Lecture 19

NP, polynomial-time mapping reductions, and NP-completeness

In the previous lecture we discussed deterministic time complexity, along with the time-hierarchy theorem, and introduced two complexity classes: P and EXP. In this lecture we will introduce another complexity class, called NP, and study its relationship to P and EXP. In addition, we will define a polynomial-time variant of mapping reductions along with the very important notion of *completeness* for the class NP.

19.1 The complexity class NP

There are two, equivalent ways to define the complexity class NP that we will cover in this lecture. The first way is arguably more intuitive and more closely connected with the way in which the complexity class NP is typically viewed. The second way directly connects the class with the notion of nondeterminism, and leads to a more general notion (which we will not have time to explore further).

A few conventions about alphabets and encodings

Before getting to the definitions, it will be helpful for us to spend a couple of moments establishing some conventions that will be used throughout the lecture. First, let us agree that for this lecture (and Lectures 20 and 21 as well) we will always take $\Sigma = \{0, 1\}$ to be the binary alphabet, and if we wish to consider other alphabets, which may or may not be equal to Σ, we will use the names Γ and Δ.

Second, when we consider an encoding $\langle x, y \rangle$ of two strings x and y, we will make the assumption that the encoding $\langle x, y \rangle$ itself is always a string over the bi-

nary alphabet Σ, and we assume that this encoding scheme is *efficient*.[1] In particular, it is possible for a DTM that stores x and y on its tape to compute the encoding $\langle x, y \rangle$ in time polynomial in the combined length of x and y, which implies that the length of the encoding $\langle x, y \rangle$ is necessarily polynomial in the combined length of x and y. Furthermore, it must be possible for a DTM that stores the encoding $\langle x, y \rangle$ on its tape to extract each of the strings x and y from that encoding in polynomial time. Finally, assuming that the alphabets from which x and y are drawn have been fixed in advance, we will demand that the length $|\langle x, y \rangle|$ of the encoding depends only on the lengths $|x|$ and $|y|$ of the two strings, and not on the particular symbols that appear in x and y. This particular assumption regarding the length of an encoding $\langle x, y \rangle$ doesn't really have fundamental importance, but it is easily met and will be convenient in the lectures to follow.

As an aside, we may observe that the sort of encoding scheme described in Lecture 13, through which a pair of strings can be encoded as a single string, has all of the properties we require, provided we use fixed-length encodings to encode symbols in a given alphabet Γ as binary strings. In more detail, if it is the case that either of the strings x and y is not a string over the binary alphabet, we first use a fixed-length encoding scheme to encode that string as a binary string, whose length is therefore determined by the length of the string being encoded. Once this is done (possibly to both strings), then we have reduced our task to encoding a pair of binary strings (x, y) into a single binary string $\langle x, y \rangle$. One may first encode the pair (x, y) into the string $x\#y$ over the alphabet $\Sigma \cup \{\#\}$, and then individually encode the elements of $\Sigma \cup \{\#\}$ as length-two strings over Σ. The resulting scheme evidently satisfies all of our requirements.

NP as certificate verification

With the assumptions concerning alphabets and encoding schemes for pairs of strings described above in mind, we define the complexity class NP as follows.

Definition 19.1. Let Γ be an alphabet and let $A \subseteq \Gamma^*$ be a language. The language A is contained in NP if there exists a positive integer k, a time-constructible function $f(n) = O(n^k)$, and a language $B \in \mathrm{P}$ (over the binary alphabet Σ) such that

$$A = \big\{ x \in \Gamma^* \,:\, \text{there exists } y \in \Sigma^* \text{ such that } |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \big\}. \quad (19.1)$$

[1] In general, when we discuss the computational complexity of deciding languages and computing functions that concern various mathematical objects (such as strings, numbers, matrices, graphs, DFAs, DTMs, and any number of other objects), we naturally assume that the encoding schemes used to describe these objects are efficient—polynomial-time computations should suffice to perform standard, low-level manipulations of the encodings.

The essential idea that this definition expresses is that $A \in$ NP means that membership in A is *efficiently verifiable*. The string y in the definition plays the role of a *proof* that a string x is contained A, while the language B represents an efficient *verification procedure* that checks the validity of this proof of membership for x. The terms *certificate* and *witness* are alternatives (to the term *proof*) that are often used to describe the string y.

Example 19.2 (Graph 3-colorability)**.** A classic example of a language in NP is the *graph 3-colorability* language. We define that an undirected graph $G = (V, E)$ is *3-colorable* if there exists a function $c : V \to \{0, 1, 2\}$ from the vertices of G to a three element set $\{0, 1, 2\}$ (whose elements we view as representing three different colors) such that $c(u) \neq c(v)$ for every edge $\{u, v\} \in E$. Such a function is called a *3-coloring* of G. For some graphs there does indeed exists a 3-coloring, and for others there does not—and a natural computational problem associated with this notion is to determine whether or not a given graph has a 3-coloring.

Now, in order to help to clarify the example, let us choose a couple of concrete encoding schemes for graphs and colorings. We will choose the encoding scheme for graphs suggested in Lecture 13, in which we suppose that each graph G has a vertex set of the form $V = \{1, \ldots, m\}$ for some positive integer m, and the encoding $\langle G \rangle$ is a binary string of length m^2 that specifies the adjacency matrix of G. For a function of the form $c : \{1, \ldots, m\} \to \{0, 1, 2\}$, which might or might not be a valid 3-coloring of a graph G with vertex set $\{1, \ldots, m\}$, we may simply encode the colors $\{0, 1, 2\}$ as binary strings (as $0 \to 00$, $1 \to 01$, and $2 \to 10$, let us say) and then concatenate these encodings together in the order $c(1), c(2), \ldots, c(m)$ to obtain a binary string encoding $\langle c \rangle$ of length $2m$.

With respect to the encoding scheme for graphs just described, define a language as follows:

$$\text{3COL} = \big\{ \langle G \rangle \,:\, G \text{ is a 3-colorable graph} \big\}. \tag{19.2}$$

The language 3COL is in NP. To see that this is so, we first define a language B as follows:

$$B = \left\{ \langle \langle G \rangle, y \rangle \,:\, \begin{array}{l} G \text{ is an undirected graph on } m \text{ vertices, and the} \\ \text{first } 2m \text{ bits of } y \text{ encode a valid 3-coloring } c \text{ of } G \end{array} \right\}. \tag{19.3}$$

Observe that the language B is contained in P. For a given input string w, we can first check that $w = \langle \langle G \rangle, y \rangle$ for G being a graph and y being a binary string whose first $2m$ bits (for m being the number of vertices of G) encode a function of the form $c : \{1, \ldots, m\} \to \{0, 1, 2\}$. We can then go through the edges of G one at a time and check that $c(u) \neq c(v)$ for each edge $\{u, v\}$. The entire process can be performed in polynomial time.

Finally, we observe that the requirements of Definition 19.1 are satisfied for the language 3COL in place of A. For the function f it suffices to take $f(n) = n^2 + 1$, which is unnecessarily large in most cases, but it is a polynomially bounded time constructible function, and we always have enough bits to encode the candidate colorings. For any graph G we have that $\langle G \rangle \in$ 3COL if and only if there exists a 3-coloring c of G, which is equivalent to the existence of a string y with $|y| = f(|\langle G \rangle|)$ such that $\langle \langle G \rangle, y \rangle \in B$; for such a string is contained in B if and only if the first $2m$ bits of y actually encode a valid 3-coloring of G.

Before we consider this example to be finished, we should observe that the particular encoding schemes we selected were not really critical—you could of course choose different schemes, so long as a polynomial-time conversion between the selected schemes and the ones described above is possible.

Remark 19.3. Graph 3-coloring is just one of thousands of interesting examples of languages in NP. It also happens to be an NP-complete language, the meaning of which will be made clear by the end of the lecture. A study of the natural computational problems represented by NP-complete languages is a part of CS 341, and so we will not venture in this direction in this course.

NP as polynomial-time nondeterministic computations

As was already suggested before, there is a second way to define the class NP that is equivalent to the one above. The way that this definition works is that we first define a complexity class NTIME(f), for every function $f : \mathbb{N} \to \mathbb{N}$, to be the class of all languages that are decided by a nondeterministic Turing machine running in time $O(f(n))$.

In more detail, to say that a nondeterministic Turing machine (or NTM) N runs in time $t(n)$ means that, for every $n \in \mathbb{N}$ and every input string w of length n, it is the case that *all* computation paths of N on input w lead to either acceptance or rejection after $t(n)$ or fewer steps. To say that such an NTM decides a language A means that, in addition to satisfying the condition relating to running time just mentioned, N has an accepting computation path on each input w if and only if $w \in A$.

Once we have defined NTIME(f) for every function $f : \mathbb{N} \to \mathbb{N}$, we define NP in a similar way to what we did for P:

$$\text{NP} = \bigcup_{k \geq 1} \text{NTIME}(n^k). \tag{19.4}$$

This definition is where NP gets its name: NP is short for *nondeterministic polynomial time*.

The equivalence of the two definitions is not too difficult to establish, but we won't go through it in detail. The basic ideas needed to prove the equivalence are (i) a nondeterministic Turing machine can first guess a polynomial-length binary string certificate and then verify it deterministically, and (ii) a polynomial-length binary string certificate can encode the nondeterministic moves of a polynomial-time nondeterministic Turing machine, and it could then be verified deterministically in polynomial time that this sequence of nondeterministic moves would lead the original NTM to acceptance.

Relationships among P, NP, and EXP

Let us now observe the following inclusions:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP}. \tag{19.5}$$

The first of these inclusions, $\mathrm{P} \subseteq \mathrm{NP}$, is straightforward. Suppose Γ is an alphabet and $A \subseteq \Gamma^*$ is a language, and assume that $A \in \mathrm{P}$. We may then define a language $B \subseteq \Sigma^*$ as follows:

$$B = \bigl\{ \langle x, y \rangle \,:\, x \in A,\ y \in \Sigma^* \bigr\}. \tag{19.6}$$

It is evident that $B \in \mathrm{P}$; if we have a DTM M_A that decides A in polynomial time, we can easily decide B in polynomial time by first decoding x from $\langle x, y \rangle$ (and completely ignoring y), and then running M_A on x. For $f(n) = n^2$, or any other polynomially bounded, time-constructible function, it holds that

$$A = \bigl\{ x \in \Gamma^* : \text{there exists } y \in \Sigma^* \text{ such that } |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \bigr\}, \tag{19.7}$$

and therefore $A \in \mathrm{NP}$.

With respect to the second definition of NP, as the union of $\mathrm{NTIME}(n^k)$ over all $k \geq 1$, the inclusion $\mathrm{P} \subseteq \mathrm{NP}$ is perhaps obvious. Every DTM is equivalent to an NTM that happens never to allow multiple nondeterministic moves during its computation, so

$$\mathrm{DTIME}(f) \subseteq \mathrm{NTIME}(f) \tag{19.8}$$

holds for all functions $f : \mathbb{N} \to \mathbb{N}$, and therefore

$$\mathrm{DTIME}(n^k) \subseteq \mathrm{NTIME}(n^k) \tag{19.9}$$

for all $k \geq 1$, which implies $\mathrm{P} \subseteq \mathrm{NP}$.

Now let us observe that $\mathrm{NP} \subseteq \mathrm{EXP}$. Suppose Γ is an alphabet and $A \subseteq \Gamma^*$ is language such that $A \in \mathrm{NP}$. This implies that there exists a positive integer k, a

time-constructible function f satisfying $f(n) = O(n^k)$, and a language $B \in \mathrm{P}$ such that

$$A = \big\{ x \in \Sigma^* \,:\, \text{there exists } y \in \Sigma^* \text{ such that } |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \big\}. \quad (19.10)$$

Define a DTM M as follows:

On input $x \in \Gamma^*$:

1. For every string $y \in \Sigma^*$ satisfying $|y| = f(|x|)$, do the following:
2. *Accept* if it holds that $\langle x, y \rangle \in B$.
3. *Reject*.

It is evident that M decides A, as it simply searches over the set of all strings $y \in \Sigma^*$ with $|y| = f(|x|)$ to find if there exists one such that $\langle x, y \rangle \in B$. It remains to consider the running time of M.

Let us first consider step 2, in which M tests whether $\langle x, y \rangle \in B$ for an input string $x \in \Gamma^*$ and a binary string y satisfying $|y| = f(|x|)$. This test takes a number of steps that is polynomial in $|x|$, and the reason why is as follows. First, we have $|y| = f(|x|)$ for $f(n) = O(n^k)$, and therefore the length of the string $|\langle x, y \rangle|$ is polynomial in $|x|$. Now, because $B \in \mathrm{P}$, we have that membership in B can be tested in polynomial time. Because the input in this case is $\langle x, y \rangle$, this means that the time required to test membership in B is polynomial in $|\langle x, y \rangle|$. However, because the composition of two polynomials is another polynomial, we have that the time required to test whether $\langle x, y \rangle \in B$ is polynomial in $|x|$.

Next, on any input of length n, the number of times $T(n)$ the for-loop is iterated is given by

$$T(n) = 2^{f(n)}. \quad (19.11)$$

Because we have $f(n) = O(n^k)$, it therefore holds that

$$T(n) = O\Big(2^{n^{k+1}}\Big). \quad (19.12)$$

Finally, there are some other manipulations that M must perform, such as computing $f(|x|)$, managing how the loop is to range over strings y with $|y| = f(|x|)$, and computing $\langle x, y \rangle$ for each choice of y. For each iteration of the loop, these manipulations can all be performed in time polynomial in $|x|$.

Putting everything together, the entire computation certainly runs in time

$$O\Big(2^{n^{k+2}}\Big), \quad (19.13)$$

which is a rather coarse upper bound that is nevertheless sufficient for our needs. We have established that M runs in exponential time, so $A \in \mathrm{EXP}$.

Now we know that $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP}$, and we also know that $\mathrm{P} \subsetneq \mathrm{EXP}$ by the time hierarchy theorem. Of course this means that one (or both) of the following proper containments must hold: (i) $\mathrm{P} \subsetneq \mathrm{NP}$, or (ii) $\mathrm{NP} \subsetneq \mathrm{EXP}$. Neither one has yet been proved, and a correct proof of either one would be a major breakthrough in complexity theory. (Determining whether or not $\mathrm{P} = \mathrm{NP}$ is, in fact, widely viewed as being among the greatest mathematical challenges of our time.)

19.2 Polynomial-time reductions and NP-completeness

We discussed reductions in Lecture 16 and used them to prove that certain languages are undecidable or non-Turing-recognizable. *Polynomial-time reductions* are defined similarly, except that we add the condition that the reductions themselves must be given by polynomial-time computable functions.

Definition 19.4. Let Γ and Δ be alphabets and let $A \subseteq \Gamma^*$ and $B \subseteq \Delta^*$ be languages. It is said that A *polynomial-time reduces* to B if there exists a polynomial-time computable function $f : \Gamma^* \to \Delta^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \tag{19.14}$$

for all $w \in \Gamma^*$. One writes

$$A \leq_m^p B \tag{19.15}$$

to indicate that A polynomial-time reduces to B, and any function f that establishes that this is so may be called a *polynomial-time reduction* from A to B.

Polynomial-time reductions of this form are sometimes called *polynomial-time mapping reductions* (and also *polynomial-time many-to-one reductions*) to differentiate them from other types of reductions that we will not consider—but we will stick with the term *polynomial-time reductions* for simplicity. They are also sometimes called *Karp reductions*, named after Richard Karp, one of the pioneers of the theory of NP-completeness.

With the definition of polynomial-time reductions in hand, we can now define NP-completeness.

Definition 19.5. Let Γ be an alphabet and let $B \subseteq \Gamma^*$ be a language.

1. It is said that B is NP-hard if, for every language $A \in \mathrm{NP}$, it holds that $A \leq_m^p B$.
2. It is said that B is NP-complete if B is NP-hard and $B \in \mathrm{NP}$.

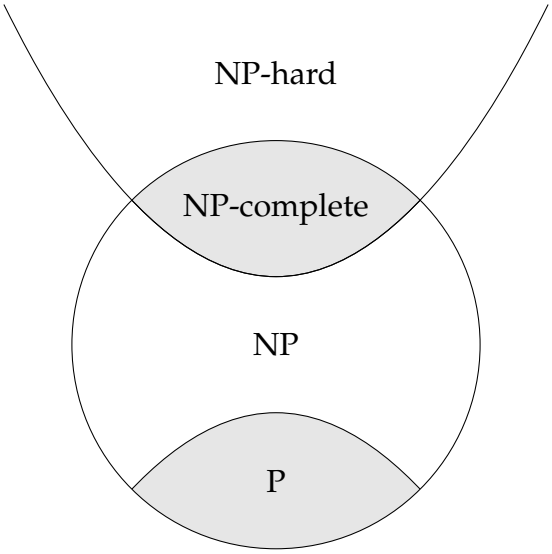

Figure 19.1: The relationship among the classes P and NP, and the NP-hard and NP-complete languages. The figure assumes P $\neq$ NP.

The idea behind this definition is that the NP-complete languages represent the hardest languages to decide in NP—*every* language in NP can be polynomial-time reduced to an NP-complete language, so if we view the difficulty of performing a polynomial-time reduction as being negligible, the ability to decide any one NP-complete language would give you a key to unlocking the computational difficulty of the class NP in its entirety. Figure 19.1 illustrates the relationship among the classes P and NP, and the NP-hard and NP-complete languages, under the assumption that P $\neq$ NP.

Now, it is not at all obvious from the definition that there should exist any NP-complete languages at all, for it is a strong condition that *every* language in NP must polynomial-time reduce to such a language. We will, however, prove that NP-complete languages do exist. There are, in fact, thousands of known NP-complete languages that correspond to natural computational problems of interest.

We will finish off the lecture by listing several properties of polynomial-time reductions, NP-completeness, and related concepts. Each of these facts has a simple proof, so if you are interested in some practice problems, try proving all of these facts. For all of these facts, it is to be assumed that A, B, and C are languages.

1. If $A \leq_m^p B$ and $B \leq_m^p C$, then $A \leq_m^p C$.
2. If $A \leq_m^p B$ and $B \in$ P, then $A \in$ P.
3. If $A \leq_m^p B$ and $B \in$ NP, then $A \in$ NP.

4. If A is NP-hard and $A \leq_m^p B$, then B is NP-hard.
5. If A is NP-complete, $B \in \text{NP}$, and $A \leq_m^p B$, then B is NP-complete.
6. If A is NP-hard and $A \in \text{P}$, then $\text{P} = \text{NP}$.

We typically use statement 5 when we wish to prove that a certain language B is NP-complete: we first prove that $B \in \text{NP}$ (which is often easy) and then look for a known NP-complete language A for which we can prove $A \leq_m^p B$. This type of problem is done in CS 341. Of course, to get things started, we need to find a "first" NP-complete language that can be reduced to others—effectively priming the NP-completeness pump. This is the content of the *Cook–Levin theorem*, which we will discuss in Lecture 21.