Notes on Complexity Theory: Fall 2005 Last updated: September, 2005

Lecture 2

Jonathan Katz

1 More $\mathcal{NP}$-Compete Languages

It will be nice to find more "natural" $\mathcal{NP}$-complete languages. To that end, we define the language *circuit-satisfiability* (CS) as follows:

$$CS = \{C \mid C \text{ is a circuit, and } \exists \sigma \text{ s.t. } C(\sigma) = 1\}.$$

Theorem 1 *CS is $\mathcal{NP}$-complete.*

Proof It is relatively easy to see that $CS \in \mathcal{NP}$. We show that CS is $\mathcal{NP}$-complete by giving a Karp reduction from any $L \in \mathcal{NP}$ to CS. Fix such an L, and let M_L be a non-deterministic machine deciding L and running in time n^c on inputs of size n. The idea is straightforward: let $M_L(x, w)$ denote an execution of M_L with input x and making non-deterministic choices w. Note that $x \in L$ iff there exists a w such that $M_L(x, w) = 1$. (We further assume that w is padded, as necessary, so that there exists such a w with length exactly n^c in this case.) Consider the deterministic, polynomial-time function $M_{L,x} : \{0,1\}^{n^c} \rightarrow \{0,1\}$ defined as $M_{L,x}(w) \stackrel{\text{def}}{=} M_L(x, w)$. Our Karp reduction will take as input x and output a circuit $C_{M_L,x}$ that computes the function $M_{L,x}$. Clearly, $x \in L$ iff $C_{M,L,x} \in CS$. So if we can construct such a circuit in polynomial time we are done. We will show that a circuit with the desired properties exists which is of size polynomial in the input x. We will then argue that in fact the circuit can be constructed in polynomial time, given x (and with implicit knowledge of M_L).

Let $t = |x|^c$. We will build a circuit having $O(t)$ levels and $O(t)$ wires/gates at each level. The single wire at level $t+1$ will be the output wire. The wires at level 0 represent the initial configuration of $M_{L,x}$, and the wires at level $i \in \{1, \ldots t\}$ represent the configuration of $M_{L,x}$ after the i^{th} step of its execution. A *configuration* of $M_{L,x}$ consists of t "blocks", with the j^{th} block storing the following information: (1) the contents of the j^{th} cell of $M_{L,x}$'s work tape, (2) a bit indicating the presence/absence of the head on the j^{th} cell of $M_{L,x}$'s work tape; (3) the value of $M_{L,x}$'s state, if the previous bit is on (we do not care how these wires are set if the previous bit is off). Note that only a constant number of wires, per block, are needed; thus, there are $O(t)$ wires per level as claimed. At level i, label the first set of wires (above) as $\vec{s}_i = s_{i,1}, \ldots, s_{i,t}$, the second set of wires $\vec{h}_i = h_{i,1}, \ldots, h_{i,t}$, and the last set of wires $\vec{q}_i = q_{i,1}, \ldots, q_{i,t}$ (note that each $s_{i,j}$ and $q_{i,j}$ is actually some constant number of wires). Call $(s_{i,j}, h_{i,j}, q_{i,j})$ the *configuration at* (i, j).

The input to the circuit is a t-bit string $w = w_1, \ldots, w_t$. Hard-wire at level 0 the values $\vec{s}_0 = x$ (padded out appropriately with spaces), $h_{0,1} = 1$, $h_{0,j} = 0$ (for $j > 1$), and $q_{0,1} = q_{\text{initial}}$. The key point is that the configuration at (i, j) is determined entirely by w_i (i.e., the i^{th} input wire) and the configuration at $(i-1, j-1)$, $(i-1, j)$, and $(i-1, j+1)$.

Note that such a statement will not be true for a general poly-time function, but it *is* true for the function $M_{L,x}$ because of the way it is defined (namely, w_i represents the choice made by $M_{L,x}$ at its i^{th} step). Thus, the value of each wire at level i is determined by the values of a *constant* number of wires (a constant number from level $i-1$ plus a single input wire), and it is not hard to see that any function on a constant number of bits can be computed by a constant-size circuit.

At the bottom level (level $t+1$), we compute the output as follows: the output of the circuit is 1 iff there exists an index j such that $h_{t,j} = 1$ and $q_{t,j} = q_{\text{accept}}$. Computing this requires $t-1$ ORs of t constant-size circuits (namely, one for each $j \in [t]$).

Since there are $O(t)$ wires/gates at each level, and $O(t)$ levels, the circuit has size $O(t^2)$ which is polynomial in $|x|$. It is not too difficult to see that the circuit can also be constructed (uniformly) in time linear in its size (in particular, the $O(t)$ constant-size circuits at each level are the same at each level, and are based on the transition function of $M_{L,x}$). ■

Using the above result, we show that *satisfiability* is $\mathcal{NP}$-complete.

Theorem 2 *SAT is $\mathcal{NP}$-complete.*

Proof It is easy to see that $SAT \in \mathcal{NP}$. We show a Karp reduction from CS to SAT. Since Karp reductions are transitive, the theorem follows.

Given a circuit C, we associate a variable with each wire. Let the variables be numbered $v_1, \ldots, v_n$. For each gate g of the circuit, we define a clause ϕ_g as follows:

– If g is a NOT gate whose input wire is associated with the variable v_i and whose output wire is associated with v_j, then let $\phi_g \stackrel{\text{def}}{=} (v_i \wedge \bar{v}_j) \vee (\bar{v}_i \wedge v_j)$.

– If g is an OR gate whose input wires are associated with the variables v_i, v_j and whose output wire is associated with v_k, then let

$$\phi_g \stackrel{\text{def}}{=} (v_k \wedge (v_i \vee v_j)) \bigvee (\bar{v}_k \wedge \bar{v}_i \wedge \bar{v}_j).$$

– If g is an AND gate whose input wires are associated with the variables v_i, v_j and whose output wire is associated with v_k, then let

$$\phi_g \stackrel{\text{def}}{=} (v_k \wedge v_i \wedge v_j) \bigvee (\bar{v}_k \wedge (\bar{v}_i \vee \bar{v}_j)).$$

– If the output wire is associated with variable v_i, define $\phi_{\text{out}} \stackrel{\text{def}}{=} v_i$.

Finally, define $\Phi = \phi_{\text{out}} \wedge \bigwedge_{g \in C} \phi_i$. Say the input wires of C are associated with $v_1, \ldots, v_\ell$. It can be verified that C evaluates to 1 on an input $x \in \{0,1\}^\ell$ iff there is a satisfying assignment of Φ in which $v_i = x_i$ for $1 \leq i \leq \ell$. The theorem follows. ■

2 Relating Search Problems to Decision Problems

For this section, let R denote a relation which is *polynomially bounded* (namely, there exists a polynomial p such that if $(x,w) \in R$ then $|w| \leq p(|x|)$) as well as *polynomially-verifiable* (i.e., $R \in \mathcal{P}$). Note that any such R defines the $\mathcal{NP}$-language L_R given by:

$$L_R \stackrel{\text{def}}{=} \{x \mid \exists w : (x,w) \in R\}$$

(this is the "dual" of the conversion from an $\mathcal{NP}$-language L to a relation R_L described earlier, although we remark that there are multiple relations R which define the same language L_R). The *decisional problem* over R is to determine, given x, whether $x \in L_R$; i.e., to determine whether there exists a w such that $(x, w) \in R$. The *search problem* over R is to find a w for which $(x, w) \in R$, assuming such a w exists.

Search problems are "harder" than decisional problems in the sense that if the search problem can be solved efficiently then so can the decisional one. What can we say about the other direction? We begin with a (somewhat informal) definition:

Definition 1 Let $\mathcal{O}_P$ denote an oracle that solves some problem P. We say a problem P_1 is *Cook-reducible* to a problem P_2 if there exists a polynomial-time oracle machine M such that $M^{\mathcal{O}_{P_2}}$ solves P_1. $\diamondsuit$

(The above notion is sometimes also referred to as *Turing reducibility*.) It is not hard to see that, using the above terminology, the decisional problem over a relation R is always Cook-reducible to the search problem over R (assuming R satisfies the criteria given above).

Definition 2 We say a relation R is *self-reducible* if the search problem over R is Cook-reducible to the decisional problem over R. $\diamondsuit$

We remark that self-reducibility is technically defined for *relations*, not *languages* (since a given language may give rise to multiple relations and thus multiple associated search problems), but we will be informal and always mean the "natural" relation associated with a language. We showed in class that SAT is self-reducible. (In fact, one can similarly show that all $\mathcal{NP}$-complete languages are self-reducible.) More difficult to see is that graph isomorphism — which is not believed to be $\mathcal{NP}$-complete — is self-reducible via the following algorithm which, given two isomorphic n-vertex graphs G_1, G_2, determines an isomorphism ϕ between them vertex-by-vertex (here, if $(i, j) \in \phi$ we take it to mean that $\phi(i) = j$):

Let v_i (resp., u_i) denote the i^{th} vertex of G_1 (resp., G_2)
Initialize $G_1' = G_1$, $G_2' = G_2$, $\phi = \emptyset$, and $J = [n]$
For $i = 1$ to n do:
    For $j \in J$ do:
        Let G_1'' be the graph derived from G_1' by rooting a tree with $i \cdot n$ vertices at v_i
        Define G_2'' analogously using u_j
        If there is an isomorphism between G_1'' and G_2'' do:
            add (i, j) to ϕ
            set $G_1' = G_1''$ and $G_2' = G_2''$
            set $J = J \setminus \{j\}$
            loop to the next value of i
Output ϕ

It is believed that not all languages in $\mathcal{NP}$ are self-reducible. One conjectured example is the natural relation derived from factoring: although composite numbers can (now) be recognized in polynomial time, factoring composite numbers in polynomial time is not believed possible.

3 Ladner's Theorem

We know that there exist $\mathcal{NP}$-complete languages. Furthermore, it is not hard to see that as long as $\mathcal{P} \neq \mathcal{NP}$, any $\mathcal{NP}$-complete language lies in $\mathcal{NP} \setminus \mathcal{P}$. Are there non-$\mathcal{NP}$-complete languages in $\mathcal{NP} \setminus \mathcal{P}$? Ladner's theorem tells us that there are.

Theorem 3 *Assuming $P \neq \mathcal{NP}$, there exists a language $A \in \mathcal{NP} \setminus \mathcal{P}$ which is not $\mathcal{NP}$-complete.*

Proof The high-level intuition behind the proof is that we construct A by taking an $\mathcal{NP}$-complete language and "blowing holes" in it in such a way that the language is no longer $\mathcal{NP}$-complete yet not in $\mathcal{P}$ either. The specific details are quite involved.

Let $M_1, \ldots$ denote an enumeration of all polynomial-time Turing machines; formally, this can be achieved by considering an enumeration[1] of $\mathcal{M} \times \mathbb{Z}$ (where $\mathcal{M}$ is the set of all Turing machines), and defining M_i as follows: if the i^{th} item in this enumeration is (M, j), then $M_i(x)$ runs $M(x)$ for at most $|x|^j$ steps. We remark that $M_1, \ldots$ also gives an enumeration of languages in $\mathcal{P}$ (with languages appearing multiple times). In a similar way, let $F_1, \ldots$ denote an enumeration of functions computable in polynomial time (such functions are defined by the poly-time Turing machine which computes them).

Define language A as follows:

$$A = \{x \mid x \in SAT \wedge f(|x|) \text{ is even}\},$$

for some function f that remains to be defined. Note that as long as we ensure that f is computable in polynomial time, then $A \in \mathcal{NP}$. We define f by a polynomial-time Turing machine M_f such that $M_f(1^n) = f(n)$. Let M_{SAT} be a machine that decides SAT (not necessarily in polynomial time, of course...), and let $f(0) = f(1) = 2$. On input 1^n (with $n > 1$), M_f proceeds in two stages, each lasting for n steps:

1. During the first stage, M_f computes $f(0), f(1), \ldots$ until it runs out of time.[2] Suppose the last value of f it was able to compute was $f(x) = k$. The output of M_f will be either k or $k+1$, to be determined by the next stage.

2a. If $k = 2i$ is even, then M_f tries to find a $z \in \{0,1\}^*$ such that $M_i(z)$ outputs the "wrong" answer as to whether $z \in A$. (That is, M_f tries to find a z such that either $z \in A$ but $M_i(z) = 0$, or such that $z \notin A$ but $M_i(z) = 1$.) This is done by computing $M_i(z), M_{\text{SAT}}(z)$, and $f(|z|)$ for all strings z in lexicographic order.

If such a string is found within the allotted time, then the output of M_f is $k+1$. Otherwise, the output of M_f is k.

2b. If $k = 2i-1$ is odd, then M_f tries to find a string z such that $F_i(z)$ is an incorrect Karp reduction from SAT to A. (That is, M_f tries to find a z such that either $z \in SAT$ but

[1] Since both $\mathcal{M}$ and $\mathbb{Z}$ are countable, it follows that $\mathcal{M} \times \mathbb{Z}$ is countable.

[2] Here and in the next stage, note that M_f never needs to compute $f(n')$ for $n' \geq n$ (which would cause self-reference problems). This is so since $M_f(1^n)$ runs out of time (in either stage) if it writes down a string of length n.

$F_i(z) \notin A$, or $z \notin SAT$ but $F_i(z) \in A$.) This is done by computing $F_i(z)$, $M_{\mathrm{SAT}}(z)$, $M_{\mathrm{SAT}}(F_i(z))$, and $f(|F_i(z)|)$.

If such a string is found within the allotted time, then the output of M_f is $k+1$; otherwise, the output is k.

It is clear from its definition that M_f runs in polynomial time. Note also that $f(n+1) \geq f(n)$ for all n.

We claim that $A \notin \mathcal{P}$. Suppose the contrary. Then A is decided by some M_i. In this case, however, the second stage of M_f with $k = 2i$ will never find a z satisfying the desired property, and so f is eventually a constant function and in particular $f(n)$ is odd for only finitely-many n. But this implies that A and SAT coincide except for finitely-many strings. This implies that $SAT \in \mathcal{P}$, a contradiction to our assumption that $P \neq \mathcal{NP}$.

Similarly, we claim that A is not $\mathcal{NP}$-complete. For, if so, then there is a polynomial-time function F_i which gives a Karp reduction from SAT to A. Now $f(n)$ will be even for only finitely-many n, implying that A is a finite language. But then $A \in \mathcal{P}$, a contradiction to our assumption that $P \neq \mathcal{NP}$. ■

As an addendum to the theorem, we note that there are no "natural" languages known to be in $\mathcal{NP} \setminus \mathcal{P}$ but not $\mathcal{NP}$-complete (assuming $\mathcal{P} \neq \mathcal{NP}$, of course). However, there are a number of languages conjectured to fall in this category, including graph isomorphism and essentially all languages derived from cryptographic assumptions (e.g., factoring, one-way functions, etc.).

Bibliographic Notes

Sections 1 and 2 are based on [2, Lecture 2]. Ladner's theorem was proven in [3], and the proof given here is based on Papadimitriou [4, Chap. 14] and a note by Fortnow [1].

References

[1] L. Fortnow. Two Proofs of Ladner's Theorem. Available from http://weblog.fortnow.com/media/ladner.pdf.

[2] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).

[3] R.E. Ladner. On the Structure of Polynomial-Time Reducibility. *J. ACM* 22(1): 155–171, 1975.

[4] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1995.