Notes on Complexity Theory Last updated: September, 2015

Lecture 3

Dov Gordon, taken from [1], Chapters 6-7

1 One more mapping reduction

We continue where we left off, giving one more example of a mapping reduction.

Theorem 1 *The language* $L_{EQ} = \{(\langle M_1 \rangle, \langle M_2 \rangle) \mid L(M_1) = L(M_2)\}$ *is undecidable.*

Proof This time we prove it by showing a reduction from $L_\emptyset$ to L_{EQ}. Since we've already shown that $L_\emptyset$ is undecidable, it follows that L_{EQ} is also undecidable. Given a TM M_{EQ} that decides L_{EQ}, we built a TM $M_\emptyset$ for $L_\emptyset$, arriving at the desired contradiction.
$\underline{M_\emptyset(\langle M \rangle):}$

1. Construct the description of a Turing machine M' that rejects all strings.
2. Run $M_{EQ}(\langle M \rangle, \langle M' \rangle)$, and output whatever it outputs.

■

If we want to formalize this as a mapping reduction, showing that $L \leq_m L_{EQ}$, we have to give a computable function f that maps $x \in L$ to $x' \in L_{EQ}$. The function is straightforward: letting $\langle M_2 \rangle$ be the description of a turing machine that rejects all strings, $f(\langle M_1 \rangle) = (\langle M_1 \rangle, \langle M_2 \rangle)$.

2 The recursion theorem

Can computers duplicate themselves? Yes they can! To give an intuition for how this is done, consider the following "program", that is interpreted by a human instead of a computer:

> Print two copies of the sentence after this one, and place the second copy in quotes.
> "Print two copies of the sentence after this one, and place the second copy in quotes."

We begin by describing a Turing machine that ignores its input and outputs its own description. We start with the following lemma.

Lemma 2 *There exists a computable function* $q : \Sigma^* \to \Sigma^*$ *that on input string* w, $q(w)$ *is the description of a TM,* M_w*, that prints* w *and halts.*

Proof To show that q is computable, we need to describe a TM Q that computes it.
$\underline{Q(w):}$

1. Construct the following TM M_w:
   $\underline{M_w(x):}$
   (a) Ignore x,

(b) Write w on the output tape.

(c) Halt.

2. Output $\langle M_w \rangle$.

■

The TM that outputs its own description has two parts, which we call A and B. A runs first, printing a description of B and then passes control to B, which prints a description of A. A is simply $M_{\langle B \rangle}$, which is the machine that computes $q(\langle B \rangle)$, as described in the previous lemma. In other words, A has a description of B hard-coded into it, and, when run, it prints out this hard-coded description. It is tempting to define B analogously, but notice that then we'd have a circular defintion! B would have a copy of itself hard-coded into itself! Because we defined A to be $q(\langle B \rangle)$, we want B to print $q(\langle B \rangle)$. B cannot have itself hardcoded into itself, but it can get a description of itself from A's output! In other words, once A has completed and printed $\langle B \rangle$ on the work-tape, B can compute $q(\cdot)$ on the value A has left. I.e. B computes $q(\langle B \rangle) = A$, exactly as desired.

Intuitively, then A is the description of a TM that outputs $\langle B \rangle$, and B is the description of a TM that, on input w – in our case, $w = \langle B \rangle$ – outputs $q(w)$, namely the description of a TM that prints w — in our case, A.

Stated again:
$A = M_{\langle B \rangle}$
and
$B =$"on input $\langle M \rangle$:

1. Compute $q(\langle M \rangle)$ and place the result on the output tape.

2. Print $\langle M \rangle$ after it on the output tape.

3. halt."

Theorem 3 *Let T be a TM that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a TM R that computes a function $r : \Sigma^* \to \Sigma^*$, where for every w, $r(w) = t(\langle R \rangle, w)$.*

Intuitively, we should think of t as being any arbitrary computation that takes a description of a TM as its first input. Then, the theorem says, there is always some turing machine R that computes t on itself.

Proof The description of R is very similar to what we described above. We construct it in 3 parts instead of 2: A, B and T, where T is the Turing machine described in the theorem. $A = M_{\langle BT \rangle}$, which is the machine that computes $q(\langle BT \rangle)$. B takes the output A and computes $q(\langle BT \rangle) = M_{\langle BT \rangle} = A$. It combines this with A's output, writing ABT to the work-tape, and passes control to T. ■

2.1 Yet another proof of the halting theorem

Suppose M_{halt} computes $L_{halt} = \{(\langle M \rangle, x) \mid M \text{ halts on } x \text{ with output } 1\}$. Let $\overline{M}_{halt}$ be the Turing machine that runs M_{halt} and reverses its output (i.e. it computes $\overline{L}_{halt}$). By the recursion theorem, there exists some machine R that on input w computes $\overline{M}_{halt}(\langle R \rangle, w)$. In other words, R checks whether it itself halts and accepts w, and then does the opposite!

3 $\mathcal{NP}$ and $\mathcal{NP}$-Completeness

Recall, we have already defined a Karp reduction:

Definition 1 *A language L is* Karp reducible *(or* many-to-one reducible*) to a language L' if there exists a polynomial-time computable function f such that $x \in L$ iff $f(x) \in L'$. We express this by writing $L \leq_p L'$.*

Let's consider a concrete example by showing that $3SAT \leq_p CLIQUE$. Recall, $3SAT$ is the set of satisfiable Boolean formulas in conjunctive normal form. For example, $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_3 \vee x_4)$. (One possible solution is $(x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0)$.) The language $CLIQUE = \{(G, k) \mid G$ has a clique of size $k\}$. We want to show a polynomial-time computable function f that takes a Boolean, CNF formula ϕ as input, and outputs the description of a pair (G, k) with the property that $(G, k) \in CLIQUE \Leftrightarrow \phi \in 3SAT$. Suppose ϕ has k clauses. We construct a graph with $3k$ nodes, and label them each with the variables of ϕ. In other words, there is a 1-1 mapping between literals in the CNF and nodes in the graph. Then, we draw edges between every pair of nodes, *except* a) we do not draw an edge between two nodes if their corresponding literals appear in the same conjunction, and b) we do not draw an edge between two nodes if their corresponding literals are the negation of one another.

This example appears in Sipser's book [1]: Suppose that some $\phi \in 3SAT$, which means that

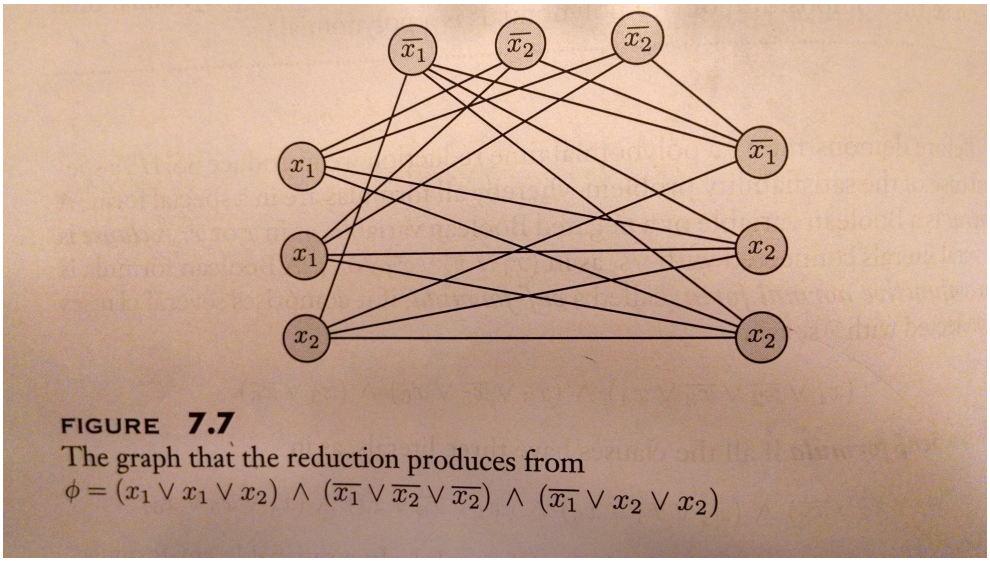

FIGURE **7.7**
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Figure 1: An example of the function f mapping ϕ to (G, k).

it has some satisfying assignment. We claim that $f(\phi) = (G, k) \in CLIQUE$. To see this, choose one literal in each clause that is assigned the value of 1 in the satisfying solution (note that there is at least 1 such value in each clause). We claim that the corresponding nodes constitute a clique in (G, k). This holds because each of the k chosen values appear in different clauses and do not contradict one another. To show that $(G, k) \in CLIQUE$ implies that $\phi \in 3SAT$, consider any clique of size k, and assign the value 1 to the variables corresponding to the nodes in the clique. Because there are edges between all of these nodes, the variables must appear in different clauses, and must not contradict one another.

3.1 $\mathcal{NP}$-Completeness

3.1.1 Defining $\mathcal{NP}$-Completeness

A problem is $\mathcal{NP}$-hard if it is "at least as hard to solve" as any problem in $\mathcal{NP}$. It is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and also in $\mathcal{NP}$. Formally:

Definition 2 *Language L' is $\mathcal{NP}$-*hard *if for every $L \in \mathcal{NP}$ it holds that $L \leq_p L'$. Language L' is $\mathcal{NP}$-*complete *if $L' \in \mathcal{NP}$ and L' is $\mathcal{NP}$-hard.*

Note that if L is $\mathcal{NP}$-hard and $L \leq_p L'$, then L' is $\mathcal{NP}$-hard as well.

3.1.2 Existence of $\mathcal{NP}$-Complete Problems

A priori, it is not clear that there should be any $\mathcal{NP}$-complete problems. One of the surprising results from the early 1970s is that $\mathcal{NP}$-complete problems exist. Soon after, it was shown that many important problems are, in fact, $\mathcal{NP}$-complete. Somewhat amazingly, we now know thousands of $\mathcal{NP}$-complete problems arising from various disciplines.

Here is a trivial $\mathcal{NP}$-complete language:

$$L = \left\{(M, x, 1^t) : \exists w \in \{0,1\}^t \text{ s.t. } M(x, w) \text{ halts within } t \text{ steps with output } 1.\right\}.$$

We will show that $3SAT$ is $\mathcal{NP}$-complete, which is a bit more interesting, and far more useful. In fact, observing that $CLIQUE \in \mathcal{NP}$, if $3SAT$ is $\mathcal{NP}$-complete, it would follow that $CLIQUE$ is as well!

To show that $3SAT$ is $\mathcal{NP}$-complete, we will actually start by showing that SAT is $\mathcal{NP}$-complete, which is the main challenge. We will then reduce from SAT to $3SAT$, which is straightforward.

Theorem 4 (Cook-Levin) *If $SAT \in \mathcal{P}$ then $\mathcal{P} = \mathcal{NP}$.*

Proof To show that SAT is $\mathcal{NP}$-complete, we start with a non-deterministic TM M and some input w. We show how to build a formula ϕ that has a solution iff M has some valid sequence of states that leads to the accepting state, when starting with input w. The size of the resulting formula will be polynomial in the run-time of the machine M. Since any language in $\mathcal{NP}$ has some non-deterministic machine that decides it in polynomial time, it follows that any language in $\mathcal{NP}$ can be reduced to SAT: simply take the machine that decides the language, together with the input w, and build ϕ as described below.

To construct ϕ from (M, w), we start by building a table of size $n^k \times n^k$, where n^k is a bound on the run-time of M on input w, for some constant k. We will think of this table as representing a single branch of the computation of M, with each row representing a single state of this branch of the computation. For example, the first row has the following content in its cells, going from left to right: $q_0, w_1, w_2, \ldots, w_n, \sqcup, \sqcup, \ldots$. Each row has exactly one state value, and we'll use the location of that value to indicate the location of the head on the tape. (We assume only a single tape.) We will say the table is accepting input x if there is an accept state somewhere in the table. Suppose that the machine has the rule $\delta(q_0, w_1) \to (q_1, 0, R)$. Then the 2nd row of this table would have content: $0, q_1, w_2, \ldots, w_n, \sqcup, \sqcup, \ldots$.

Let $V = Q \cup \Gamma$, where Q are the states of M and Γ is its alphabet. Let $C[i, j]$ denote the content of the cell in row i and column j. For every $i, j, \in n^k$ and every $v \in V$, we create a variable: $x_{i,j,v}$.

Note that the number of variables so far is polynomial in n, since there are a polynomial number of states, and since n^{2k} is polynomial in n. ϕ is a formula over these variables, and we will split it into 4 parts: $\phi = \phi_{\mathsf{cell}} \wedge \phi_{\mathsf{start}} \wedge \phi_{\mathsf{move}} \wedge \phi_{\mathsf{accept}}$. We now describe each of these parts:

$$\phi_{\mathsf{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{v \in V} x_{i,j,v} \right) \wedge \left(\bigwedge_{\substack{v,u \in V, \\ v \neq u}} \overline{x_{i,j,v}} \vee \overline{x_{i,j,u}} \right) \right]$$

Intuitively, this captures that there is exactly one value from V in every single cell of the table. (The outermost $\wedge$ is over all pairs in $n^k \times n^k$, making the requirement hold for every cell. The $\vee$ says that at least one of the values is assigned to cell i, j, and the inner $\wedge$ requires that there cannot be more than 1 in any cell.)

ϕ_{start} will be true only if the first row of the table correctly represents the machine's start state on input x. This is done by simply requiring the appropriate variables to be set to 1:

$$\phi_{\mathsf{start}} = x_{1,1,q_0} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \dots$$

ϕ_{accept} ensures that an accept state appears somewhere in the table:

$$\phi_{\mathsf{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\mathsf{accept}}}$$

Finally, ϕ_{move} ensures that each consecutive configuration follows legally from the previous row. This is done by ensuring that the changes (or lack of changes) in each row are consistent with the transition function δ of the Turing machine. We can do this by reducing the correctness check to a collection of statements about 2×3 *windows* of the table. For example,

0	q_1	1
q_2	0	0

Is a legal window if $\delta(q_1, 1) \to (q_2, 0, L)$. Something like

0	0	1
0	0	1

is always a legal window, even though we cannot "see" where the head of the tape is; wherever it is, we know that it cannot write to this part of the table.

Let $a_1, a_2, a_3, a_4, a_5, a_6$ denote the content of some window.

$$\phi_{\mathsf{move}} = \bigwedge_{\substack{1 \le i < n^k \\ 1 < j < n^k}} \bigvee_{\substack{a_1,\dots,a_6 \\ \text{is a legal window}}} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}$$

We now consider the size of the given formula, and the time that it takes to construct this formula. The size of ϕ_{cell} is $O(n^{2k})$. The size of ϕ_{start} is $O(n^k)$. The size of ϕ_{accept} is $O(n^{2k})$, as is the size of ϕ_{move}. It is easy to see that the run-time of constructing each of these sub-formulas is linear in the size of the formula. (Recall that $|V|$ is a constant when viewed as a function of the input size.) The only one that is not immediately obvious is ϕ_{move}. Note that there are only $6^{|V|}$ total possible windows, and determining whether each one is legal can be done in constant time by scanning the description of the M. ■

References

[1] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.