CS 6902 (Theory of computation) – Lecture 5

Antonina Kolokolova *

May 21, 2018

1 Complexity Classes

Unless explicitly stated, by a TM we will mean a *multi-tape* TM. When we talk about the space used by a multi-tape TM, we only count the number of cells on the worktape(s) that are touched by the TM during its computation. That is, we *do not* count the input tape or the output tape.

We will be mostly using the alphabet $\{0, 1\}$ (though any finite alphabet with at least two letters will do). Let n denote the input size, and $f : \mathbb{N} \to \mathbb{N}$ be a function.

- $\mathsf{Time}(f(n)) = \{L \mid L$ is decided by some TM M in at most $T_M(n) \in O(f(n))$ steps$\}$
- $\mathsf{Space}(f(n)) = \{L \mid L$ is decided by some TM using at most $O(f(n))$ cells of its worktapes$\}$

In particular, some common classes in complexity are

- $\mathsf{P} = \cup_{k\geq 1}\mathsf{Time}(n^k)$,
- $\mathsf{EXP} = \cup_{k\geq 1}\mathsf{Time}(2^{n^k})$,
- $\mathsf{L} = \mathsf{Space}(\log n)$,
- $\mathsf{PSPACE} = \cup_{k\geq 1}\mathsf{Space}(n^k)$.

In general, a *complexity class* is a collection of languages decidable within a given amount of computational resources. We will also talk about polynomial-time (exponential-time, log-space, etc) computable functions by extending these definitions to Turing machines that produce output (by saying that they halt and produce an output within time or space limit.)

We say that a complexity class is "robust" (intuitive notion) if:

1. no reasonable changes to the model of computation changes the class, and

*This lecture is a modification of notes by Valentine Kabanets.

2. the class contains interesting problems.

For example, the class P is a robust class: changing the definition of a TM (say, to multi-tape TMs, or Random-Access Machine such as a modern computer) will not change the class of polytime solvable problems. And composing two polynomial-time computable functions gives a polynomial-time computable function. In contrast, the class of linear-time solvable problems is not robust! As we saw earlier, the language of palindromes is in linear time for 2-tape TMs, but not for 1-tape TMs.

To make it easier to think about these classes, think of complete problems for them (recall that a problem complete for a complexity class is the "hardest problem in the class" under corresponding reductions).

- L: Is a given graph G a tree?
- P: System of linear equations (with integer coefficients) solvabilty (is there a solution x to a system $Ax = b$, where A, b are a matrix and a vector over integers?)
- PSPACE: Is there a winning strategy in tic-tac-toe from a given position on a $n \times n$ board? (similar for Hex, Reversi, Mahjong).
- EXP: Is there a winning strategy in chess from a given position on a $n \times n$ board? (similar for checkers, Shogi, Japanese Go).

We will now proceed to show

$$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$$

Note that although all these inclusions are not strict (e.g., it is not impossible that $\mathsf{P} = \mathsf{L}$ or that $\mathsf{P} = \mathsf{PSPACE}$, albeit unlikely), we can show that $\mathsf{L} \subsetneq \mathsf{PSPACE}$ and $\mathsf{P} \subsetneq \mathsf{EXP}$. The general form of such result is called the hierarchy theorems (for space and time, in this case).

2 Hierarchy Theorems

Before, we were using O-notation to show that one function asymptotically grows no faster than another. Here, we want to be able to talk about a function growing strictly slower than another. For that, we introduce "little" analogues of big-O (and big-Ω) as follows.

Definition 1. Let $g(n), f(n) \colon \mathbb{N} \to \mathbb{N}$ be functions. We say that $g(n) \in o(f(n))$ (respectively, $g(n) \in \omega(f(n))$) if $\forall c > 0 \exists n_0 \forall n > n_0$ $g(n) \leq cf(n)$ (respectively, $g(n) \geq cf(n)$ for ω).

If you know calculus, an alternative way to think about those definitions (as well as O, Ω and Θ) is using the notion of a limit. Consider $\lim_{n\to\infty} g(n)/f(n)$ such that a limit exists. If this limit is 0, then $g(n) \in o(f(n))$. If $\lim_{n\to\infty} g(n)/f(n) \to \infty$, then $g(n) \in \omega(f(n))$. If $\lim_{n\to\infty} g(n)/f(n) = c$ for some constant c, then $g(n) \in \Theta(f(n))$. And finally $O(f(n))$ and $\Omega(f(n))$ tell us that the limit is at most (respectively, at least) some constant.

For example, let $f(n) = n^2$. Then $2n^2$ and $n^2 + \log n$ are in $\Theta(n^2)$ (that is, in both $O(n^2)$ and $\Omega(n^2)$). However, $n^{1.5}$, $n \log n$ or $(\log n)^2$ are not only in $O(n^2)$, they are in $o(n^2)$: that is, they grow strictly slower than n^2, and the limit of the ratio of these functions over n^2 goes to 0. Similarly, $n^3, 2^n$ and $n^2 \log n$ are not only in $\Omega(n^2)$, but moreover in $\omega(n^2)$.

Theorem 1 (Hierarchy theorems)**.** *For every proper*[1] *function* $f(n) \geq n$*,*

$$\mathsf{Time}(o(f(n)/\log f(n))) \subsetneq \mathsf{Time}(f(n)) \qquad \text{(Time Hierarchy Theorem)}$$

For every proper function $f(n) \geq \log n$*,*

$$\mathsf{Space}(o(f(n))) \subsetneq \mathsf{Space}(f(n)). \qquad \text{(Space Hierarchy Theorem)}$$

Intuitively, these theorems say that given more time (by asymptotically more than a $\log n$ factor) or more space (by more than a constant factor), Turing machines can decide more languages. The proof is by diagonalization; we will skip it here.

As a simple application of these Hierarchy Theorems, we can prove that $\mathsf{P} \subsetneq \mathsf{EXP}$ and $\mathsf{L} \subsetneq \mathsf{PSPACE}$.

3 Relationships among complexity classes

We'll show that any language decidable in space $s(n)$ can be decided in time, roughly, $2^{s(n)}$. This will yield the following.

Theorem 2. $\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$.

The middle inclusion ($\mathsf{P} \subseteq \mathsf{PSPACE}$) is trivial: in polynomial time, no TM can touch more than polynomial number of tape cells.

To prove the other two inclusions, we recall the notion of a *configuration* of a TM. If at a given moment in time, a TM is in state q, and its tape contains symbols $\sigma_1\sigma_2 \ldots \sigma_i\sigma_{i+1} \ldots \sigma_m$, and its tape head is scanning the symbol σ_{i+1}, then the configuration of the TM at this moment in time is

$$C = \sigma_1\sigma_2 \ldots \sigma_i q \sigma_{i+1} \ldots \sigma_m.$$

(The state name is immediately to the left of the symbol currently scanned by the TM.)

Start configuration (for a one-tape TM) on input x is $q_{start}x_1x_2 \ldots x_n$.

We say that a configuration C *yields* C' in one step if a TM in configuration C goes to configuration C' in one step, using its transition function.

Example 1. Suppose that the tape contains the string "0110b" as its first $f(n) = 5$ cells, and that the head is pointing to the second occurrence of a 1, with the TM being in state q_3. The corresponding configuration is $01q_310b$.

[1] A function $f(n)$ is called *proper* if there is a TM M that, on input 1^n, outputs exactly $f(n)$ symbols and runs in time $O(f(n) + n)$ and space $O(f(n))$. The usual functions like $\log n$, $\sqrt{n}$, n^2, 2^n, $n!$ are proper. Also, if f and g are proper, then so are $f + g$, fg, $f(g)$, f^g, 2^g. In this course, we will always use proper complexity functions for bounds.

We now define the *configuration graph* of a given TM M on input x:

- Nodes = configurations of M,
- Edges = $\{(C, C') \mid$ C yields C' in one step$\}$

Easy Fact: The number of configurations of a 2-tape TM (with one read-only input tape and one work tape) is at most:

- n (input-tape head positions)
- $*f(n)$ (work-tape head positions)
- $*|Q|$ (state)
- $*|\Gamma|^{f(n)}$ (work-tape contents).

Note: the contents of the read-only input tape is not part of the configuration.

Proof of Theorem 2. For $f(n) = c \log n$, the number of configurations is at most

$$n * c \log n * c_0 * c_1^{c \log n} \leq n^{c_2},$$

a polynomial.

For $f(n) = n^c$, the number of configurations is at most

$$n * n^c * c_0 * c_1^{n^c} \leq 2^{n^{c_2}},$$

an exponential function.

To determine if a TM M accepts x,

1. construct the configuration graph of M on x;
2. check if q_{accept}-configuration is reachable from the start configuration (using, e.g., DFS); this can be done in time polynomial in the size of the graph.

Hence, for $f(n) = c \log n$, we need $\mathsf{poly}(n)$ time; and for $f(n) = n^c$, we need $\mathsf{poly}(2^{n^{c_2}})$ time. □

Theorem 3. $\mathsf{L} \subsetneq \mathsf{PSPACE}$

Proof. Note that $L \subseteq \mathsf{Space}(n)$ trivially. Now, by the Space Hierarchy Theorem, $\mathsf{Space}(n) \subsetneq \mathsf{Space}(n \log n)$. Finally, $\mathsf{Space}(n \log n) \subseteq \mathsf{PSPACE}$ trivially. □

As an immediate consequence, we obtain the following

Theorem 4. *Among the inclusions* $\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE}$ *at least one inclusion must be proper. Among the inclusions* $\mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$ *at least one inclusion must be proper.*

Proof. Again we prove just the first part. Suppose that all inclusions are equalities. Then $\mathsf{L} = \mathsf{P} = \mathsf{PSPACE}$, contradicting Theorem 3. □

Major Open Questions:

- $\mathsf{L} \stackrel{?}{=} \mathsf{P}$
- $\mathsf{P} \stackrel{?}{=} \mathsf{PSPACE}$
- $\mathsf{PSPACE} \stackrel{?}{=} \mathsf{EXP}$

By Theorem 4, at least one of these three questions must have a negative answer. The bizarre fact is: *we do not know which one!* (even though we suspect that all of these questions have negative answers.)

4 "Padding" Technique

Suppose $L \in \mathsf{EXP}$ is decided by a TM M in time 2^{n^c}, for some constant c. Define a new language

$$L_{pad} = \{x\#^{2^{|x|^c}} \mid x \in L\}$$

(here, # is a new symbol outside the alphabet of M).

The TM M can be easily modified to M' that decides L_{pad}: just ignore the #'s. Now, the running time of M' on input $x\#^{2^{|x|^c}}$ of size $n = |x| + 2^{|x|^c}$ is $O(n)$, linear!

Why is this useful?

Theorem 5. *If* $\mathsf{P} = \mathsf{L}$*, then* $\mathsf{EXP} = \mathsf{PSPACE}$*.*

Proof. Take an arbitrary $L \in \mathsf{EXP}$, decided by some TM M in time 2^{n^c}. Construct L_{pad} as explained above. Since $L_{pad} \in \mathsf{P}$, by our assumption $\mathsf{P} = \mathsf{L}$, we get that $L_{pad} \in \mathsf{L}$. That is, there is some TM M_0 deciding L_{pad} in space $\log n$. This TM M_0 can be used to decide L as follows: On input x, simulate M_0 on the padded input $x\#^{2^{|x|^c}}$; accept iff M_0 accepts.

Now, the space we used on input x of size n is the space used by M_0 on the padded input $x\#^{2^{|x|^c}}$ of size $n + 2^{n^c}$, which is at most $\log(2^{n^c+1}) = n^c + 1$, a polynomial. Hence, we have that $L \in \mathsf{PSPACE}$.

One technical point: If we actually were to write down the padded input $x\#^{2^{|x|^c}}$ on our work tape, this would make our space usage exponential in n. But, we do not need to write this padded string down! We know what it looks like to the right of x. So we can simulate M_0 on the virtual padded input $x\#^{2^{|x|^c}}$ by keeping a counter which tells us the position on the tape of M_0. If M_0 enquires about the position to the right of x, we respond with the symbol "#" (or the blank, if M_0 goes to the right of the entire padded input). Since the counter can assume the value at most 2^{n^c}, we need at most n^c bits for the counter. Thus, our new TM uses space at most polynomial in n. □

The theorem above is just one example of a more general phenomenon: "Padding" allows us to translate upwards equalities between complexity classes.

5 Scaling down from computability to complexity

In real life, we are interested whether a problem can be solved efficiently; just knowing that something is decidable is not enough for practical purposes. The Complexity Theory studies languages from the point of view of efficient solvability. But what is efficient? The accepted definition in complexity theory equates efficiency with being computable in time polynomial in the input length (number of bits to encode a problem). So a problem is efficiently solvable (a language is efficiently decidable) if there exists an algorithm (such as a Turing machine algorithm) that solves it in time $O(n^d)$ for some constant d. Thus, an algorithm running in time $O(n^2)$ is efficient; an algorithm running in time $O(2^n)$ is not. This leads us to consider

Recall that we defined, for some function $f\colon \mathbb{N} \to \mathbb{N}$, $\mathsf{Time}(f(n)) = \{L \mid L \text{ is decided by some TM in at most } f(n) \text{ steps }\}$. Now, a problem is *efficiently* solvable if $f(n)$ is $O(n^d)$ for some constant d (that is, f is polytime (polynomial-time) computable). In particular, the class of languages which are efficiently decidable is the class P that we defined at the start of this lecture. Most of the problems that you encountered in your algorithms course were solvable in polynomial time; most algorithms you studied run in polynomial time: Dijkstra's algorithm, Euclid's algorithm, depth first search, etc.

To go from computability to complexity we will augment all the definitions (of decidable, semi-decidable, co-semi-decidable and in general arithmetic hierarchy, reducibility, completeness, etc) with "efficient" and "short", meaning polynomial time and polynomial length. That is, we obtain complexity versions of classes of languages from computability by bounding all quantifiers by a polynomial in the input length.

Scaling down decidable, as we just did, we get P. Scaling down semi-decidable gives us NP; the efficient counterpart of co-semi-decidable is co-NP, and overall, arithmetic hierarchy (alternating quantifiers) scales down to polynomial-time hierarchy (polynomially-bounded alternating quantifiers). Our reductions will also become efficient (polynomial-time computable).

In this course we will spend a fair bit of time on P, NP and the major open problem asking whether the two are distinct. NP is a class of languages that contains all of P, but which most people think also contains many languages that aren't in P. Informally, a language L is in NP if there is a "guess-and-check" algorithm for L. That is, there has to be an efficient verification algorithm with the property that any $x \in L$ can be verified to be in L by presenting the verification algorithm with an appropriate, short "certificate" string y.

Remark: NP stands for "nondeterministic polynomial time", because one of the ways of defining the class is via nondeterministic Turing machines (that is, as a class of languages computed in polynomial time on a non-deterministic Turing machine). NP does *not* stand for "not polynomial", and as we said, NP includes as a subset all of P. That is, many languages in NP are very simple: in particular, all regular languages and all context-free languages are in P and therefore in NP.

We now give the formal definition. For convenience, from now on we will assume that all our languages are over the fixed alphabet Σ, and we will assume $0, 1 \in \Sigma$.

Definition 2. Let $L \subseteq \Sigma^*$. We say $L \in \mathsf{NP}$ if there is a two-place predicate $R \subseteq \Sigma^* \times \Sigma^*$

such that R is computable in polynomial time, and such that for some $c, d \in \mathbb{N}$ we have for all $x \in \Sigma^*$, $x \in L \Leftrightarrow \exists y \in \Sigma^*, |y| \leq c|x|^d$ and $R(x, y)$.

We have to say what it means for a two-place predicate $R \subseteq \Sigma^* \times \Sigma^*$ to be computable in polynomial time. One way is to say that $\{\langle x, y\rangle \,|\, (x, y) \in R\} \in \mathsf{P}$, where $\langle x, y\rangle$ is our standard encoding of the pair x, y. Another, equivalent, way is to say that there is a Turing machine M which, if given $x\#y$ on its input tape, halts in time polynomial in $|x| + |y|$, and accepts if and only if $(x, y) \in R$, just like we did for the definition of semi-decidable via existence of a verifier.

Most languages that are in NP are easily shown to be in NP, since this fact usually follows immediately from their definition,

Example 2. A *clique* in a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that there is an edge between every pair of vertices in S. That is, $\forall u, v \in V(u \in S \wedge v \in S \rightarrow E(u, v))$. The language $CLIQUE = \{< G, k > |G$ is a graph, $k \in N$, G has a clique of size $k\}$. Here, we take $n = |V|$, so the size of the input is $O(n^2)$.

We will see later that this problem is NP-complete. Now we will show that it is in NP.

Suppose that $< G, k > \in CLIQUE$. That is, G has a set S of vertices, $|S| = k$, such that for any pair $u, v \in S$, $E(u, v)$. Guess this S using an existential quantifier. It can be represented as a binary string of length n, so its length is polynomial in the size of the input. Now, it takes k^2 checks to verify that every pair of vertices in S is connected by an edge. If the algorithm is scanning E every time, it takes $O(n^2)$ steps to check that a given pair has an edge between them. Therefore, the total time for the check is $k^2 \cdot n^2$, which is quadratic in the length of the input (since E is of size n^2, the input is of size $O(n^2)$ as well).

One common source of confusion, especially among non-computer scientists, is that NP is a class of languages, not a class of optimization problems for which a solution needs to be computed. So if instead of the $CLIQUE$ problem above you formulate your problem as "compute a maximum size clique in G", or even "compute a clique of size k", these problems would not be in NP. They could still be NP-hard, but they cannot be NP-complete.