Notes on Computer Theory Last updated: September, 2015

Recursion theorem, $\mathcal{NP}$-completeness

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

1 $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-Completeness

What is complexity theory about? The fundamental question of complexity theory is to understand the inherent complexity of various languages/problems/functions; i.e., what is the most efficient algorithm (Turing machine) deciding some language? A convenient terminology for discussing this is given by introducing the notion of a *class*, which is simply a set of languages. Two basic classes are:

- $\text{TIME}(f(n))$ is the set of languages decidable in time $O(f(n))$. (Formally, $L \in \text{TIME}(f(n))$ if there is a Turing machine M and a constant c such that (1) M decides L, and (2) M runs in time $c \cdot f$; i.e., for all x (of length at least 1) $M(x)$ halts in at most $c \cdot f(|x|)$ steps.)
- $\text{SPACE}(f(n))$ is the set of languages that can be decided using space $O(f(n))$.

Note that we ignore constant factors in the above definitions. This is convenient, and lets us ignore low-level details about the model of computation.[1]

Given some language L, then, we may be interested in determining the "smallest" f for which $L \in \text{TIME}(f(n))$. Or, perhaps we want to show that $\text{SPACE}(f(n))$ is strictly larger than $\text{SPACE}(f'(n))$ for some functions f, f'; that is, that there is some language in the former that is not in the latter. Alternately, we may show that one class contains another. As an example, we start with the following easy result:

Lemma 1 *For any $f(n)$ we have $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$.*

Proof This follows from the observation that a machine cannot write on more than a constant number of cells per move. ■

1.1 The Class $\mathcal{P}$

We now introduce one of the most important classes, which we equate (roughly) with *problems that can be solved efficiently*. This is the class $\mathcal{P}$, which stands for *polynomial time*:

$$\mathcal{P} \stackrel{\text{def}}{=} \bigcup_{c \geq 1} \text{TIME}(n^c).$$

[1] This decision is also motivated by "speedup theorems" which state that if a language can be decided in time (resp., space) $f(n)$ then it can be decided in time (resp., space) $f(n)/c$ for any constant c. (This assumes that $f(n)$ is a "reasonable" function, but the details need not concern us here.)

That is, a language L is in $\mathcal{P}$ if there exists a Turing machine M_L and a polynomial p such that $M_L(x)$ runs in time $p(|x|)$, and M_L decides L.

Does $\mathcal{P}$ really capture efficient computation? There are debates both ways:

- For many problems nowadays that operate on extremely large inputs (think of Google's search algorithms), only linear-time are really desirable. (In fact, one might even want *sub*linear-time algorithms, which are only possible by relaxing the notion of correctness.) This is related to the (less extreme) complaint that an n^{100} algorithm is not really "efficient" in any sense.

  The usual response here is that n^{100}-time algorithms rarely occur. Moreover, when algorithms with high running times (e.g., n^8) *do* get designed, they tend to be quickly improved to be more efficient.

- From the other side, one might object that $\mathcal{P}$ does not capture all efficiently solvable problems. In particular, a *randomized* polynomial-time algorithm (that is correct with high probability) seems to also offer an efficient way of solving a problem. Most people today would agree with this objection, and would classify problems solvable by randomized polynomial-time algorithms as "efficiently solvable". Nevertheless, it may turn out that such problems all lie in $\mathcal{P}$ anyway; this is currently an unresolved conjecture. (We will discuss the power of randomization, and the possibility of derandomization, later in the semester.)

  As mentioned previously, *quantum* polynomial-time algorithms may also be considered "efficient". It is fair to say that until general-purpose quantum computers are implemented, this is still debatable.

Another important feature of $\mathcal{P}$ is that it is closed under composition. That is, if an algorithm A (that otherwise runs in polynomial time) makes polynomially many calls to an algorithm B, and if B runs in polynomial time, then A runs in polynomial time. See [**?**] for further discussion.

1.2 The Class $\mathcal{NP}$

Another important class of problems are those whose solutions can be *verified* efficiently. This is the class $\mathcal{NP}$. (Note: $\mathcal{NP}$ does *not* stand for "non-polynomial time". Rather, it stands for "non-deterministic polynomial-time" for reasons that will become clear later.) Formally,

Definition 1 *$L \in \mathcal{NP}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x, w)$ runs in time[2] $p(|x|)$, and (2) $x \in L$ iff there exists a w such that $M_L(x, w) = 1$.*

Such a w is called a *witness* (or, sometimes, a *proof*) that $x \in L$. Compare this to the definition of $\mathcal{P}$: a language $L \in \mathcal{P}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x)$ runs in time $p(|x|)$, and (2) $x \in L$ iff $M_L(x) = 1$.

Stated informally, a language L is in $\mathcal{P}$ if membership in L can be decided efficiently. A language L is in $\mathcal{NP}$ if membership in L can be efficiently verified (given a correct proof). A classic example is given by the following language:

$$\text{IndSet} = \left\{ (G, k) \ : \ \begin{array}{c} G \text{ is a graph that has} \\ \text{an } \textit{independent set} \text{ of size } k \end{array} \right\}.$$

[2] It is essential that the running time of M_L be measured in terms of the length of x alone. An alternate approach is to require the length of w to be at most $p(|x|)$ in condition (2).

We do not know an efficient algorithm for determining the size of the largest independent set in an arbitrary graph; hence we do not have any efficient algorithm deciding IndSet. However, if we know (e.g., through brute force, or because we constructed G with this property) that an independent set of size k exists in some graph G, it is easy to prove that $(G, k) \in$ IndSet by simply listing the nodes in the independent set: verification just involves checking that every pair of nodes in the given set is *not* connected by an edge in G, which is easy to do in polynomial time. Note further than if G does *not* have an independent set of size k then there is no proof that could convince us otherwise (assuming we are using the stated verification algorithm).

It is also useful to keep in mind an analogy with mathematical statements and proofs (though the correspondence is not rigorously accurate). In this view, $\mathcal{P}$ would correspond to the set of mathematical statements (e.g., "1+1=2") whose truth can be easily determined. $\mathcal{NP}$, on the other hand, would correspond to the set of (true) mathematical statements that have "short" proofs (whether or not such proofs are easy to find).

We have the following simple result, which is the best known as far as relating $\mathcal{NP}$ to the time complexity classes we have introduced thus far:

Theorem 2 $\mathcal{P} \subseteq \mathcal{NP} \subseteq \bigcup_{c \geq 1} \textsc{time}(2^{n^c})$.

Proof The containment $\mathcal{P} \subseteq \mathcal{NP}$ is trivial. As for the second containment, say $L \in \mathcal{NP}$. Then there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x, w)$ runs in time $p(|x|)$, and (2) $x \in L$ iff there exists a w such that $M_L(x, w) = 1$. Since $M_L(x, w)$ runs in time $p(|x|)$, it can read at most the first $p(|x|)$ bits of w and so we may assume that w in condition (2) has length at most $p(|x|)$. The following is then a deterministic algorithm for deciding L:

> On input x, run $M_L(x, w)$ for all strings $w \in \{0,1\}^{\leq p(|x|)}$. If any of these results in $M_L(x, w) = 1$ then output 1; else output 0.

The algorithm clearly decides L. Its running time on input x is $O\left(p(|x|) \cdot 2^{p(|x|)}\right)$, and therefore $L \in \textsc{time}\left(2^{n^c}\right)$ for some constant c. ■

The "classical" definition of $\mathcal{NP}$ is in terms of non-deterministic Turing machines. Briefly, the model here is the same as that of the Turing machines we defined earlier, except that now there are *two* transition functions δ_0, δ_1, and at each step we imagine that the machine makes an arbitrary ("non-deterministic") choice between using δ_0 or δ_1. (Thus, after n steps the machine can be in up to 2^n possible configurations.) Machine M is said to output 1 on input x if there exists *at least one* sequence of choices that would lead to output 1 on that input. (We continue to write $M(x) = 1$ in this case, though we stress again that $M(x) = 1$ when M is a non-deterministic machine just means that $M(x)$ outputs 1 for *some* set of non-deterministic choices.) M decides L if $x \in L \Leftrightarrow M(x) = 1$. A non-deterministic machine M runs in time $T(n)$ if for every input x and every sequence of choices it makes, it halts in time at most $T(|x|)$. The class $\textsc{ntime}(f(n))$ is then defined in the natural way: $L \in \textsc{ntime}(f(n))$ if there is a non-deterministic Turing machine M_L such that $M_L(x)$ runs in time $O(f(|x|))$, and M_L decides L. Non-deterministic space complexity is defined similarly: non-deterministic machine M uses space $T(n)$ if for every input x and every sequence of choices it makes, it halts after writing on at most $T(|x|)$ cells of its work tapes. The class $\textsc{nspace}(f(n))$ is then the set of languages L for which there exists a non-deterministic Turing machine M_L such that $M_L(x)$ uses space $O(f(|x|))$, and M_L decides L.

The above leads to an equivalent definition of $\mathcal{NP}$ paralleling the definition of $\mathcal{P}$:

Claim 3 $\mathcal{NP} = \bigcup_{c \geq 1} \textsc{ntime}(n^c)$.

The major open question of complexity theory is whether $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$; in fact, this is one of the outstanding questions in mathematics today. The general belief is that $\mathcal{P} \neq \mathcal{NP}$, since it seems quite "obvious" that non-determinism is stronger than determinism (i.e., verifying should be easier than solving, in general), and there would be many surprising consequences if $\mathcal{P}$ were equal to $\mathcal{NP}$. (See [**?**] for a discussion.) But we have had no real progress toward proving this belief.

Conjecture 4 $\mathcal{P} \neq \mathcal{NP}$.

A (possibly feasible) open question is to prove that non-determinism is even *somewhat* stronger than determinism. It is known that $\textsc{ntime}(n)$ is strictly stronger than $\textsc{time}(n)$ (see [2, 3, 4] and references therein), but we do not know, e.g., whether $\textsc{time}(n^3) \subseteq \textsc{ntime}(n^2)$.

1.3 Karp Reductions

What does it mean for one language L' to be harder[3] to decide than another language L? There are many possible answers to this question, but one way to start is by capturing the intuition that if L' is harder than L, then an algorithm for deciding L' should be useful for deciding L. We can formalize this idea using the concept of a *reduction*. Various types of reductions can be defined; we start with one of the most central:

Definition 2 *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 $L \leq_p L'$.*

The existence of a Karp reduction from L to L' gives us exactly what we were looking for. Say there is a polynomial-time Turing machine (i.e., algorithm) M' deciding L'. Then we get a polynomial-time algorithm M deciding L by setting $M(x) \stackrel{\text{def}}{=} M'(f(x))$. (Verify that M does, indeed, run in polynomial time.) This explains the choice of notation $L \leq_p L'$.

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 $(x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0)$.) The language $CLIQUE = \{(G, k) \mid G \text{ has a clique of size } k\}$. 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 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

[3] Technically speaking, I mean "at least as hard as".

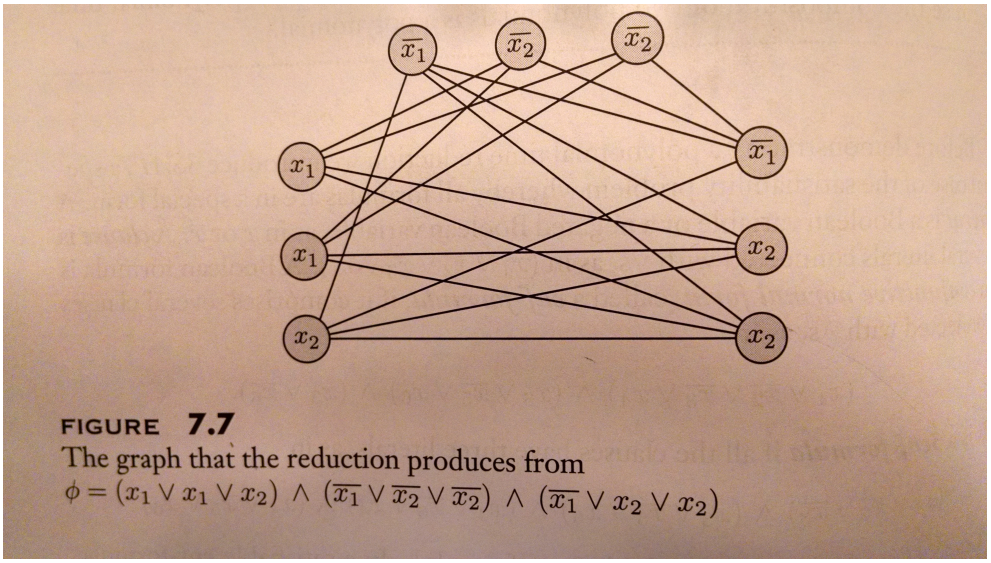

Figure 1: An example of the function f mapping ϕ to (G, k).

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.

1.4 $\mathcal{NP}$-Completeness

1.4.1 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 3 *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.

1.4.2 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 5 (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:

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

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.

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

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.

[2] R. Kannan. Towards separating nondeterminisn from determinisn. *Math. Systems Theory* 17(1): 29–45, 1984.

[3] W. Paul, N. Pippenger, E. Szemeredi, and W. Trotter. On determinism versus non-determinisn and related problems. FOCS 1983.

[4] R. Santhanam. On separators, segregators, and time versus space. *IEEE Conf. Computational Complexity* 2001.