Notes for Lecture Notes 2

In this lecture we define **NP**, we state the **P** versus **NP** problem, we prove that its formulation for search problems is equivalent to its formulation for decision problems, and we prove the time hierarchy theorem, which remains essentially the only known theorem which allows to show that certain problems are not in **P**.

1 Computational Problems

In a *computational problem*, we are given an *input* that, without loss of generality, we assume to be encoded over the alphabet $\{0,1\}$, and we want to return as *output* a solution satisfying some property: a computational problem is then described by the property that the output has to satisfy given the input.

In this course we will deal with four types of computational problems: *decision* problems, *search* problems, *optimization* problems, and *counting* problems.[1] For the moment, we will discuss decision and search problem.

In a *decision* problem, given an input $x \in \{0,1\}^*$, we are required to give a YES/NO answer. That is, in a decision problem we are only asked to verify whether the input satisfies a certain property. An example of decision problem is the 3-coloring problem: given an undirected graph, determine whether there is a way to assign a "color" chosen from $\{1,2,3\}$ to each vertex in such a way that no two adjacent vertices have the same color.

A convenient way to *specify* a decision problem is to give the set $L \subseteq \{0,1\}^*$ of inputs for which the answer is YES. A subset of $\{0,1\}^*$ is also called a *language*, so, with the previous convention, every decision problem can be specified using a language (and every language specifies a decision problem). For example, if we call 3COL the subset of $\{0,1\}^*$ containing (descriptions of) 3-colorable graphs, then 3COL is the language that specifies the 3-coloring problem. From now on, we will talk about decision problems and languages interchangeably.

In a *search* problem, given an input $x \in \{0,1\}^*$ we want to compute some answer $y \in \{0,1\}^*$ that is in some relation to x, if such a y exists. Thus, a search problem is specified by a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, where $(x,y) \in R$ if and only if y is an admissible answer given x.

Consider for example the search version of the 3-coloring problem: here given an undirected graph $G = (V,E)$ we want to find, if it exists, a coloring $c : V \to \{1,2,3\}$ of the vertices, such that for every $(u,v) \in V$ we have $c(u) \neq c(v)$. This is different (and more demanding) than the decision version, because beyond being asked to determine whether such a c exists, we are also asked to construct it, if it exists. Formally, the 3-coloring problem is specified by the relation R_{3COL} that contains all the pairs (G,c) where G is a 3-colorable graph and c is a valid 3-coloring of G.

[1] This distinction is useful and natural, but it is also arbitrary: in fact every problem can be seen as a search problem

2 P and NP

In most of this course, we will study the *asymptotic* complexity of problems. Instead of considering, say, the time required to solve 3-coloring on graphs with $10,000$ nodes on some particular model of computation, we will ask what is the best asymptotic running time of an algorithm that solves 3-coloring on all instances. In fact, we will be much less ambitious, and we will just ask whether there is a "feasible" asymptotic algorithm for 3-coloring. Here feasible refers more to the rate of growth than to the running time of specific instances of reasonable size.

A standard convention is to call an algorithm "feasible" if it runs in polynomial time, i.e. if there is some polynomial p such that the algorithm runs in time at most $p(n)$ on inputs of length n.

We denote by **P** the class of decision problems that are solvable in polynomial time.

We say that a search problem defined by a relation R is a **NP** search problem if the relation is efficiently computable and such that solutions, if they exist, are short. Formally, R is an **NP** search problem if there is a polynomial time algorithm that, given x and y, decides whether $(x, y) \in R$, and if there is a polynomial p such that if $(x, y) \in R$ then $|y| \leq p(|x|)$.

We say that a decision problem L is an **NP** decision problem if there is some **NP** relation R such that $x \in L$ if and only if there is a y such that $(x, y) \in R$. Equivalently, a decision problem L is an **NP** decision problem if there is a polynomial time algorithm $V(\cdot, \cdot)$ and a polynomial p such that $x \in L$ if and only if there is a y, $|y| \leq p(|x|)$ such that $V(x, y)$ accepts.

We denote by **NP** the class of **NP** decision problems. The class **NP** has the following alternative characterization, from which it takes its name. (**NP** stands for *N*ondeterministic *P*olynomial time.)

Theorem 1 **NP** *is the set of decision problems that are solvable in polynomial time by a non-deterministic Turing machine.*

Proof: Suppose that L is solvable in polynomial time by a non-deterministic Turing machine M: then we can define the relation R such that $(x, t) \in R$ if and only if t is a transcript of an accepting computation of M on input x (that is, the sequence of configurations of an accepting branch of M on input x) and it's easy to prove that R is an **NP** relation and that L is in **NP** according to our first definition. Suppose that L is in **NP** according to our first definition and that R is the corresponding **NP** relation. Then, on input x, a non-deterministic Turing machine can guess a string y of length less than $p(|x|)$ and then accept if and only if $(x, y) \in R$. Such a machine can be implemented to run in non-deterministic polynomial time and it decides L. □

The reason why **NP**, as a complexity class, is defined as a class of decision problems rather than search problems is that one can develop a cleaner theory for decision problems (the definition of reduction, for example, is simpler), and the complexity of decision problems in **NP** completely characterizes the complexity of search problems.

Theorem 2 *For every* **NP** *search problem there is an* **NP** *decision problem such that if the decision problem is solvable in time $t(n)$ then the search problem is solvable in time*

$O(n^{O(1)} \cdot t(n^{O(1)}))$. *In particular,* $\mathbf{P} = \mathbf{NP}$ *if and only if every* **NP** *search problem is solvable in polynomial time.*

PROOF: Let R be an **NP** relation defining an **NP** search problem. Then consider the following decision problem:

- Input: a pair of strings x, w
- Goal: determine if there is a string z such that $(x, w \circ z) \in R$, where $\circ$ denotes string concatenation.

Suppose the above decision problem is solvable in time $\leq t(n)$ by an algorithm A. Then consider the following algorithm for the search problem (ϵ denotes the empty string):

- Input x
- if $A(x, \epsilon) == FALSE$, then return "*no solution*"
- $w := \epsilon$
- while $(x, w) \notin R$
  - if $A(x, w \circ 0) == TRUE$ then $w := w \circ 0$
  - else $w := w \circ 1$
- return w

If there is no solution for x, then the algorithm discovers that at the beginning. Otherwise, every iteration of the *while* loop increases the length of w while maintaining the invariant that w is a prefix of a valid solution to x. Since R is an **NP** relation, such a solution must be of length polynomial in the length of x, and so the loop terminates after a polynomial number of iterations, with a valid solution. □

For a function $t : \mathbb{N} \to \mathbb{N}$, we define by $\mathbf{DTIME}(t(n))$ the set of decision problems that are solvable by a deterministic Turing machine within time $t(n)$ on inputs of length n, and by $\mathbf{NTIME}(t(n))$ the set of decision problems that are solvable by a non-deterministic Turing machine within time $t(n)$ on inputs of length n. Hence, $\mathbf{P} = \bigcup_k \mathbf{DTIME}(O(n^k))$ and $\mathbf{NP} = \bigcup_k \mathbf{NTIME}(O(n^k))$.

3 Diagonalization

Diagonalization is essentially the only way we know of proving separations between complexity classes. The basic principle is the same as in Cantor's proof that the set of real numbers is not countable. First note that if the set of real numbers r in the range $[0, 1)$ is countable then the set of infinite binary sequences is countable: we can identify a real number r in $[0, 1)$ with its binary expansion $r = \sum_{j=1}^{\infty} 2^{-j} r[j]$. If we had an enumeration of real numbers, then we would also have an enumeration of infinite binary string. (The only

thing to watch for is that some real numbers may have two possible binary representations, like $0.01000\cdots$ and $0.001111\cdots$.)

So, suppose towards a contradiction that the set of infinite binary sequences were countable, and let $B_i[j]$ be the j-th bit of the i-th infinite binary sequence. Then define the sequence B whose j-th bits is $1 - B_j[j]$. This is a well-defined sequence but there can be no i such that $B = B_i$, because B differs from B_i in the i-th bit.

Similarly, we can prove that the Halting problem is undecidable by considering the following decision problem D: on input $\langle M \rangle$, the description of a Turing machine, answer NO if $M(\langle M \rangle)$ halts and accepts and YES otherwise. The above problem is decidable if the Halting problem is decidable. However, suppose D where decidable and let T be a Turing machine that solves D, then $T(\langle T \rangle)$ halts and accepts if and only if $T(\langle T \rangle)$ does not halt and accept, which is a contradiction.

It is easy to do something similar with time-bounded computations.

Definition 3 (Time constructible functions) *A function $f : \mathbb{N} \to \mathbb{N}$ is time constructible if there is an algorithm that, on input n, computes the value $t(n)$ in time $O(t(n))$*

All polynomials $p(n)$ are time constructible, as well as all combinations of exponential, polynomial, and root functions such as $2^{\sqrt{n}}$, 2^n, $2^{2^{n^3-1}}$, and so on. Indeed, one needs to work quite hard to come with an example of a function that is not time constructible. One example of a function that is not time constructible is the following: define $t(n)$ to equal n^2 if the number n written in binary encodes a turing machine that halts on all inputs, and define $t(n)$ to equal $n^2 + 1$ otherwise.

Theorem 4 (Time hierarchy theorem – simplified version) *For every time-constructible function $t(\cdot)$, there is a language L such that every algorithm that decides L must run in time $> t(n)$ on infinitely many inputs, but $L \in \mathbf{DTIME}(t^{O(1)}(n))$.*

Proof: The theorem holds for any model of computation which is simulable within the model. For concreteness, we give the proof for the case of Turing machine. We define the language

$$L := \{(M, x) : \ M((M, x)) \text{ halts and rejects in } \leq t(|(M, x)|) \text{ steps } \}$$

That is, an input to the problem is a pair (M, x), where M is a Turing machine and x is a possible input for x, and the goal of the problem is to determine whether M, on input x, halts and rejects in at most $t(n)$ steps, where n is the length of (M, x).

First, we note that L is decidable in time polynomial in $t(n)$ on inputs of length n. Given an input (M, x), we compute $t(n)$, where n is the length of (M, x) in time $O(t(n))$ (here we are using the time-constructibility of $t(\cdot)$) and then we run a simulation of the computation of M on input x. If the simulation halts and reject within $t(n)$ steps, then we accept. If the simulation accepts within $t(n)$ steps, or is still running after $t(n) + 1$ steps, then we reject.

It remains to argue that every machine that decides L must run in time $> t(n)$ for infinitely many inputs.

Let M^* be a machine that decides L and let x be any string. Suppose that M^*, on input (M^*, x), halts after $\leq t(|(M^*, x)|)$ steps. Then if M^*, on input (M^*, x), rejects, then $(M^*, x) \in L$, by definition of L, and so M^* is incorrectly deciding L on input (M^*, x). If M^*, on input (M^*, x) accepts, then $(M^*, x) \notin L$, and so M^* is again incorrect. In either case we reach a contradiction to the assumption that M^* decides L.

It follows that M^* halts after $> t(|M^*, x|)$ steps on all the inputs of the form (M^*, x), of which there are infinitely many. □

We define two more complexity classes in order to derive certain simple consequences of the time hierarchy theorem.

$$\mathbf{E} := \mathbf{DTIME}(2^{O(n)})$$

$$\mathbf{EXP} := \bigcup_k \mathbf{DTIME}(2^{O(n^k)})$$

Corollary 5 $\mathbf{P} \neq \mathbf{E}$ *and* $\mathbf{E} \neq \mathbf{EXP}$

Proof: Apply the time hierarchy theorem to $t(n) = 2^n$, and let L be the resulting language. Then L is decidable in time $2^{O(n)}$, and so it is in $\mathbf{E}$. Suppose that $L \in \mathbf{P}$, and so that there is a machine that decides L in time at most $p(n)$ on inputs of length n, where $p(\cdot)$ is a polynomial. Since $p(n) \leq 2^n$ for all but finitely many values of n, contradicting the property of L guaranteed by the time hierarchy theorem.

Apply the time hierarchy theorem to $t(n) = 2^{n^2}$, and let L' be the resulting language. Then L' is decidable in time $2^{O(n^2)}$, and so $L' \in \mathbf{EXP}$, but we cannot have $L \in \mathbf{E}$ or else the machine deciding L' in time $2^{O(n)}$ would run in time $\leq 2^{n^2}$ for all but a finite number of inputs. □

Besides separating complexity classes, we can also use the hierarchy theorem to prove lower bounds for specific problems. Consider the "time-bounded acceptance" problem BA defined as

$$BA := \{(M, x, t) : M \text{ accepts } x \text{ within } t \text{ time steps}\}$$

Then we see that BA is solvable in time $2^{O(n)}$ given an input of length n. (The simulation can be performed in time polynomial in the length of M, in the length of x, and in the value of t. If n is the overall input length, then the length of M and x are at most n, and the value of t is at most 2^n.)

Now, suppose that there was an algorithm for BA running in time $2^{o(n)}$: we show that this implies that every problem in $\mathbf{E}$ can also be solved in $2^{o(n}$ time. Indeed, let L be a problem in $\mathbf{E}$, and let M be a machine that solves L in time $\leq 2^{cn}$ for a constant c on inputs of length n. Then, given an input x for L of length n, we can prepare the input $(M, x, 2^{cn})$ for BA, and we see that $x \in L$ if and only if $(M, x, 2^{cn}) \in BA$. Since $(M, x, 2^{cn})$ is an input of BA length $n' = O(1) + n + cn$, it can be decided in time $2^{o(n')}$, which is $2^{o(n)}$. Now, deciding all problems in $\mathbf{E}$ in time $2^{o(n)}$ is a violation of the time hierarchy theorem, and so the assumption that there is a $2^{o(n)}$-time algorithm for BA must be false.