[CSS.203.1]: Computational Complexity (2025-I)

Summary scribes

git info: f4c15c4 , (2025-07-28 11:28:10 +0530)

Lectures

A dummy's guide

Theorem 1. *This is a cool theorem*

In fact we can refer to theorems using Theorem 1.

You could also define lemma, corollary etc. (take a look at `thmmacros.tex` for the environments).

Other useful macros are present in `lazy_macros.tex` and `common_macros.tex`. You can add more to them if required.

One pet-peeve: There are many times when people have to define a function called 'blah'. There are multiple ways of doing this:

- (worst) `$blah$` which renders as $blah$
- (bad) `$\text{blah}$` which renders as blah
- (better, but not ideal) `$\mathrm{blah}$` which renders as blah
- (right) `$\operatorname{blah}$` which renders as blah

Here is a place where you can see the difference between these:

$blah\theta$ $\text{blah}\theta$ $\mathrm{blah}\theta$ $\operatorname{blah}\theta$

And the same within an emphasised text

$blah\theta$ *$\text{blah}\theta$* *$\mathrm{blah}\theta$* *$\operatorname{blah}\theta$*

Here is a general guide for deciding which to use:

> If you just want to use text within a math block, then use `\text`. For examples such as defining a set called 'PRIMES' (which are not used as functions or operators), you may use `\mathrm{PRIMES}`. If you are defining functions or operators, then use `\operatorname{blah}` as it adds the right spacing around it.
>
> Using `$blah$` should only be used when you are multiplying four variables called b, l, a and h.

Lecture 1

Introduction to the course

Scribe: Aindrila Rakshit

Topics covered in this lecture

1. Introduction to Computational Complexity
2. Examples of problems and Reduction
3. Automata

1.1 Introduction to Computational Complexity

Computational Complexity is the study of understanding the resource constraint of a computational model when it comes to solving a task. So given some objects that we wish to study, we look at the procedural way of computing them, i.e. their computational models and the resources required to do so.

Σ^* : a string of arbitrary finite length with elements of the alphabet Σ.
E.g.- $\{0,1\}^*$: binary string of some arbitrary finite length

Some examples of Objects, Computational Models, and Resources:

- For Boolean functions (i.e., $f : \{0,1\}^n \to \{0,1\}$), the computational model could be circuits, or Turing machines, or python programs. The resource we care about could be time taken, memory used, number of API calls etc.
- For polynomials (i.e, $f(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$), the computational model could be algebraic circuits / formulas. The resource we care about could be the number of basic operations, depth, etc.

Broadly speaking, Complexity theory deals with classifying 'objects' based on 'resource required' by certain 'computational models'. Here 'classifying' means creating a gradation of complexity (hardness) among objects/tasks.

1.2 Examples of problems and Reduction

We will mostly be dealing with computational tasks that have a yes/no answer (or 0/1).

1. Graph reachability (undirected / directed): Given a graph G, and two vertices s and t, check if there is a path from s and t.
2. Perfect matching: Given a graph G on an even number of vertices, check if there is a perfect matching in the graph.
3. Primality: Given a number N, check if it is prime.
4. Linear programming. Given a matrix A and a vector b, check if there is a vector x satisfying $Ax \leq b$ (component-wise) with every entry of x being non-negative.
5. Integer programming. Given a matrix A and a vector b, check if there is a vector x satisfying $Ax \leq b$ (component-wise) with every entry of x being a non-negative integer.
6. Vertex cover: Given a graph G and a number k, check if G has a vertex cover of size at most k.
7. Chess: Given a chess position (on a generalised $n \times n$ chess board), check if white has a winning strategy.

It seems that Problem 1 is the easiest and Problem 7 is the hardest. In fact, the problems seem to be gradually increasing in hardness with some problems having similar levels of complexity. Our aim in this course would be to quantify these notions of complexity.

Definition. *For an alphabet Σ, a language is just a subset of Σ^*. Equivalently, one could consider the 'membership function' $f : \Sigma^* \to \{0,1\}$ such that $f(x) = 1$ if and only if $x \in L$.*

Given an arbitrary function $f : \Sigma^* \to \{0,1\}$, we may sometimes use L_f to denote the language associated with it. (i.e., $L_f = \{x \in \Sigma^* : f(x) = 1\}$) $\Diamond$

1.3 Automata

Automata are primitive computational models.

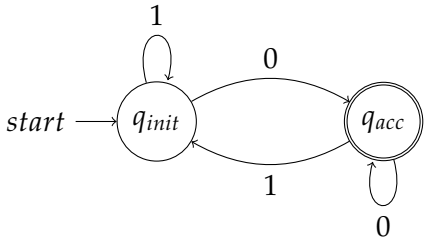


Figure 1.1: Example of an automata

An automata is specified by providing a set of states Q, a specified start state q_{init}, a set of accepting states $F \subseteq Q$, and a transition function:

$$\Gamma : Q \times \Sigma \to Q$$

that specifies what state to move to after consuming one letter in the given string. If at the end of consuming the entire string the automata is in any fo the states in F, the automata is said to have *accepted* the string (and *rejected* otherwise).

Lecture 2

Introduction to Turing Machines

Scribe: Nishant Das

2.1 Deterministic Turing Machine

A Deterministic Turing Machine (DTM) is essentially a Deterministic Finite Automata (DFA) with a piece of (infinitely long) paper. It is defined by

1. **Tape alphabet Σ:** The tape alphabet is an extension of the input alphabet with some extra symbols such as start ($\triangleright$), stop (#) and blank space ($\sqcup$). We will abuse the notation and represent both the input and the tape alphabet as Σ.

2. **Work Tapes:** The work tapes are like scratch sheets for the DTM. It can read and write on these tapes to store information that might be helpful later.

3. **Transition function:** The transition function for a DFA tells it which state to go to as a function of it's current state and the input symbol. DTM also has the following additional freedom:

   (a) change the values of the tapes at the position the DTM is currently reading.

   (b) A DFA only has an input tape and after reading a symbol, it transitions to its next state and reads the next symbol. We can picturize this as a DFA having a "head" which moves right one step forcefully after reading a symbol. A DTM has a head for each of the tapes, and it can keep the head at place or move it left or right by a step.

   Hence, the Transition Function, denoted by Γ, is a function of the following kind

$$\Gamma : Q \times \Sigma^{k+1} \to Q \times \Sigma^{k+1} \times \{-1, 0, 1\}^{k+1}$$

   Here, Q denotes the state space of the DTM and the set $\{-1, 0, 1\}$ denotes the commands "move head left", "stay at place" and "move head right".

4. **Start, Accept and Reject States**. A unique element of Q is the start state. It should have at least one accept state. A DTM can also have some reject states, which is essentially like a sink in a DFA.

2.1.1 Halting TMs

A Turing machine (TM) M is said to be a **halting TM** if for every input x in Σ^*, M *halts* (accepts or rejects) on x. Henceforth, we will only work with halting TMs.

Given a TM M and an input x, does M halt on x? This is known as the *halting problem* and it is *undecidable*.

2.1.2 Time Complexity of a TM

Let M be a TM. Consider the function $T_M : \mathcal{N} \to \mathcal{N}$ |

$$T_M(n) = \max_{x \in \Sigma^*, |x|=n} \{\text{number of steps for } M \text{ to halt on } x\}.$$

This function is called the **time complexity** of M.

2.2 Alphabet and Tape Reduction

Whenever we are trying to build a TM, we can work with as many tapes and as many symbols as we like. The following two theorems guarantee us that there is an equivalent TM with only 2 tapes and 5 letters. Moreover, their proofs are constructive, providing an explicit method for constructing such an equivalent machine.

1. **Alphabet Reduction:** Suppose M is a halting TM with alphabet Σ and time complexity T_M. Then, $\exists$ an equivalent TM $\tilde{M}$ with alphabet $\tilde{\Sigma} = \{\triangleright, \#, \sqcup, 0, 1\}$ such that $T_{\tilde{M}} = \mathcal{O}(T_M)$.

2. **Tape Reduction:** Suppose M is a halting TM with k tapes. Then we can build an equivalent TM $\tilde{M}$ with 2 tapes such that $T_M = \mathcal{O}(T_M \log T_M)$.

2.2.1 Proof Sketches

The main idea to prove alphabet reduction is to encode all letters in the alphabet as strings of 0s and 1s. This introduces a factor of at most $\log_2 |\Sigma|$ in the computation time. Since the new machine operates on a smaller alphabet, to simulate each step of the original machine, it may have to read or write multiple bits sequentially. This can be achieved by introducing additional states and a few special symbols.

For the proof of tape reduction, the key idea is to interweave all the working tapes into a single tape in a structured manner. Perhaps the first idea that comes to mind is to store all k tapes on a single tape by interleaving them into fixed-size blocks, where each block contains one symbol from each tape in sequence. However, a slightly more efficient way is to stick all

of the working tapes end-to-end into a single working tape and introduce new symbols to the alphabet to mark the beginning of each working tape. This gives us a TM that runs in $\mathcal{O}(T_M^2)$, but a more clever way of interweaving exists which improves this bound to $\mathcal{O}(T_M \ln T_M)$.

2.3 Universal Turing Machine

Let $\langle M \rangle$ denote some string encoding of the TM M. This can be interpreted as the "code" of M. Note that M has infinitely many string encodings. A **Universal Turing Machine**, U, is a machine that simulates any Turing machine M given its encoding $\langle M \rangle$. It takes $(\langle M \rangle, x, 1^t)$ as input and accepts the input if M accepts x in at most t steps and rejects otherwise. To simulate M, U needs to store and look up $\langle M \rangle$. The time taken by U on $(\langle M \rangle, x, 1^t) \leq \mathcal{O}_{|M|}(t \log t)$ where the symbol $\mathcal{O}_{|M|}(f)$ means "$\mathcal{O}(f)$ upto constants depending on M".

Remark. *An important point (that would be useful later on in the course) is that the the constant in the above bound depends on the* machine *M and not the encoding. In particular, if we are given a* really long *encoding of the same machine M, the constant does not change (i.e., it doesn't scale if the encoding of the machine is made longer as long as we are dealing with the same machine).* $\Diamond$

Lecture 3

Non-determinism

Scribe: Soumyadeep Paul

3.1 Nondeterministic Finite Automata

Definition 3.1 (NFA)**.** *A Nondeterministic Finite Automaton (NFA) is a tuple of the form* $(Q, \Sigma, \delta, s, F)$*, where* Q *is a finite set of states,* Σ *is a finite set of alphabets,* $\delta : Q \times \Sigma \to 2^Q$ *is the transition function,* s *is the start state and* F *is the set of final states.*

A word is accepted by an NFA if a final state can be reached via valid transitions, on reading the word. ◊

The difference from deterministic automata is that now, on reading a symbol the automaton can "guess" and move to any of the states in its transition function.

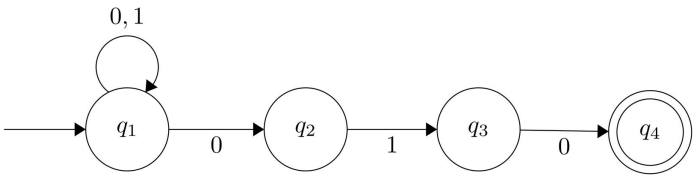


Figure 3.1: An NFA that accepts all words ending with 010.

Definition 3.2 (ε-transition)**.** *In an NFA with ε-transition, there are some transitions labelled with* ε *and the automaton is allowed to take this transition without reading a symbol.* ◊

3.2 Regular Expressions

Definition 3.3. *A regular expression is a pattern of symbols from* $\Sigma \cup \{\varepsilon, +, \cdot, *\}$*. Where* Σ *is a finite set of symbols,* $+$ *stands for "or",* $\cdot$ *is for concatenation and* * *is Kleene star.* ◊

Example 3.4. $\{0+1\}^*01\{0+1\}^*$ *is the regular expression for the language of all words which contains* 01 *as a substring.* ◊

It is known, for every regular expression there is an NFA that accepts the same language as the regular expression. This enables us to check if a given word satisfies a regular expression, in $O(n^2)$ time, by simulating run of that word on the corresponding automaton.

3.3 Nondeterministic Turing Machine

Definition 3.5 (NTM)**.** *A Nondeterministic Turing Machine is the same as a deterministic Turing machine, except it now has two transition functions* Γ_0, Γ_1.

A word is x accepted if there exists a sequence of transitions that results in q_{accept} *state.* ◊

Definition 3.6 (Halting TM)**.** *A Turing machine is said to be halting if for every input and every computational path, the TM accepts/rejects in finite steps.* ◊

We only consider halting Turing machines from hereon. We define the time taken by a Turing machine M,

$$T_M(x) := \max_{\text{all computational paths } P} T_M(x, P)$$

$$T_M(n) := \max_{|x|=n} T_M(x)$$

For any halting machine, the above definitions are well-defined as we are guaranteed to halt on every computational path.

Definition 3.7 (Circuit-SAT)**.** Circuit-SAT $= \{C \mid C \text{ is a boolean circuit and } \exists x \text{ such that } C(x) = 1\}$. ◊

Definition 3.8 (Circuit-Eval)**.** Circuit-Eval $= \{(C, x) \mid C \text{ is a boolean circuit and } C(x) = 1\}$. ◊

We notice that there is an efficient algorithm, for Circuit-Eval, by simply simulating x on C. Now, we can get an NTM for $CictuitSAT$ that runs in polynomial time by "guessing" the input for the circuit and then running Circuit-Eval.

Definition 3.9 (DTIME)**.** *For a function* $t(n) : \mathbb{N} \to \mathbb{N}$*, we will denote by* $\mathsf{DTIME}(t(n))$ *the class of languages that can be computed by deterministic TMs in* $O(t(n))$ *time. Formally,*

$$\mathsf{DTIME}(t(n)) := \left\{ L \ : \begin{array}{c} \text{there is a deterministic halting TM } M \text{ with} \\ L(M) = L \text{ that runs in time } O(t(n)) \end{array} \right\}$$

◊

Definition 3.10 (NTIME)**.** *For a function* $t(n) : \mathbb{N} \to \mathbb{N}$*, we will denote by* $\mathsf{NTIME}(t(n))$ *the class of languages that can be computed by non-deterministic TMs in* $O(t(n))$ *time. Formally,*

$$\mathsf{NTIME}(t(n)) := \left\{ L \ : \begin{array}{c} \text{there is a non-deterministic halting TM } M \text{ with} \\ L(M) = L \text{ that runs in time } O(t(n)) \end{array} \right\}$$

◊

Therefore, Circuit-SAT $\in \mathsf{NTIME}(\text{poly}(n))$.

Definition 3.11.

$$\mathsf{P} := \bigcup_{c \geq 1} \mathsf{DTIME}(n^c)$$
$$\mathsf{NP} := \bigcup_{c \geq 1} \mathsf{NTIME}(n^c)$$

$\Diamond$

Definition 3.12 (Co-nondeterministic TM)**.** *A* co-nondeterministic Turing machine *is a machine with the same syntax as a non-deterministic TM (i.e., has two transition functions* Γ_0 *and* Γ_1*), but with the acceptance criteria modified. A word* x *is said to be acccepted by the TM if and only if* every *computational path accepts.* $\Diamond$

We can similarly define coNTIME and coNP by replacing non-deterministic TMs with co-nondeterministic TMs. Furthermore, it can be easily seen that

$$L \in \mathsf{NTIME}(t(n)) \Leftrightarrow \overline{L} \in \mathsf{coNTIME}(t(n)).$$

Lecture 4

Many-One Reductions

Scribe: Soham Chatterjee

We will mainly focus on polynomial time reductions and the one reduction which we will use most of the time is Many-One Reduction

Definition 4.1 (Many-One Reduction)**.** *$L_1, L_2 \subseteq \Sigma^*$ are two languages. Then L_1 is reducible to L_2 under many-one reduction if and only if $\exists\ f : \Sigma^* \to \Sigma^*$ such that $\forall\ x \in \Sigma^*$*

$$x \in L_1 \iff f(x) \in L_2$$

If f is polynomial time computable function then we say L_1 is reducible to L_2 under polynomial time many-one reduction and denote it as $L_1 \leq_m^{\text{poly}}$. ◊

4.1 Vertex Cover ≡ Independent Set ≡ Clique

4.1.1 Vertex Cover $\leq_m^{\text{poly}}$ Ind Set $\leq_m^{\text{poly}}$ Vertex Cover

Want: $(G, k) \mapsto (H, k')$

Observation 4.2. *For any $S \subseteq V$ vertex cover in $G \iff \overline{S} = V \setminus S$ is independent set in G*

So $f : \Sigma^* \to \Sigma^*$ will be $f(G, k) = (G, n - k)$.

Remark 4.3. *Here f is a bijection between independent sets and vertex cover. That may not be the case always while constructing the reductions.* ◊

Therefore we actually obtain both Vertex Cover $\leq_m^{\text{poly}}$ Ind Set and Ind Set $\leq_m^{\text{poly}}$ Vertex Cover. So

$$\text{Vertex Cover} \equiv_m^{\text{poly}} \text{IndSet}$$

4.1.2 Ind Set $\leq_m^{\text{poly}}$ Clique $\leq_m^{\text{poly}}$ Ind Set

Want: $(G, k) \mapsto (H, k')$

Observation 4.4. *For any $S \subseteq V$, S is independent set of $G \iff S$ is a clique in $\overline{G} = (V, \overline{E})$.*

So $f : \Sigma^* \to \Sigma^*$ will be $f(G,k) = (\overline{G},k)$. Here also notice that f is a bijection between independent sets and cliques. Therefore we obtain Ind Set $\leq_m^{\text{poly}}$ Clique and Clique $\leq_m^{\text{poly}}$ Ind Set. Hence

$$\text{IndSet} \equiv_m^{\text{poly}} \text{Clique}$$

4.2 Circuit-SAT $\equiv$ 3 CNF-Sat

We already have 3 CNF-Sat $\leq_m^{\text{poly}}$ Circuit-SAT by the identity function.

The idea is to convert the function of each gate into a clause and then take a AND over all of them to ensure all the gates are working as they are supposed to.

So from a given circuit C of size s whose input variables are $x_1, \ldots, x_n$ we will construct a 3CNF with $n+s$ variables and $s+1$ constraints. The variables will be

$$\{x_i \mid i \in [n]\} \cup \{y_i \mid i \in [s]\}$$

For each gate g in C, y_g is the variable which corresponds to g. We will design the constraints in the following way:

- $g = g_1 \vee g_2$ or $g = g_1 \wedge g_2$: Then $y_g = y_{g_1} \vee y_{g_2}$ or $y_g = y_{g_1} \wedge y_{g_2}$ respectively.
- $g = x_i \vee x_j$ or $g = x_i \wedge x_j$: Then $y = x_i \vee x_j$ or $y_g = x_i \wedge x_j$ respectively.
- y_{root}

We do this for all gates and take a big $\bigwedge$ over all these constraints. Each of the equality can be converted to a constant size 3 CNF-Sat formula by comparing the answers from the truth table. This conversion of the C to the 3 CNF-Sat formula φ is polynomial time doable. Therefore Circuit-SAT $\leq_m^{\text{poly}}$ 3 CNF-Sat

4.3 NP-hardness

Lemma 4.5. *Suppose f is a polynomial time many-one reduction from A to B where $A, B \subseteq \Sigma^*$ and $B \in \mathsf{NP}$ ($B \in \mathsf{coNP}$). Then $A \in \mathsf{NP}$ ($A \in \mathsf{coNP}$).*

Proof. $B \in \mathsf{NP} \implies \exists$ nondeterministic machine M_B for B. Since f is polynomial time computable function $\exists$ a polynomial time deterministic turing machine M_f computing f. Then construct M_A which on input x first runs M_f on x and then runs M_B on output of M_f. So

$$M_A = M_B \circ M_f$$

Therefore M_A is an nondeterministic turing machine whose language is $A \in \mathsf{NP}$. Similar proof works for $A \in \mathsf{coNP}$. □

Definition 4.6 (NP-hardness and completeness)**.** *A language $L \subseteq \Sigma^*$ is said to be NP-hard if for any language $L' \in \mathsf{NP}$ there is a polynomial time many-one reduction $f_{L'} : \Sigma^* \to \Sigma^*$ from L' to L i.e. $x \in L' \iff f_{L'}(x) \in L$*

The language L is said to be NP-complete if it is NP-hard and also in NP. ◊

The same definition extends to any complexity class and you can similarly define coNP-completeness etc.

Lecture 5

Cook-Levin Theorem and NP-Completeness

Scribe: Shubham A. Bhardwaj

Remark. *In this lecture, Reduction will always mean "Polynomial Time-Many One" Reduction.* ◊

We defined Many-One Reductions and the notion of NP-Complete problems in the last lecture. Circuit-SAT was the first problem to be proven NP-Complete, independently by Stephen Cook and Leonid Levin in 1970s. This result is known as Cook-Levin Theorem.

5.1 Cook-Levin Theorem

Theorem 5.1. Circuit-SAT *is* NP*-Complete.*

Proof. To prove that Circuit-SAT is NP-Complete, we need to prove that there exist reductions from all languages in NP to Circuit-SAT. Let L be some language in NP. We need to show that *there exist a polynomial time computable function f, which on input x, will output a circuit C_x such that $x \in L$ if and only if $C_x \in$* Circuit-SAT.

To accomplish the above, we somehow need to simulate the NTM in our circuit. Observe that at any point of time, the configuration of a TM can be completely characterised by following 3 things - the contents of tape, current state of the TM, and the location of head. Now, let M be the NTM for language L that runs in cn^k time. Let $R_i(x)$ be the configuration of M after i steps. Now, $x \in L$ if and only if there exist sequence of configurations $R_1(x), R_2(x), \ldots, R_{cn^k}(x)$ such that $R_{cn^k}(x)$ is an accepting configuration and $\forall i \leq cn^k$, $R_i(x)$ follows from $R_i(x)$. All these configurations can be arranged in a $cn^k \times cn^k$ table that we call "computational tableau". Thus, our circuit just need to check whether such a tableau exist or not.

Now, observe that a maximum of 3 entries change from one row to the next of a tableau and these 3 entries are consecutive, thus we can have a sliding window of size 2×3 and only need to check the entries in the window are valid or not. For this, our circuit can have a constant number of clause for each position of sliding window to check whether the entries in window are valid or not. As the tableau is of size $cn^k \times cn^k$, there are a total of $\mathcal{O}(n^{2k})$ position for the

sliding window. Thus our circuit will have $\mathcal{O}(n^{2k})$ clauses with each clause having a constant no of literals (since the window is of constant size). Check the tableau first row coorespond to input x and a valid configuration and the last row of tableau correspond to an accepting configuration can be done in a constant number of clauses of size $\mathcal{O}(n^k)$. Thus, the circuit needed to simulate NTM is of polynomial size. Thus given the description of machine M, we can build the TM N that outputs C_x on input x which runs in polynomial time. This proves there exist reductions from all languages $L \in \mathsf{NP}$ to Circuit-SAT.

□

5.2 More NP-Complete Problems

It turns out that not just Circuit-SAT, but a whole bunch of problem are NP-Complete. And having already proved that Circuit-SAT being NP-Complete, To prove that some language $L \in \mathsf{NP}$ is NP-Complete, we just need to give a reduction from Circuit-SAT to L.

We saw an simple reduction from 3-CNF-Sat to Ind-Set proving that Ind-Set is NP-Complete. We also defined the language Int-Program as follows:

$$\text{Int-Program} = \{(A, b) | A \in \{0,1\}^{n \times n}, b \in \{0,1\}^n \text{ and } \exists x \in \{0,1\}^n \text{ such that } Ax \leq b\}$$

There exist a simple reduction from 3-CNF-Sat to Int-Program. For each clause in the 3-CNF, we can write an corresponding linear inequality. For example, for the clause, $x_1 \vee \neg x_2 \vee x_3$, the corresponding linear inequality will be $x_1 + (1 - x_2) + x_3 \geq 1$. This way we'll get a system of linear inequality which will have a feasible solution if and only if the formula is satisfied. This shows that Int-Program is NP-Complete.

Lecture 6

Time Hierarchy Theorem

Scribe: Vivek Karunakaran

6.1 Deterministic Time Hierarchy Theorem

Theorem 6.1 (Deterministic time hierarchy theorem)**.** *If $f, g : \mathbb{N} \to \mathbb{N}$ are nice functions (time-constructible) satisfying $f(n) \log f(n) = o(g(n))$, then*

$$\mathsf{DTIME}(f) \subsetneq \mathsf{DTIME}(g).$$

Proof. We prove this for the particular case when $f(n) = n^3$ and $g(n) = n^4$ for the notational convenience and the general argument follows the same. Note that $\mathsf{DTIME}(n^3) \subset \mathsf{DTIME}(n^4)$. In order to prove that they are not equal, We construct a Turing machine whose Language belongs to $\mathsf{DTIME}(n^5)$ but does not belong to $\mathsf{DTIME}(n^3)$.

Let the Turing machine that we construct be D. On an input input x,the machine D does the following:

1. If the input x is not of the form "$\langle M \rangle \#0^t$" where t is an arbitrary integer and $\langle M \rangle$ is a valid machine description, it rejects x.
2. We may now assume that $x = \langle M \rangle \#0^t$; let its length be n.
3. Create a counter with value n^4 on a worktape.
4. Using the UTM for at most n^4 steps to simulate the machine M on input x (do one step of simulation, decrement counter etc.). If the simulation did not complete within n^4 steps of the UTM, then reject x. Else,
   - If M accepted x, then the machine D rejects x.
   - If M rejected x, then the machine D accepts x.

Let the language defined by this Turing machine D be L_D. Clearly $L_D \in \mathsf{DTIME}(n^4)$ since the simulation ends in n^4 steps and the other steps are negligible. Suppose $L_D \in \mathsf{DTIME}(n^3)$.

Then, There exists a Turing machine M which runs in $O(n^3)$ time and $M(x) = D(x), \forall x$. For the UTM to completely simulate M on any input x of length n, the UTM would require at most $cn^3 \log n$ for some constant c (that could depend on some parameters of M such as number of tapes, alphabet size etc.). Consider $x = \langle M \rangle \#0^t$ with t chosen large enough so that $n^4 > cn^3 \log n$ for $n = |x|$. This always possible since $n^3 \log n = o(n^4)$.

Now, note that D accepts x if and only if M rejects x (since we have ensured that n^4 steps is sufficient to simulate M on x completely). Thus, clearly $L(M) \neq L(D)$ □

Remark. *This method of proving is called the Diagonalization Technique. Consider the tableau where the turing machines corresponding to* $\mathsf{DTIME}(f(n))$ *are listed out in the rows and their corresponding string representation in the columns. Each cell represents the result when the corresponding machine is run on the corresponding string. So, Each row represents the language accepted by the corresponding turing machine. Now, If we form a language by flipping all the values along the diagonal, then There cannot be any machine in this list which can accept this language. We construct a machine that accepts this language, which is also in* $\mathsf{DTIME}(g(n))$*, thus proving the time hierarchy. The outline for the Non-deterministic Time Hierarchy theorem was given in this lecture. However, The details are deferred to the next.* ◊

Lecture 7

Non-deterministic time hierarchy theorem and Oracle Turing Machines

Scribe: Aindrila Rakshit

Topics covered in this lecture

1. Non-deterministic time hierarchy theorem
2. Oracle Turing Machines
3. The Baker-Gill-Solovay theorem

7.1 Time hierarchy theorems

Definition 7.1 (Time-constructible functions)**.** *A function $f : \mathbb{N} \to \mathbb{N}$ is said to be time-constructible if g can be computed by a deterministic Turing machine in time $O(f(n))$ on input 1^n.* $\Diamond$

Theorem 7.2 (Deterministic time hierarchy theorem)**.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ be two time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$. Then, $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$.*

Theorem 7.3 (Non-deterministic time hierarchy theorem)**.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ be time-constructible functions with $f(n) = o(g(n))$. Then, $\mathsf{NTIME}(f(n)) \subsetneq \mathsf{NTIME}(g(n))$.*

We present the following proof of the non-deterministic time hierarchy theorem which is due to Lance Fortnow and Rahul Santhanam.

Proof. For simplicity let us assume that $f(n) = n^3$ and $g(n) = n^4$. Consider the description of M via a non-deterministic Turing Machine D computing it.

> All strings in the language $L(D)$ are of the form $(\langle M \rangle, x, p)$ for input x and computational path p with the following constraints: let $n = |\langle M \rangle| + |x|$
>
> - $\langle M \rangle$ represents the encoding of a non-deterministic *TM*.

- If $|p| < n^4$, then we accept the input $(\langle M \rangle, x, p)$ if and only if M accepts both $(\langle M \rangle, x, p0)$ and $(\langle M \rangle, x, p1)$.
- If $|p| \geq n^4$, then we accept the input $(\langle M \rangle, x, p)$ if and only if M on input $(\langle M \rangle, x, \varepsilon)$ rejects on the computational path p.

Claim 7.4. $L(D) \in \mathsf{NTIME}(n^4)$.

Proof. If $|p| < n^4$, we only need to simulate M on two strings $(\langle M \rangle, x, p0)$ and $(\langle M \rangle, x, p1)$. Using non-determinism, we can guess computational paths for both subproblems and accept only if both those simulations resulted in an accept. The overall running time is at most $O(n^4)$.

If $|p| \geq n^4$, then we simulate M deterministically which can be done in $O(f(n))$ time. □

Claim 7.5. $L(D) \notin \mathsf{NTIME}(n^3)$.

Proof. Suppose on the contrary that there was a non-deterministic machine M with running time $O(n^3)$ satisfying $L(M) = L(D)$. Consider the input $\left(\langle M \rangle, 0^k, \varepsilon\right)$

$$
\begin{aligned}
M \text{ accepts } \left(\langle M \rangle, 0^k, \varepsilon\right) &\Leftrightarrow \left(\langle M \rangle, 1^\ell, \varepsilon\right) \in L(D) \\
&\Leftrightarrow M \text{ accepts } \left(\langle M \rangle, 0^k, 0\right) \text{ and } \left(\langle M \rangle, 0^k, 1\right) \\
&\Leftrightarrow \left(\langle M \rangle, 0^k, 0\right), \left(\langle M \rangle, 0^k, 1\right) \in L(D) \\
&\Leftrightarrow M \text{ accepts } \left(\langle M \rangle, 0^k, 00\right), \ldots, \left(\langle M \rangle, 0^k, 11\right) \\
&\quad \vdots \\
&\Leftrightarrow M \text{ accepts } \left(\langle M \rangle, 0^k, p\right) \text{ for all } p \in \Sigma^{n^3} \\
&\Leftrightarrow \left(\langle M \rangle, 0^k, \varepsilon\right) \notin L(D) \\
&\Leftrightarrow M \text{ rejects } \left(\langle M \rangle, 0^k, \varepsilon\right)
\end{aligned}
$$

□

yielding the required contradiction. □

7.2 Oracle Turing Machines

This is a formalization of the option of TMs that have access to a subroutine for membership in a specific language. Now, the TM has a special "oracle tape", and three special states $q^{\text{oracle}}_{\text{query}}, q^{\text{oracle}}_{\text{yes}}, q^{\text{oracle}}_{\text{no}}$, along with the ususal input tape and work tape. Whenever we look at M^A, where A is some language, we will have the following functionality — when the TM enters the state $q^{\text{oracle}}_{\text{query}}$, the machine is moved to the state $q^{\text{oracle}}_{\text{yes}}$ if the string currently on the oracle tape is in A, and moved to the state $q^{\text{oracle}}_{\text{no}}$ if the string is *not* in A.

Definition 7.6 ($\mathsf{DTIME}^A(f(n))$:)**.** $\mathsf{DTIME}^A(f(n)) = L(M^A)$: *A is a deterministic Oracle TM running in $O(f(n))$ time.* ◊

Similarly define $\mathsf{NTIME}^A(f(n))$ etc.

So, time hierarchy theorem holds with oracles, i.e. for an arbitrary language A, $\mathsf{DTIME}^A(f(n)) \subsetneq \mathsf{DTIME}^A(g(n))$, for $f, g : \mathbb{N} \to \mathbb{N}$ time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$ and the proof is almost the same, since the technique of diagonalisation continues to work even in this setting as all we needed for our proof was some way encoding the TMs (or oracle TMs), and having some way of simulating such a machine on an input (which in case of oracle TMs, the simulator has access to A and can make the necessary oracle queries). Therefore, the time hierarcy theorems continue to hold for any language A. That is, the technique of diagonalisation is insensitive to the presence of oracles.

7.3 The Baker-Gill-Solovay theorem

The fact that the technique is insensitive to the presence of oracles also points out the weakness of this technique. The following theorem by Baker, Gill and Solovay shows that the " P vs NP " is not oblivious to the presence of oracles.

Theorem 7.7 (Baker, Gill and Solovay)**.** *There are two languages $A, B \subseteq \Sigma^*$ such that*

1. $\mathsf{P}^A = \mathsf{NP}^A$,
2. $\mathsf{P}^B \neq \mathsf{NP}^B$

So, the diagonalization technique alone cannot help us prove a result, which is sensitive to the presence of oracles.

Proof. Take A as an EXP-complete language. We will show that $\mathsf{P}^A = \mathsf{NP}^A$.

$\mathsf{P}^A \subseteq \mathsf{NP}^A$: Trivial, since every deterministic machine is also a non-deterministic machine.

$\mathsf{NP}^A \subseteq \mathsf{P}^A$: We will show that $\mathsf{EXP} \subseteq \mathsf{P}^A$ and $\mathsf{NP}^A \subseteq \mathsf{EXP}$.

Claim 7.8. *Fix any non-deterministic oracle TM M. Then, $L(M^A) \in \mathsf{EXP}$.*

$\mathsf{NP}^A \subseteq \mathsf{EXP}$: On each poly length path, whenever NP^A machine makes a query, EXP solves A, on all possible paths.

$\mathsf{EXP} \subseteq \mathsf{P}^A$: : Since A is EXP-complete, there is a poly-time reduction f such that $x \in \mathsf{EXP}$ iff $f(x) \in A$.

Thus, $\mathsf{P}^A = \mathsf{NP}^A$. □

Lecture 8

Oracle machines, and Baker-Gill-Solovay Theorem

Scribe: Nishant Das

In this lecture, the proof of Baker-Gill-Solovay was completed and the complexity classes P^{NP} and $\mathsf{NP}^{\mathsf{NP}}$ were introduced and analyzed.

8.1 Baker-Gill-Solovay continued

Theorem 8.1 (Baker-Gill-Solovay)**.** *There are two languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$ but $\mathsf{P}^B \neq \mathsf{NP}^B$.*

In the previous lecture we saw the proof for the existence of the language A. Now we will give a proof sketch for the existence of the language B.

8.1.1 Proof Sketch

For any language B, we can extract its lengths set defined as $L_B := \{1^n : \exists y \in B, |y| = n\}$. Notice that $L_B \in \mathsf{NP}^B$ for any B. We will now construct a language B such that $L_B \notin \mathsf{P}^B$.

- First, we will enumerate all the encodings of deterministic oracle TMs as $M_1, M_2, \ldots$. We initialize B as an empty language and strings will be added to it as this procedure runs.
- We will divide our construction of B into phases. In phase i, we pick a length n_i such that
  - no string of length n_i has been added to or removed from the language.
  - $2^{n_i} > {n_i}^i$

  The goal of the i^{th} phase is to ensure that M_i is wrong on 1^{n_i}.
- We run M_i on 1^{n_i} for ${n_i}^i$ steps. Whenever M_i queries B about a string which B has already decided upon (whether it's in the language or not), B answers honestly. This ensures consistency of B. However, if a query is made regarding a string whose presence in the

language has not yet been decided, B rejects it from the language (and answers "NO"). At best, M_i will be able to make ${n_i}^i$ queries to the oracle but there are a total of 2^{n_i} strings of length n_i.

- If M_i accepts 1^{n_i}, B rejects all strings of length n_i. However, if M_i rejects 1^{n_i}, B accepts one of the un-queried strings.
- We run this procedure for the entire list $M_1, M_2, ...$ to successfully construct B.

Claim 8.2. $L_B \notin P^B$

For the sake of contradiction, let us assume there exists some deterministic oracle TM M^B such that $L_B \in L(M^B)$. Say it's runtime is n^k. Choose M_i which is an encoding of M^B which also satisfies ${n_i}^i > n^k$. Since M_i faulted on 1^{n_i}, M^B which is the same machine only running for a fewer steps will necessarily fault on 1^{n_i}.

Remark 8.3. *Diagonalization based arguments are agnostic of oracles. Because of Baker-Gill-Solovay, the* P *vs* NP *problem cannot be solved via Diagonalization.* ◊

8.2 The classes P^{NP} and $\mathsf{NP}^{\mathsf{NP}}$

Recall, P^A stands for the class of problems that can be solved by a TM in polynomial time using an oracle of the language A. When a complexity class (e.g. NP) is in the superscript, we mean a complete problem from that complexity class is given as an oracle. So, $\mathsf{P}^{\mathsf{NP}} \equiv \mathsf{P}^{\text{Circuit-SAT}} \equiv \mathsf{P}^{\text{Vertex-Cover}} \equiv \mathsf{P}^{\text{Your-Favourite-NP-Complete-Problem}}$.

Observation 8.4. $\mathsf{NP} \subseteq \mathsf{P}^{\mathsf{NP}}$, $\mathsf{coNP} \subseteq \mathsf{P}^{\mathsf{NP}}$ *and* $\mathsf{P}^{\mathsf{NP}} = \mathsf{P}^{\mathsf{coNP}}$

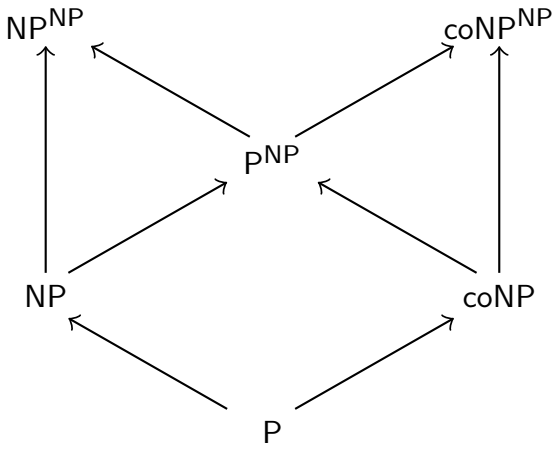


Figure 8.1: A sneak peak into the polynomial heirarchy

A complete problem in $\mathsf{NP}^{\mathsf{NP}}$ is the Σ_2-SAT $:= \{C : \exists x \forall y,\ C(x,y) = 1\}$. It is easy to see that it is in $\mathsf{NP}^{\mathsf{NP}}$ because the correct x, if it exists, can be guessed and then oracle can be queried "$\neg C(x, \_)$ in SAT?" and finally, we flip the oracle's response. Similarly, Π_2-SAT $:= \{C : \forall x \exists y,\ C(x,y) = 1\}$ is a complete problem for $\mathsf{coNP}^{\mathsf{NP}}$.

Lecture 9

$\mathsf{NP}^{\mathsf{NP}}$ and the Polynomial Hierarchy

Scribe: Soumyadeep Paul

In this lecture, we saw how to construct an NTM for every language in $\mathsf{NP}^{\mathsf{NP}}$ which queries the oracle only once. We also constructed the polynomial hierarchy.

9.1 More on $\mathsf{NP}^{\mathsf{NP}}$

Theorem 9.1. *Let $L \in \mathsf{NP}^{\mathsf{NP}}$. Then there is an $\mathsf{NTIME}(\mathrm{poly}(n))$ machine for L that makes a single query to oracle on any computational path.*

9.1.1 Proof sketch

Let M be a $\mathsf{NP}^{\mathsf{NP}}$ machine for L. Given an input x and a path P we construct a new machine M' which guesses what oracle queries are made.

$$\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_r$$
$$b_1, b_2, \cdots, b_r$$
$$\sigma_1, \sigma_2, \cdots, \sigma_r$$

where $\mathcal{C}_i$ is the machine's guess of the i^{th} query, b_i is the machines's guess for the answer of the query $\mathcal{C}_i$ and σ_i is a guess for a satisfying assignment for $\mathcal{C}_i$ if $b_i =$ true. Then the machine M' runs the following check:

- If $b_i = 1$ then check if $\mathcal{C}_i(\sigma_i) =$ ture.
- Check that $\mathcal{C}_i$'s were indeed the right queries provided the b_i's are correct.
- Confirm that

$$\bigvee_{i=1, b_i=0}^{r} \mathcal{C}_i \text{ is UNSAT}$$

using 1 oracle query.

9.2 Polynomial Hierarchy

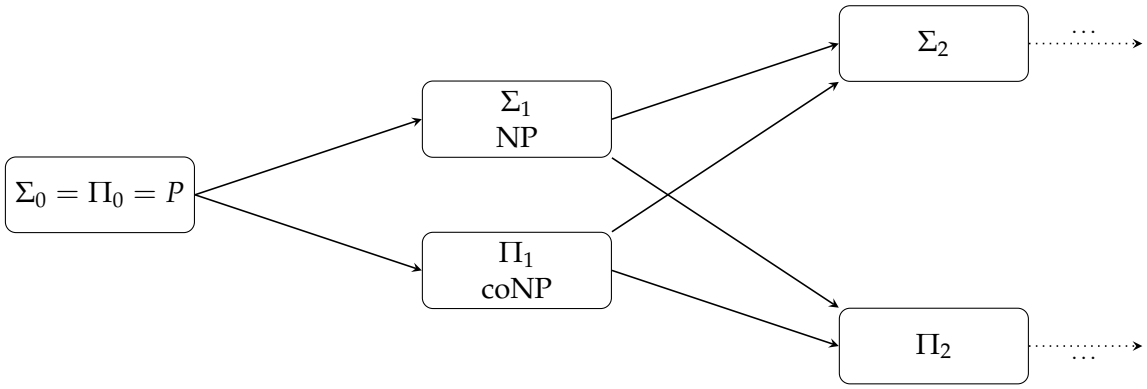


Figure 9.1: Polynomial Hierarchy

Definition 9.2 (Polynomial Hierarchy)**.** *We define the class* Σ_2 *as the class* $\mathsf{NP}^{\mathrm{SAT}}$ *which we also denote by the notation* $\mathsf{NP}^{\mathsf{NP}}$*. Similarly, we define* $\Pi_2 = \mathsf{coNP}^{\mathsf{NP}}$*. In general we define*

$$\Sigma_i = \mathsf{NP}^{\Sigma_{i-1}\text{-SAT}} = \mathsf{NP}^{\Sigma_{i-1}}$$
$$\Pi_i = \mathsf{NP}^{\Sigma_{i-1}\text{-SAT}} = \mathsf{NP}^{\Sigma_{i-1}}$$

We define the the polynomial Hierarchy as

$$\mathsf{PH} = \bigcup_{i=0}^{\infty} \Sigma_i = \bigcup_{i=0}^{\infty} \Pi_i$$

$\Diamond$

Lemma 9.3. *If* $\Sigma_1 = \Pi_1$ *then*

$$\mathsf{PH} = \Sigma_1 = \Pi_1.$$

Proof. Let $L \in \mathsf{PH}$. Thus $L \in \Sigma_i$ for some i.

Now we prove by induction on i that $\Sigma_i \subseteq \Sigma_1$ and $\Pi_i \subseteq \Pi_1$.

The base case for $i = 1$ is trivially true. We assume the statement is true for $i - 1$.

Now let $L \in \Sigma_i$. Then, by definition $x \in L \Leftrightarrow \exists y_1 \forall y_2 \cdots \psi_x(y_1, \cdots, y_i)$. We show a reduction from L to Σ_{i-1}-SAT.

Given x and a fixed y_1 we map it to the instance $\forall y_2 \exists y_3 \cdots \psi_x(y_1, \cdots, y_i)$. This is in Π_{i-1} and therefore by our assumption, in $\Pi_1 = \Sigma_1$ and therefore reduces to a formula $\exists z \psi'(x, y_1, z)$. Therefore, we are able to construct a machine in Σ_1 by considering $\exists y_1 \exists z M'(x, y_1, z)$.

$\square$

We now look at the language

$$\text{TQBF} = \{\text{All formulas of the form } \exists x_1 \forall x_2 \cdots Q_n x_n \varphi(x_1, \cdots, x_n) \text{ that are True } \}.$$

We note that TQBF is PH-hard and we believe that TQBF $\notin$ PH as we believe the PH doesn't collapse.

We note that $\mathsf{P} = \mathsf{NP} \implies$ PH collapses.

But this doesn't mean Σ_2-SAT $\in \mathsf{P}^{\mathsf{NP}}$ since for the reduction it is important that we know the source code of the machine while with oracle access we only get access to the answers of specific queries and not the source code of the machine.

Lecture 10

Self-Reducibility of NP & Implications for P vs. NP

Scribe: Soham Chatterjee

10.1 Self-Reducibility of NP

Since SAT is NP-complete it is enough to show that we can find a satisfying assignment given access to a SAT oracle. So given any formula φ we will run the following algorithm:

Step 0: First we ask oracle on the given boolean formula φ. If it accepts then φ is in SAT. Otherwise reject because it is not satisfiable, hence no satisfying assignment exists. Suppose it accepts.

Step 1: Then fix $x_1 = 1$. Now putting the value of x_1 in φ we have a new boolean formula $\varphi'(x_2, \ldots, x_n) = \varphi(1, x_2, \ldots, x_n)$. We ask the oracle on φ', if it accepts then keep $x_1 = 1$. Otherwise keep $x_1 = 0$ since we know there is a satisfying assignment from step 0 and each variable can have only two values 0 or 1.

Step 2: Now in φ' fix $x_2 = 1$ and repeat the same process as step 1. Then again keep the value for x_2 and fix x_3 Repeat then process for all variables $x_1, \ldots, x_n$ till every variable is fixed.

Step 3: At the end of the recursive process of step 2 we have a satisfying assignment $\overline{x}$ for the boolean formula φ.

10.2 EXP $\neq$ NEXP $\Rightarrow$ P $\neq$ NP

Theorem 10.1. *If* $\mathsf{P} = \mathsf{NP}$ *then* $\mathsf{EXP} = \mathsf{NEXP}$

Idea. *Padding* $\Diamond$

Proof. Suppose $L \in \mathsf{NEXP} \implies \exists\, c \in \mathbb{N}$ such that $L \in \mathsf{NTIME}\left(2^{n^c}\right)$. Suppose M_L be the turing machine. Then consider the language

$$L_{\text{PAD}} = \{(x, 1^m) \mid x \in L, m = 2^{|x|^c}\}$$

Claim 10.2. $L_{\text{PAD}} \in \mathsf{NP}$

Proof. $\left|\left(x, 1^{2^{|x|^c}}\right)\right| = 2^{O(|x|^c)}$. Consider the turing machine M'_L

M_L: Input $\tilde{x}$.
If input is not of the form $(x, 1^m)$ where $m = 2^{|x|^c}$ reject.
Run M_L on x and return the same answer.

This M'_L takes the same answer as M_L but with respect to $\left(x, 1^{2^{|x|^c}}\right)$ it takes linear time. □

Since $\mathsf{P} = \mathsf{NP}$, $L_{\text{PAD}} \in P$. Hence there exists a deterministic turing machine M_{PAD} such that $L_{\text{PAD}} \in \mathsf{DTIME}(n^{c'})$ for some $c' \in \mathbb{N}$. Then consider the turing machine M:

M: Input x
Compute $m = 2^{|x|^c}$ and build $\tilde{x} = (x, 1^m)$.
Run M_{PAD} on $\tilde{x}$ and return the same answer.

The second step takes $2^{|x|^c}$ time at most. Hence M takes $2^{O(|x|^c)}$ time. So $L \in \mathsf{EXP}$. □

10.3 Unary NP-Complete $\Rightarrow$ P = NP

Theorem 10.3. *If an unary language is* NP*-hard then* $\mathsf{P} = \mathsf{NP}$

Idea. *Keep a bag of satisfiable smaller formulas such that* $\varphi \in \text{SAT}$ *if and only if there is a formula in the bag satisfiable.* ◊

Proof. Let L is an unary language which is NP-hard. Let f is a reduction from SAT to L. Suppose $\exists\, c \in \mathbb{N}$ such that f is computed in n^c time.

Let $\mathcal{B}$ be a bag of formulas following the invariant that:

$$\varphi \in \text{SAT} \iff / \mathcal{B} \cap \text{SAT} \neq \emptyset$$

We can do two operations in $\mathcal{B}$: EXPAND or PRUNE. EXPAND returns a bigger $\mathcal{B}$ and PRUNE returns a smaller.

$\text{EXPAND}(\mathcal{B}) = \{\varphi(x_i = 0), \varphi(x_i = 1) \mid \varphi \in \mathcal{B}\}$

Now PRUNE returns a smaller $\mathcal{B}'$ such that

$$\mathcal{B} \cap \text{SAT} \neq \emptyset \iff \mathcal{B}' \cap \text{SAT} \neq \emptyset$$

So PRUNE$\mathcal{B}$ is applied when size of $\mathcal{B}$ is bigger, $|\mathcal{B}| > n^{2c}$. Now let $\mathcal{B} = \{\varphi_1, \ldots, \varphi_t\}$. Let $y_i = f(\varphi_i)$. So if any of the y_i is not of the form 1^k then PRUNE throws them away. If $y_i = y_j$ for $i \neq j$ then both are satisfiable or both not. So we can throw one of them away.

PRUNE:($\mathcal{B}$): Put φ in it if $f(\varphi)$ is of the form 1^k for some $k \in \mathbb{N}$ and if for $\varphi, \varphi' \in \mathcal{B}$, $f(\varphi) = f(\varphi')$ put only one of them.

So upon repeated EXPAND and PRUNE after n steps $\mathcal{B}$ only contains literals and if one of them is true then we can return true. So the algorithm is

Algorithm 1: SAT-Fast-Algo

Input: n-variate boolean formula φ

Output: Is φ satisfiable.

```
1 begin
2     B ⟵ {φ}
3     for i = 1, ..., n do
4         B ⟵ EXPAND(B)
5         if |B| > n^{2c} then
6             PRUNE(B)
7     if any element of B is true then
8         return True
9     return False
```

This algorithm is a polynomial time algorithm. Therefore $L \in \mathsf{P}$. Hence $\mathsf{P} = \mathsf{NP}$. □

10.4 Sparse NP-Complete $\Rightarrow$ P = NP

Definition 10.4 (Sparse Language)**.** *L is n^c-sparse if $|L \cap \{0,1\}^n| \leq n^c$. Hence L is n^c-cosparse if $\overline{L}$ is n^c-sparse.* ◊

Proposition 10.5 (Mahaney's Theorem)**.** *If a n^c-cosparse language is* NP*-hard then* $\mathsf{P} = \mathsf{NP}$*.*

Idea. *Same idea as unary language with modifications.* ◊

Proof. Let L is an n^c-cosparse language which is NP-hard. Let f is a reduction from SAT to L. Suppose $\exists\, k \in \mathbb{N}$ such that f is computed in n^k time. Let φ is a given formula.

Suppose $\mathcal{B}$ be the bag of smaller formulas. And we define the EXPAND operation like in the unary case. The PRUNE operation if for any $\varphi \in \mathcal{B}$, $f(\varphi)$ is not of the form 1^l for some $l \in \mathbb{N}$ throw it away. And if for some $\varphi, \varphi' \in \mathcal{B}$ if $f(\varphi) = f(\varphi')$ then we can throw one of them. So suppose all $f(\varphi)$ for all $\varphi \in \mathcal{B}$ are different.

Now suppose φ is unsatisfied $\implies$ $\mathcal{B}$ only has unsatisfied formulas. Suppose for all $\varphi \in \mathcal{B}$, $|f(\varphi)| \leq n^l$ for some $l \in \mathbb{N}$. Then $|\overline{L} \cap \{0,1\}^{\leq n^l} \leq (n^l)^{c+1} = n^{l(c+1)}$. So if $|\mathcal{B}| = t > n^{(c+1)l}$ then we are getting t many $f(\varphi)$'s to be in $\overline{L}$. But $\overline{L}$ can not have that many entries. So we reject immediately. □

Lecture 11

Introduction to Space Complexity

Scribe: Shubham A. Bhardwaj

In this lecture, we defined space complexity and various complexity classes associated with space and investigated some basic relationship between these classes.

11.1 Space Complexity

One way to define space used by a Turing Machine M on input x is to count the total number of distict cells used during computation by M on x, but since we want to make sense of computation using sublinear space and the input itself is of linear size, we need to slightly modify our approach. We do this by having a distinguished input tape which is 'read only' and the cells used in this read only tape do not count toward space used by TM. With this in mind, we give some basic definitions.

$$\begin{aligned}
\mathsf{SPACE}(M,x) &:= \#\text{ cells used by } M \text{ on } x \text{ in non input tapes}\\
\mathsf{SPACE}(M,n) &:= \max_{x:|x|=n} \mathsf{SPACE}(M,x)\\
\mathsf{DSPACE}(s(n)) &:= \{L : L = L(M) \text{ for some DTM with } \mathsf{SPACE}(M,n) = O(s(n))\}\\
\mathsf{NSPACE}(s(n)) &:= \{L : L = L(M) \text{ for some NTM with } \mathsf{SPACE}(M,n) = O(s(n))\}\\
\mathsf{PSPACE} &:= \mathsf{DSPACE}(\mathrm{poly}(n))\\
\mathsf{NPSPACE} &:= \mathsf{NSPACE}(\mathrm{poly}(n))\\
\mathsf{L} &:= \mathsf{DSPACE}(\log(n))\\
\mathsf{NL} &:= \mathsf{NSPACE}(\log(n))
\end{aligned}$$

11.2 Space Hierarchy Theorem

Theorem 11.1. *If f and g are space constructible functions, such that $f(n) = o(g(n))$, then*

$$\begin{aligned}
&\mathsf{DSPACE}(f(n)) \subsetneq \mathsf{DSPACE}(g(n))\\
&\mathsf{NSPACE}(f(n)) \subsetneq \mathsf{NSPACE}(g(n))
\end{aligned}$$

Proof. We skip the proof as it is similar to the proof of Time Hierarchy theorems. □

11.3 Basic Observations

Observation 11.2. $\mathsf{P} \subseteq \mathsf{PSPACE}$ *and* $\mathsf{NP} \subseteq \mathsf{PSPACE}$

Since any machine running in polynomial time can use atmost polynomial space, we have $\mathsf{P} \subseteq \mathsf{PSPACE}$. Also, for any Circuit-SAT instance, we can check all possible assignments in lexicographic order. To do this, we just need to store a counter to keep track of which assignment we are currently checking and the current assignment. Clearly, this can be done in $O(n)$ space, this implies $\mathsf{NP} \subseteq \mathsf{PSPACE}$.

Observation 11.3. Σ_2 -SAT $\in \mathsf{PSPACE}$

Σ_2 -SAT $= \{\varphi \quad | \exists x, \forall y \quad \varphi(x, y) = 1\}$ Again similar to Circuit-SAT, we just iterate over all possible assignments of x and check if $\varphi(x,_{)}$ becomes tautology. Also, we can extend this to show that Σ_i -SAT $\in \mathsf{PSPACE}$. Thus, we have the following

Observation 11.4. $\mathsf{PH} \in \mathsf{PSPACE}$

11.4 Configuration Graph

We introduce the notion of configuration graphs. Let M be a TM running in space $s(n)$. At any step, we need to store the current state, head positions and content of the tape, thus configuration of M at any step is a subset of $Q \times s(n)^k \times n \times 2^{s(n)}$ where k is the number of tapes. The configuration graph of M then consist of all configurations as nodes and edges between configurations c_1 and c_2 if we can take a transition from c_1 to go to c_2. Notice that configuration graph for DTM will have outgoing degree atmost 1 whereas for NTMs, outgoing degree can be upto 2.

Now, to check if $x \in L(M)$, we can just check whether there exist a path from initial configuration c_1 to some accept configuration, and since the number of vertices and edges in the graph are of order $2^{O(s(n))}$, we can check whether there exists such a path in $2^{O(s(n))}$ time. Thus,we have the following lemmas:

Lemma 11.5. $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{NTIME}(2^{O(s(n))})$.

Lemma 11.6. $\mathsf{NPSPACE} \subseteq \mathsf{EXP}$.

Lecture 12

PSPACE Completeness

Scribe: Vivek Karunakaran

In this lecture, We proved that TQBF is PSPACE-complete and used the proof to see the Savitch's Theorem. And We proved that the Generalized Geography is PSPACE-complete by the reduction from TQBF to Generalized Geography.

12.1 TQBF is PSPACE-complete

The TQBF problem can be solved using a recursive algorithm that evaluates the formula based on the quantifiers:

- If the formula is a Boolean constant (e.g., `true` or `false`), return its value.
- If the formula is of the form "$\exists x : \varphi$", recursively evaluate φ for both $x = 0$ and $x = 1$, returning `true` if at least one satisfies φ.
- If the formula is of the form "$\forall x : \varphi$", recursively evaluate φ for both $x = 0$ and $x = 1$, returning `true` only if both cases satisfy φ.

The recursive evaluation does not require storing all variable assignments simultaneously. Instead, the algorithm uses *depth-first search (DFS)* with backtracking, as in recursive function calls. Since there are at most n variables, the recursion depth is at most n. Thus, the function call stack has at most $O(n)$ depth, which is reused for all branches. If the expression is of length m, the total space used is $O(m \cdot n)$. Hence, TQBF $\in$ PSPACE.

Now let us prove that TQBF is PSPACE-hard. Let $L \in$ PSPACE and let M be its corresponding machine that takes space $S(n)$. In the corresponding configuration graph, there are at most $2^{O(S(n))}$ states. Given an input x, we aim to construct a TQBF instance $\Psi(s, t)$ such that:

$$x \in L \iff \Psi(C_{\text{start}}, C_{\text{accept}}) = \text{TRUE}$$

That is, $\Psi(C_{\text{start}}, C_{\text{accept}})$ has to encode the statement that there exists a path of length at most $2^{O(S(n))}$ between the start state C_{start} and the accepting state C_{accept} in the configuration graph, $G_{M,x}$.

Let $\Psi_l(C_1, C_2)$ be the statement that there exists a path of length at most l from C_1 to C_2 in $G_{M,x}$. Consider $\Psi_{2l}(C_1, C_2)$. This means that there exists another node C_{mid} such that both the path from C_1 to C_{mid} and the path from C_{mid} to C_2 are of length at most l. That is:

$$\Psi_{2l}(C_1, C_2) = \exists C_{\text{mid}} \ : \ \Psi_l(C_1, C_{\text{mid}}) \wedge \Psi_l(C_{\text{mid}}, C_2).$$

However, this leads to an exponential increase in the size of the formula. To avoid this, we use a universal quantifier to capture the same statement as follows:

$$\Psi_{2l}(C_1, C_2) = \exists C_{\text{mid}} \, \forall (\alpha, \beta) \in \{(C_1, C_{\text{mid}}), (C_{\text{mid}}, C_2)\} \ : \ \Psi_l(\alpha, \beta).$$

The base cases $\Psi_1(C_1, C_2)$ can be expressed similarly to the proof of the Cook-Levin theorem. Therefore, the formula size satisfies:

$$\text{size}(\Psi_{2l}) \leq \text{size}(\Psi_l) + O(S).$$

This implies that the final quantified formula has size $O(S^2)$. This completes the reduction, proving that TQBF is PSPACE-complete.

12.1.1 PSPACE = NPSPACE

Note that nowhere in the reduction do we require that each node in the configuration graph has degree one. This means that the same reduction applies to *nondeterministic* Turing machines as well. Consequently, TQBF is NPSPACE-hard too, leading to the conclusion:

$$\mathsf{PSPACE} = \mathsf{NPSPACE}.$$

12.1.2 Savitch's Theorem

Theorem 12.1. *For any space-constructible function $S(n) \geq \log n$,*

$$\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DSPACE}(S(n)^2).$$

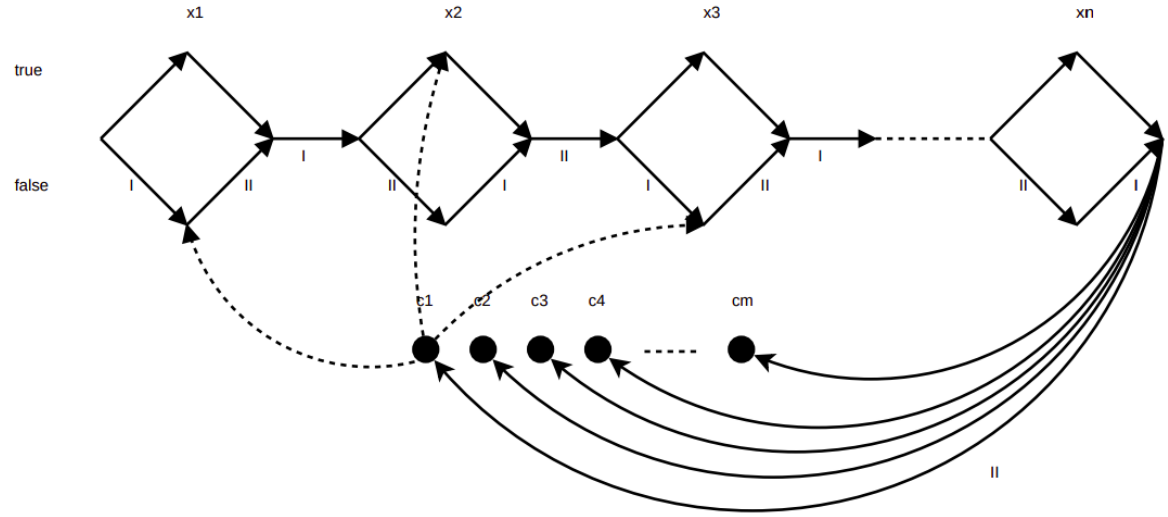
Consider $L \in \mathsf{NSPACE}(S(n))$. By following the above reduction, We can reduce the instance of L to the equivalent instance of TQBF of size $O(S^2)$. We know that TQBF is in PSPACE and precisely this TQBF $\in \mathsf{DSPACE}(S^2)$. Therefore, $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DSPACE}(S(n)^2)$.

12.2 Generalized Geography is PSPACE-complete

We have a directed graph G with the starting vertex given. There are two players: Player1 and Player2. The game starts with the Player1 and they take turns choosing vertices from G such that the vertices form a simple path (no vertices repeated) from the starting vertex. A player loses when they are unable to continue the path. We want to decide if Player1 has a winning strategy for geography on G starting at some initial vertex s.

Generalized Geography is in PSPACE. We can do the recursive algorithm as we have done with TQBF. We start with the start vertex. Let $A(u)$ be the function that returns True if the player starting with u has a winning strategy. Suppose u has edges to $v_1, v_2, .., v_k$. Then, $A(u)$ is True iff $\exists v \in \{v_1, v_2, ..., v_k\}$ such that $A(v)$ is False after removing u and the corresponding edges from the graph. This is the recursive algorithm with the recursion stack getting at most n (number of vertices) levels. So, The space consumed by the recursive stack is polynomial and hence, GenGeog is in PSPACE.

Now we construct a reduction TQBF $\leq$ GenGeog. Assume without loss of generality that we have an equal number of quantifiers, and the formula is a 3CNF-formula. That is, $\varphi = \exists x_1 \forall x_2 ... \forall x_n : c_1 \wedge c_2 \wedge ... \wedge c_m$ where c_i is a 3-clause. Then, the construction is as follows:



Note that the first diamond is played by Player1 and the second diamond is played by Player2 and so on. So, at last, the diamond corresponding to the variable x_n is played by Player2. So, Eventually Player2 chooses some clause and Player1 has to choose some literal in order to end the game if φ is True.

If φ is True, then, Player1 has to choose accordingly at his turns during the existential quantifier so as to make the expression True with the chosen values for the variables. Finally when his turn is to choose the literal, he has to choose the literal chosen by him which turned out to be true which is always possible, if φ is True. This ensures Player1 can win always. If φ is False, Player2 can choose the clause in which all the 3 literals end up as False. So, whichever literal chosen by Player1, Player2 can always end the game and hence winning. Similarly if the Player1 has the winning strategy,then that corresponds to the values of the variables with the existential quantifier that will make φ True and If the Player2 has the winning strategy, that corresponds to the values of the variables with the forall quantifier that will make φ False.

Therefore, Generalized Geography is PSPACE-hard and hence, PSPACE-complete.

Lecture 13

Logspace Reductions and NL-Completeness

Scribe: Aindrila Rakshit

Topics covered in this lecture

1. Logspace Reductions
2. NL-Completeness of directed s-t connectivity

13.1 Logspace Reductions

We would like to know problems that are complete for NL, and thus would like to define a type of reduction for that class. We cannot use polynomial-time reduction since L is a weaker class and the notion of reduction shouldn't be more powerful than the class. So we will introduce logspace reductions, which are computable my a deterministic TM running in logarithmic space.

Definition 13.1 (Logspace reduction)**.** *A function $f : \Sigma^* \to \Sigma^*$ is logspace computable, if f is polynomially bounded and the languages $L_f = \{\langle x, i\rangle : f(x)_i = 1\}$ and $L'_f = \{\langle x, i\rangle : i \leq |f(x)|\}$ are in* L*.*

A language A is logspace reducible to language B, i.e, $A \leq_{\mathsf{L}} B$, if there exists a logspace computable function f such that $x \in A$ if and only if $f(x) \in B$. ◊

Some properties of logspace reductions:

1. $A \leq_{\mathsf{L}} B$ and $B \leq_{\mathsf{L}} C$, then $A \leq_{\mathsf{L}} C$.

   Proof. We have machines M_1 and M_2 that compute the reductions $A \leq_L B$ and $B \leq_L C$ respectively. M_2 needs access to its input tape which M_1 writes to and it cannot store the output of M_1 in a work tape since that would exceed the space available to us. Instead, we assume M_2 has access to the output of M_1. Each time M_2 needs to read a bit from the

input tape, we execute M_1 and output the requested bit. This can be done in logspace (since we aren't storing the whole output at any point). Hence, $A \leq_L C$. □

2. $A \leq_{\mathsf{L}} B$ and $B \in \mathsf{L}$, then $A \in \mathsf{L}$.

   Proof. Suppose $f : \Sigma^* \to \Sigma^*$ is the log-space computable reduction from A to B. The main issue with the naive approach is again that the final machine cannot afford to write down the output of $f(x)$ entirely. However, we can always re-compute it as and when we need. Essentially, we begin running M (that accepts B) on "$f(x)$" but whenever this machine needs to read the i-th bit of $f(x)$, we run the reduction f, maintain a counter for output bits, and wait for the i-th bit of $f(x)$ to be computed (discarding all other bits). □

Definition 13.2 (NL-Completeness)**.** *A language L is* NL*-hard under logspace reductions if for every language $L' \in \mathsf{NL}$ there is a logspace reduction f such that $L' \leq_{\mathsf{L}} L$. L is* NL*-complete if it is* NL*-hard and $L \in \mathsf{NL}$.* ◊

13.2 NL-Completeness of directed s-t connectivity

Definition 13.3 (Dir-Path)**.** *The language* Dir-Path *is defined as*

$$\{\langle G, s, t\rangle \ : \ G \text{ is a directed graph and there is a directed path from } s \text{ to } t\}.$$

◊

Theorem 13.4. Dir-Path *is* NL*-Complete under logspace reductions.*

Proof. **Showing** Dir-Path $\in$ NL**:**

A nondeterministic log-space Turing machine can solve the directed s-t connectivity problem as follows:

- It starts at the source vertex s and maintains the index of the current vertex.
- Using nondeterminism, it guesses a neighbor of the current vertex and moves to it.
- If it reaches t within at most n steps (where n is the number of nodes), it accepts.
- If it has taken n steps and has not reached t, it rejects.

The machine only needs to store:

- The current vertex index (requiring $O(\log n)$ bits).
- A counter for the number of steps taken (also requiring $O(\log n)$ bits).

Since the total space used is $O(\log n)$, we conclude that Dir-Path $\in$ NL.

Showing Dir-Path **is** NL**-hard.**

To show that every language $L \in \mathsf{NL}$ reduces to Dir-Path, let M be an NL-machine that decides L in $O(\log n)$ space. We define a log-space computable function f that maps an input x of size n to a directed graph $G_{M,x}$, called the configuration graph.

- The vertices of $G_{M,x}$ correspond to all possible configurations of M on input x.
- The number of possible configurations is at most $2^{O(\log n)} = \text{poly}(n)$.
- There is an edge from configuration C_1 to C_2 if and only if C_2 is one of the (at most two) possible next configurations of C_1 according to M's transition function.
- The special start node C_{start} represents the initial configuration of M on x.
- The special accept node C_{acc} represents any accepting configuration of M.

Thus, $x \in L$ if and only if there exists a directed path from C_{start} to C_{acc} in $G_{M,x}$.

The Reduction is Log-Space Computable. The adjacency matrix of $G_{M,x}$ can be computed using $O(\log n)$ space. Specifically, given two configurations (C_1, C_2), a deterministic machine can:

- Verify whether C_2 is a valid next configuration of C_1 using $O(\log n)$ space.
- Construct the adjacency matrix entry for (C_1, C_2) in log-space.

Since we only generate edges dynamically without storing the entire graph, the reduction uses at most $O(\log n)$ space.

Since Dir-Path is both in NL and NL-hard, we conclude that Dir-Path is NL-complete. □

Other complete problems for NL include 2-SAT, checking if a graph is bipartite.

Theorem 13.5 (Reingold)**.** Undir-Path $\in$ L.

Lecture 14

The Immerman-Szelepcsényi theorem

Scribe: Nishant Das

14.1 Read-Once Certificates for NL

In the case of NP, we can characterize membership using a deterministic polynomial-time verifier with access to a certificate (or witness). A natural question is whether a similar characterization holds for NL. However, a direct adaptation of the NP definition fails and incorrectly place problems such as SAT within NL, which is unlikely. To address this, we impose the restriction that the witness is *read-once* and *read-only*.

Definition 14.1 (Read-Once Certificates for NL)**.** *A language L is a polynomially-sized read-once certifiable in log-space if there exists a deterministic log-space verifier V such that $x \in L$ if and only if:*

The input x is provided to V on a read-only input tape.

There exists a certificate $w \in \{0,1\}^m$ with $m = |x|^c$ (for some constant c), provided on a separate read-once, read-only witness tape to V.

V has access to an additional work tape of size $O(\log |x|)$.

V accepts (x, w).

$\Diamond$

This definition precisely captures NL, as formalized in the following lemma.

Lemma 14.2. *A language L is polynomially-sized read-once certifiable in log-space if and only if $L \in$ NL.*

Example: Directed Path Problem For the language Dir-Path, where (G, s, t) is in the language if there exists a path from s to t in the directed graph G, a valid certificate is simply the sequence of vertices along such a path. The log-space verifier reads this path once, checking validity by storing only the current and previous vertex.

14.2 Immerman-Szelepcsényi Theorem

Theorem 14.3. $\overline{\text{Dir-Path}} \in$ NL.

In the previous lecture, we established that the **directed path problem** (DirPath) is NL-complete. Consequently, its complement, $\overline{\text{Dir-Path}}$, must be coNL-complete. If we can show that $\overline{\text{Dir-Path}}$ is in NL, then we establish that $\mathsf{NL} = \mathsf{coNL}$.

To achieve this, we use the technique of **inductive counting**, which enables us to track the number of reachable vertices in a structured manner while staying within NL.

Define a sequence of sets B_i, representing the vertices reachable from a source node s within distance i:

$$B_i = \{u \mid \text{dist}(s,u) \leq i\} = B_{i-1} \cup \{v \mid (u,v) \in E, u \in B_{i-1}\}.$$

That is, each layer consists of the previous layer plus all nodes that can be reached in one more step. Our goal is to **certify that a target node t does not belong to B_n**, meaning that t is not reachable from s.

Suppose we already know the size of each reachability layer, denoted $r_i = |B_i|$. We construct a certificate proving that $t \notin B_i$ as follows:

1. List the r_i vertices in B_i, say $u_1, u_2, \ldots, u_{r_i}$ in lexicographically increasing order.
2. For each vertex u_k in B_i, provide an explicit path from s to u_k with length at most i.
3. The verifier, using only logspace, checks each path to confirm that these vertices belong to B_i.

Since the certificate explicitly lists all reachable vertices, if t is not included, we can conclude that $t \notin B_i$. Now, let's see how given $r_i = |B_i|$, we can certify $r_{i+1} = |B_{i+1}|$ by providing certificates for **all** vertices:

- If $u \in B_{i+1}$, provide a path from s to u of length at most $i+1$.
- If $u \notin B_{i+1}$, enumerate all vertices in B_i along with their paths and verify that u is not reachable from any of them in one step.

This ensures that the verifier can check the correctness of each B_i, leading to a final certificate that confirms $t \notin B_n$, thereby proving that t is not reachable from s.

The complete certificate consists of the base case: $B_0 = \{s\}$ and certificates for each layer $|B_1|, |B_2|, \ldots, |B_n|$. The verifier, working in logspace, checks the validity of these certificates. If t is not in B_n, it concludes that there is no path from s to t, proving that $\overline{\text{Dir-Path}}$ is in NL. This establishes the fundamental result that $\mathsf{NL} = \mathsf{coNL}$.

Lecture 15

Catalytic computation

Scribe: Soumyadeep Paul

15.1 Catalytic computation

Definition 15.1 (Catalytic computation)**.** *We consider that in addition to the work tape of size $w(n)$ the machine is given larger catalytic tape of size $c(n)$. The catalytic tape must be returned to its original state at the end of the computation.* ◊

Definition 15.2 (CSPACE)**.** *We let $\mathsf{CSPACE}(w(n), c(n))$ be the set of languages which are computable by catalytic machines with catalytic tape size $c(n)$ and work tape size $c(n)$.* ◊

We generally work with the convention $\mathsf{CSPACE}(w(n)) = \mathsf{CSPACE}(w(n), 2^{w(n)})$. We ovserve that

$$\mathsf{L} \subseteq \mathsf{CL}\,.$$

15.2 Reversible computation

We are given n registers $R_1, R_2, \cdots, R_n \in \mathcal{R}$ where $\mathcal{R}$ is a ring. We are allowed the instructions of the form

$$R_i \leftarrow R_i \pm u_j u_k,$$

where u_j, u_k are either constants or other registers.

Definition 15.3 (Transparent computation)**.** *A reversible program P transparently computes a function $f(x_1, \cdots, x_n)$ on register R_1 if for all $\tau_1, \cdots, \tau_n$, they get taken to $\tau_1', \cdots, \tau_n'$ where $\tau_1' = f(\overline{x}) + \tau_1$.* ◊

Definition 15.4. *We say a reversible program computes f cleanly on register R_1 if if $\tau_1' = f(\overline{x}) + \tau_1$ and $\tau_i' = \tau_i$ for all $2 \leq i \leq n$.* ◊

Theorem 15.5. *Suppose there is an m register program P that transparently computes $f(\overline{x})$, then there is an $m+1$ register program P' that cleanly computes f and $|P'| \leq 2|P| + 2$.*

Proof. Suppose R_0 is a new register, and P is a program that transparently compute $f(\overline{x})$ on R_1, then consider the following program.

- $R_0 \leftarrow R_0 - R_1$
- Run P
- $R_0 \leftarrow R_0 + R_1$
- Run P^{-1}

It is easy to see that this program cleanly computes f on register R_0. □

Lemma 15.6. *Suppose $f(\overline{x})$ can be transparently computed by P_f, and $g(\overline{x})$ can be transparently computed by P_g then we can transparently compute $f + g$ by $P = P_f^r \cdot P_g^r$.*

Lemma 15.7. *Assume that we have at least 3 registers. If P_f cleanly computes f, and P_g cleanly computes g, then we can also cleanly compute fg.*

Proof. Let σ be the instruction $R_1 \leftarrow R_1 - R_2R_3$. We can cleanly compute fg via the following program:

$$P_f^{(2)} \cdot \sigma \cdot P_g^{(3)} \cdot \sigma^{-1} \cdot (P_f^{(2)})^{-1} \cdot \sigma \cdot (P_g^{(3)})^{-1} \cdot \sigma^{-1}.$$

Once can easily check that this program clearly compute fg on register R_1. The length of the program is at most $4\max(|P_f|, |P_g|) + 4$. □

Theorem 15.8 (Ben-Or & Cleve)**.** *If f is computable by a fan-in 2 depth d formula, then we can cleanly compute f via a program of length $\leq 4^d$ using 3 registers.*

From the above proof, we can make a simple observation — Theorem 15.8 also works when $\mathcal{R}$ is not commutative.

Theorem 15.9 (Brent, Spira)**.** *If f is computable by a formula of size s, then it is also computable by a formula of size* $\text{poly}(s)$ *and depth* $O(\log s)$.

Theorem 15.10. *If $f(x) = (M_1 \cdots M_s)_{1,1}$ and each $M_i \in \mathcal{R}^{k\times k}$, then f can be cleanly computed by a program of length $O(s^2k^3)$ with $O(k^2)$ registers.*

Proof sketch. Assume each super-register now holds a matrix rather than an element of the underlying ring. Each super-register is will be internally thought of as k^2 registers, one for each entry. Clearly, each super-register instruction can be translated to $O(\text{poly}(k))$ instructions on the standard registers, and this eventually yields a program of length $O(s^2 \cdot \text{poly}(k))$ with $O(k^2)$ registers. □

Theorem 15.11. $\mathsf{DirPath} \in \mathsf{CL}$, *implying* $\mathsf{NL} \subseteq \mathsf{CL}$.

Proof sketch. Checking if there is a directed path from s to t can be obtained by inspecting the (s, t)-entry of A^n where A is the adjacency matrix of the graph (with self-loops on each vertex) and n is the number of vertices in G.

One catch is that since we have to deal with registers holding integer values and that might make their size unbounded (and potentially the entries of the matrix A^n could be as large as n^n and we can't store that in our workspace). This can be fixed via Chinese Remaindering. □

The above proof actually extends to show that a class called $\mathsf{GapL} \subseteq \mathsf{CL}$.

With more work, one can not just work with f computable by small formulas or branching programs, but in fact by any polynomial-sized log-depth circuits. This yields the following theorem.

Theorem 15.12. $\mathsf{LOGCFL} \subseteq \mathsf{CL}$.

(Check the handwritten notes for the proof).

In fact, the largest class that is currently known to be in CL is the class TC^1.

Theorem 15.13 (BCKLS)**.** *Uniform* $\mathsf{TC}^1 \subseteq \mathsf{CL}$.

In terms of upper-bounds for CL, the best we know of currently is the following result.

Theorem 15.14 (BCKLS)**.** $\mathsf{CL} \subseteq \mathsf{ZPP}$.

Lecture 16

Tree Evaluation Problem

Scribe: Soham Chatterjee

Definition 16.1 (Tree Evaluation Problem ($\mathsf{TEP}_{h,k}$))**.** *Given a tree of height h and alphabet size k where every leaf $v_u \in \{0,1\}^k$ for all $u \in \{0,1\}^h$ is an element of $\{0,1\}^k$ and every internal node is a function $f_u : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ and the ouput of the tree is the value at the root where the tree is evaluated bottom-up. We denote it by $\mathsf{TEP}_{h,k}$.* ◊

The input size is $2^h \cdot k + (2^h - 1)2^{2k} \cdot k = 2^{O(k+h)}$.

Observation 16.2. $\mathsf{TEP}_{h,k} \in \mathsf{P}$

A naive low-space algorithm for $\mathsf{TEP}_{h,k}$ will be to evaluate each subtree of the root separately and the return the value of the root based on the value of the subtrees:

$\mathsf{TEP}_{h,k}$:

$\alpha_L = \mathsf{TEP}_{h-1,k}(\text{Left Subtree})$

$\alpha_R = \mathsf{TEP}_{h-1,k}(\text{Right Subtree})$

Return $f_{\text{root}}(\alpha_L, \alpha_R)$

The space used by this algorithm is

$$\text{Space}(h,k) = \text{Space}(h-1,k) + O(k) = O(h \cdot k) = O(\log^2 n)$$

The interesting case for us is where $h, k = O(\log n)$. So for this case we will reference it as TEP from now on.

Theorem 16.3 ([CM21])**.** $\mathsf{TEP} \in \mathsf{DSPACE}\left(\frac{\log^2 n}{\log\log n}\right)$

Theorem 16.4 ([CM24])**.** $\mathsf{TEP}_{h,k} \in \mathsf{DSPACE}((h+k)\log k)$

The recent result by Cook and Mertz gives us the result

Corollary 16.5. $\mathsf{TEP} \in \mathsf{DSPACE}(\log n \log\log n)$

Idea. *Do the naive algorithm using catalytic computation* ◊

For all $f : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ we can find a polynomial $F(x_1, \ldots, x_k, y_1, \ldots, y_k) \in \mathbb{F}[x_1, \ldots, x_k, y_1, \ldots, y_k]$ such that $F(a_1, \ldots, a_k, b_1, \ldots, b_k) = f(a_1, \ldots, a_k, b_1, \ldots, b_k)$ for $a_i, b_j \in \{0,1\}$ where $F(\overline{x}, \overline{y}) = \sum_{\overline{a},\overline{b} \in \{0,1\}^k} f(\overline{a}, \overline{b}) \cdot \delta_{\overline{a},\overline{b}}(\overline{x}, \overline{y})$ where $\delta_{\overline{a},\overline{b}}(\overline{x}, \overline{y}) = \prod_{i=1}^{k} (a_i x_i + (1 - a_i)(1 - x_i))(b_i y_i + (1 - b_i)(1 - y_i))$. $\deg F \leq 2k$

Observation 16.6. *Given $f : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ and $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k \in \mathbb{F}$ we can compute $F(\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k)$ in $\mathsf{DSPACE}(k + \log |\mathbb{F}|)$ space.*

Now we will discuss the Cook and Mertz algorithm. The program memory model is 4 memories: $(u, \tau_1, \tau_2, \tau_3)$ where u is of size h, τ_i is of size k which stores values $\mathbb{F}^k$. Goal is the build a P_u that does

$$(u, \tau_1, \tau_2, \tau_3) \to (u, \tau_1, \tau_2, \tau_3 + v_u)$$

where $v_u = f_u(v_{u0}, v_{u1})$ for all $u \in \{0,1\}^{<h}$. Our inductive assumption is we have P_{u0}, P_{u1} which does the following jobs respectively

$$(u, \tau_1, \tau_2, \tau_3) \xrightarrow{P_{u0}} (u, \tau_1 + v_{u0}, \tau_2, \tau_3)$$

and

$$(u, \tau_1, \tau_2, \tau_3) \xrightarrow{P_{u1}} (u, \tau_1, \tau_2 + v_{u1}, \tau_3)$$

Attempt 1:
- ▷ Run P_{u0}
- ▷ Run P_{u1}
- ▷ $R_3^{(i)} \longleftarrow R_3 + F_u^{(i)}(R_1, R_2)$
- ▷ Run P_{u0}^{-1}
- ▷ Run P_{u1}^{-1}

But we end with $\tau_3 + F_u(\tau_1 + v_{u0}, \tau_2 + v_{u1})$ in R_3 where as we wanted $\tau_3 + F_u(v_{u0}, v_{u1})$.

Attempt 2:
- ▷ Scale R_1 and R_2 by $\alpha \in \mathbb{F} \setminus \{0\}$
- ▷ Run P_{u0}, P_{u1}
- ▷ $R_3^{(i)} \longleftarrow R_3 + F_u^{(i)}(R_1, R_2)$
- ▷ Run P_{u0}^{-1}, P_{u1}^{-1}
- ▷ Scale R_1 and R_2 by α^{-1}

Now R_3 holds $\tau_3 + F_u(\alpha\tau_1 + v_{u0}, \alpha\tau_2 + v_{u1})$. $G(t) = F_u(t \cdot \tau_1 + t, t \cdot \tau_2 + t)$. Therefore $G(0) = \sum_{i=0}^{2k} \gamma_i G(\alpha_i) = \sum_{i=0}^{2k} \gamma_i F_u(\alpha_i \tau_1 + v_{u0}, \alpha_i \tau_2 + v_{u1})$. Now interpolate. So we do the above process $2k + 1$ times for distinct $\alpha_i \in \mathbb{F}$.

Algorithm 2: Program P_u for register R_3

for $i = 1, \ldots, 2k+1$ **do**
- Scale R_1 and R_2 by $\alpha_i \in \mathbb{F} \setminus \{0\}$
- Run P_{u0} for register R_1.
- Run P_{u1} for register R_2.
- $R_3 \leftarrow R_3 + \gamma_i \cdot F_u(R_1, R_2)$
- Run P_{u0}^{-1} for register R_1.
- Run P_{u1}^{-1} for register R_2.
- Scale R_1 and R_2 by α_i^{-1}.

The local space that the above algorithm needs to maintain is just the current value of i (which requires $O(\log k)$ bits), and any space needed to computes F_u, the value of α_i, γ_i, can be during the recursion. Thus,

$$\text{LocalSpace}(P_u) = \text{LocalSpace}(P_{ub}) + O(\log k) \implies \text{Space} = O((h+k)\log k)$$

Therefore for $h, k = O(\log n)$ we have $\text{Space} = O(\log n \log\log n)$.

Remark. *With some additional ideas (using multivariate interpolation), the above bound can be improved to* $O(k + h\log k)$. ◊

Lecture 17

Simulating Time With Square-Root Space

Scribe: Shubham A. Bhardwaj

In this lecture, we went over the recent result of Ryan Williams on the relation between time and space. We'll prove the following theorem

Theorem 17.1 ([Wil25])**.** $\mathsf{DTIME}(t(n)) \subseteq \mathsf{DSPACE}(\sqrt{t(n)\log t(n)})$

17.1 Proof Outline

Very Broadly, the idea is to reduce simulation of a time $t(n)$ TM to a tree evaluation problem and then use the recent result of [CM24] to get the theorem. Let M be a TM running in time $t(n)$. We now go over the basic ideas of the proof.

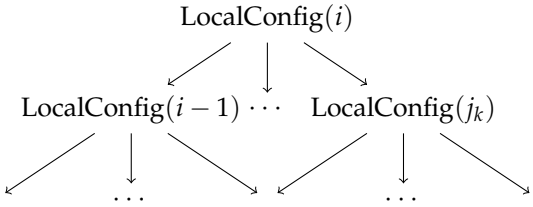
- Divide the space $t(n)$ into epochs of size B each. Now, if we know the "configuration" of the TM at the end of the i-th epoch, we can figue out the configuration of the TM at the end of the $(i+1)$-th epoch. The space needed to compute the configuration of the TM at the end of the $(i+1)$-th epoch from i-th epoch is $O(B)$ since we just need to do a time B simulation of the TM. So, we can store the configuration of the TM at the end of the i-th epoch in $O(B)$ space.
- Observe that we do not need to know the complete configuration of the TM at the end of the i-th epoch, Since, in B time, only B cells of the tape can change, if we just know the local configuration at the end of i-th epoch , we can simulate B steps of the TM and give the "local configuration" at the end of $(i+1)$-th epoch. Content of the rest of tape remains same.
- We divide the tape into blocks of size B each. From the Assignment, we know that we can construct a block respecting TM with the same language with only a constant factor increase in time. Thus, we can assume that our machibe is Block respecting. With this, local configuration at the end of i-th epoch is just the content of the blocks in which our heads are at the end of i-th epoch.

- Define the following functions:

$$\begin{aligned}
\text{TapeBlock}(h,i) &= \text{block in which head of tape } h \text{ is} \\
\text{Content}(h,i) &= \text{content of the TapeBlock}(h,i) \text{ after epoch } i \\
\text{LocalConfig}(i) &= \text{Local Configuration after epoch } i \\
\text{last}(h,i) &= \max\{j < i : \text{TapeBlock}(h,j) = \text{TapeBlock}(h,i)\}
\end{aligned}$$

- If the head does not change block from between $(i-1)$-th epoch and i-th epoch, then $\text{LocalConfig}(i-1)$ suffices to compute $\text{LocalConfig}(i)$. Otherwise, we also need to know the content of the curent block after the last time the TM operated on the block, which is $\text{LocalConfig}(\text{last}(h,i))$.

- Now, observe that now we have the following DAG to compute $\text{LocalConfig}(i)$:



  We can open up the DAG to get a tree and then it is a tree evaluation problem $\mathsf{TEP}_{h,k,d}$. The parameters for TEP instance are $h = O(t/B)$,$d = \#\text{tapes} + 1$ and $k = O(B)$. But, we don't know the tree. So, now we need to find a way to compute the tree.

- Instead of computing the tree, we'll guess the tree. Define the following function:

$$\mathrm{m}(h,i) = \begin{cases} 1 & \text{head } h \text{ moves one block right in epoch } i \\ -1 & \text{head } h \text{ moves one block left in epoch } i \\ 0 & \text{head } h \text{ stays in same block in epoch } i \end{cases}$$

  Now, if we have the value of $\mathrm{m}(h,i)$ for all h and i, we can compute $\text{TapeBlock}(h,i)$ as

$$\begin{aligned}
\text{TapeBlock}(h,i) &= \sum_{j<i} m_{h,j} + 1 \\
\text{last}(h,i) &= \max\{k : \sum_{j \le k} \mathrm{m}(h,j) = i\}
\end{aligned}$$

  Computing $\text{TapeBlock}(h,i)$ and $\text{last}(h,i)$ requires space $O(\log(t/b))$ only. Thus, once we know the value of $\mathrm{m}(h,i)$, we can compute the tree easily

- Now, we need to compute $\mathcal{M} = \{m_{h,i}\}$. Instead of computing $\mathcal{M}$, we can guess $\mathcal{M}$ and then check if the guess is correct. We'll also guess a state sequence $(q_1, q_2, \ldots, q_{t/B}) \in Q^{t/B}$. Also, observe that the leaves will be the leaves corresponding to nodes where some block is being accessed for the first time.

- Now, the only thing we need to check is that the guess is correct. To do this, at each node, we add a check for consistency between consecutive states and heads, if at any point, the guess is inconsistent, we abort. Also, it's easy to see that this check catches all the wrong guess and cab be done in $O(B)$ time.

The above points give us an algorithm in which we guess $\mathcal{M}$ and the state sequence and then check if the guess is correct. During every guess, we need to solve an instance of the tree evaluation problem. Thus, accounting for the total space used, we get the following result:

$$\text{Total Space} = O(t/B) + O(B) + O(\log \frac{t}{B}) + O(B + \frac{t}{B} \log B) = O(B + \frac{t}{B} \log B)$$

Setting $B = \sqrt{t \log t}$, we get:

$$\text{Total Space} = O(\sqrt{t \log t})$$

This completes the proof of theorem.

Lecture 18

Introduction to Circuits

Scribe: Vivek Karunakaran

18.1 Boolean circuits

A Boolean circuit is the directed acyclic graph with n sources (leaves), one unique sink representing the output. The leaves represent variables and constants and the inner nodes are gates - $\wedge, \vee$ *and* $\neg$ and each inner node has 1 or 2 in-degrees depending on the gate. The circuit can hence be viewed as the function $f : \{0,1\}^n \to \{0,1\}$.

18.2 Circuit family

A Circuit family C is the sequence of circuits C_1, C_2, C_3... where C_i is the circuit on i bits and together they compute $f : \{0,1\}^* \to \{0,1\}$. That is, for all $x \in \{0,1\}^*$, we have $f(x) = C_{|x|}(x)$.

Size of the circuit C_i is denoted by $|C_i|$ which is the number of vertices in the circuit.

We say that the circuit family C decides the language L if for all lengths n and $x \in \{0,1\}^n$, we have $x \in L \iff C_n(x) = 1$.

18.3 Circuit size classes

$$\mathsf{SIZE}(S(n)) = \{L : \textit{There exists a circuit family } \{C_i\} \textit{ that computes } L \textit{ and } |C_n| = O(S(n))\}.$$

In words, $\mathsf{SIZE}(S(n))$ refers to the set of languages that are computable by some circuit family of size $S(n)$.

$\mathsf{SIZE}(\text{poly}(n))$ is also called as P/poly (we will justify this notation later).

Note that $\mathsf{P} \subseteq \mathsf{SIZE}(\text{poly}(n))$. This is because the Turing machine that runs in $O(S(n))$ time has $O(S(n))$ configuration sequence which can be encoded as the logical circuit as we have done in Cook-Levin Theorem.

$\mathsf{SIZE}(\text{poly}(n))$ however recognizes some undecidable languages as well. Note that all unary languages are in $\mathsf{SIZE}(\text{poly}(n))$ as each circuit C_n just indicates whether 1^n exists in

the language. We can encode the halting problem as the unary language which then belongs to $\mathsf{SIZE}(\text{poly}(n))$ as well. That is, $L_{\text{UHALT}} = \{1^n : \ M_n \text{ halts when run on a blank input tape}\}$ belongs to $\mathsf{SIZE}(\text{poly}(n))$.

18.4 Size Hierarchy Theorem

Consider the function $f : \{0,1\}^n \to \{0,1\}$. The circuit that can compute this, can be of size at most $2^n \cdot n$. So, Any function can be computed using the circuit family of size $2^n \cdot n$.

Let us calculate the number of circuits of size S. Note that the circuit can be represented as the adjacency list and each node has at most two incoming edges. So, This can be represented as the string of size $O(S \cdot \log S)$. So, Number of circuits of size $S \leq 2^{O(S \cdot \log S)}$. In fact, This does not exceed $2^{10(S \cdot \log S)}$.
Suppose $2^{2^n} > 2^{10(S \cdot \log S)} => 10S \cdot \log S < 2^n => S < 2^n/(10n)$.

So, There are functions f which are computed by the circuits of size $2^n * n$ but not computed by the circuits of size $2^n/(10n)$. Therefore, $\mathsf{SIZE}(2^n/(10n)) \subsetneq \mathsf{SIZE}(2^n \cdot n)$.

One can actually easily extend (left as an exercise) the above observation to show the following:

Theorem 18.1 (Size hierarchy theorem)**.** *For all $s : \mathbb{N} \to \mathbb{N}$ such that $s(n) = \omega(1)$ and $s(n) = o(2^n/n^2)$ we have*

$$\mathsf{SIZE}(s(n)) \subsetneq \mathsf{SIZE}(s(n) \cdot (\log s)^2).$$

18.5 Karp-Lipton-Sipser Theorem

Could it be the case that $\mathsf{SIZE}(\text{poly})$ contains all of NP? The following theorem says that this is quite unlikely.

Theorem 18.2 (Karp-Lipton-Sipser theorem)**.** *If $\mathsf{NP} \subseteq \mathsf{SIZE}(\text{poly})$, then $\mathsf{PH} = \sum_2 \bigcap \prod_2$.*

Proof. The rough idea of the proof is as follows. Assume that $\mathsf{NP} \subseteq \mathsf{SIZE}(\text{poly})$. So, SAT can be computed by a circuit family C of polynomial size. By self-reducibility of SAT, we can get the exact satisfying assignment with polynomial overhead when given access to the circuit family C. Hence, we have circuit family C' that outputs the satisfying assignment given the SAT instance. Note that C' is also of poly size.

Suppose $L \in \prod_2$. So, $x \in L \iff \forall y \exists z M(x,y,z) \iff \forall y \exists z \ \varphi_x(y,z)$. Note that $\exists z \varphi_x(y,z)$ is the NP problem and z is the satisfying assignment for $\varphi_x(y,z)$. We can get the satisfying assignment z using the circuit family C'. That is, $C'_{|z|}(\varphi_x, y) = z$. So, $\forall y \exists z \ : \ \varphi_x(y,z) \iff \forall y \ : \ \varphi_x(y, C'_{|z|}(\varphi_x, y))$. Now we can simply guess the circuit $C'_{|z|}$ itself as it is also of polynomial size. That is, $\forall y \exists z \ : \ \varphi_x(y,z) \iff \exists C'_{|z|} \forall y \ : \ \varphi_x(y, C'_{|z|}(\varphi_x, y))$. So, We got the equivalent Σ_2 statement.

Since we assumed that there does indeed exist a circuit family for SAT, the above formula is true when $x \in L$ since we can instantiate C with the *right* circuit family.

On the other hand, if the Σ_2 statement is true for some instantiation of C, eventhough that may not be the *right* circuit family, it did nevertheless yield a satisfying assignment for $\varphi_x(y)$. Therefore we must have that $x \in L$.

Overall, we have shown that every $L \in \Pi_2$ can also be shown to be in Σ_2. Thus, the polynomial hierarchy collapses to the second level. □

The above proof can infact be generalised to show the following:

Theorem 18.3 (Meyer's theorem)**.** *If* $\mathsf{EXP} \subseteq \mathsf{SIZE}(\text{poly}(n))$*, then* $EXP = \Sigma_2^P$.

18.6 Some common circuit classes

NC^i: A language is in NC^i if there exists a poly size circuit with two fan-ins and $O(\log^i n)$ depth that decides it.
AC^i: A language is in AC^i if there exists a poly size circuit with unbounded fan-ins and and $O(\log^i n)$ depth that decides it.
Since unbounded polynomial fan-in can be simulated using a tree of ORs/ANDs of depth $O(\log n)$, $\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$.

Lecture 19

What AC^0 can and cannot do

Scribe: Aindrila Rakshit

Topics covered in this lecture

1. $\text{ADD}_2 \in \mathsf{AC}^0$
2. $\text{Parity}_n \notin \mathsf{AC}^0$

In this lecture, we saw what the complexity class AC^0 (constant depth, polynomial size, unbounded fan-in) can and cannot do. Recall that the size of a circuit refers to the number of gates in it, the depth corresponds to the length of the longest path from an input gate to the output gate and the fan-in of a gate is the number of input wires coming into it.

19.1 Some functions in AC^0

Obvious functions in AC^0: OR, AND, CNF. We will first see a slightly surprising function in AC^0, namely the task of adding two n-bit numbers.

ADD_2: Adds two binary numbers, i.e. $\text{ADD}_2 : (X, Y) \to X + Y$, where X, Y are numbers in binary.

Addition of two numbers at first glance seems to be not in AC^0, since it appears to require a sequential operation wherein the i^{th} bit of both X and Y are added to find both the i^{th} bit of the output and the carry needed for the addition of the $(i+1)^{th}$ bits. The circuit made using this sequential algorithm would have $O(n)$ depth.

Proposition 19.1. $\text{ADD}_2 \in \mathsf{AC}^0$.

Proof. We will construct an AC^0 circuit for ADD_2 using a *carry-lookahead generator*. Define for each bit position i:

$g_i = X_i \wedge Y_i$ (generate a carry at position i if $X_i = Y_i = 1$)
$p_i = X_i \vee Y_i$ (propagate a carry through position i if at least one of X_i or Y_i is 1)

C_i is the carry coming into position i

Assume $C_0 = 0$ is the initial carry.
The carry into position i is 1 iff:

- Some lower position $j < i$ generated a carry ($g_j = 1$), and
- All bits from $j+1$ to $i-1$ propagate the carry ($p_k = 1$ for $j+1 \leq k \leq i-1$)

Then, for $i = 1, \ldots, n$:

$$C_i = \bigvee_{j=0}^{i-1} \left(g_j \wedge \bigwedge_{k=j+1}^{i-1} p_k \right)$$

The i^{th} bit of the output can now be easily computed from X_i, Y_i and C_i giving an AC^0 circuit for addition. □

Turns out, we can infact add $O(\log n)$ numbers in AC^0 as well (this is a non-trivial theorem).

Theorem 19.2. $\mathrm{ADD}_{\log n} \in \mathsf{AC}^0$.

Other surprising functions in AC^0 are "small thresholds".

Theorem 19.3. $\mathrm{Th}_{\log n} \in \mathsf{AC}^0$, *where* $\mathrm{Th}_{\log n}$ *is the task of checking if the number of ones in the input is at least* $\log n$.

19.2 $\mathrm{Parity}_n \notin \mathsf{AC}^0$

We now move on to showing a function that *cannot* be computed by AC^0 circuits. This was first proved by Furst-Saxe-Sipser and there have been many subsequent proofs (by Håstad, Razborov, Smolensky, etc.). We will see the proof by Razborov and Smolensky.

Parity_n: Computes the parity of the n input bits.

Theorem 19.4 (Razborov-Smolensky)**.** $\mathrm{Parity}_n \notin \mathsf{AC}^0$*: Any depth* Δ *circuit computing PARITY must have size* $\geq 2^{\Omega(n^{1/2\Delta})}$.

Håstad proved the *optimal* lower bound for Parity: Any depth Δ circuit computing Parity_n must have size $\geq 2^{\Omega(n^{1/\Delta-1})}$.

Intuition: Parity_n is a sensitive function, changing any input bit changes the output.

To prove the theorem, we will rely on the classic method of approximating a function $f : \{0,1\}^n \to \{0,1\}$ by a polynomial. E.g., the function $OR(x_1, \ldots, x_n)$ is computed by the polynomial $1 - \prod_{i=1}^{n}(1 - x_i)$ of degree n, but we would want it to be described by a polynomial of degree less than n. It can be approximated by the polynomial which is identically 1 on all inputs, and thus is correct on every input except on the one where all x_i's are 0's. Similarly, the function $AND(x_1, \ldots, x_n) = \prod_{i=1}^{n}(x_i) \approx 0$, which is correct on every input except on the

one where all x_i's are 1's. Given any circuit C which is polynomially approximated by f, $\neg C$ can be polynomially approximated by $1 - f$.

But instead of asking for the polynomial to be correct on "most" inputs, we will ask for the stronger requirement that we want to be right on *every* input with high probability. For this, it would be imperative that the polynomial is chosen from some distribution (what do we take the probability over otherwise?). This leads us to the following notion of approximating polynomials (which is actually a *distribution* on polynomials).

Definition 19.5 (Approximating polynomials)**.** *Approximating polynomial is a distribution $\mathcal{D}$ on a set of polynomials over $\mathbb{F}$ that ε-approximates a Boolean function $f : \{0,1\}^n \to \{0,1\}$ if for all $X \in \{0,1\}^n$:*

$$\Pr_{P \sim \mathcal{D}}[P(X) \neq f(X)] \leq \varepsilon.$$

In words, for every input x, if P is picked according to the distribution then $P(X)$ agrees with $f(x)$ with probability at least $1 - \varepsilon$.

We shall say that this approximating polynomial has degree d if every P in the support of $\mathcal{D}$ has degree at most d. ◊

Here, we will be working over the field $\mathbb{F}_3$ consisting of three elements $\{0,1,2\}$ (or $\{0,1,-1\}$).

The main theorem would follow from the following two lemmas:

Lemma 19.6. *If f is computable by a size s, depth Δ circuit, then for any ε there is an ε-approximating polynomial of degree $\leq O((\log s/\varepsilon)^{2\Delta})$.*

Lemma 19.7. *Suppose $\mathcal{D}$ is an approximating polynomial family that 0.1-approximates Parity_n, then degree of $\mathcal{P} \geq \sqrt{n}/100$.*

This would immediately yield that $(10 \log s)^{2\Delta} = \sqrt{n}/100$ which implies that $s = 2^{\Omega(n^{1/2\Delta})}$.

19.2.1 Approximating polynomial for $\text{OR}(X_1, \ldots, X_n)$

We begin by providing a 2/3-approximating polynomial for OR (fan-in n).

The sampling process for the polynomials would be by choosing some elements $r_i \in \mathbb{F}$ uniformly at random.

$$P_{\bar{r}}(\bar{x}) = (r_1 x_1 + \ldots + r_n x_n)^2, \text{ where } r_i \in_R \mathbb{F}$$

If all $x_i = 0$, then $P_{\bar{r}}(\bar{x}) = 0$ irrespective of $\bar{r}$, so it correctly computes OR with probability 1.

Observation 19.8. *If some $\text{OR}(\bar{x}) = 1$, then $Pr_{\bar{r} \in_R \mathbb{F}}[r_1 x_1 + \ldots + r_n x_n = 0] = 1/3$.*

In particular, for all $\bar{x}$, $Pr_{\bar{r} \in_R \mathbb{F}}[(r_1 x_1 + \ldots + r_n x_n)^2 = \text{OR}(\bar{x})] \geq 2/3$.

Amplifying the success: This can be amplified by taking t samples of $\bar{r}$ instead of 1.

$$P_{\bar{r_1}, \ldots, \bar{r_t}}(\bar{x}) = 1 - \prod_{i=1}^{t} (1 - P_{\bar{r_i}}(\bar{x})$$

(each $\overline{r_i} \in \mathbb{F}_3^n$)

Observation 19.9. *$P_{\overline{r_1},\ldots,\overline{r_t}}(\overline{x})$ approximates $\mathrm{OR}(x_1,\ldots,x_n)$ with error $\varepsilon \leq (1/3)^t$.*

Note that $\deg(P_{\overline{r_1},\ldots,\overline{r_t}}(\overline{x})) = 2t$.

Handling NOT gates NOT gates have fixed fan in 1. It is easy to see that if P is an approximating polynomial for f, then $1 - P$ is an approximating polynomial for $\neg f$

19.2.2 Approximating polynomial for $\mathrm{AND}(x_1,\ldots,x_n)$

Using De Morgan's law, we know that $\mathrm{AND}(x_1,\ldots,x_n) = \neg\,\mathrm{OR}(\neg x_1,\ldots,\neg x_n)$. Therefore, the following is the natural polynomial that approximates $\mathrm{AND}(x_1,\ldots,x_n)$:

$$Q_{\overline{r}}(\overline{X}) = 1 - P_{\overline{r}}(1 - x_1,\ldots,1 - x_n).$$

Thus, AND can be $(1/3)^t$ approximated by the degree $2t$ polynomial $Q_{r_1,\ldots,r_t}$.

19.2.3 Composing approximating polynomials

Suppose we now have a circuit that is an AND of OR's, we can just compose the polynomials for the corresponding gates.

$$T(\overline{x}) = Q_{\overline{r}}(P_{\overline{r}}(\overline{x}),\ldots,P_{\overline{r}}(\overline{x}))$$

Observation 19.10. *Error of $T(\overline{x})) \leq$ number of gates $\times$ error at any gate*

To approximate an depth Δ circuit, use the same construction for OR, AND, NOT and compose $Q'(\overline{x})$. If we have s gates, the total error we may incur (if each P or Q has error $(1/3)^t$) is bounded by $s \cdot (1/3)^t$. To ensure that this is at most ε, we can instantiate $t = O(\log(s/\varepsilon))$.

Furthermore, if we have a circuit of depth Δ, then the degree of T is at most $(O(\log s/\varepsilon))^{2\Delta}$.

This completes the proof of Lemma 19.6. We will see the proof of Lemma 19.7 in the next lecture.

Lecture 20

Randomized Complexity Classes

Scribe: Nishant Das

20.1 Randomized Computation

Recall nondeterministic computation, where the machine has two transition functions Γ_0 and Γ_1. At each step, it either *guesses* which path to take or *follows both* transitions simultaneously. The machine accepts if at least one computational path leads to acceptance (depending on your view of nondeterministic Turing machines).

In **randomized computation**, instead of guessing, the machine chooses between Γ_0 and Γ_1 *uniformly at random*. Thus, each computational path has a probability associated with it.

We now ask: what is the probability that the machine reaches an accepting state?

Alternatively, we can model randomized computation as having access to an additional *random tape* filled with independent random bits. The machine then determines its transition based on this random tape.

20.2 Randomized Complexity Classes

Depending on the acceptance criteria, we define different complexity classes:

Definition 20.1 (Bounder-error probabilistic polynomial time (BPP))**.** *Let* $0 < s < c < 1$*. Then*

$$\mathsf{BPP}_{c,s} = \left\{ \begin{array}{l} L \mid \exists \textit{ a randomized polynomial-time machine } M \textit{ such that:} \\ x \in L \Rightarrow \Pr[M(x,r) \textit{ accepts}] \geq c \\ x \notin L \Rightarrow \Pr[M(x,r) \textit{ accepts}] \leq s \end{array} \right\}$$

$\Diamond$

Definition 20.2 (Randomised polynomial time (RP))**.** *Let* $0 < c < 1$*. Then*

$$\mathsf{RP}_{c} = \left\{ \begin{array}{l} L \mid \exists \textit{ a randomized polynomial-time machine } M \textit{ such that:} \\ x \in L \Rightarrow \Pr[M(x,r) \textit{ accepts}] \geq c \\ x \notin L \Rightarrow \Pr[M(x,r) \textit{ accepts}] = 0 \end{array} \right\}$$

◊

Definition 20.3 (co-randomized polynomial time (coRP))**.** *Let* $0 < s < 1$*. Then*

$$\mathsf{coRP}_s = \left\{ \begin{array}{l} L \mid \exists \textit{ a randomized polynomial-time machine } M \textit{ such that:} \\ x \in L \Rightarrow \Pr[M(x,r) \textit{ accepts}] = 1 \\ x \notin L \Rightarrow \Pr[M(x,r) \textit{ accepts}] \leq s \end{array} \right\}$$

◊

Inclusion Relations

- $\mathsf{P} \subseteq \mathsf{BPP} \cap \mathsf{RP} \cap coRP \subseteq \mathsf{PSPACE}$
- $\mathsf{RP} \subseteq \mathsf{NP}$, since NP only requires one accepting path
- $\mathsf{coRP} \subseteq \mathsf{coNP}$

Conjecture: The prevailing belief in the community is that $\mathsf{BPP} = \mathsf{P}$ and we will see why towards the end of the course. .

20.3 Error Reduction

Randomized algorithms allow a small probability of error, but we can reduce this error exponentially through repetition.

Error Reduction in RP

Recall: $L \in \mathsf{RP}_c$ if there exists a randomized poly-time machine M such that:

- If $x \in L$, then $\Pr[M(x) \text{ accepts}] \geq c$
- If $x \notin L$, then $\Pr[M(x) \text{ accepts}] = 0$

To reduce error, define M' that runs M independently t times and accepts if *any* run accepts.

- If $x \in L$, success probability is at least $1 - (1-c)^t$
- If $x \notin L$, all runs reject

To achieve error δ, it suffices to set $t = \mathcal{O}(\frac{1}{c} \log \frac{1}{\delta})$. Hence, $\mathsf{RP}_{\frac{1}{2}} = \mathsf{RP}_{1-\frac{1}{2^{n^c}}}$ for any constant c, and we simply refer to the class as RP.

Error Reduction in BPP

In contrast to RP, the class $\mathsf{BPP}_{c,s}$ allows two-sided error:

- If $x \in L$, $\Pr[M(x) \text{ accepts}] \geq c$
- If $x \notin L$, $\Pr[M(x) \text{ accepts}] \leq s$, with $c > s$

We reduce error by repeating M independently t times and accepting if the majority of runs accept. Let $X_i \in \{0,1\}$ be indicators for acceptance in the i-th run, and define $X = \sum X_i$. The expected value $\mu = \mathbb{E}[X]$ will differ depending on whether $x \in L$ or $x \notin L$. To bound the probability that the majority vote is wrong, we use the Chernoff bound:

$$\Pr\left[|X - \mu| \geq \varepsilon\mu\right] \leq \exp\left(-\frac{\varepsilon^2\mu}{3}\right)$$

For instance, suppose $c = \frac{2}{3}$, $s = \frac{1}{3}$, and t is the number of repetitions. Then:

- If $x \in L$, $\mu \geq \frac{2t}{3}$, and error occurs if $X < \frac{t}{2}$, i.e., deviation $\geq \frac{t}{6}$
- By Chernoff, this error is $\leq \exp\left(-\frac{t}{24}\right)$

To reduce error to δ, we need:

$$t = \mathcal{O}\left(\frac{1}{(c-s)^2}\log\frac{1}{\delta}\right)$$

Hence, even with a very small initial advantage (e.g., $c = \frac{1}{2} + \frac{1}{n}$), we can amplify the gap to reach extremely small error like $1/2^{n^c}$. Therefore:

$$\mathsf{BPP}_{\frac{1}{2}+\frac{1}{\text{poly}(n)},\frac{1}{2}-\frac{1}{\text{poly}(n)}} = \mathsf{BPP}_{1-\frac{1}{2^{n^c}},\frac{1}{2^{n^c}}} = \mathsf{BPP}$$

Lecture 21

More on randomised classes

Scribe: Soumyadeep Paul

21.1 Zero-error probability polynomial time (ZPP)

Definition 21.1 (ZPP)**.** ZPP *is the class of all such languages which can be accepted by a randomised Turing Machine such that it makes no mistake in its output and the expected runtime is polynomial in the size of input.* ◊

Lemma 21.2. $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$.

Proof. ($\Rightarrow$)

- $\mathsf{ZPP} \subseteq \mathsf{RP}$.

  Let M be a ZPP machine for L with expected runtime $f(n)$.

  We make a new machine M' which simulates M for $100f(n)$ steps. If it gets an output from M, M' also outputs that. Otherwise, it outputs *reject*.

- $\mathsf{ZPP} \subseteq \mathsf{coRP}$.

  Let M be a ZPP machine for L with expected runtime $f(n)$.

  We make a new machine M' which simulates M for $100f(n)$ steps. If it gets an output from M, M' also outputs that. Otherwise, it outputs *accept*.

  The correctness of the algorithms can be seen by using the Markov inequality, M makes an error only when it took more than $100f(n)$ steps. The probability of this happening is less than $\frac{1}{100}$.

  ($\Leftarrow$) To show the other direction, let M_1 be an RP machine for L and M_2 be a coRP machine for L. (We assume that the probability of error is $\frac{1}{2}$ and the runtime of both of them is $f(n)$).

  We make a new machine M that does the following:

  – Run M_1 and M_2 on x.

- If M_1 accepts, then we accept.
- If M_2 rejects, we reject.
- Otherwise, we just try again.

In other words, the machine will continue to run until either M_1 ends up accepting x, or M_2 rejects x (and in either case, we know our answer is correct).

The expected runtime of this machine would be

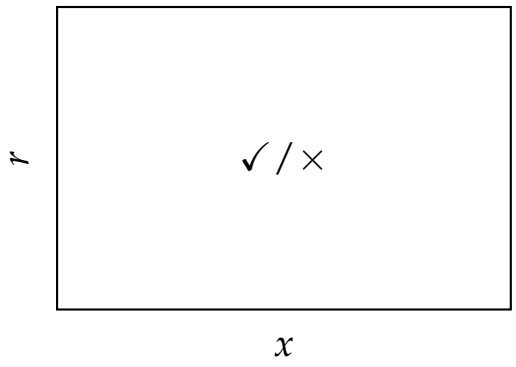
$$\mathbb{E}[\text{runtime}] \leq f(n) + 2f(n)\frac{1}{2} + 3f(n)\frac{1}{4} + \cdots = O(f(n))$$

□

21.2 BPP and circuits

Theorem 21.3 (Adleman)**.** $\mathsf{BPP} \subseteq \mathsf{SIZE}(\text{poly})$.

Proof. We consider the table of all possible pairs of inputs x and random strings r used by the BPP machine. We mark the index with a ✓ if the machine on the random bits r, computed x correctly and we mark it with a $\times$ otherwise.



Let $\mathbb{P}[\text{error}] \leq \delta$ and, say, the BPP machine uses m random bits and the input size be n.
By the probability of error,

$$\text{Each column has } \leq \delta 2^m \text{ crosses}$$

Therefore, number of crosses $\leq \delta 2^m 2^n$.
If each of the rows had a cross,

$$\text{Number of crosses } \geq 2^m$$

Therefore, if we set $\delta < \frac{1}{2^n}$, we would have a row with all checkmarks, that is, a string r for which the machine is correct on all inputs of size n. We can hardwire this random string to get a circuit.

Remark: This string can be hardcoded into the circuit and only depends on the size of the input but might be hard to find. □

21.2.1 Machines with advice

$L \in \mathcal{C}/n^2$ if there is a $\mathcal{C}-$machine M and a sequence $\{z_i\}_{i\in\mathbb{N}}$, $z_i \in \{0,1\}^*, \forall i$ such that

$$x \in L \Leftrightarrow M(x, z_{|x|}) \text{accepts}$$

and $\text{len}(z_i) = O(n^2)$.

Therefore, $\mathsf{P}/_{\text{poly}} = \mathsf{SIZE}(\text{poly})$.

Lecture 22

Relation of BPP with Other Classes

Scribe: Soham Chatterjee

22.1 BPP in Polynomial Hierarchy

Theorem 22.1 (Gacs-Sipser, Lautteman)**.** $\mathsf{BPP} \subseteq \Sigma_2 \cap \Pi_2$

Proof. Suppose $L \in \mathsf{BPP}$ and let M is a randomized machine for L. On input x suppose M uses m random bits and δ be the error. We'll choose δ later as required. Now consider the set $S = \{r \colon M(x,r) = \text{Accept}\}$. Now

$$x \in \implies |S| \geq (1-\delta)\cdot 2^m \qquad \text{and} \qquad x \notin L \implies |S| \leq \delta \cdot 2^m$$

Since if $x \in L$ then size of S is very large if we shift all the elements of S by some $a \in \{0,1\}^m$. Let we denote the set $\{b \oplus a \mid b \in a\}$ by $S \oplus a$. Then with very few shifts we can cover all the $\{0,1\}^m$ i.e. $\exists\, t \in \{0,1\}^{\mathsf{poly}(n)}$ and $\exists\, a_1, \ldots, a_t \in \{0,1\}^m$ such that

$$\{0,1\}^n = \bigcup_{i=1}^{t} S \oplus a_i$$

Therefore we can construct the following quantified boolean formula

$$\exists\, a_1, \ldots, a_t \in \{0,1\}^m \,\forall\, y \in \{0,1\}^t \bigvee_{i=1}^{t} (M(x, a_i + y) = \text{Accept})$$

Let the boolean formula $\Psi(x) := \bigvee_{i=1}^{t} (M(x, a_i + y) = \text{Accept})$. Then our gaol is to show $\Psi(x) = \text{True} \iff x \in L$.

Suppose $x \notin L$. Then we should have $\Psi(x) = \text{False}$. Hence we should have $\bigcup_{i=1}^{t} S \oplus a_i \subsetneq \{0,1\}^m$. Now

$$\left| \bigcup_{i=1}^{t} S \oplus a_i \right| \leq t \cdot \delta \cdot 2^m$$

Hence if we choose δ such that $t \cdot \delta \cdot 2^m < 2^m \iff t \cdot \delta < 1$ we are done. So it suffices to have $\boxed{t \cdot \delta < 1}$

Suppose $x \in L$. Then we should get $\Psi(x) = \text{True}$. Now,

$$\begin{aligned} x \in L &\implies \mathop{\mathbb{P}}_{a}[a + y \notin S] \leq \delta \qquad [\text{Since } |S| \geq (1-\delta)2^m \text{ and } a+y \text{ is uniform}] \\ &\implies \mathop{\mathbb{P}}_{a_1,\ldots,a_t}[a_i + y \notin S,\ \forall\, i \in [t]] \leq \delta^t \\ &\implies \mathop{\mathbb{P}}_{a_1,\ldots,a_t}[a_i + y \notin S,\ \forall\, i \in [t]][\exists\, y \colon a_i + y \notin S,\ \forall\, i \in [t]] \leq 2^m \delta^t \end{aligned}$$

Hence it suffices to have $\boxed{2^m \cdot \delta < 1}$.

So if we choose $\delta = 2^{-n}$ and $t = \frac{m}{n} + 1$ then both the inequalities are satisfied. Hence we got $\mathsf{BPP} \subseteq \Sigma_2$. Since BPP is closed under complementation and complement of Σ_2 is Π_2 we also have $\mathsf{BPP} \subseteq \Pi_2$. Therefore we have $\mathsf{BPP} \subseteq \Sigma_2 \cap \Pi_2$. □

22.2 Randomized Analogue of NP

We can think that NP is basically P with a $\exists$ quantifier before it, i.e. NP essentially $\exists$P. Similarly we can think of BPP as BP · P where it means there is a BP quantifier before P which introduces the randomized computation. So $L \in \mathsf{BP} \cdot \mathsf{P}$ if

$$x \in L \implies \mathop{\mathbb{P}}_{r}[M(x,r) = \text{Accept}] \geq \frac{2}{3} \quad \text{and} \quad x \notin L \implies \mathop{\mathbb{P}}_{r}[M(x,r) = \text{Accept}] \leq \frac{1}{3}$$

So in this view we can also define $\mathsf{BP} \cdot \exists\mathsf{P}$ and $\exists \cdot \mathsf{BPP}$.

Definition 22.2 ($\mathsf{BP} \cdot \exists\mathsf{P}$)**.** *A language $L \subseteq \Sigma^*$, $L \in \mathsf{BP} \cdot \exists\mathsf{P}$ if there is a deterministic turing machine M such that*

$$\begin{aligned} x \in L &\implies \mathop{\mathbb{P}}_{r}\left[\exists\, w \in \{0,1\}^{\mathit{poly}(|x|)},\ M(x,w,r) = \mathit{Accept}\right] \geq \frac{2}{3} \\ x \notin L &\implies \mathop{\mathbb{P}}_{r}\left[\exists\, w \in \{0,1\}^{\mathit{poly}(|x|)},\ M(x,w,r) = \mathit{Accept}\right] \leq \frac{1}{3} \end{aligned}$$

◊

Definition 22.3 ($\exists \cdot \mathsf{BPP}$)**.** *A language $L \subseteq \Sigma^*$, $L \in \exists \cdot \mathsf{BPP}$ if there is a deterministic turing machine M such that*

$$\begin{aligned} x \in L &\implies \exists\, w \in \{0,1\}^{\mathit{poly}(|x|)} \mathop{\mathbb{P}}_{r}[M(x,w,r) = \mathit{Accept}] \geq \frac{2}{3} \\ x \notin L &\implies \forall\, w \in \{0,1\}^{\mathit{poly}(|x|)} \mathop{\mathbb{P}}_{r}[M(x,w,r) = \mathit{Accept}] \leq \frac{1}{3} \end{aligned}$$

◊

Remark 22.4. *The class $\exists \cdot \mathsf{BPP}$ is also known as* MA *and $\mathsf{BP} \cdot \exists\mathsf{P}$ also known as* AM ◊

Consider a matrix where the rows are indexed by all possible n length input strings and the columns are indexed by the all possible random strings of length $\mathsf{poly}(n)$. At (w, x) index of the matrix there is a 1 if w gets accepted on the random string r and 0 otherwise.

Then $L \in \mathsf{MA}$ means if $x \in L$ there exists a row where most entries have 1 and $x \notin L$ means all the rows have very few entries with 1.

Similarly $L \in \mathsf{AM}$ means if $x \in L$ most of the columns have an entries with 1 and $x \notin L$ means very few columns have an entry 1.

Lecture 23

GraphNonIso $\in$ AM

Scribe: Shubham A. Bhardwaj

In this lecture, we explore the complexity class AM (Arthur-Merlin) and the problem of graph non-isomorphism (GraphNonIso). We will show that GraphNonIso is in AM and discuss the implications of this result.

The graph non-isomorphism problem (GraphNonIso) is the problem of determining whether two given graphs G_1 and G_2 are not isomorphic. Two graphs are said to be isomorphic if there exists a bijection between their vertex sets that preserves adjacency. Formally, G_1 and G_2 are isomorphic if there exists a permutation π of the vertices such that $(u, v) \in E(G_1)$ if and only if $(\pi(u), \pi(v)) \in E(G_2)$.

23.1 Private-Coin protocol for GraphNonIso

There exists a simple private-coin protocol for GraphNonIso. Let the input graphs be G_1 and G_2, both on n vertices. The protocol proceeds as follows:

1. Arthur randomly selects 2 'bits' $b_1, b_2 \in \{1, 2\}$ and 2 random permutation π_1, π_2 of the vertices of G_{b_1} and G_{b_2}.
2. Arthur sends the permuted graph $\pi_1(G_{b_1})$ and $\pi_2(G_{b_2})$ to Merlin.
3. Merlin, who knows G_1 and G_2, must determine whether the received graphs corresponds to G_1 or G_2. Merlin sends back a guess (b_1', b_2') to Arthur.
4. Arthur accepts if and only if $(b_1', b_2') = (b_1, b_2)$.

Observe that the above protocol works since if the graphs are non-isomorphic, then merlin can determine whether $\pi_i(G_{b_i})$ are isomorphic to G_1 or G_2 and then can return the correct bits. If the graphs are isomorphic, then merlin has no way to figure out which graph Arthur chose and thus at best can return a random guess. Thus, the probability of Arthur accepting is $1/4$ in this case.

23.2 Public-Coin protocol for GraphNonIso

The above protocol does not work if the random bits selected by Arthur are public. Thus, we need to modify the protocol to work with public coins.

Consider the following set:

$$S = \{(H, \sigma) | \, H \cong G_1 \text{ or } H \cong G_2 \text{ and } \sigma \in \text{aut}(H)\}.$$

Observe that if $G_1 \cong G_2$, then $|S| = n!$ and if $G_1 \ncong G_2$, then $|S| = 2(n!)$. Also, note that membership in S can be certified in polynomial time.

Let $S \subseteq \{0,1\}^m$ and $N = 2(n!)$ and k be such that $2^{k-2} \leq N < 2^{k-1}$. Let $p = N/2^k$.

Let $\mathcal{H}$ be the family of all functions from $\{0,1\}^m$ to $\{0,1\}^k$. Now, suppose Arthur sends a random function h from $\mathcal{H}$ and a random string from $\{0,1\}^k$ to Merlin. Merlin is now supposed to return z such that $h(z) = y$ and a certificate $c(z)$ to certify that $x \in S$. Arthur accepts if and only if $h(z) = y$ and $c(z)$ is a valid certificate for z.

The following lemmas will help us prove that the above protocol works.

Lemma 23.1. *If $|S| = N/2$, then for any $y \in \{0,1\}^k$, we have*

$$\Pr_{h \in \mathcal{H}}[\exists z \in S,\ h(z) = y] \leq \frac{1}{2}p$$

Proof. Since $|S| = N/2$, the size of the image of S under h is at most $|S|$. Thus, the probability that y is an element of this image is at most $\frac{N/2}{2^k} = \frac{p}{2}$. □

Lemma 23.2. *If $|S| = N$, then for any $y \in \{0,1\}^k$, we have*

$$\Pr_{h \in \mathcal{H}}[\exists z \in S,\ h(z) = y] \geq \frac{3}{4}p$$

Proof. Using the lower bound from principle of inclusion-exclusion, we have

$$\begin{aligned}
\Pr_{h \in \mathcal{H}}[\exists z \in S,\ h(z) = y] &\geq \sum_{z \in S} \Pr_{h \in \mathcal{H}}[h(z) = y] - \sum_{z \neq z'} \Pr_{h \in \mathcal{H}}[h(z) = y \wedge h(z') = y] \\
&= \sum_{z \in S} \frac{1}{2^k} - \sum_{z \neq z'} \frac{1}{2^{2k}} \\
&= \frac{|S|}{2^k} - \frac{\binom{|S|}{2}}{2^{2k}} \\
&\geq \frac{|S|}{2^k} - \frac{|S|^2}{2^{2k+1}} \\
&= p\left(1 - \frac{p}{2}\right) \geq \frac{3}{4}p \qquad \text{since } p \leq \frac{1}{2}
\end{aligned}$$

□

Thus, above lemmas implies:

$$\Pr[\text{Arthur accepts}] = \begin{cases} \geq \frac{3}{4}p & \text{if } G_1 \cong G_2 \\ \leq \frac{1}{2}p & \text{if } G_1 \ncong G_2 \end{cases}$$

But there's a problem with this approach. To send a random hash function h from $\mathcal{H}$, Arthur needs to send full description of h to Merlin. This is not possible since $|\mathcal{H}| = 2^{k2^m}$ and hence even the size of the *description* of h is exponential in m. But observe that we only used the following two properties of $\mathcal{H}$:

- For any $z \in \{0,1\}^m$ and $y \in \{0,1\}^k$, we have $\Pr_{h\in\mathcal{H}}[h(z) = y] = \frac{1}{2^k}$.
- For any $z, z' \in \{0,1\}^m$ with $z \neq z'$, and $y, y' \in \{0,1\}^k$ we have

$$\Pr_{h\in\mathcal{H}}[h(z) = y \wedge h(z') = y'] = \frac{1}{2^{2k}}.$$

The above two properties are called *pairwise independence* and any family $\mathcal{H}$ that satisfies the above two properties is called a *pairwise independent hash family*. Fortunately, there exists succinct families of hash functions that are pairwise independent. Consider the following family of hash functions:

$$\mathcal{G} = \{h_{A,b} : \{0,1\}^m \to \{0,1\}^k |\ A \in \{0,1\}^{k\times m}, b \in \{0,1\}^k \text{ and } h_{A,b}(x) = (Ax + b) \bmod 2\}$$

It us easy to see that the above family of hash functions is pairwise independent. Also, to send a hash function from $\mathcal{G}$, Arthur only needs to send A and b which is of size $O(km)$. Thus, with this family of hash functions, the above protocol works. Also note that the probability of error can be amplified by sending multiple hash functions and taking the majority vote. Thus, we get the following theorem:

Theorem 23.3 (Goldwasser-Sipser)**.** GraphNonIso $\in$ AM.

Lecture 24

Pseudorandomness

Scribe: Vivek Karunakaran

Randomness is a powerful tool in computation. For instance, finding the MAXCUT in a graph is an NP-hard problem. However, we can obtain a $\frac{1}{2}$-approximation using a simple randomized algorithm: Given a graph $G = (V, E)$, assign each vertex independently and uniformly at random to either V_1 or V_2. This process yields a cut whose expected value is at least $\frac{1}{2}$ the number of edges (and hence at least $\frac{1}{2}$ the MAXCUT).

The central idea of pseudorandomness is to generate strings using deterministic algorithms that "look" random.

24.1 Pseudorandom Generator

Suppose we have randomized algorithm $A(x, y)$ where x is the input to the algorithm and y is the random bits It uses, $y = (r_1, r_2, ..., r_m)$.

We call $S \subseteq \{0,1\}^m$ is ε-pseudorandom for A if for any x, we have

$$|\Pr_{\bar{r}}[A(x, \bar{r}) = \text{accept}] - \Pr_{\bar{r} \in S}[A(x, \bar{r}) = \text{accept}]| \leq \varepsilon.$$

Definition 24.1 (Pseudorandom generators (PRGs) for size s circuits)**.** *A map $G : \{0,1\}^\ell \to \{0,1\}^m$ is ε-PRG for size s circuits on m bits, if for every circuit C of size s, we have*

$$\left| \Pr_{x \in \{0,1\}^m}[C(x) = 1] - \Pr_{z \in \{0,1\}^\ell}[C(G(z)) = 1] \right| \leq \varepsilon.$$

The PRG is said to be efficient if $G(y_1, \ldots, y_\ell)$ can be computed in $\text{poly}(\ell, m)$ *time.* $\Diamond$

Note that if we have an efficient PRG, $G : \{0,1\}^\ell \to \{0,1\}^m$, then, we can find the value of $\Pr_{x \in \{0,1\}^m}[C(x) = 1]$ approximately (up to an additive error of ε) $poly(s, 2^\ell)$ time (by simply running over all $G(y_1, \ldots, y_\ell)$).

The main question is of course, are there such pseudorandom generators?

24.2 PRGs from hardness assumptions

The following theorem (that we will not prove in this course) shows that PRGs exist under some very believable hardness assumptions.

Theorem 24.2 (Impagliazzo-Wigderson)**.** *Suppose there exists a language $L \in \mathsf{E} = \mathsf{DTIME}(2^{O(n)})$ and a constant $\varepsilon > 0$ such that $L \notin \mathsf{SIZE}(2^{\varepsilon n})$, then $\mathsf{P} = \mathsf{BPP}$.*

That is, unless every language computable in $2^{O(n)}$ time is also computable by sub-exponential-sized ($2^{o(n)}$) circuits families, $\mathsf{P} = \mathsf{BPP}$.

The above theorem builds over a differnet theorem of Nisan and Wigderson that we will see in this lecture and next.

Definition 24.3 ((s,ε)-average-case hard function)**.** *A boolean function $f : \{0,1\}^n \to \{0,1\}$ is (s,ε)-average-case hard if for every circuit C of size s, we have*

$$\left|Pr_{x\in\{0,1\}^n}[C(x) = f(x)] - 1/2\right| \leq \varepsilon.$$

That is, no circuit of size s does much better at than blindly guessing to find the value of f at a random input x. ♢

Theorem 24.4 (Nisan and Wigderson (informal))**.** *Given a family $\{f_n\}$ of boolean functions that is average-case hard, we can use $\{f_n\}$ to build PRG's (whose seed-length will depend on the hardness provided).*

We will see a formal statement of the theorem in the next lecture but to begin with, let us try to construct some non-trivial PRG.

Let us define a generator G as follows (albeit stretching just one additional bit)

$$G(z_1, z_2, ..., z_\ell) = (z_1, z_2, ..., z_\ell, f_\ell(z)) \text{ where } z = (z_1, z_2, ..., z_\ell).$$

Claim 24.5. *If f_ℓ is (s,ε)-average-case-hard, then G is indeed an ε-PRG for size $s/2$ circuits.*

Proof sketch. Suppose G is not a PRG. Then, there exists a circuit of size s that distinguishes the output of G from uniform. That is,

$$|\Pr[C(z_1, z_2, ..., z_\ell, z_{\ell+1}) = \text{accept}] - \Pr[C(z_1, z_2, ..., z_\ell, f_\ell(z)) = \text{accept}]| > \varepsilon$$

Without loss of generality, we may assume that

$$\Pr[C(z_1, z_2, ..., z_\ell, z_{\ell+1}) = \text{accept}] - \Pr[C(z_1, z_2, ..., z_\ell, f_\ell(z)) = \text{accept}] > \varepsilon$$

by replacing C with $\neg C$ if required.

Let us build a circuit $D : \{0,1\}^n \to \{0,1\}$ that guesses the value of f_ℓ significantly better than random-guessing.

D **on input** $(z_1, \ldots, z_\ell)$**:**

- Compute $b_0 = C(z_1, z_2, ..., z_\ell, 0)$ and $b_1 = C(z_1, \ldots, z_\ell, 1)$.
- If $b_0 = b_1$, then just guess the output randomly.
- Else, output i such that $b_i = 0$. (That is, output the value of $z_{\ell+1}$ that resulted in the circuit rejecting)

It is not hard to see that the circuit D guesses the value of f correctly on at least $1/2 + \varepsilon$ fraction of inputs, contradicting the assumption that f_ℓ is (s, ε) average-case hard. Although the above is a randomized circuit, we can freeze random bits to obtain a standard circuit with the same guarantee.

Therefore, G is indeed a PRG for size $s/2$ circuits. □

The above is a generator that just provides one additional random bit. But we will see how to generate way more random bits and thereby create a generator with a large stretch. We will see that in the next lecture.

Lecture 25

Nisan-Wigderson pseudorandom generators

Scribe: Aindrila Rakshit

Topics covered in this lecture

1. Nisan-Wigderson theorem: PRGs from average-case hard functions
2. Average-case hardness, combinatorial designs

25.1 The Nisan-Wigderson theorem

For the sake of simplicity, we will refer to the following notion of hardness (which is essentially Definition 24.3 but with $\varepsilon = 1/s$).

Definition 25.1 (s-hardness)**.** *A function $f : 0,1^n \to 0,1$ is s-hard to guess if for every circuit of size at most s, satisfies $Pr_x[C(x) = f(x)] - 1/2 \leq 1/s$.* $\Diamond$

Theorem 25.2 (Nisan-Wigderson)**.** *Suppose there is a $L \in \mathsf{EXP}$ that is $s(n)$-hard to guess then BPP can be derandomized as per the following table:*

Hardness	**Consequence**
$s(n) = 2^{n/100}$	$\mathsf{BPP} = \mathsf{P}$
$s(n) = 2^{\sqrt{n}}$	$\mathsf{BPP} \subseteq \mathsf{DTIME}(n^{\text{poly}\log n})$
$s(n) = n^{\omega(1)}$	$\mathsf{BPP} \subseteq \mathsf{DTIME}(2^{n^{o(1)}})$

Table 25.1: Hardness and its consequences

25.1.1 How to build PRGs

Idea: Stretch a small number of bits to many-many bits. How many bits you start with (seed-length) depends on the hardness.

Toy example of a PRG: Assume that f is hard to guess, then a PRG that stretches by just 1 bit:

$$\mathcal{G} : (z_1, \ldots, z_n) \mapsto (z_1, \ldots, z_n, f(\bar{z}))$$

Suppose there is some circuit C that distinguishes the output of the PRG from the uniform distribution, that is

$$\Pr_{x \sim \mathcal{U}}[C(x) = 1] - Pr_{z \sim \mathcal{U}}[C(z, f(z)) = 1] > \varepsilon(*)$$

Note: We might as well assume that there is no absolute quantity, since we can just work with $\neg C$ if the above quantity is negative.

The goal now is essentially to show that if the above happened, the assumption that f is hard to guess is false. We wish to build $D : 0,1^n \to 0,1$ be a randomized circuit that guesses the value of f.

Intuition: The above condition says that it is slightly more likely for the circuit to accept if the last bit was random as opposed to if the last bit was actually $f(\bar{z})$

- Compute $b_0 = C(z_1, \ldots, z_n, 0)$, and $b_1 = (z_1, \ldots, z_n, 1)$
- If $b_1 = b_0$, then guess randomly.
- Else, lean on the side of rejection. That is, return i such that $b_i = 0$.

Claim 25.3. $\Pr_{x,D}[D(x) = f(x)] > 1/2 + \varepsilon$

Proportion of strings	$C(z_1, \ldots, z_n, f(z))$	$C(z_1, \ldots, z_n, \neg f(z))$
α_1	Accept	Accept
α_2	Accept	Reject
α_3	Reject	Accept
α_4	Reject	Reject

Table 25.2: Table for what C does on the input strings and the percentage of input strings in each category

Proof. We can write each of the relevant probabilities based on $\alpha_1, \ldots, \alpha_4$.

$$\begin{aligned}
\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 &= 1 \\
\Pr_{x \sim \mathcal{U}}[C(x) = 1] &= \alpha_1 + \frac{\alpha_2}{2} + \frac{\alpha_3}{2} \\
\Pr_{z \sim \mathcal{U}}[C(z, f(z)) = 1] &= \alpha_1 + \alpha_2 \\
\Pr_{z \sim \mathcal{U}}[C(z, f(z)) = 1] - \Pr_{x \sim \mathcal{U}}[C(x) = 1] &= \frac{\alpha_3 - \alpha_2}{2} > \varepsilon. \\
\Pr_{x,D}[D(x) = f(x)] &= \frac{\alpha_1}{2} + \frac{\alpha_4}{2} + \alpha_3
\end{aligned}$$

$$= \alpha_3 + \frac{1-\alpha_2-\alpha_3}{2} = \frac{1}{2} + \frac{\alpha_3-\alpha_2}{2}$$
$$> \frac{1}{2} + \varepsilon.$$

□

25.2 Towards greater stretch

The above gave us a stretch of one additional bit. There is a natural way to get a stretch of a few more bits via the following:

$$\mathcal{G} : (z^{(1)}, \ldots, z^{(k)}) \to (z^{(1)}, \ldots, z^{(k)}, f(z^{(1)}), \ldots, f(z^{(k)}))$$

This doesn't improve the 'rate of stretch'. If f is acting on disjoint subset of seed bits, then output length/input length $\leq (n+1)/n$

Key idea: What if f acts on "almost disjoint" subsets.

Definition 25.4 (Combinatorial designs: (n, ℓ, k, a))**.** *A collection of subsets $S_1, \ldots, S_n \subseteq [l]$ is an (n, ℓ, k, a)-combinatorial design if they satisfy the following:*

- $|S_i| = k$
- $|S_i \cap S_j| \leq a$, *whenever* $i \neq j$. ◊

There are such designs with efficient constructions as well.

Lemma 25.5. *For any k, a satisfying $n \leq 2^a$, $k \geq 2a$, there are (n, ℓ, k, a) designs with $\ell = k^2/a$.*

Think of $a = \log n, k = 100 \log n$ and the above says that are designs consisting of n sets from a universe with $\ell = O(\log n)$ elements.

Nisan-Wigderson PRGs

Let us fix an $(n\ell, k, a)$-combinatorial design $S_1, \ldots, S_n$. Given a function $f : \{0,1\}^k \to \{0,1\}$, we define the following generator:

$$\mathcal{G}_f^{(NW)} : \{0,1\}^\ell \to \{0,1\}^n$$
$$\mathcal{G}_f^{(NW)} : (z_1, \ldots, z_l) \mapsto (f(z|_{S_1}), \ldots, f(z|_{S_n}))$$

Lemma 25.6. *Suppose C is an ε-distinguisher for $\mathcal{G}_f^{(NW)}$. Then there is a circuit D of size $\leq 2(|C| + n \cdot 2^a)$ such that $\Pr_y[D(y) = f(y)] > 1/2 + \varepsilon/n$.*

Therefore, if f was hard-to-guess, then the above is indeed a pseudorandom generator.

Proof. If C is an ε-distinguisher then

$$\Pr_{x \sim \mathcal{U}_n}[C(x) = 1] - \Pr_{z \sim \mathcal{U}_l}[C(\mathcal{G}(z)) = 1] > \varepsilon.$$

Define a sequence of hybrid distributions: Such that initially it is a bunch of uniform bits $x_i \sim \mathcal{U}$, then it gets prefixed by i outputs of NW generator.

$$
\begin{aligned}
D_0 &:(x_1, \ldots, x_n) \\
D_1 &:(f(z|_{S_1}), \ldots, x_n) \\
&\vdots \\
D_i &:(f(z|_{S_1}), \ldots, f(z|_{S_i}), x_{i+1}, \ldots, x_n) \\
&\vdots \\
D_n &:(f(z|_{S_1}), \ldots, f(z|_{S_n})).
\end{aligned}
$$

By averaging argument, since the total distinguishing advantage is ε there must be some i such that

$$\Pr_{x \sim D_i}[C(x) = 1] - \Pr_{x \sim D_{i+1}}[C(x) = 1] > \varepsilon/n$$

Recall,

$$
\begin{aligned}
D_i &:(f(z|_{S_1}), \ldots, f(z|_{S_i}), x_{i+1}, \ldots, x_n) \\
D_{i+1} &:(f(z|_{S_1}), \ldots, f(z|_{S_i}), f(z|_{S_{i+1}}), x_{i+2}, \ldots, x_n)
\end{aligned}
$$

The only difference between D_i and D_{i+1} is the bit at position $i+1$: uniform vs. $f(z|_{S_{i+1}})$.

Let us set all variables outside $z|_{S_{i+1}}$ to some values while preserving

$$\Pr_{x \sim D_i}[C(x) = 1] - \Pr_{x \sim D_{i+1}}[C(x) = 1] > \varepsilon/n$$

Now, we wish to use C to construct a guesser for f. Consider an input $y_1, \ldots, y_k$ (which is identified with the variables $z|_{S_{i+1}}$). In order to feed this into C, we also need to provide to C with $z|_{S_j}$ for all $j \leq i$. Fortunately, since we have frozen all bits outside S_{i+1}, for each $j \leq i$ we have at most a variables in S_j that are unset. Hence, we can always compute the function $f(z|_{S_j})$ after freezing the other bits by circuits of size at most 2^a.

Overall, we are able to build a circuit D of size $2(|C| + n \cdot 2^a)$ that guesses f with non-trivial advantage. □

Theorem 25.7 (Nisan-Wigderson)**.** *If f is $(s + n2^a)/2$ -hard to guess, then $\mathcal{G}_f^{(NW)} : 0,1^l \to 0,1^n$ is a PRG for size s circuits.*

A strengthening of the above theorem is the following result of Impagliazzo and Wigderson that construct PRGs from worst-case hardness rather than average-case hardness.

Theorem 25.8 (Impagliazzo-Wigderson)**.** *If $\exists \varepsilon \geq 0$ such that some $L \in \mathsf{EXP}$ is not in $\mathsf{SIZE}(2^{\varepsilon n})$. Then $\mathsf{P} = \mathsf{BPP}$.*

Bibliography

[CM21] James Cook and Ian Mertz. Encodings and the tree evaluation problem. In *Electron. Colloquium Comput. Complex*, volume 54, 2021.

[CM24] James Cook and Ian Mertz. Tree evaluation is in space $O(\log n \cdot \log \log n)$. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, page 1268–1278, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649664.

[Wil25] R. Ryan Williams. Simulating time with square-root space, 2025. URL: https://arxiv.org/abs/2502.17779, arXiv:2502.17779.