Notes on Theory of Computation Last updated: September, 2021

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

Evgenios Kornaropoulos & 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?

We now give the definitions of running time for deterministic Turing machines.

Definition 1 *Let M be a deterministic Turing machine that halts on all inputs. The* running time *(or time complexity) of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M takes on any input of length n. Customarily, we use n to represent the length of the input.*

A convenient terminology for capturing "complexity" is given by introducing the notion of a *class*, which is simply a set of languages. Two basic classes are:

- $\textsc{time}(f(n))$ is the set of languages decidable in time $O(f(n))$. Formally, $L \in \textsc{time}(f(n))$ 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 such that $|x| \geq 1$, $M(x)$ halts in at most $c \cdot f(|x|)$ steps.
- $\textsc{space}(f(n))$ 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 $L \in \textsc{time}(f(n))$. Or, perhaps we want to show that $\textsc{space}(f(n))$ is strictly larger than $\textsc{space}(f'(n))$ 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 $\textsc{time}(f(n)) \subseteq \textsc{space}(f(n))$.*

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

[1]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.)

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*:

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

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 runs in polynomial time but it 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 (where the output 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.

We note here that through the rest of these notes we may use the terms "language" and "problem" interchangeably. As we discussed in the previous lectures, all problems can be encoded as a language, e.g., the problem "Given a graph $G = (V, E)$ is it possible to construct a Hamiltonian path?" concerns graphs, nevertheless it can be expressed as a decision problem for the appropriate language.

1.2 The Class $\mathcal{NP}$

The languages that belong to class $\mathcal{P}$ can be solved efficiently. On the contrary, for a lot of interesting problems, we do not know any efficient way to determine whether they admit a solution. Some of these problems though have a distinctive feature that is called *polynomial verifiability.* For example, going back to the Hamiltonian path problem, even though we do not know of any efficient algorithm to determine whether a given graph admits a Hamiltonian path, if someone knew of a Hamiltonian path for the given graph, then it would be easy to persuade us of its existence by simply providing the path, also called a *witness*.

The above 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 2 *$L \in \mathcal{NP}$ 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 $M_L(x, w) = 1$.*

With w we denote the *witness* (or, sometimes, a *proof*) that is used to persuade us that $x \in L$. Notice here that the TM of the above definition requires a witness w as an input. Compare this to the definition of $\mathcal{P}$ where the TM decides without any witness: "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 verified efficiently (given a correct proof). Another example is given by the following language:

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

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 $(G, k) \in \text{IndSet}$ 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 *$\mathcal{P} \subseteq \mathcal{NP} \subseteq \bigcup_{c \geq 1} \text{TIME}(2^{n^c})$.*

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:

[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).

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 $L \in \text{TIME}\left(2^{n^c}\right)$ 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.) Non-deterministic TM 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 $\text{NTIME}(f(n))$ is then defined in the natural way: $L \in \text{NTIME}(f(n))$ 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 $\text{NSPACE}(f(n))$ 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 $\mathcal{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c)$.

The major open question of complexity theory is whether $\mathcal{P} \overset{?}{=} \mathcal{NP}$; in fact, this is one of the outstanding questions in mathematics today. The Clay Mathematics Institute is offering a 1 million US dollars reward to anyone who has a formal proof that $\mathcal{P} = \mathcal{NP}$ or that $\mathcal{P} \neq \mathcal{NP}$. 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 $\mathcal{NP}$. 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 $\text{NTIME}(n)$ is strictly stronger than $\text{TIME}(n)$ (see [3, 4, 5] and references therein), but we do not know, e.g., whether $\text{TIME}(n^3) \subseteq \text{NTIME}(n^2)$.

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

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

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 3 *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 $L \leq_p L'$, where the subscript p refers to the polynomial-time computability of f.*

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 $M(x) \overset{\text{def}}{=} M'(f(x))$. (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 $CLIQUE = \{(G, k) \mid G$ has a clique of size $k\}$. In order to show that $3SAT$ is Karp reducible to $CLIQUE$ we need to construct 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]:

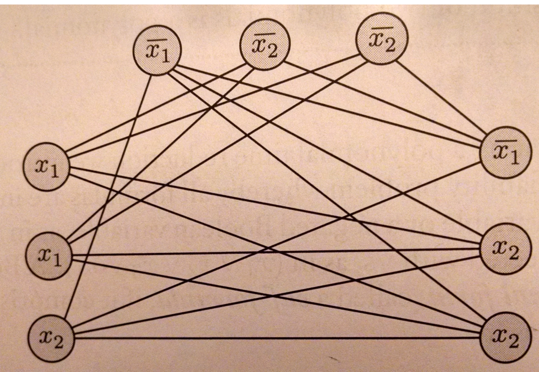

FIGURE 7.7
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

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

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 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 4 *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. This follows from the transitive property of reductions. In fact one can prove the following theorems:

Theorem 5 *If language L is $\mathcal{NP}$-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.*

Theorem 6 *If language L is $\mathcal{NP}$-complete, then $L \in \mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.*

An $\mathcal{NP}$-complete is as hard as any other problem in class $\mathcal{NP}$, or to put it differently any $\mathcal{NP}$ problem can be transformed into any of the $\mathcal{NP}$-complete problems. The last theorem explains the term "complete", i.e., to resolve the question $\mathcal{P} \overset{?}{=} \mathcal{NP}$ it is enough to study whether *any* $\mathcal{NP}$-complete can be decided in polynomial time.

We highlight here a very important detail. In the previous lectures we talked about mapping reductions and we argued that if we know that L is undecidable and we show that $L \leq_m L'$ for a new language L', then L' is undecidable as well. In this subsection the language of interest is the L' in the relation $L \leq_p L'$ and language L is *any* member of $\mathcal{NP}$, as opposed to a fixed and undecidable language. This shows the power and the versatility of reductions.

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:

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

The above language is in $\mathcal{NP}$ because given an input it is simple to verify whether M accepts the input by simulating M. It is $\mathcal{NP}$-complete because the verifier for any instance of a problem in $\mathcal{NP}$ can be encoded as polynomial TM machine that verifies if a given input halts within t steps with output 1.

We refer the reader to Section 2.3 [2] for the proof that $3SAT$ is $\mathcal{NP}$-complete. To show that $3SAT$ is $\mathcal{NP}$-complete, we 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.

References

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

[2] Sanjeev Arora, Boaz Barak: *Computational Complexity - A Modern Approach*. Cambridge University Press 2009, ISBN 978-0-521-42426-4, pp. I-XXIV, 1-579. `https://theory.cs.princeton.edu/complexity/book.pdf`

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

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

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