Computational Complexity Notes

Leart Ajvazaj

January 2024

Notes from my Computational Complexity class at Cambridge with Tim Gowers. Any mistake is with very high certainty mine.

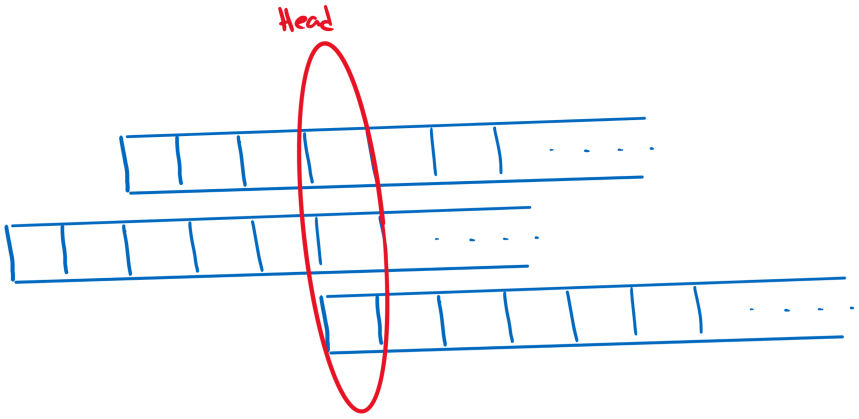
Computational Problems

Lecture 1

Example: Given a graph G with n vertices and $x, y \in V(G)$. Problem: is there a path from x to y?
A problem has variable input and output. If the output belongs to $\{0,1\}$ or $\{\text{no}, \text{yes}\}$ the problem is called a **decision problem**.
Write $\{0,1\}^*$ for the set $\bigcup_{n=1}^{\infty}\{0,1\}^n$, then a decision problem can be encoded as a **Boolean function**, that is, a function $f : \{0,1\}^* \to \{0,1\}$.
The set $\{x \in \{0,1\}^* : f(x) = 1\}$ is called the **language** defined by f.

Turing Machines

A **Turing machine** formalizes the notion of an algorithm. A **k-tape Turing machine** consists of: A collection of k tapes, where a tape is an infinite sequence of **cells**. There's also a finite set A called the **alphabet** and each cell contains an element of A. There's also a **head** which is in a **state** (an element of a finite set S of states) and in a **position** in each type. 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 according to the A^k component of the image; changes state according to the S component; shifts the tapes according to the $\{L, N, R\}^k$ component. One tape is designated as the input tape and doesn't change. Another is the output tape. All tapes except for the input tape start full of zeros.
If the machine reaches state S_{halt}, it stops. If the input is x and output y, we say that the machine computed y, given x.
Variants: Can assume that $A = \{0, 1\}$: that $k = 1$ (with a different convention: divide the tape such that "tape 1" is the places 1 (mod k), ... "tape k" is the places 0 (mod k).

Some Complexity Classes

The **complexity class** P consists of all Boolean functions $f : \{0,1\}^* \to \{0,1\}$ such that there exists a Turing machine T and polynomial p such that for any x, T computes $f(x)$ in at most $p(|x|)$ steps, where $|x| = n$ if $x \in \{0,1\}^n$.
Example: The problem st-CONN (input: directed graph G and two vertices, output: 0 or 1) belongs to P.
Another example: input: $m, n \in \mathbb{N}$ and output mn.

Lecture 2

NP - Nondeterministic Polynomial Time

Example: input: a graph G with n vertices. Output: 1 if and only if G contains a Hamilton cycle.
Loosely, a nondeterministic algorithm is one that doesn't fully specify what it does. H outputs $f(s) = 1$ if and only if there's 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 $f(x) = 1$ if and only if there is some sequence of choices that leads to output 1 when input is x.
NP is the class of functions computable in polynomial time by a nondeterministic Touring machine.
Alternative definition: $f \in NP$ if there's a polynomial p and a function $g \in P$ such that for

every $x \in \{0,1\}^*$, $f(x) = 1$ if and only if $\exists y \in \{0,1\}^{p(|x|)}$ such that $g(x,y) = 1$.
To see that this is equivalent, observe that if f satisfies the second definition, then we can write down y non-deterministically and apply g. In the other direction, g encodes the choices made by the nondeterministic Turing machine (NDTM), so given y, the computation can be done deterministically.
Big open problem: Does $P = NP$?

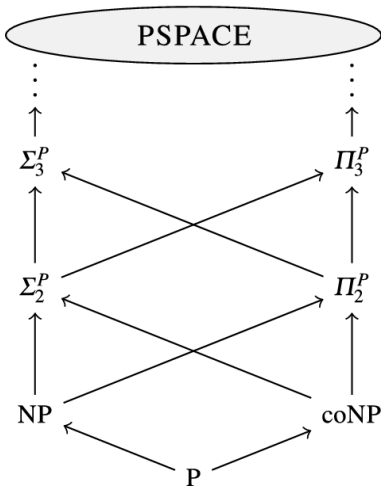
co-NP

$$f \in \textbf{co-NP} \iff \neg f \text{ or } 1 - f \in NP.$$

Alternatively, $f \in$ co-NP if and only if there exists a polynomial p and $g \in P$ such that for all $x \in \{0,1\}^*$ $f(x) = 1$ if and only if $\forall y \in \{0,1\}^{p(|x|)}$: $g(x,y) = 1$.

The Polynomial Hierarchy

Define $\sum_0^P$ and $\prod_0^P$ to be P. If $\sum_k^P$ and $\prod_k^P$ have been defined then $f \in \sum_{k+1}^P$ if and only if there exists a polynomial p and $g \in \prod_k^P$ such that $f(x) = 1$ if and only if $\exists y$ such that $g(x,y) = 1$. $f \in \prod_{k+1}^P$ if and only if there exists a polynomial p and $g \in \sum_k^P$ such that $f(x) = 1$ if and only if for all y: $g(x,y) = 1$. In both cases $y \in \{0,1\}^{p(|x|)}$.
Example: $f \in \sum_5^P$ if and only if $\exists h \in P$ such that $f(x) = 1$ if and only if $\exists y_1 : \forall y_2 \exists y_3 : \forall y_4 \exists y_5$: $h(x, y_1, \ldots, y_5) = 1$.
We define the **Polynomial Hierarchy** as $PH = \bigcup_{k=0}^{\infty} \sum_k^P \cup \prod_k^P$.



Proposition

If P=NP, then P=HP.

Proof. Note that if P=NP, then P=co-NP. If $f \in \sum_{k+1}^P$ then there exists $g \in \prod_k^P$ such that $f(x) = 1$ if and only if $\exists y$: $g(x,y) = 1$. By induction, $g \in P$ so $f \in$NP implies $f \in$P. The proof for $\prod_{k+1}^P$ is similar (or just look at the negation). □

Exercises on sheet: If NP=co-NP, then PH=NP=co-NP. If $\sum_k^P = \sum_{k+1}^P$ or $\sum_k^P = \prod_k^P$ then PH=$\sum_k^P$.

PSPACE

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

Proposition

NP⊂PSPACE.

Proof. Here's a sketch of the proof. If $f(x) = 1$ if and only if $\exists y$: $g(x, y) = 1$ where $g \in P$ then to compute f by a brute force search on y. All you need to remember is which y you have got to (in a sensible order) and whether you have formed y such that $g(x, y) = 1$. You can clear the space needed to compute each $g(x, y)$. □

Exercise on sheet: PH⊂PSPACE.

EXPTIME

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

Lecture 3

Proposition

PSPACE⊂EXPTIME.

Proof. Given a Turing machine in the middle of a computation define its **configuration** to be its state, its position on each tape, and the values in all the cells in 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 the alphabet. So if the computation goes on for longer than this, the configuration repeats (by pigeonhole) so it's eventually periodic, so doesn't halt. □

NEXPTIME

$$f \in \textbf{NEXPTIME} \iff \exists g \in \text{EXPTIME} : f(x) = 1 \iff \exists y : g(x, y) = 1.$$

EXPSPACE

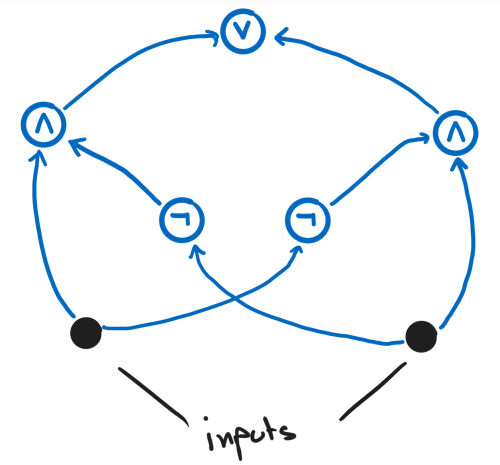
f is in **EXPSPACE** if f can be computed using at most $\exp(p(n))$ space for an input of size n.

To recap, so far we have

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

Circuit Complexity

A **circuit** is a directed acyclic graph (DAG) such that each vertex is labelled as an **input**, an **AND** gate, an **OR** gate, or a **NOT** gate. An input is a vertex of in-degree 0. A NOT gate has an in-degree 1. All vertices of in-degree > 1 are AND gates or OR gates. Vertices of out-degree 0 are outputs.



If the vertex preceding a NOT gate has value x then the vertex at the NOT gate has value $1 - x$.
The value at the AND gate is the minimum of the values of its predecessors.
The value at the OR gate is the maximum of the values at its predecessors.
Using these roles we get a well-defined function from $\{0,1\}^I$ to $\{0,1\}^O$ where I is the set of inputs and O the set of outputs.
If every AND gate and OR gate has in-degree $\leq k$ then we say that the circuit is of **fanin** $\leq k$. We often restrict to circuits of fanin ≤ 2.

Straight-Line Computations

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

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

Proposition

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's a family of circuits (C_n) such that $|C_n| = O(t(n)^{k+2})$ and C_n computes f for inputs of size n.

Proof. Let $S = \{s_1, \ldots, s_r\}$ be the set of states of T. (assume the alphabet is $\{0,1\}$). Then we can encode the configuration of T at time t using variables $\sigma_1(t), \ldots, \sigma_r(t), \pi_i^h(t)$ where $1 \leq i \leq t(n)$ and $1 \leq h \leq k$, $v_i^h(t)$ where $\sigma_i(t) = 1$ if and only if T is in state s_i at time t. $\pi_i^h(t) = 1$ if and only if the head is at position i on tape h at time t. $v_i^h(t) =$ value in cell i of tape h at time time t.
Note that $\sigma_i(t) = 1$ if and only if $\exists j \leq r,\ i_1, \ldots, i_k$ such that $\sigma_j(t-1) = 1$ and $\pi_{i_h}^h(t-1) = 1$ for $h = 1, \ldots, k$ and $\tau(s_j, v_{i_1}^1(t-1), \ldots, v_{i_k}^k(t-1))$ has state component equal to s_i.
For any given $i_1, \ldots, i_k$ we have a function of $k+1$ variables to evaluate which we can do with a circuit of bounded (in terms of k) size. Hence we can calculate $\sigma_i(t)$ from the previous configuration in time $O(t(n)^k)$.
Similarly, we can compute each position variable $\prod_i^h(t)$ from the configuration at time $t-1$ with a circuit of size $O(t(n)^k)$. 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. □

Lecture 4

P/poly

This complexity class has three equivalent definitions. $f \in$**P/poly** if one of the following holds:

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

(2) There is a polynomial p and a sequence (y_n) with $|y_n| = p(n)$ and a function $g \in P$ such that $f(x) = 1$ if and only if $g(x, y_{|x|}) = 1$.

(3) There is a sequence (T_n) of Turing machines and a polynomial p such that T_n has $\leq p(n)$ 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 $1^n = \underbrace{(1, \ldots, 1)}_{n-\text{times}}$, it outputs C_n).

Proof.
(1) $\Rightarrow$ (2): 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.
(2) $\Rightarrow$ (1): Fix an input size n. Let C'_n compute g and let C_n be C'_n with the last $p(n)$ inputs restricted to y_n.
(2) $\Rightarrow$ (3): 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 $g(x, y_n)$.
(3) $\Rightarrow$ (2): Let y_n be an encoding of T_n and let $g(x, y) = 1$ if the Turing machine encoded by y outputs 1 with input x. □

Search Problems & Decision Problems

Let g be a Boolean function of two variables. Then we get the decision problem "does there exists y such that $g(x, y) = 1$?" a corresponding **search problem** is "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

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

The way you essentially prove this is by playing a game of 20 questions. You want to ask "is it possible for the i-th digit to be 1?"

Proof. For each i, let g_i be the function that takes as input x and u_i where $|u_i| = i$ and outputs 1 if and only if $\exists v$, $|v| = p(x) - i$ such that $g(x, u, v) = 1$.
Now run the following procedure: start by calculating $g_1(x, 1)$ in polynomial time (sincere $P = NP$) and let $u_1 = g_1(x, 1)$. Note that if $\exists y$ such that $g(x, y) = 1$ then $\exists v$ such that $g_1(x, u, v) = 1$. 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 $g(x, y) = 1$. □

Lemma

If $NP \subset P/poly$ then for $f$as above, 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. 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.
Continue all the way to $C'_{p(n)}$

Then the output if there exists y such that $g(x,y)=1$ will be such a y □

Theorem : The Karp-Lipton Theorem

If $NP \subset P/poly$ then $\sum_2^P = \prod_2^P$ (and therefore $PH = \sum_2^P = \prod_2^P$)

Lecture 5

Proof. Let $f \in \prod_2^P$ and let $h \in P$ be such that $f(x)=1 \iff \forall\, y\ \exists\, z:\ h(x,y,z)=1$ (where y and z are of appropriate polynomial size depending on $|x|$). Define $g(x,y) = 1 \iff \exists z:\ h(x,y,z)=1$. Therefore $g \in NP$, so by our hypothesis and the previous lemma there exists a circuit family (C_n) of polynomial size such that for every x, if $|x|=n$ and $g(x,y)=1$, then $h(x,y,C_n(x,y))=1$ ($C_n(x,y)$=output of C_n with input x,y). So $f(x)=1$ implies that there exists C_n such that for all y we have $h(x,y,C_n(x,y))=1$. 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 the implies is actually an $\iff$. Hence $f \in \prod_2^P$. By doing the same for $1-f$ we get the reverse inclusion. □

Lemma

For every k there's 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. Exercise Sheet. □

Theorem

For every k there is a boolean function $f \in \prod_4^P$ 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 ($\prec$) on circuits of size $\leq n^{k+1}$. Let $f_n(x)$ (if $|x|=n$) be $C_n(x)$ where C_n is the first circuit (in this ordering) such that $|C_n| \leq n^{k+1}$ and no circuit of size $\leq n^k$ computes the same f_n as C_n. Then let $f=(f_n)_{n=1}^{\infty}$ (if $n<n_0$, f_n can be arbitrary).
Then if $|x|=n$, we have that $f(x)=1$ if and only if there exists C_n such that $|C_n| \leq n^{k+1}$ and for all circuits D with $|D| \leq n^k$ $\exists y$ such that $C_n(y) \neq D(y)$ and for all $E \prec C_n$ there exists F such that $|F| \leq n^k$ so that for all z, $E(z)=F(z)$ and $C_n(x)=1$. The $\exists\forall\exists\forall$ shows that $f \in \sum_4^P$. □

Corollary

In fact, for every k there is a function $f \in \sum_2^P \cap \prod_2^P$ that cannot be computed by a circuit family of size n^k.

Proof. By Karp-Lipton if $NP \subset P/poly$ then $PH \subset \sum_2^P \cap \prod_2^P$ so the function just defined does the job. If $NP \not\subset P/poly$ then we're done as we get the stronger result that there is $f \in \sum_1^P$ (=NP) that cannot be computed by any circuit family of polynomial size. □

L

Roughly **L** is functions that can be computed with a logarithmic amount of memory. More formally, $f \in L$ 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 you want several outputs, you have a write-only output tape (i.e. can't read it))

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 $f \in NL$ if there is a Turing machine with a read-only input tape, 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 $f(x) = 1 \iff \exists y\ (|y| \leq p(|x|))$ that can be put in the certificate tape such that the Turing machine outputs 1.

Proposition

$$NL \subset P$$

Lecture 6

We prove the fact that $NL \subset P$.

Proof. Let $f \in NL$, let T be a non-deterministic Turing machine that computes f in log space. Let G be the **configuration graph** of T. That is, the directed graph whose vertices are the configurations of T and C is joined to C' if and only if it is possible for T to go from C to C' in one step. Then $f(x) = 1$ if and only if there is a directed path in G from the initial configuration to 1 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. □

Low-depth Computation

The class NC^i ($i \in \mathbb{N}_0$) consists of all functions that can be computed by a family of circuits of polynomial size and fanin ≤ 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 fanin is allowed. We define $NC = \bigcup_{i=0}^{\infty} NC^i$ and $AC = \bigcup_{i=0}^{\infty} AC^i$. It's easy to see that $NC = AC$ since $AC^i \subset NC^{i+1}$. It's usual to impose a uniformity condition on NC^i. The favored condition is log space uniformity. i.e. $f \in u-NC^i$ if the circuits can be generated in log space.
Major open problem: Is $P \subset NC$? In fact, it is not known whether $PH \subset NC^1$.

Randomized Computations

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

$$\mathbb{P}_{y \in \{0,1\}^m}[g(x,y) = 1] \begin{cases} = 0 \text{ if } f(x) = 0 \\ \geq \frac{1}{2} \text{ if } f(x) = 1 \end{cases}$$

Note that if we run the computation on independent strings $y_1, \ldots, y_k$ then $\mathbb{P}[\exists i : g(x, y_i) = 1] \begin{cases} = 0 \text{ if } f(x) = 0 \\ \geq 1 - 2^{-k} \text{ if } f(x) = 1 \end{cases}$ So we can make the error probability very small without much cost.
$f \in$ **co-RP** if and only if $1 - f \in RP$.
ZPP (zero-error probabilistic polynomial time) $= RP \cap co - RP$.
$f \in$ **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 $g(x, y_1), \ldots, g(x, y_k)$ ($y_1, \ldots, y_k$ independent) 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

$$BPP \subset P/poly$$

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

Lecture 7

Theorem : Sipser-Lautemann

$$BPP \subset \sum_2^P \cap \prod_2^P.$$

Proof. If $f \in BPP$ then there exists a polynomial p and $g \in P$ such that if $|x| = n$, then $\mathbb{P}_{y \in \{0,1\}^{p(n)}}[g(x,y) = f(x)] \geq 1 - 2^{-n}$ (by the accuracy boosting argument). For each x, let $A_x \subset \{0,1\}^{p(n)}$ be $\{y : g(x,y) = 1\}$. 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 $y_1, \ldots, y_r$ 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 $\oplus$ is pointwise mod 2 addition.
So if we choose $r > \dfrac{p(n)}{n}$ this probability is less than 2^{-n}, so $f(x) = 1 \Rightarrow \exists y_1, \ldots, y_r$ such that $\bigcup_{i=1}^r A_x \oplus y_i = \{0,1\}^{p(n)}$.

Equivalently, $f(x) = 1 \Rightarrow \exists y_1, \ldots, y_r \; \forall y \in \{0,1\}^{p(n)} \overbrace{\exists i : \; g(x, y \oplus y_i) = 1}^{\text{in } P}$. But if $f(x) = 0 \Rightarrow A_x$ has density $\leq 2^{-n}$ and r is polynomial in n, so (for sufficiently large n) it's not possible that $\bigcup_{i=1}^r A_x \oplus y_i = \{0,1\}^{p(n)}$ so the $\Rightarrow$ is a $\iff$ and therefore $f \in \sum_2^P$. By symmetry of BPP (i.e. $f \in BPP \iff 1 - f \in BPP$), $f \in \prod_2^P$. □

Major open problem: Does $BPP = P$?

P

#P is a class for **counting problems**. A function $f : \{0,1\}^* \to \mathbb{N}_0$ belongs to $\#P$ if there exists a polynomial p and $g \in P$ such that for every $x \in \{0,1\}^*$, $f(x) = |\{y \in \{0,1\}^{(|x|)} : g(x,y) = 1\}|$. Note that if we can count solutions, then we can detect whether solutions exist, so $\#P$ is harder than NP.
Example: Input a graph G. Output the number of Hamilton cycles in G.
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.

Algebraic Complexity

Algebraic complexity is about building up multivariable polynomial 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 variable $x_1, \ldots, x_n$. As with Boolean circuits 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's a constant and 1 if it is a variable. The degree at a + gate is the maximum of degrees of its inputs. The degree at a $\times$ gate is the sum of the degrees of its inputs.

VP

VP is the class of families of polynomials such that p_n can be computed by an arithmetic circuit of polynomial size and polynomial degree.
Example: The determinant (of a matrix $(x_{ij})_{i,j=1}^n$) is in VP. (proof 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 $f_n(x) = \sum_{y=(y_1,\ldots,y_n)\in\{0,1\}^m} g_n(x,y)$ where $m = p(\#\text{variables in } x)$.

Lecture 8

Example: a permanent belongs to VNP.

Proof. We must find a suitable way of representing the polynomial $\sum_{\pi\in S_n}\prod_{i=1}^n x_{i\pi(i)}$. For each i, let $r_i(y) = \sum_{j=1}^n y_{ij}\pi_{j'\neq j}(y_{ij} - y_{ij'})$ and note that $r_i(y) = 1 \iff$ there's exactly one j such that $y_{ij} = 1$. Let $r(y) = \prod_{i=1}^n r_i(y)$. Similarly define a polynomial $c(y)$ that takes the value 1 iff y has exactly one 1 in each column. Then

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

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

2. Completeness

Lecture 8

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

NP-Completeness

Let f, 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 $g = f \circ h$. If $f \in P$ and g be polynomial-time reduced to f, then $g \in P$.
We sometimes write $g \prec_P f$. Note that the relation $\prec_P$ is transitive.
f is said to be **NP-hard** if $g \prec_P f$ for every $g \in NP$. f is **NP-complete** if f is NP-hard

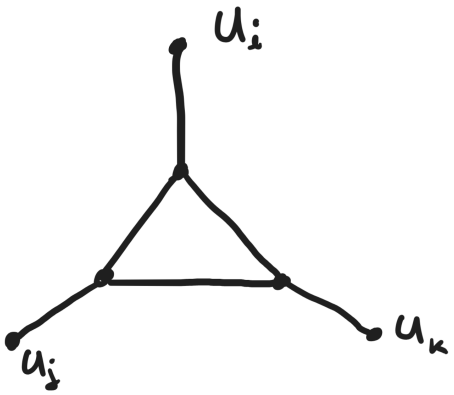
and $f \in NP$.
Example 1 (boring): The input is a Turing machine T, some $x \in \{0,1\}^*$ and two unary sequences 1^m and 1^t. Then $\phi(T, x, 1^m, 1^t) = 1$ if and only if there exists y with $|y| = m$ such that T halts on input (x, y) in time $\leq t$ and outputs 1.
Clearly $\phi \in NP$. If $f \in NP$ and p, q are polynomials and $g \in P$ 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 $g(x, y)$ in time $\leq q(|x|)$. Then $f(x) = 1 \iff \phi(T, x, 1^{p(|x|)}, 1^{q(|x|)}) = 1$. Call that problem **TSAT**.
Example 2 (CIRCUITSAT): Here the input is a circuit C with $n + m$ inputs and some $x \in \{0,1\}^n$. The output is 1 if and only if $\exists y \in \{0,1\}^n$ such that $C(x, y)$.
We can reduce TSAT to CIRCUITSAT. 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 $|x| + m$. 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. A **variable** is a symbol x that stands for an element of $\{0,1\}$. A **literal** is either x or $\neg x$ for some variable x. A **clause** is an expression of the form $u_1 \vee u_2 \vee \cdots \vee u_k$ where $u_1, \ldots, u_k$ are literals. A **formula** is a conjunction of clauses. Eg $x_1 \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee x_3)$. A formula ϕ is **satisfiable** if it is possible to choose variables such that ϕ evaluates to 1.
Example 3: **SAT** takes as input a formula and outputs 1 if and only if it is satisfiable.
3SAT is the special case where each clause has size 3.

Proof. We shall reduce CIRCUITSAT to 3SAT. First, let $v_1, \ldots, v_m$ be some straight-line computation.
Assume fanin≤ 2. 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 $v_i \vee v_j$ (we're doing $P \Rightarrow Q \iff P \vee \neg Q$). 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, x, set the inputs corresponding to x to be those values and simplify the formula to some ϕ_x which is satisfiable if and only if $\exists y$ such that $C(x, y) = 1$. □

Lecture 9

VERTEX COVER is the problem where the input is a graph G and a positive integer m and the output is 1 if and only if there is a subset $A \subset V(G)$, $|A| = m$ such that every edge has a vertex in A.
To see that this is NP-complete we shall reduce 3SAT to it. This we do with the help of "gadgets". Let ϕ be a formula in $x_1, \ldots, x_n$ of 3SAT (For convenience suppose that all clauses have size exactly 3). For each x_i, build a "variable gadget"

which is an edge with vertices labelled x_i and $\neg x_i$. For each clause $u_i \vee u_j \vee u_k$ we build a "clause gadget" as follows:



i.e. take a triangle and connect the three vertices to the vertices labelled u_i, u_j, u_k.
Observe that every vertex cover of the resulting graph has size at least $n + 2m$, where m is the number of clauses.
Claim that there is a vertex cover of size $n + 2m$ if and only if ϕ is satisfiable.
proof of claim: If ϕ is satisfiable, let $u_1, \ldots, u_n$ be a satisfying assignment and pick u_i from the i-th variable gadget and then for any clause gadget we have at least one of the outer edges covered, so can cover the rest with 2 vertices.
If ϕ is not satisfiable, then for any choice of $u_1, \ldots, u_n$ there will be some clause gadget with no outer edge covered so we'll need all three inner vertices. □

Theorem : (Ladner)

If $P \neq NP$, then there is a problem in NP that is neither in P nor NP-complete.

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be a function to be defined. Then let $g : \{0,1\}^* \to \{0,1\}$ be defined as follows: If $u \in \{0,1\}^*$ has the form $x01^{f(|x|)}$ and x encodes a satisfiable formula (in some sensible way), then $g(u) = 1$. Otherwise $g(u) = 0$.
We shall define functions $i : \mathbb{N} \to \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$ recursively as follows. Enumerate all Turing machines as $T_1, T_2, \ldots$ in such a way that each Turing machine occurs infinitely often. Also constrain T_i to halt within in^i steps. Observe that if T is any Turing machine that halts in polynomial time, then it agrees with some T_i.
If we have defined $i(1), \ldots, i(n-1), f(1), \ldots, f(n-1)$ then we set $i(n)$ to be $i(n-1)$ if $T_{i(n-1)}$ agrees with g on all inputs of size $\leq \log n$ and $i(n-1)+1$ otherwise. $f(n) = n^{i(n)}i(n)$. Note that this isn't a circular definition as we have enough values of f to compute g on inputs of size $\leq n$. Note also that f is polynomial-time in n computable since we can do a brute-force search.
Claim 1: $g \notin P$.
proof of claim 1: If $g \in P$, then g is computed by some T_i. But then $i(n)$ never goes above i. So $f(n) \leq in^i$ for all n but then the amount of padding is polynomial in $|x|$, so if we can compute g in polynomial time then $SAT \in P$ contradicting that $P \neq NP$.

Claim 2: g is not NP-complete.
proof of claim 2: If g is NP-complete then we have a polynomial time reduction of SAT to g. If the reduction is in time kn^k, then the reduction is to an instance u of g of size at most kn^k. But for n sufficiently large, $i(n)n^{i(n)} > kn^k$ which implies that the "x" part of u has size less than n. So in polynomial time we reduce an instance of SAT to a smaller instance of SAT, so $SAT \in P$, but $P \neq NP$. $\rightarrow\leftarrow$. □

Completeness in Space Classes

Let $s : \mathbb{N} \to \mathbb{N}$ then $DSPACE(s(n))$ is the class of functions computable by a Turing machine in space $O(s(n))$ and $NSPACE(s(n))$ is the class of functions computable by a non-deterministic Turing machine in space $O(s(n))$.

Theorem : (Savitch)

If $s(n)$ grows at least as fast as $\log n$then $NSPACE(s(n)) \subset DSPACE(s(n)^2)$.

Proof. Let $f \in NSPACE(s(n))$ and let T be a NDTM that computes f in space $s(n)$, and for any x let G_x be the configuration graph of T with input x. Note that $|V(G)| = 2^{O(s(n))}$ (since $s(n) \geq c \log n$). Then $f(x) = 1 \iff$ there is a path in G_x from the initial configuration to the halt with output 1 vertex

Lecture 10

Let $\sigma(k)$ be the amount of space needed to determine whether there is a path of length $\leq 2^k$ between two vertices of the configuration graph and let $g_k(u, v) = 1$ if such a path exists from u to v.
Then $g_k(u, v) = 1 \iff \exists w$ such that $g_{k-1}(u, w) = 1$ and $g_{k-1}(w, v) = 1$. Therefore, $\sigma(k) \leq O(s(n)) + \sigma(k - 1) + 1$ (reuse space!)
Also $\sigma(0) = O(s(n))$, so $\sigma(k) = O(ks(n))$. Choose $k = O(s(n))$ such that $2^k > |V(G_x)|$ and we get the bound stated. □

A PSPACE-complete problem

A **quantified Boolean formula** (QBF) is a statement of the form

$$Q_1 x_1, Q_2 x_2 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$$

where ϕ is built out of $x_1, \ldots, x_n$ using $\neg, \vee, \wedge$ and each Q_i is $\exists$ or $\forall$.
TQBF is the problem of determining whether a QBF is true.

Theorem

TQBF is PSPACE-complete.

Proof. First, let's show that TQBF is in PSPACE. Let $S(n,m)$ be the amount of space needed to solve TQBF for formulas of size m with n variables, where constants are allowed. Let P be the QBF $Q_1x_1 \dots Q_nx_n\phi(x_1,\dots,x_n)$ where ϕ has size m. Then if $Q_1 = \forall$ we get that P is true if and only if $P|_{x_1=0}$ and $P|_{x_1=1}$ is true.
The space needed to determine whether this is the case is at most $S(n-1,m)+1+O(m)$. Also $S(0,m) = O(m)$. Similarly, if $Q_1 = \exists$. So $S(n,m) = O(nm)$.
Now let $f \in PSPACE$. We want a polynomial-time reduction of f to TQBF. Let $x \in \{0,1\}^*$, let T be a Turing machine that computes f in polynomial space, and let G be the configuration graph of T with input x. Then $f(x) = 1$ if and only if there's a path in G from v_{init} to V_{accept}. Observe that $|V(G)| = 2^{O(p(n))}$ for some polynomial p.
For any k,u,v let $Q(k,u,v)$ be the statement that there is a path in G of length $\leq 2^k$ from u to v. Let $S(k)$ be the maximum size of a QBF that expresses $Q(k,u,v)$. Then

$$\begin{aligned} Q(k,u,v) &\iff \exists w : Q(k-1,u,w) \wedge Q(k-1,w,v) \\ &\iff \exists w\ \forall y,z\ ((y=u \wedge z=w) \vee (y=w \wedge z=v)) \Rightarrow Q(k-1,y,z). \end{aligned}$$

It follows that $S(k) \leq O(p(n)) + S(k-1)$. Picking k such that $2^k \geq |V(G)|$ we get $S(k) = O(p(n)^2) + S(0)$.
But $Q(0,u,v) = 1 \iff u = v$ or $uv \in E(G)$. But we can find in polynomial time a formula of size polynomial in n that determines whether one configuration is the successor of another - see proof of $P \subset P/poly$. □

0.0.1 NL-Completeness

A Boolean function $f : \{0,1\}^* \to \{0,1\}^*$ is **implicitly log space computable** if for every x and every i the functions $1_{\{i \leq |f(x)|\}}$ and $f(x)_i$ are in L (the latter only if $i \leq |f(x)|$).

Lemma

If f and g are implicitly log space computable, then so is $g \circ f$.

Lecture 11

Write ILC for implicitly logspace computable.

Lemma

If f and g are ILC, then so is $g \circ f$.

Proof. Look at recording 22/02/2024 □

Definition: Let $f,g : \{0,1\}^* \to \{0,1\}$. Then g is **logspace reducible** to f if there is an ILC function $h : \{0,1\}^* \to \{0,1\}^*$ such that $g = f \circ h$. f is **NL-hard** if every $g \in NL$ is logspace reducible to f. f is **NL-complete** if $f \in NL$ and f is NL-hard.
Notation: If g is logspace reducible to f, write $g \prec_L f$.

Lemma

(i) If $h \prec_L g$ and $g \prec_L f$, then $h \prec_L f$.

(ii) If f is NL-complete, $g \in NL$ and $f \prec_L g$, then g is NL-complete.

The proofs are obvious from the previous lemma.
Example of an NL-complete problem

Proposition

STCONN is NL-complete.

Proof. Let $f \in NL$. Let T be a NDTM that computes f. Let $x \in \{0,1\}^*$. Let G be the configuration graph of T with input x. Then G is easily seen to be implicitly logspace computable. But $f(x) = 1$ if and only if there is a directed graph in G from v_{init} to v_{accept}, so we have a logspace reduction of f to STCONN. □

P-completeness

Definition: A function $f \in P$ is **P-complete** if every $g \in P$ is logspace reducible to f.

Lemma

$NL \subset u - AC^1$.

Proof. Let $f \in NL$, $x \in \{0,1\}^*$, T a NDTM that computes f in logspace. Then examining the proof of $P \subset P/poly$, we see that there is a circuit of depth $O(1)$ that inputs a pair of configurations (κ, κ') and outputs 1 if and only if (κ, κ') belongs to the configuration graph. Moreover, this circuit is ILC. It follows that the adjacency matrix A of G is ILC. Also the number of configurations is $2^{O(\log n)}$, so polynomially bounded. Let $p(n)$ be a polynomial upper bound. Then $f(x) = 1 \iff$ there is a path of length $\leq p(n)$ from v_{init} to v_{accept}.
Given two 01 matrices A and B define their **Boolean product** AB by $(AB)_{ij} = \bigvee_k A_{ik} \wedge B_{kj}$. If A and B are square matrices with 1s on the diagonal then $(AB)_{ij} = 1 \iff$ there's a path from i to j with first edge in A and second in B.
Therefore $(A^r)_{ij} = 1$ (if A is a square matrix) if and only if there's a path of length $\leq r$ from i to j. It follows that we're done if we can find a depth $O(\log n)$ circuit for computing $A^{p(n)}$ (where A is the adj. matrix of G) But from definition of Boolean product there's a circuit of depth 2 for computing it so by repeated squaring we get a circuit of depth $O(\log n)$ for computing $A^{p(n)}$. It also has polynomial size. □

Corollary

If f is P-complete, then if $f \in u - NC$ then $P = NC$.

Proof. let $g \in P$. Then g is logspace reducible to f, so $\exists h$ ILC such that $g = f \circ h$. Since $L \subset u-AC^1 \subset u-NC^2$, h can be computed by a logspace uniform family of circuits of depth $O(\log(n)^2)$. Combining this with the circuits that compute f, we get that $g \in NC$. □

0.1 Lecture 12

CIRCUITEVAL is the problem which takes as input a circuit C with n inputs and $x \in \{0,1\}^n$ and outputs $C(x)$.

Proposition

CIRCUITEVAL is P-complete.

Proof. Obviously CIRCUITEVAL is in P. If $f \in P$, and f is computable by T in polynomial time, then there is an ILC family (C_n) of circuits that emulate T. So $f(x) = 1 \iff$ CIRCUITEVAL $(C_{|x|}, x) = 1$. So f is logspace reducible to CIRCUITEVAL. □

3. Less Obvious Relationships between Complexity Classes

Lecture 12

Say that $f : \mathbb{N} \to \mathbb{N}$ is **nice** if

(i) $f(n) \geq n$ for all n.

(ii) f is increasing.

(iii) The function $1^n \mapsto f(n)$ can be computed in time $O(f(n))$.

The next theorem is a weakening of a result called the time hierarchy theorem.

Theorem

There exists a polynomial p such that for every nice function f, there is a function ϕ that can be computed in time $p(f(n))$ such that if ψ is any function that can be computed in time $f(n)$, then there are infinitely many x such that $\phi(x) \neq \psi(x)$.

The proof uses diagonalization and is in some regards similar to the proof of the halting problem.

Proof. The basic idea is as follows: We define $\phi(x)$ to be $1 - T(x)$ if x encodes the Turing machine T and T halts in time $f(|x|)$. Otherwise let $\phi(x)$ be arbitrary.
Details: organize the encoding in such a way that if $s = t0^m$ and one of s, t encodes T, then so does the other. Suppose that T is some TM that halts in time $f(n)$. If t encodes T, then $\phi(t) = 1 - T(t)$, so ϕ disagrees with T on t. But t can be arbitrarily long, so ϕ and T disagree infinitely often.
It remains to prove that ϕ can be computed in time $p(f(n))$ for some (fixed) polynomial p. To compute ϕ, we simulate T using a universal TM and the encoding t. It's not hard to see that if T takes time m the simulation takes time $O(m^2)$. So we run the UTM for time $O(f(m)^2)$ until the simulated machine has taken $f(n)$ steps. That is enough to compute $\phi(x)$ when $|x| = n$. □

Let $A : \{0,1\}^* \to \{0,1\}$. An **oracle Turing machine with oracle A** is a TM with an extra **oracle tape**, and at any time it is allowed to "query the oracle" - i.e. if y is the content of the oracle tape, it can obtain the value of $A(y)$ - and this takes one step.
Given any uniform complexity class, the "A version" is what you get by replacing the TM by an OTM with oracle A.
Example:

- P^A =functions computable by an OTM with oracle A in polynomial time.
- NP^A =functions computable by an NDOTM with oracle A in polynomial time. You can also give the certificate version.

We shall now find oracles A, B such that $P^A = NP^A$ and $P^B \neq NP^B$.
Let $\text{EXPCOM}(T, x, 1^n) = T(x)$ if T computes $T(x)$ in time 2^n.
Claim: $P^{\text{EXPCOM}} = NP^{\text{EXPCOM}} = \text{EXPTIME}$.

Proof. Let $f \in \text{EXPTIME}$ and suppose that $f(x)$ can be computed in time $2^{p(|x|)}$ for each x by a TM T. Given x, create $(T, x, 1^{p(|x|)})$ on the oracle tape and query the oracle. This shows that $f \in P^{\text{EXPCOM}}$. Clearly $P^{\text{EXPCOM}} \subset NP^{\text{EXPCOM}}$ and $g \in P^{\text{EXPCOM}}$ and q a polynomial be such that $f(x) = 1 \iff \exists y : |y| = q(|x|)$ such that $g(x, y) = 1$. Each time g queries EXPCOM the oracle tape consists at most polynomially many bits, so we can carry out the computation in time $2^{\text{poly}(n)}$. At most polynomially many queries are made so $g \in \text{EXPTIME}$. But we can calculate $g(x, y)$ for every y by running $2^{q(n)}$ of these computations. So $f \in \text{EXPTIME}$. Hence $\text{NP}^{\text{EXPCOM}} \subset \text{EXPTIME}$. Putting everything together, $\text{EXPTIME} \subset \text{P}^{\text{EXPCOM}} \subset \text{NP}^{\text{EXPCOM}} \subset \text{EXPTIME}$. □

Proposition

There exists B such that $\text{P}^B \neq \text{NP}^B$.

Proof. Suppose that we've chosen B. Then define a function ϕ by $\phi(x) = 1 \iff \exists y$ such that $|y| = |x|$ and $B(y) = 1$. Then $\phi \in \text{NP}^B$.

Lecture 13

Now we shall find $n_1 < n_2 < \ldots$, sequences $x_1, x_2, \ldots$ with $|x_i| = n_i$, and $Y_1 \subset Y_2 \subset \ldots$ finite subsets of $\{0,1\}^*$ and at each stage we should ensure that B is defined on Y_i and (given an enumeration $T_1, T_2, \ldots$ of OTMs such that each one appears infinitely often) that if $T_i(x_i) = 1$ and T_i halts in time $\leq 2^{\frac{n_i}{10}}$ then $B(y) = 0$ for all y of size n_i, and if $T_i(x_i) = 0$ and T_i halts in time $\leq 2^{\frac{n_i}{10}}$ then $\exists y : \ |y| = n$, such that $B(y) = 1$.
Suppose we have done this up to $i-1$. Let n_i be bigger than every $|y|$ with $y \in Y_{i-1}$. Now pick x_i with $|x_i| = n_i$. Run T_i for $2^{\frac{n_i}{10}}$ steps. Each time it queries in B with some query y if $B(y)$ has been chosen (i.e. $y \in Y_{i-1}$) then carry on.
If $y \notin Y_{i-1}$ then set $B(y) = 0$. If T_i doesn't halt after $2^{\frac{n_i}{10}}$ steps, then we're done. If it does and outputs 1, set $B(y) = 0$ for every y with $|y| = n_i$ (This is consistent with our earlier choices). If it outputs 0, then pick some y for which $B(y)$ is not yet chosen (possible as $2^{\frac{n_i}{10}} < 2^{n_i}$) and set $B(y) = 1$.
So now if T is any OTM with oracle B it equals T_i for infinitely many i, so for infinitely many x_i, T either doesn't halt in time $2^{\frac{n_i}{10}}$ or gets $\phi(x_i)$ wrong. So $\phi \notin P^B$ □

The result above is known as the Baker-Gill-Solovay theorem.
A proof is said to **relativize** if it still works when TMs are replaced by OTMs for any given oracle. So no proof that $P \neq NP$ can relativize, but diagonal arguments do tend to relativize.
Here's a corollary of the weak time-hierarchy theorem that we forgot to cover earlier.

Corollary

P $\neq$ EXPTIME.

Proof. Apply the theorem with $f(n) = n^{\lfloor \log_2 n \rfloor}$ then we obtain ϕ computable in time $p(f(n))$ (so way subexponential) and if $\psi \in P$ then we can find ψ', computable in time $f(n)$ that agrees with ψ for all sufficiently large x. Then by the theorem ψ' disagrees with ϕ infinitely often, so $\psi \neq \phi$. □

Theorem : Immerman-Szelepcsenyi

NL = co-NL.

Proof. Since STCONN is NL-complete, it's enough to prove that STCONN $\in$ co-NL.
Claim 1: If we know how many vertices can be reached from s in $\leq k$ steps and x cannot be reached in k steps then there's a certificate that x cannot be reached.
proof of claim 1: Let the vertices of G be $y_1, \ldots, y_n$ in lexicographical order of how they're encoded. Suppose that m vertices are reachable in $\leq k$ steps. Then let $y_{i_1}, \ldots, y_{i_m}$ be the reachable vertices with $i_1 < i_2 < \cdots < i_m$. The certificate is a path from s to y_{i_1}, path from s to y_{i_2},... path from s to y_{i_m}. In a log amount of space we can check that these paths have

length $\leq k$, that $y_{i_1} \prec y_{i_2} \prec \cdots \prec y_{i_m}$ and that there are m vertices y_{i_j} and that no $y_{i_j} = x$.
Claim 2: If we know how many vertices are reachable with a path of length $k-1$ and x is not reachable in k steps, then there's a certificate of this.
proof of claim 2: use the certificate of claim 1 but with $k-1$ replacing k, and with "no $y_{i_j} = x$" replaced by "no y_{i_j} is joined to x by a directed edge".
Claim 3: For every k, m if exactly m vertices are reachable from s in $\leq k$ steps, then there is a certificate of this.
proof of claim 3: Induction on k. $k = 0$ is trivial. If we know how many vertices are reachable in $\leq k-1$ steps, and m vertices are reachable in $\leq k$ steps, then enumerate the vertices $y_1, \ldots, y_n$ in order and for each y_i give either a certificate that it is reachable in $\leq k$ steps (if it is) or a certificate that it isn't (if it isn't) using claim 2. We can use this in log space to count how many vertices are reachable in $\leq k$ steps. □

Remark: The proof can easily be generalized to larger amounts of space, e.g. to show that PSPACE = NPSPACE

Lecture 14

Characterizing (non-uniform) NC^1

Recall that a formula is defined inductively as follows: x_i is a formula of size 1, if ϕ is a formula of size m, then so is $\neg\phi$, if ϕ_1, ϕ_2 are formulas of sizes m_1, m_2 then $\phi_1 \vee \phi_2$ and $\phi_1 \wedge \phi_2$ are formulas of size $m_1 + m_2$. The **formula size** or **formula complexity** of a function $f : \{0,1\}^n \to \{0,1\}$ is the size of a smallest formula that computes f.

Proposition

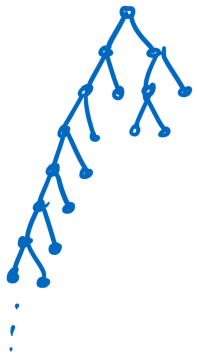
$f \in \mathrm{NC}^1 \iff f$ can be computed by a formula of depth $O(\log n)$ (and hence of polynomial size).

Proof. "$\Leftarrow$": is trivial.
"$\Rightarrow$": Suppose f can be computed by a fanin 2 circuit of depth d. Then $f = \neg g$ or $f_1 \wedge f_2$ or $f_1 \vee f_2$ for some functions g, f_1, f_2 that can be computed by circuits of depth $d-1$. So the result follows by induction. □

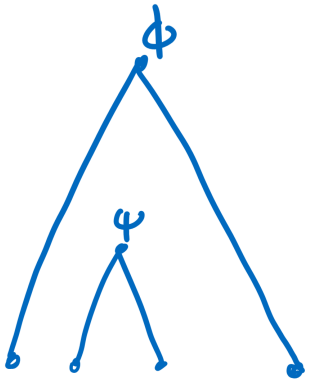
Lemma

f can be computed by a formula of polynomial size $\iff$ f can be computed by a formula of polynomial size and logarithmic depth.

Proof. Suppose that ϕ is a formula of size m that computes f, and that $m > 1$.



Starting at the top of the tree corresponding to ϕ walk down to a leaf, at each stage choosing the larger of the two subtrees (or in the case of $\neg$, the unique subtree) just below. Then the size of the subformula at each step drops by a factor of at most 2, so ϕ must have a subformula ψ of size between $\frac{m}{3}$ and $\frac{2m}{3}$.



Let ϕ_0 and ϕ_1 be the results of replacing ψ in ϕ by 0 and 1, respectively. Then ϕ computes the same function as $(\phi_0 \wedge \neg\psi) \vee (\phi_1 \wedge \psi)$.
So if $\delta(m)$ is the depth needed for a formula of size m, then $\delta(1) = 1$ and for $m > 1$, $\delta(m) \leq \delta(\frac{2m}{3}) + 2$. So $\delta(m)$" $\leq$ "$2\log_{3/2} m < 4\log_2 m$. □

Definition: A **branching program** of width k and length m in n variables $x_1, x_2, \ldots, x_n$ is a sequence $(i_t, p_t, q_t)_{t=1}^m$ where $i_t \in \{1, 2, \ldots, n\}$ and p_t, q_t are functions from $[k]$ to $[k]$. The **yield** of the program with input x is the function $r_m \circ r_{m-1} \circ \cdots \circ r_1$, where $r_t = \begin{cases} p_t & x_{i_t} = 0 \\ q_t & x_{i_t} = 1 \end{cases}$
If $F \subset [k]^{[k]}$, then we can define $f(x)$ to be 1 if and only if the yield belongs to F on input x. Then the branching program (with F) is said to compute f. If all p_t, q_t are permutations and $\alpha \in S_k$ is a non-identity permutation, we say that a branching program α-computes f if the yield is α when $f(x) = 1$ and id when $f(x) = 0$.

Theorem : Barrington

$f \in \mathrm{NC}^1$ if and only if f can be computed by a branching program of bounded width and polynomial length.

Proof. "$\Leftarrow$": an exercise on sheet 3.
"$\Rightarrow$": Let f be computable by a formula of depth d. We shall show that f can be computed by a branching program of width 5 and length $\leq 4^d$.
Let α and β be 5-cycles and suppose that f is α-computable. We can find some $\gamma \in S_n$ such that $\gamma\alpha\gamma^{-1} = \beta$. So if we replace p_1, q_1 by $p_1\gamma^{-1}$ and $q_1\gamma^{-1}$ and p_m, q_m by γp_m and γq_m, then the yield changes from α to $\gamma\alpha\gamma^{-1} = \beta$. Therefore f is α-computable $\iff$ f is β-computable.
Now let $f = \neg g$. Given a branching program that α-computes g, replace p_1, q_1 by $p_1\alpha^{-1}$, $q_1\alpha^{-1}$. This shows that f is α^{-1}-computable, so if α is a 5-cycle then f is α-computable by a branching program of the same length.
Next, suppose that $f = g \wedge h$ and that g, h are α-computable by branching programs of length $\leq m$. Pick a 5-cycle β such that $\alpha\beta\alpha^{-1}\beta^{-1}$ is a 5-cylce. (e.g. if $\alpha = (12345)$, then $\beta = (13542)$ works). Now g is α-computable and α^{-1}-computable and h is β-computable and β^{-1}-computable. Compose these branching programs: α-computable $g \circ \beta$-computable $h \circ \alpha^{-1}$-computable $g \circ \beta^{-1}$-computable h. If either $g(x) = 0$ or $h(x) = 0$, the yield will be $\beta\beta^{-1}$ or $\alpha\alpha^{-1}$ or id, i.e. id. If $g(x) = h(x) = 1$, the yield is $\alpha\beta\alpha^{-1}\beta^{-1}$, which is a 5-cycle, so f is α-computable by a program of length $\leq 4m$.
By writing $g \vee h$ as $\neg(\neg g \wedge \neg h)$ and combining the two steps above we get a similar conclusion when $f = g \vee h$. So the conclusion follows. □

4. Two Randomized Algorithms

Lecture 15

Polynomial Identity Testing

Input: Polynomials P, Q over a field $\mathbb{F}$.
Output: 1 if P and Q are the same polynomial.

Lemma : (Schwartz-Zippel)

Let P be a polynomial of total degree d in n variables $x_1, x_2, \ldots, x_n$ over a field $\mathbb{F}$ and let $S \subset \mathbb{F}$. Then the number of $s \in S^n$ such that $P(s) = 0$ is at most $d|S|^{n-1}$.

Proof. Ex Sheet 3. Induct on n. □

Now the algorithm is very simple. Given a polynomial of degree $d < |\mathbb{F}|$, choose $S \subset \mathbb{F}$ such that $d < |S|$ (if possible). Then randomly pick $s \in S^n$ and compute $P(s)$ and $Q(s)$ (we assume P, Q are presented in a way that makes this computation efficient). If $P(s) = Q(s)$

then output $P = Q$. Otherwise, output $P \neq Q$. If $P = Q$, the algorithm outputs $P = Q$ with probability 1. If $P \neq Q$ it outputs $P \neq Q$ with probability $1 - \frac{d}{|S|}$ by Schwartz-Zippel on $P - Q$. If $1 - \frac{d}{|S|} < \frac{1}{2}$, then repeat k times to boost to $1 - \left(\frac{d}{|S|}\right)^k$ choosing k appropriately such that $1 - \left(\frac{d}{|S|}\right)^k \geq \frac{1}{2}$.
This shows that polynomial identity testing is in co-RP.

Primality Testing

The problem takes input n and outputs 1 $\iff$ n is prime. We shall show that this is in co-RP.

Lemma

The polynomials $(X+1)^n$ and $X^n + 1$ are equal (as polynomials) mod n if and only if n is prime.

Proof. If n is prime, then $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is a multiple of n. If n is composite, pick a prime p such that $p \mid n$. Then $\frac{1}{n}\binom{n}{p} = \frac{(n-1)\cdots(n-p+1)}{p!}$ which isn't an integer since p doesn't divide the numerator. □

We can't get anywhere by testing for identity of $(X+1)^n$ and $X^n + 1$. Example: if n is a Carmichael number ($\forall a$, $a^n \equiv a \pmod{n}$ and n isn't prime) then $(X+1)^n$ and $X^n + 1$ take the same values mod n.
So instead we do this:

- Pick a random monic polynomial Q of degree d ($\sim 2\log n$) with coefficients in $\mathbb{Z}/n\mathbb{Z}$.
- Calculate $(X+1)^n$ and $X^n + 1$ in the ring $\frac{\mathbb{Z}/n\mathbb{Z}[x]}{Q(x)}$
- If they're equal, output n is prime. Else, output n is composite.

The second step asks if $(X+1)^n$ and $X^n + 1$ are congruent mod n,$Q(x)$.
We can test this by repeated squaring in time $O(\log n)O(d^2)$, which is polynomial in $\log n$.
If n is prime, then $(X+1)^n$ and $X^n + 1$ are the same mod n, so the same mod n,$Q(x)$ for all Q. So the algorithm outputs "n is prime".
To deal with the case of composite n, we need a lemma.

Lemma

Let p be a prime and let d be a positive integer. Then over $\mathbb{F}_p$, $X^{p^d} - X$ is the product of all irreducible polynomials of degree dividing d.

Corollary

The number of monic irreducible polynomials of degree d over $\mathbb{F}_p$ is $(1 - o(1))\frac{p^d}{d}$.

Proof. Let $\phi(k)$ be the number of monic irreducible polynomials of degree k. Then by the lemma, $p^d = \sum_{k|d} k\phi(k)$. So by Mobius inversion,

$$d\phi(d) = \sum_{k|d} \mu(k) p^{\frac{d}{k}}.$$

The leading term is p^d and remaining terms at most $p^{\frac{d}{2}}$ so the result follows. □

Suppose that n is composite and let p be a prime that divides n. If $\mathbb{P}$[algo outputs "n is composite"] $\leq \frac{1}{2d}$, then the probability that $Q(x)$ is irreducible and $Q(x) \mid (X+1)^n - X^n - 1$ in $\mathbb{F}_p[X]$ is at least $\frac{1}{2d} - o(1)$. But any two distinct monic irreducible polynomials are coprime, so this shows that $d(\frac{1}{2d} - o(1))p^d \leq n$. This is a contradiction if $d \geq 2\log n$. So if n is composite and $d \geq 2\log n$, then $\mathbb{P}$[alg outputs "n is composite"] $\geq \frac{1}{2d} - o(1)$. By repeating $O(d)$ times we can boost this to $\frac{1}{2}$.