board).

There are also EXP-complete languages, and NEXP-complete languages, and even EXPSPACE-complete languages. All of these tasks are even harder than NP-complete problems. These come up more rarely, however.

What about coNP-complete problems? Well, as you might expect, if A is an NP-complete language, then $\overline{A}$ is complete for coNP, and vice versa. This is not hard to show using the fact that reductions are preserved when we take the complement of the languages.