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 $= \Sigma_k^P$.

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. $\square$

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 $\le k$, then we say that the circuit is of fan–in $\le k$. 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 $f_1, f_2, \dots, f_m$ (we call m the **length** of the computation) such that $f_i(x) = x_i$ for $1 \le i \le n$, and for each $i > n$, either $f_i = f_{j_1} \wedge \dots \wedge f_{j_k}$ for some $j_1, \dots, j_k < i$, or $f_i = f_{j_1} \vee \dots \vee f_{j_k}$ for some $j_1, \dots, j_k < i$, 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. □

1.3 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 1.6. 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 1.7. 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 1.8 (Karp–Lipton Theorem)**.** If NP $\subset$ P/poly, then $\Sigma_2^P = \pi_2^P$ (and therefore PH $= \Sigma_2^P = \Pi_2^P$).

01 Feb 2024, Lecture 5

Proof. Let $f \in \Pi_2^P$ and $h \in P$ be such that $f(x) = 1 \iff \forall y \; \exists z \; h(x, y, z) = 1$ (for y, z of appropriate polynomial size depending on $|x|$). Define $g(x, y)$ to be 1 if and only if $\exists z \; h(x, y, z) = 1$. Then $g \in$ NP, so by our hypothesis and the lemma above, there is a circuit family (C_n) of polynomial size such that for every x, if $|x| = n$ and $g(x, y) = 1$, then $h(x, y, C_n(x, y)) = 1$ (with $C_n(x, y)$ the output of C_n with input x, y). So $f(x) = 1 \implies \exists C_n \; \forall y \; h(x, y, C_n(x, y)) = 1$. Checking whether $h(x, y, C_n(x, y)) = 1$ can be done in polynomial time. If $f(x) = 0$, then $\exists y \; \forall z \; h(x, y, z) = 0$, so the $\implies$ is in fact an if and only if. Therefore $f \in \Sigma_2^P$. By doing the same for $1 - f$, we get the reverse inclusion. □

Lemma 1.9. For every k, there is a Boolean function f that can be computed by a circuit family of size n^{k+1}, but not by a circuit family of size n^k.

Proof. This is on Ex. Sheet 1. Note that Gowers is also pretty happy with a proof for n^{2k}, but n^{k+1} can be achieved. □

Theorem 1.10. For every k, there is a Boolean function $f \in \Sigma_4^P$ that cannot be computed by a family of circuits of size n^k.

Proof. For n sufficiently large, the lemma gives us $f'_n : \{0,1\}^n \to \{0,1\}$ that can be computed by a circuit of size n^{k+1}, but not by a circuit of size n^k. Choose a sensible ordering on circuits of size $\le n^{k+1}$. Let $f_n(x)$ (if $|x| = n$) be $C_n(x)$, where C_n is the first circuit (in this ordering) such that $|C_n| \le n^{k+1}$ and no circuit of size $\le n^k$ computes the same function as C_n. Then let $f = (f_n)_1^\infty$ (for large enough n, for $n < n_0$, let f_n be arbitrary).

Then if $|x| = n$, we have $f(x) = 1$ if and only if

$$\begin{gathered}\exists C_n, |C_n| \le n^{k+1} \text{ and } \forall D, |D| \le n^k, \exists y \; C_n(y) \ne D(y) \\ \text{and } \forall E, E \prec C_n, \exists F \; |F| \le n^k \; \forall z \; E(z) = F(z) \text{ and } C_n(x) = 1.\end{gathered}$$

(Here $E \prec C_n$ means E preceeds C_n in our ordering.) Then $\exists \; \forall\exists \; \forall$ shows that $f \in \Sigma_4^P$. □

Corollary 1.11. In fact, for every k there is a function $f \in \Sigma_2^P \cap \Pi_2^P$ that cannot be computed by a circuit family of size n^k.

Proof. By the Karp–Lipton theorem, if NP $\subset$ P/poly, then PH $\subset \Sigma_2^P \cap \Pi_2^P$, so the function just defined does the job.

If NP $\not\subset$ P/poly, then we get the stronger result that there is $f \in \Sigma_1^P$ (= NP) that can't be computed by any circuit family of polynomial size. □

The complexity class L. Roughly, L is the set of functions that can be computed with a logarithmic amount of memory. More formally, $f \in \mathrm{L}$ if there is a Turing machine that computes f with a read–only input tape that contains the input and cannot be modified and a work tape of size $O(\log n)$ for inputs of size n. (If we want several outputs instead of 1, then we also have a write–only output tape).

For example, during long multiplication in our head is in L, since we can do the computations digit–by–digit locally.

The complexity class NL. NL is like L, except that the Turing machine is nondeterministic. We can also give a certificate definition of NL, but one has to be careful. We say that $f \in \mathrm{NL}$ if there is a Turing machine with a read–only input tape, a work tape of size $O(\log n)$ and a read–once certificate tape (on which the head can only ever stay still or move to the right) such that $f(x) = 1$ if and only if there is some y with $|y| \le p(|x|)$ that can be put in the certificate tape such that the Turing machine outputs 1.

Proposition 1.12. $\mathrm{NL} \subset \mathrm{P}$.

06 Feb 2024, Lecture 6

Proof. Let $f \in \mathrm{NL}$ and let T be a NDTM that computes a function f and uses a logarithmic amount of space. The **configuration graph** of T is a directed graph, whose vertices are the configurations, and $\kappa \to \kappa'$ if and only if one of the transition functions takes κ to κ' in one step. Let G be the configuration graph of T. Then $f(x) = 1 \iff$ there is a directed path in G from the initial configuration to one such that T has halted with output 1. Since T uses $O(\log n)$ space, the number of vertices of G is polynomial in n. But REACHABILITY, the problem of determining whether there is a directed path from a vertex x to a subset S of vertices in a directed graph is easily seen to be in P. □

Remark. $|V(G)| \le 2^{O(\log n)}\,|S| = n^{O(1)}$, giving the polynomial bound in the above.

1.4 Low–depth computation

The class of NC^i (for $i \in \mathbb{Z}_{\ge 0}$) consists of all functions that can be computed by a family of circuits of polynomial size, fan–in 2 and depth $O((\log n)^i)$, where the **depth** of a circuit is the length of the longest directed path in the associated DAG.

AC^i is like NC^i, except that unbounded fan–in is allowed. It is an exercise on Ex. Sheet 1 to show that $\mathrm{AC}^i \subset \mathrm{NC}^{i+1}$. We write $\mathrm{NC} = \bigcup_{i=0}^{\infty} \mathrm{NC}^i, \mathrm{AC} = \bigcup_{i=0}^{\infty} \mathrm{AC}^i$ (which are of course equal, but both names come up in literature).

It is usual to impose a uniformity condition on NC^i. The favored condition is log space uniformity, i.e. $f \in \text{u–NC}^i$ if the circuits can be generated in log space (i.e. L).

Major open problem: is P $\subset$ NC? In fact, it is not known whether PH $\subset$ NC^1.

1.5 Randomized computation

A function f is in RP (randomized polynomial time) if there is a polynomial p and a function $g \in$ P such that if $|x| = n$ and $m = p(n)$, then

$$\mathbb{P}_{y\in\{0,1\}^m}\left(g(x,y)=1\right) = \begin{cases} 0 & f(x)=0 \\ \geq \frac{1}{2} & f(x)=1. \end{cases}$$

Note that if we run the computation on independent strings $y_1, \dots, y_k$, then $\mathbb{P}(\exists i\; g(x,y_i)=1) = \begin{cases} 0 & f(x)=0 \\ \geq 1-\frac{1}{2^k} & f(x)=1\dots \end{cases}$ So we can make the error probability very small without much cost.

We say $f \in$ co-RP if $1-f \in$ RP. We write ZPP = RP $\cap$ co-RP (ZPP stands for zero–error probabilistic polynomial time).

We say $f \in$ BPP (bounded–error probabilistic polynomial time) if there exists g as above such that $\mathbb{P}_{y\in\{0,1\}^m}(g(x,y)=f(x)) \geq \frac{2}{3}$. Again we can shrink the error probability. To do so, pick some k and calculate $g(x,y_1), \dots, g(x,y_k)$ for $(y_1,\dots,y_k)$ independent random elements of $\{0,1\}^m$ and take the majority. The probability of getting the wrong answer is at most $e^{-k/48}$ by Chernoff estimates for the sum of independent Bernoulli random variables.

Corollary 1.13. BPP $\subset$ P/poly.

Proof. If $k \geq 48n$. then the probability that the majority of $g(x,y_i)$ is wrong is $< 2^{-n}$. Therefore, there exist $y_1,\dots,y_k$ such that for every x, the majority vote is correct. This $y_1,\dots,y_k$ serves as an advice string, together with the function that computes the majority vote. □

Proposition 1.14. BPP $\subset \Sigma_2^P \cap \Pi_2^P$.

Proof. Let g be such that $\mathbb{P}(g(x,y)=f(x)) \geq \frac{2}{3}$. Then we have seen that we can find h such that $\mathbb{P}(h(x,y)=f(x)) \geq 1-2^{-n}$ (this y is $48n$ times as long as the previous y, call this new length m). For each x, let $A_x = \{y \in \{0,1\}^m \mid h(x,y)=f(x)\}$. □

08 Feb 2024, Lecture 7

Theorem 1.15 (Sipser, Lautemann)**.** BPP $\subset \Sigma_2^P \cap \Pi_2^P$.

Proof. If $f \in$ BPP, then there exists a polynomial p and $g \in$ P such that if $|x|=n$, then $\mathbb{P}_{y\in\{0,1\}^{p(n)}}\left(g(x,y)=f(x)\right) \geq 1-2^{-n}$ (by the "boosting" argument given earlier). For each x, let $A_x \subset \{0,1\}^{p(n)}$ be $\{y \mid g(x,y)=1\}$. Then if $f(x)=1$, A_x has density $\geq 1-2^{-n}$ and if $f(x)=0$, then A_x has density $\leq 2^{-n}$.

Suppose that $f(x) = 1$ and let $y_1, \dots, y_r$ be chosen independently and uniformly from $\{0,1\}^{p(n)}$. For each y, $\mathbb{P}\left(y \notin \bigcup_{i=1}^r A_x \oplus y_i\right) \leq 2^{-rn}$, where $A_x \oplus y_i$ is pointwise mod 2 addition. So if we choose $r > \frac{p(n)}{n}$, this probability is $< 2^{-p(n)}$, so $f(x) = 1 \implies \exists y_1, \dots, y_r$ such that $\bigcup_{i=1}^r A_x \oplus y_i = \{0,1\}^{p(n)}$.

Equivalently, $f(x) = 1 \implies \exists y_1, \dots, y_r \;\; \forall y \in \{0,1\}^{p(n)} \; \exists i \; g(x, y \oplus y_i) = 1$. But if $f(x) = 0$, then A_x has density $\leq 2^{-n}$ and r is polynomial in n, so it is (for sufficiently large n) not possible that $\bigcup_{i=1}^r A_x \oplus y_i = \{0,1\}^{p(n)}$, so the implication $\Longleftarrow$ is a double implication $\iff$. Therefore $f \in \Sigma_2^P$. By symmetry of BPP, $f \in \Pi_2^P$. $\square$

Major open problem: does BPP = P?

#P. This is a class of **counting problems**. A function $f : \{0,1\}^* \to \mathbb{Z}_{\geq 0}$ belongs to #P if there exists a polynomial p and $g \in P$ such that for every $x \in \{0,1\}^*$, $f(x) = \left|\{y \in \{0,1\}^{p(|x|)} \mid g(x,y) = 1\}\right|$. Note that if we can count solutions, we can detect whether solutions exist, so #P is at least as hard as NP.

Example 1.5. Input: a graph G. Output: the number of Hamilton cycles in G.

Example 1.6 (Important example.)**.** Input: a bipartite graph with vertex sets of the same size. Output: the number of perfect matchings.

If G is such a graph with two copies of $[n]$ as its vertex sets, and if $A_{ij} = \begin{cases} 1 & ij \in E(G), \\ 0 & \text{otherwise,} \end{cases}$ then the number of perfect matchings is

$$\sum_{\sigma \in S_n} \prod_{i=1}^n A_{i\sigma(i)}.$$

This is called the **permanent** of the $n \times n$ matrix A. It is the "determinant without signs". It is an example of a problem which is in P when we're considering it as a decision problem, but is very difficult when considering it as a counting problem in #P.

Algebraic complexity is about building up multivariable polynomials using $+$ and $\times$. An **arithmetic circuit** is a DAG where each vertex is either an input, a $+$-gate, or a $\times$-gate. The inputs are constants from some given field $\mathbb{F}$ and variables $x_1, \dots, x_n$. As with Booleans, we can talk instead about straight-line computations.

The **degree** of a circuit is defined inductively. The degree of an input vertex is 0 if it is a constant and 1 if it is a variable, the degree at a $+$-gate is the maximum of the degrees of its inputs and the degree of a $\times$-gate is the sum of the degrees of its inputs.

VP. This is the class of families of polynomials (p_n) such that p_n can be computed by an arithmetic circuit of polynomial size and polynomial degree.

Notice the condition of polynomial degree here – without this, x^{2^n} would be in our class since we could compute it in linear time.

Example 1.7. The determinant (of a matrix $(x_ij)_{i,j=1}^n$) is in VP. (The proof will be given later).

VNP. A polynomial family (f_n) is in VNP if there is a polynomial p and a polynomial family (g_n) in VP such that

$$f_n(x) = \sum_{y=(y_1,\dots,y_m)\in\{0,1\}^m} g_m(x,y).$$

for $m = p(\#$ of variables in $x)$.

Example 1.8. The permanent is in VNP. This is a good exercise to try before next time.

13 Feb 2024, Lecture 8

We must find a suitable way of representing the polynomial $\sum_{\pi\in S_n}\prod_{i=1}^n x_{i\pi(i)}$. Write this as $\sum_{y\in\{0,1\}^{[n]^2}}\prod_{i=1}^n(\sum_{j=1}^n x_{ij}y_{ij})$. For each i, let

$$r_i(y) = \sum_{j=1}^n y_{ij}\prod_{j'\neq j}(y_{ij} - y_{ij'})$$

and note $r_i(y) = 1$ if and only if there is exactly one j such that $y_{ij} = 1$. Let $r(y) = \prod_{i=1}^n r_i(y)$. Similarly define a polynomial $c_i(y)$ that takes the value 1 if and only if y has exactly 1 in each column. Then

$$\sum_{y\in\{0,1\}^{[n]^2}}\prod_{i=1}^n(\sum_{j=1}^n x_{ij}y_{ij})\cdot r(y)c(y) = \mathrm{perm}(x)$$

and the above formula gives a polynomial size and polynomial degree arithmetic circuit.

2 Completeness

For many complexity classes, there are problems in the classes that can be shown to be at least as hard as any other problem in the class.

2.1 NP–completeness

Let f and g be Boolean functions. A function $h : \{0,1\}^* \to \{0,1\}^*$ is a **polynomial-time reduction** from g to f if h can be computed in polynomial time and $g = f \circ h$.

Hence if $f \in P$ and g can be polynomial-time reduced to f, then $g \in P$. We sometimes write $g \prec_P f$ for this. Note that this relation $\prec_P$ is transitive.

f is said to be **NP-hard** is $g \prec_P f$ for every $g \in$ NP. f is **NP-complete** if f is NP-hard and $f \in$ NP.

Example 2.1 (Boring example)**.** Input: Turing machine T, some $x \in \{0,1\}^*$ and two sequences $1^m, 1^t$. Then $\phi(T, x, 1^m, 1^t) = 1 \iff$ there exists y with $|y| = m$ such that T halts on input (x, y) in time $\leq t$ and outputs 1.

Clearly $\phi \in$ NP. If $f \in$ NP and p, q are polynomials and g a function in P such that $f(x) = 1 \iff \exists y \in \{0,1\}^{p(|x|)}$ such that $g(x, y) = 1$, then let T, q be such that T computes $q(x, y)$ in time $\leq q(|x|)$. Then $f(x) = 1 \iff \phi(T, x, 1^{p(|x|)}, 1^{q(|x|)}) = 1$.

Call this problem **TSAT**. (This is nonstandard name!)

Example 2.2. CIRCUITSAT. Here the input is a circuit C with $n+m$ inputs and $x \in \{0,1\}^n$. The ouput is 1 $\iff \exists y \in \{0,1\}^n$ such that $C(x, y) = 1$.

We can reduct **TSAT** to **CIRCUITSAT** as follows: Given an instance $(T, x, 1^m, 1^t)$ of **TSAT**, build a circuit C_T that emulates T for t steps with inputs of size $|x| + m$. (If we look at the proof that P $\subset$ P/poly, we will see that such circuits exist and can be built by an algorithm in polynomial time.) Then $\phi(T, x, 1^m, 1^t) = 1 \iff \exists y$ such that $C_T(x, y) = 1$, completing the reduction.

For the next example, we need some definitions.

Definition 2.1.
- A **variable** is a symbol x that stands for an element of $\{0,1\}$ (or $\{\text{false}, \text{true}\}$).
- A **literal** is either x or $\neg x$ for some variable x.
- A **clause** is an expression of the form $u_1 \vee \ldots \vee u_k$ for $u_1, \ldots, u_k$ literals.
- A **formula** (in conjunctive normal form) is a conjunction of clauses, for example $x_1 \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee x_3)$.
- A formula ϕ is **satisfiable** if it is possible for to choose values for the variables such that ϕ evaluates to 1.

Example 2.3. SAT takes as input a formula x and outputs 1 if and only if it is satisfiable.

3SAT is the special case where each clause has size at least 3.

We can reduce **CIRCUITSAT** to **3SAT**. Assume fan-in 2 for convenience. First, let $v_1, v_2, \ldots, v_m$ be some straight-line computation. For each v_i, we take a variable, which we shall also call v_i.

- If $v_i = \neg v_j$ for some $j < i$, put in clauses $\neg v_i \vee \neg v_j$ and $v_i \vee v_j$.
- If $v_i = v_j \wedge v_k$ for some $j, k < i$, put in $\neg v_i \vee v_j$, $\neg v_i \vee v_k$, $\neg v_j \vee \neg v_k \vee v_i$.
- If $v_i = v_j \vee v_k$ for some $j, k < i$, put in $\neg v_j \vee v_i$, $\neg v_k \vee v_i$, $\neg v_i \vee v_j \vee v_k$.

So far, if $v_1, \ldots, v_m$ satisfy the formula, then they must take values corresponding to the straight line computation. Now add the clause v_m to force the output to be 1. Finally, given C and x, set the input corresponding to x to be those values (i.e. plug in the given values) and simplify the resulting formula to some ϕ_x which is satisfiable if and only if $\exists y$ such that $C(x, y) = 1$.