Notes on Complexity Theory Last updated: October, 2011

Lecture 3

Jonathan Katz

1 Natural $\mathcal{NP}$-Complete Problems

Last time we saw a "non-natural" $\mathcal{NP}$-complete language. Other important $\mathcal{NP}$-complete languages are SAT (satisfiable boolean formulae in conjunctive normal form) and 3-SAT (satisfiable boolean formulae in conjunctive normal form, where each clause contains at most 3 literals). Besides being more "natural" languages, they are useful for proving $\mathcal{NP}$-completeness of other languages.

Theorem 1 (Cook-Levin Theorem) SAT *is $\mathcal{NP}$-complete.*

Proof We give a detailed proof sketch. (Note that the proof we give here is different from the one in [1]; in particular, we do not rely on the existence of oblivious Turing machines.)

Let L be a language in $\mathcal{NP}$. This means there is a Turing machine M and a polynomial p such that (1) $M(x, w)$ runs in time $p(|x|)$, and (2) $x \in L$ if and only if there exists a w for which $M(x, w) = 1$. Note that we may assume that any such w, if it exists, has length exactly $p(|x|) - |x| - 1$. We also assume for simplicity (and without loss of generality) that M has a single tape (that is used as both its input tape and work tape) and a binary alphabet.

A simple observation is that we can represent the computation of $M(x, w)$ (where $|x| = n$) by a *tableau* of $p(n) + 1$ rows, each $O(p(n))$ bits long. Each row corresponds to the entire configuration of M at some step during its computation; there are $p(n) + 1$ rows since M always halts after at most $p(n)$ steps. (If $M(x, w)$ halts before $p(n)$ steps, the last rows may be duplicates of each other. Or we may assume that $M(x, w)$ always runs for exactly $p(|x|)$ steps.) Each row can be represented using $O(p(n))$ bits since a configuration contains (1) the contents of M's tape (which can be stored in $O(p(n))$ bits — recall that $\text{SPACE}(p(n)) \subseteq \text{TIME}(p(n))$); (2) the location of M's head on its tape (which can be stored in $p(n)$ bits[1]); and (3) the value of M's state (which requires $O(1)$ bits).

Moreover, given a tableau that is claimed to correspond to an accepting computation of $M(x, w)$, it is possible to *verify* this via a series of "local" checks. (This notion will become more clear below.) Specifically, letting $p = p(n)$ and assuming we are given some tableau, do:

1. Check that the first row is formed correctly. (The tape should contain x, followed by a space and then a sequence of bits representing w; M's head should be in the left-most position; and M should be in its start state.)

2. Number the rows from 0 to T, and recall that these correspond to time steps of M's execution. Let $t_{i,j}$ denote the value written in cell j at time i. Then for $i = 1, \ldots, T$ and $j = 1, \ldots, T$, check that $t_{i,j}$ has the correct value given $t_{i-1,j-1}, t_{i-1,j}$, and $t_{i-1,j+1}$ and the value of the state at time $i - 1$. We also need to check that the state at time i takes the correct value; this is discussed in detail below.

3. Check that the state in the final row is the accepting state.

[1] In fact, $O(\log p(n))$ bits suffice, but for this proof it is somewhat simpler to use a more wasteful representation.

Each of these checks involves looking at only a small (in fact, constant) part of the tableau. This is important, as it implies that each check can be represented as a constant-size CNF formula. Then correctness of the entire tableau can be represented as a conjunction of a polynomial number of these formulae. We give further details below.

We begin with the following claim:

Claim 2 *Any function $f : \{0,1\}^\ell \to \{0,1\}$ can be expressed as a CNF formula of size at most $\ell \cdot 2^\ell$.*

Proof Let $x = (x_1, \ldots, x_\ell)$ denote the input variables of f. For some fixed string $y \in \{0,1\}^\ell$, we can express the predicate "$\mathsf{neq}_y(x) \stackrel{\text{def}}{=} [x \neq y]$" as

$$(x_1 \neq y_1) \vee \cdots \vee (x_\ell \neq y_\ell);$$

remembering that y is fixed, this is just a disjunctive clause in the ℓ variables $x_1, \ldots, x_\ell$. If we let $Y \subseteq \{0,1\}^\ell$ denote the set of inputs on which f evaluates to 0, then we can express the predicate "$f(x) = 1$" by

$$\textstyle\bigwedge_{y \in Y} \mathsf{neq}_y(x_1, \ldots, x_\ell),$$

which is a CNF formula of size at most $\ell \cdot 2^\ell$. ■

To prove the theorem, we need to show a polynomial-time transformation f that outputs CNF formula with the property that $x \in L$ iff $f(x) \in \mathsf{SAT}$. Our transformation f will output a CNF formula corresponding to the verification of an accepting tableau of the computation of $M(x, w)$ for some w. For a given x of length $n = |x|$, let $p = p(n)$; then $f(x)$ does as follows:

- Create variables $\{t_{i,j}\}$ for $i = 0$ to p and $j = 1$ to p. Each $t_{i,j}$ represents the value written in cell j at time i. (Each $t_{i,j}$ will actually be two bits, since we need two bits to represent the 0, 1, start symbol, and space character.)
- Create variables $u_{i,j}$ for $i = 0$ to p and $j = 1$ to p. Each $u_{i,j}$ is a single bit indicating whether the head is in position j at time i.
- Create variables $\vec{s}_i \stackrel{\text{def}}{=} (s_{i,1}, \ldots, s_{i,q})$ for $i = 1$ to p and some constant q that depends on the number of states that M uses. (Namely, if the set of states is Q then $q = \lceil \log |Q| \rceil$.)
- Create the following CNF formulae:
  - χ_0 checks that row 0 is correct: namely, that $t_{0,1}, \ldots, t_{0,p}$ contains a start symbol, followed by $x_1, \ldots, x_\ell$, followed by a blank, and then $\{0,1\}$ in the remaining positions; furthermore, $u_{0,1} = 1$ and $u_{0,j} = 0$ for all $j > 1$, and $\vec{s}_0$ encodes the start state of M. Even though χ_0 involves $O(p)$ variables, it is easy to see that it can be expressed as a CNF formula of size $O(p)$.
  - For $i, j = 1$ to p, let $\phi_{i,j}$ be a CNF formula that checks correctness of cell j at time i. This is a formula in the variables $t_{i,j}$, $u_{i,j}$, the three[2] cells in the neighborhood of cell j at the previous time period (namely, $N_{i-1,j} \stackrel{\text{def}}{=} \{t_{i-1,j-1}, u_{i-1,j-1}, t_{i-1,j}, u_{i-1,j}, t_{i-1,j+1}, u_{i-1,j+1}\}$), and the current and previous states $\vec{s}_i, \vec{s}_{i-1}$. This formula encodes the following predicate:

[2] Of course, if $j = 1$ or $j = p$ then the cell has only two neighbors.

$t_{i,j}, u_{i,j}$ contain the correct values given $N_{i-1,j}$ and $\vec{s}_{i-1}$.

and

if $u_{i,j} = 1$, then $\vec{s}_i$ contains the correct value given $N_{i-1,j}$ and $\vec{s}_{i-1}$.

The above can be a complicated predicate, but it involves only a *constant* (i.e., independent of n) number of variables, and hence (by Claim 2) can be encoded by a CNF formula of constant size.

- χ_p simply checks that $\vec{s}_p$ encodes the accepting state of M.
- The output of f is $\Phi = \chi_0 \wedge \left(\bigwedge_{i,j} \phi_{i,j}\right) \wedge \chi_p$.

One can, somewhat tediously, convince oneself that Φ is satisfiable if and only if there is some w for which $M(x, w) = 1$. ■

To show that 3-SAT is $\mathcal{NP}$-complete, we show a reduction from any CNF formula to a CNF formula with (at most) 3 literals per clause. We illustrate the idea by showing how to transform a clause involving 4 literals to two clauses involving 3 literals each: given clause $a \vee b \vee c \vee d$ we introduce the auxiliary variable z and then output $(a \vee b \vee z) \wedge (\bar{z} \vee c \vee d)$; one can check that the latter is satisfiable iff the former is satisfiable.

1.1 Other $\mathcal{NP}$-Complete Problems

SAT and 3-SAT are useful since they can be used to prove many other problems $\mathcal{NP}$-complete. Recall that we can show that some language L is $\mathcal{NP}$-complete by demonstrating a Karp reduction from 3-SAT to L. As an example, consider IndSet (see [1] for more details): Given a formula ϕ with n variables and m clauses, we define a graph G with $7m$ vertices. There will be 7 vertices for each clause, corresponding to 7 possible satisfying assignments. G contains edges between all vertices that are inconsistent (including those in the same cluster). One can check that there is an independent set of size m iff ϕ has a satisfying assignment.

2 Self-Reducibility and Search vs. Decision

We have so far been talking mainly about decision problems, which can be viewed as asking whether a solution *exists*. But one often wants to solve the corresponding search problem, namely to *find* a solution (if one exists). For many problems, the two have equivalent complexity.

Let us define things more formally. Say $L \in \mathcal{NP}$. Then there is some polynomial-time Turing machine M such that $x \in L$ iff $\exists w : M(x, w) = 1$. The *decision problem* for L is: given x, determine if $x \in L$. The *search problem for* L is: given $x \in L$, find w such that $M(x, w) = 1$. (Note that we should technically speak of the *search problem for L relative to M* since there can be multiple non-deterministic Turing machines deciding L, and each such machine will define its own set of "solutions". Nevertheless, we stick with the inaccurate terminology and hope things will be clear from the context.) The notion of reducibility we want in this setting is *Cook-Turing reducibility*. We define it for decision problems, but can apply it to search problems via the natural generalization.

Definition 1 *Language L is* Cook-Turing reducible to *L' if there is a poly-time Turing machine M such that for any oracle $\mathcal{O}'$ deciding L', machine $M^{\mathcal{O}'(\cdot)}$ decides L. (I.e., $M^{\mathcal{O}'(\cdot)}(x) = 1$ iff $x \in L$.)*

Note that if L is *Karp*-reducible to L', then there is also a Cook-Turing reduction from L to L'. In general, however, the converse is not belied to hold. Specifically, any language in $\mathsf{co}\mathcal{NP}$ is Cook-Turing reducible to any $\mathcal{NP}$-complete language, but there is no Karp-reduction from a $\mathsf{co}\mathcal{NP}$-complete language to a language in $\mathcal{NP}$ unless $\mathsf{co}\mathcal{NP} = \mathcal{NP}$.

Returning to the question of search vs. decision, we have:

Definition 2 *A language $L \in \mathcal{NP}$ is* self-reducible *if there is a Cook-Turing reduction from the search problem for L to the decision problem for L. Namely, there is polynomial-time Turing machine M such that for any oracle $\mathcal{O}_L$ deciding L, and any $x \in L$ we have $(x, M^{\mathcal{O}_L(\cdot)}(x)) \in R_L$.*

(One could also ask about reducing the decision problem to the search problem. For languages in $\mathcal{NP}$, however, such a reduction always exists.)

Theorem 3 SAT *is self-reducible.*

Proof Assume we have an oracle that tells us whether any CNF formula is satisfiable. We show how to use such an oracle to find a satisfying assignment for a given (satisfiable) CNF formula ϕ. Say ϕ is a formula on n variables $x_1, \ldots, x_n$. If $b_1, \ldots, b_\ell \in \{0,1\}$ (with $\ell \leq n$), then by $\phi|_{b_1,\ldots,b_\ell}$ we mean the CNF formula on the variables $x_{\ell+1}, \ldots, x_n$ obtained by setting $x_1 = b_1$, $\ldots$, $x_\ell = b_\ell$ in ϕ. ($\phi|_{b_1,\ldots,b_\ell}$ is easily computed given ϕ and $b_1, \ldots, b_\ell$.) The algorithm proceeds as follows:

- For $i = 1$ to n do:
  - Set $b_i = 0$.
  - If $\phi_{b_1,\ldots,b_i}$ is not satisfiable, set $b_i = 1$. (Note: we determine this using our oracle for SAT.)
- Output $b_1, \ldots, b_n$.

We leave it to the reader to show that this always returns a satisfying assignment (assuming ϕ is satisfiable to begin with). ■

The above proof can be generalized to show that *every* $\mathcal{NP}$-complete language is self-reducible.

Theorem 4 *Every $\mathcal{NP}$-complete language L is self-reducible.*

Proof The idea is similar to above, with one new twist. Let M be a polynomial-time non-deterministic Turing machine such that

$$L = \{x \mid \exists w : M(x,w) = 1\}.$$

We first define a new language L':

$$L' = \{(x,b) \mid \exists w' : M(x, bw') = 1\}.$$

I.e., $(x,b) \in L'$ iff there exists a w *with prefix* b such that $M(x,w) = 1$. Note that $L' \in \mathcal{NP}$; thus, there is a Karp reduction f such that $x \in L'$ iff $f(x) \in L$. (Here is where we use the fact that L is $\mathcal{NP}$-complete.)

Assume we have an oracle deciding L; we design an algorithm that, given $x \in L$, finds w with $M(x,w) = 1$. Say the length of w (given x) is $n = \mathsf{poly}(|x|)$. The algorithm proceeds as follows:

- For $i = 1$ to n do:

- Set $b_i = 0$.
- If $f((x, b_1, \ldots, b_i)) \notin L$, set $b_i = 1$. (We run this step using our oracle for L.)

- Output $b_1, \ldots, b_n$.

We leave it to the reader to show that this algorithm gives the desired result. ■

Other languages in $\mathcal{NP}$ (that are not $\mathcal{NP}$-complete) may be self-reducible as well. An example is given by graph isomorphism, a language that is not known (or believed) to be in $\mathcal{P}$ or $\mathcal{NP}$-complete. On the other hand, it is believed that not all languages in $\mathcal{NP}$ are self-reducible. One conjectured example is the natural relation derived from factoring: although compositeness can be decided in polynomial time, we do not believe that polynomial-time factoring algorithms exist.

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.