Part III
Introduction to Computational Complexity

Lectured by Timothy Gowers

Artur Avameri

Contents

0 Introduction

A good book for the course is the first few chapters of *Computational Complexity: A modern approach* by Arora and Barack.

0.1 Computational problems

A problem in complexity theory is somewhat different from a mathematical problem – it is more like a class of problems.

Example 0.1. Given: A graph G with n vertices and $x, y \in V(G)$. Problem: Is there a path from x to y? This is a decision problem (yes/no answer).

A problem has a variable input and an output. If the output belongs to $\{0,1\}$, i.e. {no, yes}, then the problem is called a **decision problem**. Write $\{0,1\}^*$ for the set of $\bigcup_{n=1}^{\infty}\{0,1\}^n$. Then a decision problem can be encoded as a **Boolean function**, that is, a function $f : \{0,1\}^* \to \{0,1\}$.

The set $\{x \in \{0,1\}^* \mid f(x) = 1\}$ is called the **language** defined by f.

0.2 Turing machines

A **Turing machine** formalizes the notion of an algorithm. There are many ways to formalize it – we (handwavily) describe a few.

A k–**tape Turing machine** consists of several ingredients. The first is a collection of k **tapes**, where a tape is an infinite sequence of **cells**. There is also a finite set A called the **alphabet**, and each cell contains an element of A. There is also a **head** which is in a **state** (an element of a finite set S of states) and in a **position** in each state. S contains two special states, S_{init} and S_{halt}.

A state is a function that takes as input an element of $A^k \times S$ and outputs an element of $A^k \times S \times \{L, N, R\}^k$. If S is this "transition function", then the machine rewrites $(a_1, \ldots, a_k)$ according to the A^k component of the image, changes the state of the head according to the S component, and shifts each tape according to the $\{L, N, R\}^k$ component.

One tape is designated as the input tape and is never changed, another is the output tape. All tapes other than the input tape start full of zeroes. If the machine reaches the state S_{halt}, it stops. If the input is x and the output is y, we say that the machine computed y given input x.

Variants assume that $A = \{0,1\}$; that $k = 1$ (with a different convention about input–output tapes); that tapes are two–sided, etc.

1 Some complexity classes

Definition 1.1. The complexity class P consists of all Boolean functions (i.e. problems) $f : \{0,1\}^* \to \{0,1\}$ such that there exists a Turing machine T and a polynomial p such that for every $x \in \{0,1\}^*$, T computes $f(x)$ in at most $p(|x|)$ steps, where $|x| = m$ if $x \in \{0,1\}^m$.

Example 1.1. The problem st-CONN (input: directed graph G and two vertices s, t; output: 1 if there is a directed path from s to t) belongs to P.

Example 1.2. Input: positive integers m, n and output: mn. This is polynomial in the number of digits.

23 Jan 2024, Lecture 2

We now talk about NP, which stands for nondeterministic polynomial time.

Example 1.3. Input: a graph G with n vertices. Output: 1 iff G contains a Hamilton cycle (a cycle that contains every vertex).

Loosely, a nondeterministic algorithm is one that doesn't fully specify what it does. It outputs $f(x) = 1$ if there is some sequence of choices that leads to $f(x) = 1$. More formally, a **nondeterministic Turing machine** is one that has not one but two transition functions. At each step, it applies one or the other. We say that it computes f if $f(x) = 1 \iff$ there is some sequence of choices that leads to output 1 when the input is x.

Definition 1.2. NP is the class of functions computable in polynomial time by a nondeterministic Turing machine.

Definition 1.3 (Alternative definition)**.** $f \in \text{NP} \iff$ there is a polynomial p and a function $g \in \text{P}$ such that for every $x \in \{0,1\}^*$, $\exists y \in \{0,1\}^{p(|x|)}$ such that $g(x, y) = 1$.

To see that this is equivalent, observe first that if f satisfies the second definition, then we can write down y nondeterministically and apply g. In the other direction, y encodes choices made by the NDTM, so given y, the computation can be done deterministically.

Big open problem: does P = NP?

Definition 1.4 (co–NP)**.** $f \in \text{co–NP} \iff 1 - f \in \text{NP}$.

Alternatively, $f \in \text{co–NP} \iff$ there exists a polynomial p and $g \in P$ such that $\forall x \in \{0,1\}^*$, $f(x) = 1 \iff \forall y \in \{0,1\}^{p(|x|)}, g(x, y) = 1$.

For example, primality testing is in both NP and co–NP (for co–NP, we need to find a polynomial time algorithm that verifies a number is prime, see Ex. Sheet 1).

1.1 The polynomial hierarchy

Definition 1.5. Define Σ_0^P and Π_0^P to be P. If Σ_k^P and Π_k^P have been defined, then $f \in \Sigma_{k+1}^P \iff \exists$ a polynomial p and $g \in \Pi_k^P$ such that $f(x) = 1 \iff \exists y, g(x, y) = 1$.

Also $f \in \Pi_{k+1}^P \iff \exists$ a polynomial p and $g \in \Sigma_k^P$ such that $f(x) = 1 \iff \forall y, g(x, y) = 1$ (here $y \in \{0, 1\}^{p(x)}$).

Example 1.4. $f \in \Sigma_5^P \iff \exists h \in P$ such that

$$f(x) = 1 \iff \exists y_1 \, \forall y_2 \exists y_3 \, \forall y_4 \exists y_5, h(y_1, y_2, y_3, y_4, y_5) = 1.$$

We define PH (the polynomial hierarchy) as

$$\mathrm{PH} = \bigcup_{k=0}^{\infty} \Sigma_k^P \cup \Pi_k^P.$$

Proposition 1.1. If P = NP, then P = PH.

Proof. Note first that if P = NP, then P = co–NP. If $f \in \Sigma_{k+1}^P$, then $\exists g \in \Pi_k^P$ such that $f(x) = 1 \iff \exists y \; g(x, y) = 1$. By induction, $g \in P$, so $f \in$ NP, so $f \in$ P. The proof for Π_{k+1}^P is similar, just look at the negation. □

Exercises on Ex. Sheet 1:

- If NP = co–NP, then PH = NP = co–NP.
- If $\Sigma_k^P = \Sigma_{k+1}^P$ or $\Sigma_k^P = \Pi_k^P$, then PH = Σ_k^P.

PSPACE. This consists of functions that can be computed by a Turing machine that uses only a polynomial amount of tape.

Proposition 1.2. NP $\subset$ PSPACE.

Again a reminder: all proofs involving Turing machines are really more proof sketches.

Proof. If $f(x) = 1 \iff \exists y \; g(x, y) = 1$ for $g \in P$, then compute f by a brute–force search. All we need to remember is which y we've got to (in any sensible ordering) and whether we've found a value of y that works. Note importantly that we can reuse the space needed to compute each $g(x, y)$. □

Exercise on Ex. Sheet 1: PH $\subset$ PSPACE.

There are functions in PSPACE but not in PH. Good examples are games: "there exists a move I can play such that for any move you play there exists a move that I can play...". For example, take a game of Go with board size n going to infinity to get an example.

EXPTIME. This is the class of functions that can be computed in time $\exp(O(n^k))$ for some k.

25 Jan 2024, Lecture 3

Proposition 1.3. PSPACE $\subset$ EXPTIME.

Proof. Given a Turing machine in the middle of a computation, define its **configuration** to be its state, position on each tape, and the values in all the cells on the tapes. If T uses only a polynomial amount of space $p(n)$ for input of size n and has k tapes, then the number of possible configurations is at most $|S| \cdot p(n)^k \cdot |A|^{kp(n)}$, where A is our alphabet (usually $\{0,1\}$). Hence if the computation goes on for longer than the above amount of time, then the configuration repeats (by pigeonhole) and so is eventually periodic, so doesn't halt. □

NEXPTIME. $f \in$ NEXPTIME $\iff \exists g \in$ EXPTIME with $f(x) = 1 \iff \exists y\ g(x,y) = 1$.

EXPSPACE. This is the set of f that can be computed using at most $\exp(p(n))$ space for an input of size n.

So far we have

$$\text{P} \subset \text{NP} \subset \text{PSPACE} \subset \text{EXPTIME} \subset \text{NEXPTIME} \subset \text{EXPSPACE}.$$

1.2 Circuit complexity

A **circuit** is a directed acyclic graph (DAG) such that each vertex is labeled as an **input**, an AND gate, an OR gate, or a NOT gate. An input is a vertex of indegree 0, a NOT gate has to have indegree 1. All vertices of indegree >1 are AND gates or OR gates. Vertices of outdegree 0 are **outputs**. If the vertex preceeding a NOT gate has value x, then the vertex at the NOT gate has value $1 - x$. The value at an AND gate is the minimum of the values at its predecessors, and the value at an OR gate is the maximum of the values at its predecessors.

Using these rules, we get a well–defined function from $\{0,1\}^I$ to $\{0,1\}^O$, where I is the set of input vertices and O is the set of output vertices. If every AND and OR gate has indegree $\leq k$, then we say that the circuit is of fan–in $\leq k$. Often we restrict to circuits of fan–in ≤ 2.

Straight–line computations. Let $f : \{0,1\}^n \to \{0,1\}$. A straight–line computation of f is a sequence of functions $f_1, f_2, \ldots, f_m$ (we call m the **length** of the computation) such that $f_i(x) = x_i$ for $1 \leq i \leq n$, and for each $i > n$, either $f_i = f_{j_1} \wedge \ldots \wedge f_{j_k}$ for some $j_1, \ldots, j_k < i$, or $f_i = f_{j_1} \vee \ldots \vee f_{j_k}$ for some $j_1, \ldots, j_k < i$, or $f_i = 1 - f_j$ for some $j < i$, and $f_m = f$.

By considering a total ordering on the vertices of a DAG such that if $v_1 \to v_2$ in the DAG, then $v_1 < v_2$, we see that the smallest circuit that computes f is the smallest length of a straight–line computation that computes f.

Lemma 1.4. Every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size at most exponential in n.

Proof. Left as an exercise on Ex. Sheet 1. □

Proposition 1.5. Let f be a function that can be computed by a Turing machine T with k tapes in time $t(n)$ for inputs of size n. Then there is a family (C_n) of circuits such that $|C_n| = O(t(n)^{k+2})$ and C_n computes f for inputs of size n.

Proof. Let $S = \{s_1, \dots, s_r\}$ be the set of states of T and assume that the alphabet is $\{0,1\}$. Then we can encode the configuation of T at time t using $\sigma_1(t), \dots, \sigma_r(t), \pi_i^h(t)$ for $1 \le i \le t(n), 1 \le h \le k$ and $v_i^h(t)$, where $\sigma_i(t) = 1 \iff T$ is in state s_i at time t, $\pi_i^h(t) = 1 \iff$ the head is in position i on tape h at time t, and $v_i^h(t)$ is the value in cell i of tape h at time t.

Note that $\sigma_i(t) = 1 \iff$ there exist $j \le r$ and $i_1, \dots, i_k$ such that $\sigma_j(t-1) = 1$ and $\pi_{i_b}^h(t-1) = 1$ for $1 \le h \le k$ and $\tau(s_j, v_{i_1}^1(t-1), \dots, v_{i_k}^k(t-1))$ has state component equal to s_i (here τ is the transition function).

For any given $i_1, \dots, i_k$, we have a function of $k+1$ variables to evaluate, which we can do with a circuit of bounded size. Hence we can calculate $\sigma_i(t)$ from the previous configuration in time $O((k+1)t(n)^k) = O(t(n)^k)$. Similarly we can compute each position variable $\pi_i^k(t)$ from the configuration at time $t-1$ with a circuit of size $O(t(n)^k)$, and also each state variable $\sigma_j(t)$. So we can compute the configuration at time t from the configuration at time $t-1$ with a circuit of size $O(t(n)^{k+1})$, so the result follows. □

30 Jan 2024, Lecture 4

P/poly. This complexity class has three equivalent definitions. We say $f \in$ P/poly if one of the following (and hence all) holds:

(i) There is a family (C_n) of polynomial size circuits such that C_n computes $f(x)$ when $|x| = n$. (So trivially, from Proposition 1.5, P $\subset$ P/poly.)

(ii) There is a polynomial p and a sequence (y_n) with $|y_n| = p(n)$ and a function $g \in$ P such that $f(x) = 1 \iff g(x, y_{|x|}) = 1$. (Think of y_n as "a piece of advice we will give the algorithm").

(iii) There is a sequence (T_n) of Turing machines and a polynomial p such that T_n has $\le p(n)$ states and T_n computes $f(x)$ when $|x| = n$.

A sequence (C_n) of circuits is **P-uniform** if there is a polynomial time algorithm that generates it (i.e. given input $1^n := \underbrace{\{1, \dots, 1\}}_{n \text{ times}}$, it outputs C_n).

Sketch proof of equivalence. (i) $\implies$ (ii): Let Y_n be an encoding of C_n and let $g(x, y) = 1$ if the circuit encoded by y outputs 1 with input x.

(ii) $\implies$ (i): Fix an input size n, let C'_n compute g (using our proposition) on inputs of size n and let C_n be C with the last $p(n)$ inputs restricted to Y_n.

(ii) $\implies$ (iii): Fix n. Let T compute g and let T_n be a Turing machine that prints out Y_n and then uses T to compute $g(x, y_n)$.

(iii) $\implies$ (ii): Let y_n be an encoding of T_n and let $g(x, y) = 1$ if the Turing machine encoded by y_n outputs 1 with input x. □

2 Search problems and decision problems

Let g be a Boolean function of two variables. Then we get the decision problem "Does there exist y such that $g(x, y) = 1$?" and a corresponding **search problem** "If there exists y such that $g(x, y) = 1$, then find one.". A solution to a search problem is an algorithm that outputs y such that $g(x, y) = 1$ if it exists.

Proposition 2.1. If P = NP and $f \in$ NP and $f(x) = 1 \iff \exists y$ of size $|y| = p(|x|)$ with $g(x, y) = 1$ for $g \in$ P, then there is a polynomial time algorithm that computes $h : \{0, 1\}^* \to \{0, 1\}^*$ such that if $f(x) = 1$, then $g(x, h(x)) = 1$.

Proof. For each i, let g_i be the function that takes as input x and u_i with $|u_i| = i$, and outputs 1 if and only if $\exists v$ with $|v| = p(|x|) - i$ such that $g(x, u, v) = 1$. Now run the following procedure: start by calculating $g_1(x, 1)$ in polynomial time (since P = NP) and let $u_1 = g_1(x, 1)$. Note that if $\exists y$ such that $g(x, y) = 1$, then $\exists v$ such that $g_1(x, u_1, v) = 1$. Now let $u_2 = g_2(x, u_1), u_3 = g_3(x, u_2)$ and so on. At the end, we obtain $u = (u_1, \ldots, u_{p(|x|)})$ such that $g(x, u) = 1$. □

The idea of this proof is just to play 20 questions: can the first digit be 1? Can the second digit be 1? etc.

Lemma 2.2. If NP $\subset$ P/poly, then for f as in the previous proposition, there is a polynomial sized family of circuits (C_n) such that if $|x| = n$, then C_n with input x computes y such that $g(x, y) = 1$.

Proof. Use the notation from the previous proof. For each i, since NP $\subset$ P/poly, there is a polynomial sized circuit C'_i that computes g_i. Now put together the circuits $C'_1, \ldots, C'_{p(n)}$ as follows: C'_1 takes inputs $x_1, \ldots, x_n, 1$ and outputs u_1. C'_2 takes inputs $x_1, \ldots, x_n, u_1, 1$ and outputs u_2. C'_3 takes inputs $x_1, \ldots, x_n, u_1, u_2, 1$. Continue all the way until $C'_{p(n)}$. Then the output if $\exists y$ such that $g(x, y) = 1$ will be such a y. □

Theorem 2.3 (Karp–Lipton Theorem)**.** If NP $\subset$ P/poly, then $\Sigma_2^P = \pi_2^P$ (and therefore PH $= \Sigma_2^P = \Pi_2^P$).