[CSS.203.1]: Computational Complexity (2024-I)

Summary scribes

Lectures

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
4. Turing Machines

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.

$\sum^*$: 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

	Objects	Computational model	Resources
1.	Boolean functions $f : \{0,1\}^\star \longrightarrow \mathbb{N}$	Python Programs	Time / Memory/ No. of API calls
2.	$f : \mathbb{R}^n \to \mathbb{R}$	Query a few times and do something	No. of queries
3.	$f : \Sigma^* \times \Sigma^* \to \Sigma^*$ say $f : x \times y \to f(x,y)$	Communication between two parties where each party has some part of the input and they communicate amongst themselves to combine the inputs to produce an output	No. of bits of communication

Defn: Complexity - Classifying 'objects' based on 'resource required' by 'computational model'.
Here 'classifying' means creating a gradation of complexity (hardness) among objects/tasks.

1.2 Examples of problems and reductions

1. **Graph Reachability (undirected vs directed)**

   Input: Graph G
   Question: Given a vertex set, we want to find a path from s to t.

2. **Perfect Matching**

   Input : Graph G on even no. of vertices
   Question: Is every vertex in G adjacent to exactly one edge in G.

3. **Linear Programs**

$$\begin{aligned} \max \quad & c^T x \\ \text{st} \quad & Ax \leq b \end{aligned}$$

4. **Primality**

   Given n, check if n is prime

5. **Vertex Cover** (set of vertices that includes at least one endpoint of every edge of the graph)

   Input: Graph G, integer k
   Question: Does G have a vertex cover of size $\leq k$

6. **Independent set** (Subset of vertices with no edge b/w them)

   Input: Graph G, integer k
   Question: Does G have an independent set of size $\geq k$

7. **Chess**

   Input: Position
   Question: Does white have a winning strategy

Surely 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. For example, in Problem 5 & 6 it would be easier to check whether they are at the same level of complexity if we are given a subset of vertices. Our aim in this course would be to quantify these notions of complexity.

Reductions: Transforming an instance of one problem to an instance of another problem such that learning the solution to one problem instance tells us that the other problem instance is solvable.

1.3 Automata

Languages:

$$f : \Sigma^* \longrightarrow \{0, 1\}$$
$$L_f = \{y \in \Sigma^* : f(y) = 1\}$$

Languages L_f are subsets of Σ^*.

Automata are primitive computational models.

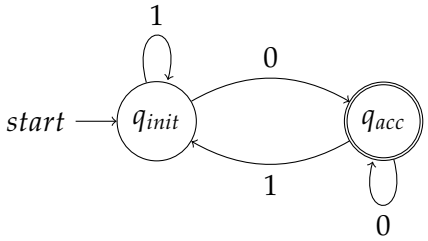


Figure 1.1: Example of an automata

We have a set of states Q and there is a specified start state q_{init} and a final state or accepting state q_{acc} The language accepted by the automata is the set of strings that the automata accepts.

1.4 Turing Machine

Turing Machines are an abstraction of how computers work. It is a combination of a set of states Q, with initial state q_{init} and the final states q_{acc} , q_{rej} and the transition function δ, which defines the Turing Machines. It consists of an infinite-length input tape, some infinite-length work tape, used for computation, and one output tape which is used for writing only once. The finite state machine points to the current symbol it is reading on each of the input and

work tapes using a head marker.

Transition function:

$$\delta : Q \times \Sigma_{\substack{\text{input tape} \\ \text{symbols}}} \times \Sigma_{\substack{\text{work tape} \\ \text{symbols}}} \to Q \times \Sigma_{\substack{\text{to write on} \\ \text{input tape}}} \times \Sigma_{\substack{\text{to write on} \\ \text{work tape}}} \times \underbrace{\{L, R, S\}^2}_{\substack{\text{decides whether the} \\ \text{head moves left,} \\ \text{right or stays at} \\ \text{the same position}}}$$

It could also be possible that the machine never reaches the final state, i.e. doesn't halt but in this course, we will only be interested in problems where it does halt.

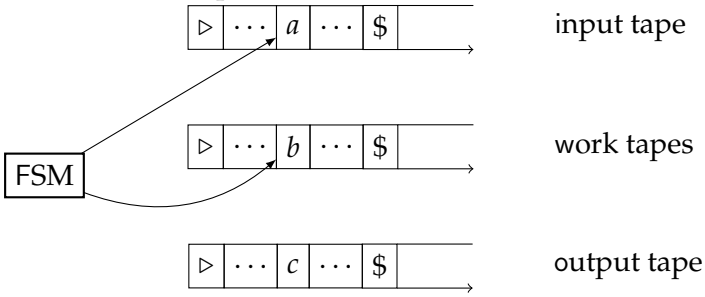
Lecture 2

Tape / alphabet reduction and Universal Turing Machines

Scribe: Siddharth Choudhary

2.1 Turing Machines

We consider the objects $f : \Sigma^* \to \Sigma^*$ where Σ is a fixed finite alphabet. A Turing Machine is a computational model with fixed finite number of states, along with access to a set of finite (say k) unbounded tapes.



With the tape alphabet as Σ', states Q of the Finite State Machine and the transition function

$$\delta : Q \times \Sigma' \times \Sigma'^k \to Q \times \Sigma' \times \Sigma'^k \times \{L, R, S\}^{k+1} \times \underbrace{\Sigma'}_{\text{output}}$$

Given a TM M, on an input x, if δ^∞ eventually reaches $q_{exit} \in Q$, then output is defined as the content of the output tape. Else, we say that M did not halt on x.
A machine is said to be *halting* if it halts on every $x \in \Sigma^*$.

We say M computes f iff for every $x \in \Sigma^*$, M on x halts and outputs $f(x)$.

For the set of functions $f : \Sigma^* \to \{0, 1\}$, the language $L_f = \{x \in \Sigma^* \mid f(x) = 1\}$ is the set of words which output 1 on f.

Given a halting machine M, let us denote $T_M(x)$ to be the time taken by machine M to halt on the input $x \in \Sigma^*$. Then, we define $T_M(n) = \max_{x \in \Sigma^n} T_M(x)$ to be the maximum time taken among all inputs of length n. Hence we get $T_M : \mathbb{N} \to \mathbb{N}$ to represent the time complexity of a language corresponding to the machine M.

2.2 Reductions

2.2.1 Alphabet Reduction

Problem: Given a TM M with alphabet Σ, can we build another equivalent machine M' with tape alphabet $\Sigma' = \{0, 1, \sqcup, \triangleright\}$?
Solution: Yes. This can be done by giving any prefix-free binary encoding of Σ and making copies of states of M to keep track of partially read encoding of a single character from Σ. Hence, we get the new machine M' with finite set of states s.t. $|Q'| \approx |Q|^{k|\Sigma|}$.
Hence we get that if $T_M(x) = n$ then $T_{M'}(x) \leq n * k * \ln |\Sigma|$. Therefore $T_{M'} = O(T_M)$.

2.2.2 Tape Reduction

Problem: Given a TM M with k working tapes, can we build another equivalent machine M' with single working tape?
Solution: Yes. This can be done by appending the k tapes content in a single tape with new alphabet which marks the pointer location of the corresponding tapes. Then the machine can scan entire tape every iteration to read the set of k pointer characters and make corresponding transition.
Hence we get that if $T_M(x) = n$ then $T_{M'}(x) \leq n^2$. Therefore $T_{M'} = O(T_M^2)$.

Note: There is method by interweaving the tapes to reduce the time s.t. $T_{M'} = O(T_M \ln(T_M))$.

2.3 Universal Turing Machine

Denoted by U, a Universal Turing Machine takes the input as an encoding of a Turing Machine M and its input x, to output $U(\langle M \rangle, x) = M(x)$.
This can be done by copying the encoding of M and x into two working tapes, and referring to the transition function δ in the encoding of M to do the corresponding transition using as many work tapes as given in the encoding of M.
Since each transition of U scans the work tape storing description of M to refer to the its transition function, we get that a Universal Turing Machine U with 1 working tape takes $T_{U(\langle M \rangle, \_)}(x) = O(T_M(x) \ln(T_M(x)))$ time.

Lecture 3

Non-determinism, and classes P, NP and coNP

Scribe: Jainam Khakhra

Before we dive into non-determinism, it would be a good idea to take a short detour into automata theory.

3.1 Regular Expressions

A regular expression, is a sequence of characters used to check if a given string follows a required pattern.Using quantifiers such as OR, AND, NOT etc., one can construct complex regular expressions.

For a long time, backtracking would be used to check the match of a regular expression. Backtracking proved to be an inefficient process because it would take a lot of time to match a string that did not match the required pattern.

Regular expressions can be converted into finite state machines as defined below.

3.1.1 Some basics on Automata

A finite state machine (or finite state automata) (FSM) is a machine that can exist in only finitely many states. There are two types of FSM, Deterministic Finite Automation (DFA) and Non-Deterministic Finite Automation (NFA).

In DFA, moving from the current state to the next state is uniquely determined, while in NFA, the next state is not uniquely determined by the machine. There may exist more than one possible next state from the current state.

Mathematically, a DFA is completely specified by a single transition function $\delta : Q \times \Sigma \to Q$ which is the transition from the current state to the next.

For an NFA, we have *two* transition functions δ_0 and δ_1 defined similarly as

$$\delta_0 : Q \times \Sigma \to Q$$
$$\delta_1 : Q \times \Sigma \to Q.$$

The NFA is said to accept a string if there is any "valid run" (choice of one of the transitions at every step) that results in the automata ending at the accept state.

3.2 Deterministic and Non Deterministic Turing Machines

In a Deterministic Turing Machine, the machine would perform either no action or exactly one action from a given configuration (state + whatever the heads were reading at that point), while in a Non Deterministic Turing Machine, the machine could perform any action among a group of possible actions from a given state. This is again specified by giving two transition functions for the machine to choose from.

A Non Deterministic Turing Machine must halt on every path, also known as the halting property, and is said to 'Accept' an input string if one of the possible paths lead to an accept.

A Co-Non Deterministic Turing Machine must also halt on every path but is said to 'Accept' a string if all the paths lead to accept.

3.3 Classes P, NP and coNP

3.3.1 Class P

The Class P is defined to be the set of languages L such that there is a Deterministic Halting Turing Machine M and a constant 'c' such that

- $L(M) = L$
- The running time of M is $O(n^c)$, i.e in polynomial time.

Example: The Circuit Evaluation Problem (Circuit-Eval). The Circuit-Eval problem is the task of computing the evaluation of an given circuit on a given input.

$$\text{Circuit-Eval} = \{(\langle C \rangle, x) \ : \ \langle C \rangle \text{ encodes a circuit and } C(x) = \text{True}\}$$

It is easy to see that Circuit-Eval $\in$ P.

3.3.2 Class NP

The Class NP is defined to be the set of languages L such that there is a Non Deterministic Halting Turing Machine M such that

- $L(M) = L$

- The running time of M is $O(n^c)$ i.e in polynomial time.

The running time of M is measured as that of the worst case over all possible inputs and paths.

Example: The Circuit Satistifiability Problem (Circuit-SAT). The Circuit-SAT problem is the task of determining if a given a Boolean Circuit is *satisfiable*, i.e. is there a set of inputs for which the output could be determined as 'True'.

It is easy to see that Circuit-SAT $\in$ NP.

3.3.3 Class coNP

The class coNP is the set of languages L whose complement language where in the running time of M is in polynomial time.

Example: Tautology (TAUT). Given a circuit C, determine if C is a tautology, i.e. is C true on every input x.

It is easy to see that TAUT $\in$ coNP.

Note that in order to check if a circuit is a tautology, we only need to check if there is any input that makes the circuit $\neg C$ true. That is, we merely need to check if $\neg C \in$ Circuit-SAT. Indeed, it is the case that a language $L \in$ NP if and only if $\overline{L} \in$ coNP.

Lecture 4

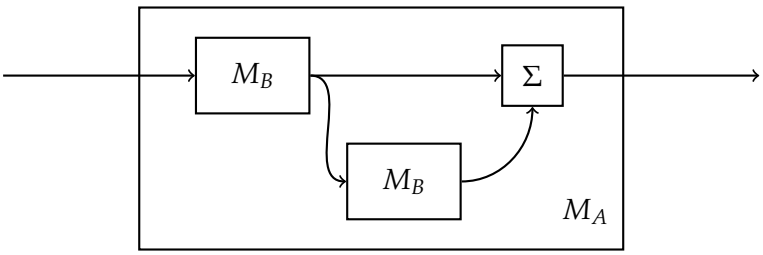
Reductions

Scribe: Bikshan Chatterjee

When is one problem harder than another? A reduction from problem A to problem B is a way to construct a machine that solves problem A given a machine that solves problem B, usually using this machine for B as a subroutine, and without requiring too much overhead. This shows that problem A is no harder than problem B.
(Here problems are languages and solving is outputing 1 for strings in the language and 0 for others).

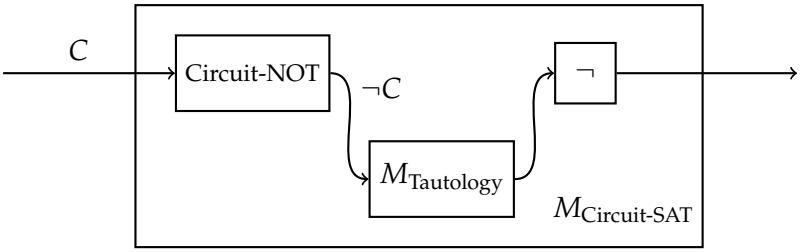
4.1 Turing Reduction

The machine M_B (solving B) can be used multiple times and the output of M_A can be computed based on any postprocessing of the outputs from M_B.



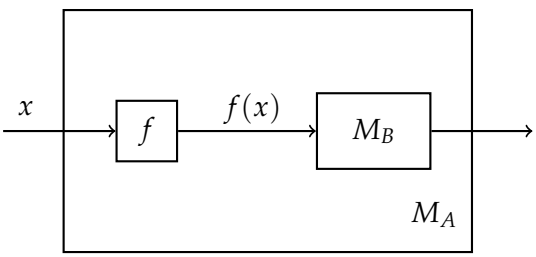
Reduction Time: time required for computations other than running M_B.

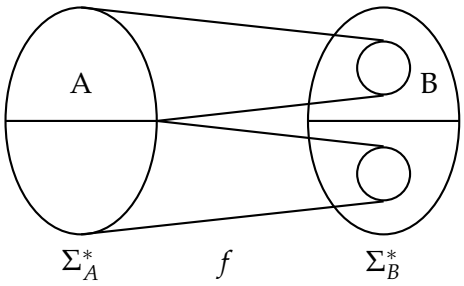
Example:
Circuit-SAT $\rightarrow$ Tautology
A circuit C is satisfiable if and only if $\neg C$ is not a tautology.



4.2 Many-one Reduction

The machine M_B (solving B) can only be used once at the end of the process, the output of M_B must be the output of M_A. It is a function $f : \Sigma_A^* \to \Sigma_B^*$ (computable by a TM) mapping instances of problem to A to instances of problem B such that $x \in A$ if and only if $f(x) \in B$.





Reduction Time: time for computing f.

Since f must be computable in polynomial time (including the time taken to write the output), $|f(x)|$ cannot be much larger than $|x|$ (bounded by some polynomial in $|x|$).

Many-one reduction from Circuit-SAT to Tautology is not known.

Theorem 4.1 (Cook-Levin)**.** *Given any language $L \in \mathsf{NP}$, there is a poly time many-one reduction f_L from L to* Circuit-SAT*. "L is no harder than* Circuit-SAT*".*

4.2.1 Examples

Languages:
(i) Ind-Set $= \{ (G,k) \mid G$ has an independent set of size $\geq k \}$.
(ii) Clique $= \{ (G,k) \mid G$ has a clique of size $\geq k \}$.
(iii) Vertex-Cover $= \{ (G,k) \mid G$ has a vertex cover of size $\leq k \}$.

1. Ind-Set $\to$ Clique: Let $G(V,E)$ be a graph. A clique of k vertices $S \subseteq V$ in the complement graph $\bar{G}(V,\bar{E})$ (edges and non-edges flipped), would become an indepnendent set of size k in G.
Reduction: $f : (G,k) \mapsto (\bar{G},k)$. Same reduction works for Clique $\to$ Ind-Set.

2. Vertex-Cover $\rightarrow$ Ind-Set : Let $G(V, E)$ be a graph. If $S \subseteq V$ is an independent set then $V \setminus S$ would be a vertex cover (no edges among vertices in S, so every edge has at least one endpoint in $V \setminus S$).
Reduction: $f : (G(V, E), k) \mapsto (G(V, E), |V| - k)$. Same reduction works for Ind-Set $\rightarrow$ Vertex-Cover.

4.3 Hardness and Completeness

C-Hardness A language L is said to be C-hard under "type of reduction" (usually polytime many-one reduction) if for all $L' \in C$ there is such a reduction from L' to L (for polytime many-one reductions, a polytime computable function $f : \Sigma^*_{L'} \rightarrow \Sigma^*_L$).

C-Complete If in addition $L \in C$, then L is said to be C-complete.

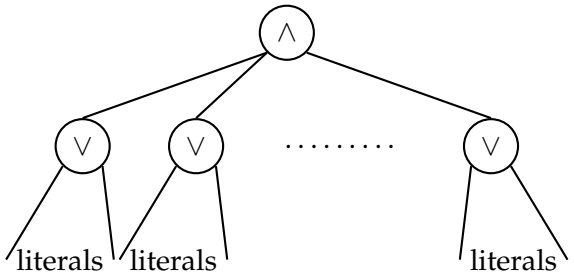
Theorem 5.8 says Circuit-SAT is NP-hard (and in NP so NP-complete). If we want to show some L is NP-complete, it is enough to show $L \in$ NP and a polytime reduction from Circuit-SAT to L. Then any language in NP can be reduced first to Circuit-SAT then to L.

4.4 Circuit-SAT $\rightarrow$ CNF-Sat

CNF: Boolean formulas of the form

$$(x_1 \vee \bar{x}_2 \vee \ldots) \wedge (\bar{x}_3 \vee x_4 \vee \ldots) \wedge \ldots$$

Special case of circuits with 2 levels:



CNF-Sat $= \{$ Formulas/circuits φ in CNF form $\mid$ φ is satisfiable $\}$.

Reducing Circuit-SAT **to** CNF-Sat:

The circuit C input to Circuit-SAT can be given as a graph with vertices and edges representing gates and wires. The fan in of each gate can be assumed to be 2: a gate with fan in n can be replaced by $n - 1$ gates of fan in 2, circuit size remains comparable (increase bounded by a factor of number of wires in C).

In the corresponding formula φ_C, there would be literals for all literals in the circuit C and new ones for every gate.

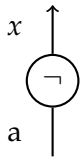
For each gate i there would be a CNF formula that corresponds to "gate i is correct". The final formula is of the form

$$\varphi_C = \bigwedge_i (\text{gate } i \text{ is correct}) \wedge (\text{output is } 1)$$

C will be satisfiable if and only if φ_C is satisfiable.

Gate is correct CNFs:

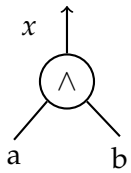
NOT gates



a	x	correct
0	0	x
0	1	✓
1	0	✓
1	1	x

CNF: $\overline{(a \wedge x) \vee (\neg a \wedge \neg x)} = (\neg a \vee \neg x) \wedge (a \vee x)$

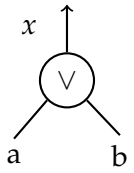
AND gates



a	b	x	correct
0	0	0	✓
0	0	1	x
0	1	0	✓
0	1	1	x
1	0	0	✓
1	0	1	x
1	1	0	x
1	1	1	✓

CNF: $\overline{(\neg a \wedge \neg b \wedge x) \vee (\neg a \wedge b \wedge x) \vee (a \wedge \neg b \wedge x) \vee (a \wedge b \wedge \neg x)}$
$= (a \vee b \vee \neg x) \wedge (a \vee \neg b \vee \neg x) \wedge (\neg a \vee b \vee \neg x) \wedge (\neg a \vee \neg b \vee x)$

OR gates



a	b	x	correct
0	0	0	✓
0	0	1	x
0	1	0	x
0	1	1	✓
1	0	0	x
1	0	1	✓
1	1	0	x
1	1	1	✓

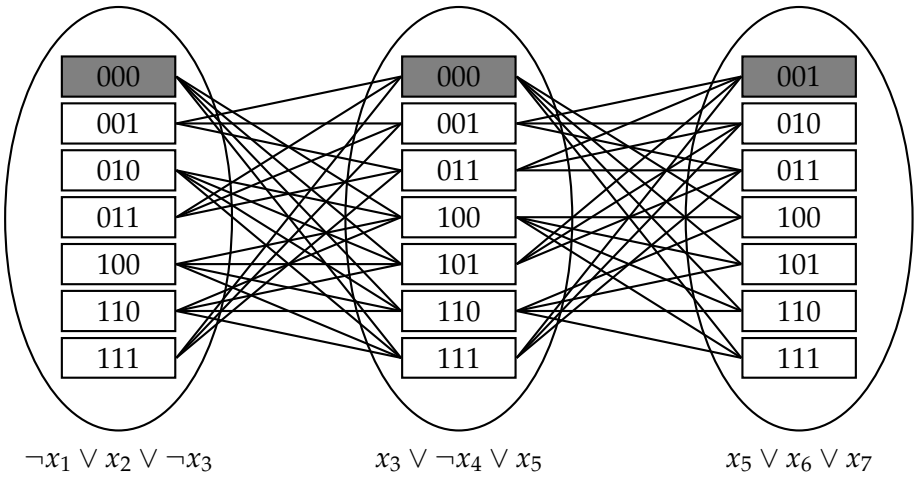
CNF: $\overline{(\neg a \wedge \neg b \wedge x) \vee (\neg a \wedge b \wedge \neg x) \vee (a \wedge \neg b \wedge \neg x) \vee (a \wedge b \wedge \neg x)}$
$= (a \vee b \vee \neg x) \wedge (a \vee \neg b \vee x) \wedge (\neg a \vee b \vee x) \wedge (\neg a \vee \neg b \vee x)$

4.5 3-CNF-Sat $\rightarrow$ Ind-Set

3-CNF-Sat : Each clause has 3 literals.
Given a formula with m clauses, create a graph with m complete subgraphs containing 7 vertices each. The vertices in the subgraphs correspond to the 7 assignments to the variables in the clause that satisfy it. Then join the incosistent assignments across the clauses. An independent set of m vertices now must take exactly one vertex from each clause (it's assignment will satisfy the clause) with no inconsistent assignments.

A formula with m clauses is satisfiable if and only if the corresponding graph has an independent set of size $\geq m$ (cannot be $> m$ so $= m$).

Example:
The CNF $(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge (x_5 \vee x_6 \vee x_7)$ maps to the graph



The ellipses are the cliques corresponding to the clauses and the rectangles are vertices corresponding to assignments that satisfy the clause. The assignments are to the variables not

literals eg. 000 in the first clique corresponds to the assignment $x_1 = 0, x_2 = 0, x_3 = 0$.

The shaded vertices give an independent set with satisfying assignment: $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 0, x_7 = 1$.

If a formula is satisfiable, take a satisfying assignment. Each clique would contain one vertex that agrees with the assignment (because it satisfies every clause). The vertices do not have edges among them because the assignment is consistent. These vertices form a clique of size m.

If there is an independent set of size m, take one independent set. It would contain one vertex from each clique (only way to select m vertices). The vertices give assignments that satisfy the clauses corresponding to the cliques. And they are consistent because there is no edge among them.

Lecture 5

Cook-Levin Theorem

Scribe: Yeshwant Pandit

Previously, we saw a web of reductions assuming Cook-Levin theorem. In this section, we will see a proof of Cook-Levin theorem. Before that, let us see few definitions.

Definition 5.1. *(class* $\mathsf{DTIME}(\cdot)$*) : A language* L *is in* $\mathsf{DTIME}(T(n))$ *iff there is a deterministic TM* M_L *that runs in time* $c \cdot T(n)$ *for some constant* $c > 0$ *and decides* L*.*

◊

Definition 5.2. *(class* P*) :* $\mathsf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c)$.

◊

Definition 5.3. *(class* EXP*) :* $\mathsf{EXP} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{n^c})$.

◊

Definition 5.4. *(class* $\mathsf{NTIME}(\cdot)$*) : A language* L *is in* $\mathsf{NTIME}(T(n))$ *iff there is a non-deterministic TM* M_L *that runs in time* $c \cdot T(n)$ *for some constant* $c > 0$ *and decides* L*.*

◊

Definition 5.5. *(class* NP*) :* $\mathsf{NP} = \bigcup_{c \geq 1} \mathsf{NTIME}(n^c)$.

◊

Definition 5.6. *(class* NEXP*) :* $\mathsf{NEXP} = \bigcup_{c \geq 1} \mathsf{NTIME}(2^{n^c})$.

◊

Before we state and prove Cook-Levin theorem, let us look at a simple example of an NP-complete language.

5.1 An example of an NP-complete language

Recall, a language L is NP-complete if

- $L \in \mathsf{NP}$
- For all $L' \in \mathsf{NP}$, we have $L' \leq_{\mathsf{P}} L$

Consider the language $L = \{(\langle M \rangle, x, 1^k) : M \text{ encodes an NDTM that accepts } x \text{ in } k \text{ steps}\}$. Assume k to be $p(|x|)$, where p is some polynomial.

Theorem 5.7. *L is* NP*-complete.*

Proof. Firstly, let us show that $L \in \mathsf{NP}$. We construct a NDTM M_L for language L as follows

- Using a non-deterministic universal Turing machine, simulate M on input x. The execution can cause at most a polynomial time slowdown.
- Accept if M accepts x
- Reject if M rejects x or M exceeds k steps

This shows that $L \in \mathsf{NP}$. Now, let us show that L is NP-hard. Consider any $L' \in \mathsf{NP}$. By definition, there is an NDTM M' and a polynomial p such that $M'(x) = 1 \Leftrightarrow x \in L'$ and M' runs in $p(|x|)$ steps.
If $x \mapsto (\langle M' \rangle, x, 1^{p(|x|)})$ then we have $x \in L' \Leftrightarrow (\langle M' \rangle, x, 1^{p(|x|)}) \in L$. This proves that L is NP-hard. Therefore, L is NP-complete □

5.2 Proof of Cook-Levin theorem

Theorem 5.8 (Cook-Levin)**.** Circuit-SAT *is* NP*-complete.*

Proof Sketch. It is easy to see that Circuit-SAT $\in \mathsf{NP}$ because a witness would be a satisfying assignment. The non-trivial part is to prove that Circuit-SAT is NP-hard.

Before we dive into the proof, let us introduce two notions namely Snaphot of a TM, and Computational Tableau.

- **Snapshot**: A snaphot of a TM at any step comprises of state of the TM, tape contents and the positions of tape heads.
- **Computational Tableau**: A computational tableau is a table with each row consisting of a snapshot of a TM and a bit i representing the transition function δ_i chosen by the TM. For example, $i = 0$ means TM selects transition function δ_0. The snapshots in the first and the last row correspond to the start and the end configuration of the TM. The snapshot of k^{th} row of the table is determined by the snapshot of $(k-1)^{th}$ row and the transition δ_i chosen by the TM in $(k-1)^{th}$ row.

Having introduced these two notions, we are now ready to show that Circuit-SAT is NP-hard. To show this consider any $L \in \mathsf{NP}$. Since $L \in \mathsf{NP}$, we know there is a NDTM M_L that runs in polynomial time and decides L. Let n^c be the running time of this machine on inputs of length n. We construct a boolean circuit $C_{L,x}$ that is satisfiable iff some branch of M_L's computation accepts x.

Since, a computational tableau consists of the snapshots attained by a NDTM on some branch of its computation tree, determining $x \in L$ is equivalent to searching whether the snapshot corresponding to the last row of the tableau contains the state q_{accept}.

We now obtain a boolean circuit $C_{L,x}$ as follows:

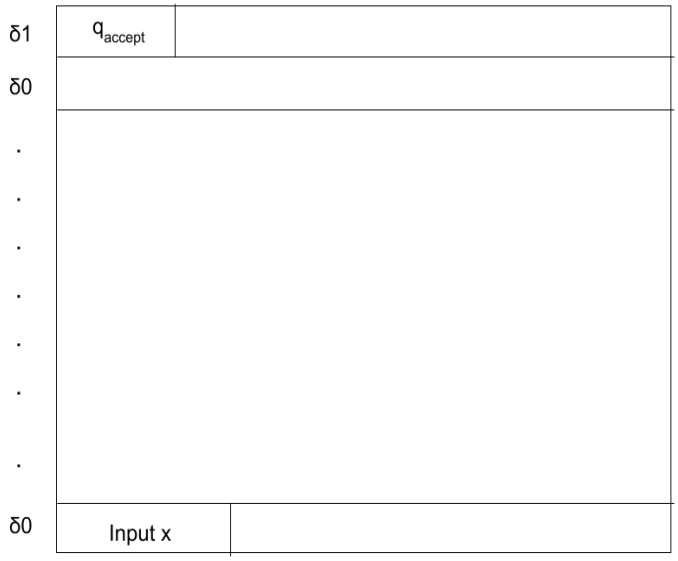


Figure 5.1: Computational Tableau

- For every cell in the tableau, we define a propositional variable which is true iff the cell contains a valid tape symbol. Note that the machine M_L accepts x iff
  - Every cell is well-defined.
  - The first row represents a valid start configuration.
  - Each row can be determined from the previous row using the transition function chosen by the machine M_L.
  - The last row of the tableau contains the state q_{accept}.
- The conditions mentioned above can be captured in boolean formulae and combining all of them using AND gives us the final boolean formula and hence a boolean circuit $C_{L,x}$.
- It is easy to see that the circuit $C_{L,x}$ is satisfiable iff some branch of M_L's computation accepts x

It is yet to be demonstrated that the reduction is computable in polynomial time. Since, we assumed that the machine M_L runs in n^c time and as a result it cannot use more than n^c cells of a tape, the size of the tableau (number of cells) is $O(n^{2c})$. Since, we had one propositional variable corresponding to each cell in the tableau, the number of literals in the circuit $C_{L,x}$ is $O(n^{2c})$. The fact that the size of each formula constructed is $O(n^{2c})$ and there are constant number of formulae, allows us to infer that the reduction is carried out efficiently. Hence, Circuit-SAT is NP-hard. □

Lecture 6

Padding and the deterministic hierarchy theorem

Scribe: Soumyajit Pyne

6.1 Collapses and Separations of Complexity Classes

In this section we will discuss about two key statements:

- " Collapses scale up."
- " Separations scale down."

Intuitively, the initial statement suggests that if a certain capability (such as non-determinism) cannot differentiate between two classes, then it shouldn't be capable of distinguishing between two bigger classes. The next theorem is an example of such statement.

Theorem 6.1. *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{EXP} = \mathsf{NEXP}$.

Proof. To prove this theorem we will use a technique called padding. Suppose $L \in \mathsf{NEXP}$ and M_{NEXP} is the non deterministic machine that decides L in time $\mathcal{O}(2^{|x|^c})$ where x is the input and c is a positive constant. Now we will define another language L' as follows.

$$L' = \left\{ x\#1^{2^{|x|^c}} \ : \ x \in L, \ \# \text{ is a special character} \right\}$$

Claim 6.2. $L' \in \mathsf{NP}$.

Proof. Suppose we are given an input y. We define a non deterministic polytime Turing machine for L' as follows.

1. we check if input y is of the form $x\#1^{2^{|x|^c}}$ where x is a binary string. If it does, proceed to step 2; otherwise, return 0.

2. Return $M_{\mathsf{NEXP}}(x)$.

The first step of the algorithm can be done in time $\mathcal{O}(|Y|)$. Note that if the input y is not of the form $x\#1^{2^{|x|^c}}$, then $y \notin L'$. Conversely, if the input y is of the form $x\#1^{2^{|x|^c}}$, then $x \in L$ if and only if $y \in L'$. As $L \in \mathsf{NEXP}$, M_{NEXP} takes $\mathcal{O}(2^{|x|^c})$ time on input x. Therefore, the above non-deterministic Turing machine runs in poly($|y|$) time. □

Claim 6.3. *$L \in EXP$.*

Proof. By the hypothesis that $\mathsf{P} = \mathsf{NP}$, we have$L' \in \mathrm{P}$ (Claim 6.2). Suppose M_P is the deterministic polytime Turing machine that decides L'. Now we define a deterministic exponential time Turing machine M_{EXP} that decides L as follows.

1. Given an input x, create $y = x\#1^{2^{|x|^c}}$.
2. Return $M_P(y)$.

Observe that $x \in L'$ if and only if $x\#1^{2^{|x|^c}} \in L$. It is easy to verify that M_{EXP} runs in exponential time. Therefore, M_{EXP} decides L is exponential time. □

From Claim 6.3 we conclude that $\mathsf{EXP} = \mathsf{NEXP}$. □

An example of the second statement "Separations scale down." would be the contrapositive of Theorem 6.1.

6.2 Deterministic Time Hierarchy Theorem

The Time Hierarchy Theorem demonstrates that providing Turing machines with additional computation time increases the class of languages that they can decide.

Definition 6.4 ($\mathsf{DTIME}(\cdot)$)**.** *Let $f : \mathbb{N} \to \mathbb{N}$ be non-decreasing function. The class $\mathsf{DTIME}(f(n))$ contains languages for which then there exists a constant $c > 0$ and a deterministic Turing Machine that can decide L in with a running time bounded by $c \cdot f(n)$ for all large enough n (where n represents the input size).* ◊

Theorem 6.5 (Deterministic time hierarchy theorem)**.** *If $f, g : \mathbb{N} \to \mathbb{N}$ are non-decreasing* time-constructible[1] *functions satisfying $f(n) \log f(n) = o(g(n))$, then*

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

Proof. To describe the proof idea we will prove a simpler statement $\mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^4)$. It is easy to see that $\mathsf{DTIME}(n^2) \subseteq \mathsf{DTIME}(n^4)$. Now we will prove the strict inclusion.

Note that all Turing machines can be described using a finite string (some representation of the number of states, number of tapes, the alphabet, and the transition function). This is like providing the 'source code' of the Turing machine. We will choose some systematic way of encoding Turing Machines, with the ability to 'add comments' (for instance, we could always allow encodings of the form "$\langle M \rangle$ *#aaaaa*" where just ignore the string after the last '#' symbol). All this is to ensure the following property of the encodings.

[1] we'll define this in the next lecture

Observation 6.6. *Each Turing machine has infinitely many possible encodings.*

We can now go ahead to define the language L that will witness the separation and we will do so via a Turing machine D running in time $O(n^4)$.

1. If the input x is not a valid encoding of a deterministic Turing machine, reject it.
2. Let M_x be the Turing machine encoded by x, and let $|x| = n$.
3. Run the Universal Turing Machine U for at most n^4 steps and simulate M_x on input x. If by within that time the machine M_x rejects the string x, then our machine D accepts the string. Else (i.e., either M_x has accepted the string x by then, or hasn't finished computation), we reject the string x.

By the very definition of the above machine, since we are only running U for n^4 steps, we immediately have the following

Claim 6.7. $L \in \mathsf{DTIME}(n^4)$.

We need to only show that $L \notin \mathsf{DTIME}(n^2)$.

Claim 6.8. $L \notin \mathsf{DTIME}(n^2)$.

Proof. For the sake of contradiction suppose M decides L and the running time of M is $O(n^2)$. By a theorem we saw in an earlier lecture, the Universal Turing Machine can simulate M in $O(n^2 \log n)$ steps. More precisely, there is a constant c and an integer n_0 such that for all $n \geq n_0$ the UTM U can simulate M in inputs of length n in at most $cn^2 \log n$ steps.

Since $n^2 \log n = o(n^4)$, there is a large enough integer $n_1 \geq n_0$ such that for all $n \geq n_1$, we have $cn^2 \log n < n^4$. Also, by Observation 6.6, let x be an encoding of the machine M of length at least n_1. We are going to show that the machine M makes a mistake on the string x.

Firstly, since $n \geq n_1 \geq n_0$, we note that n^4 steps of the UTM U can entirely simulate the running of M on the string x. Note that by the definition of L, we have $x \notin L$ if and only if the machine encoded by the string x (namely the machine M) accepts the string x within n^4 steps of the UTM simulation. Thus, we have that $x \in L$ if M rejects x, and $x \notin L$ if M accepts x. Either way, the language accepted by M is not equal to L. □

Claim 6.7 and 6.8 together conclude that $\mathsf{DTIME}(n^2) \subset \mathsf{DTIME}(n^4)$. □

Lecture 7

Non-Deterministic Time Hierarchy Theorem

Scribe: Ashutosh Singh

In the previous lecture, we saw the deterministic time hierarchy theorem which states that there are languages that are decidable in time, say n^2, but not in, say n^4. In general it states that for any two *time-constructible* functions $f, g : \mathbb{N} \to \mathbb{N}$, such that $f(n) \log f(n) = o(g(n))$, we have $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$.

In this lecture, we saw the non-deterministic time hierarchy theorem. Before stating and proving the non-deterministic time hierarchy theorem, we define this notion of time-constructible functions.

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

We also defined this notion of *clocked* UTM simulation for our convenience as follows.

Definition 7.2 (Clocked UTM Simulation)**.** *For a given TM description μ and input x, let $n = |\mu| + |x|$. Then for a function $f : \mathbb{N} \to \mathbb{N}$, a clocked UTM simulation means that we use UTM U to simulate μ on input x for at most $f(n)$ steps.* $\Diamond$

Then we saw the following weaker version of the non-deterministic time hierarchy theorem.

Theorem 7.3. *Suppose $f, g : \mathbb{N} \to \mathbb{N}$ are non-decreasing time constructible functions with $f(n+1) \log f(n+1) = o(g(n))$, then*

$$\mathsf{NTIME}(f(n)) \subsetneq \mathsf{NTIME}(g(n))$$

This is a weaker statement of the non-deterministic time hierarchy theorem which will be stated later. Regardless, the first question that we would like to address is that we have seen the deterministic time hierarchy theorem and its proof. What could go wrong in attempting to extend the same proof here? i.e., We had defined the following TM D, that on input x,

1. Simulate M_x on x for $|x|^4$ steps of the UTM.

2. *Flip* the answer

The question is, Why does the language defined by the above TM D is not enough to prove the theorem at hand? As we saw, the problem is that the machine M_x now is a non-deterministic machine. As we know for a non-deterministic machine, the acceptance criteria is that some path leads to an accepting state, but for rejection, it is that all paths lead to a rejecting state. Now the machine D itself is non-deterministic. Then think about, how would we go about defining the acceptance criteria for the machine D which essentially is trying to *flip* the answer of the machine M_x?
So now that we understand the non-deterministic time hierarchy isn't as easy to prove as it may first occur, we see the following proof due to Fortnow and Santhanam. We are again considering, $f(n) = n^2$ and $g(n) = n^4$ for convenience.

Proof. The idea is again to define the language by the machine that decides it. We define the Turing machine D which this time takes an ordered pair (x, y) as input and computes $D(x, y)$ as follows,

1. If $|y| < |x|^4$:

   (a) Run clocked UTM simulation of M_x for $(|x| + |y| + 1)^4$ steps on $(x, y0)$ and on $(x, y1)$.

   (b) Accept if both simulations end up accepting

2. Else($|y| \geq |x|^4$):

   (a) Run clocked UTM simulation of M_x for $|x|^4$ steps on $(x, "\ ")$ deterministically on the computational path y.

   (b) Flip the answer

As the very first thing, it is easy to see that the above construction of D doesn't suffer from the abnormality that we discussed above, and as was the case with the deterministic time hierarchy theorem, by the very definition of the machine D, $L(D) \in \mathsf{NTIME}(n^4)$. So all we need to show is that $L(D) \notin \mathsf{NTIME}(n^2)$. Assume that we have a machine M which claims to run on input x such that $|x| = n$ in time $O(n^2)$ and correctly decides $L(D)$. By the definition of Big-Oh, we have that there exist constants c_0, n_0, such that for all $n \geq n_0$ length inputs, machine M runs in time at most $c_0 n^2$. Now the UTM simulation of machine M on input of length n takes $O(n^2 \log n)$ time which again implies that there are constants, say c_1, n_1, such that for all input of length $n \geq n_1$, UTM can simulate M on the corresponding input in time at most $c_1 n^2 \log n$. Lastly, since, $n^2 \log n = o(n^4)$, there are constants c_2, n_2, such that for $n \geq n_2$, $n^2 \log n < c_2 n^4$. Then consider the equivalent description of the TM M such that $|\langle M \rangle| \geq \max(n_0, n_1, n_2)$. Then what we essentially have is that a UTM simulation of this M can be completed in time n^4. Now, Since $L(M) = L(D)$, then on input $(\langle M \rangle, "\ ")$, we have

$$
\begin{aligned}
M(\langle M\rangle, " ") = 1 &\Leftrightarrow D(\langle M\rangle, " ") = 1 \\
&\Leftrightarrow M(\langle M\rangle, 0), M(\langle M\rangle, 1) = 1 \\
&\Leftrightarrow D(\langle M\rangle, 0), D(\langle M\rangle, 1) = 1 \\
&\Leftrightarrow M(\langle M\rangle, 00) \cdots M(\langle M\rangle, 11) = 1 \\
&\Leftrightarrow D(\langle M\rangle, 00) \cdots D(\langle M\rangle, 11) = 1 \\
&\vdots \\
&\Leftrightarrow M(\langle M\rangle, 0^{|\langle M\rangle|^4}) \cdots M(\langle M\rangle, 1^{|\langle M\rangle|^4}) = 1 \\
&\Leftrightarrow D(\langle M\rangle, 0^{|\langle M\rangle|^4}) \cdots D(\langle M\rangle, 1^{|\langle M\rangle|^4}) = 1 \\
&\Leftrightarrow M \text{ on input } (\langle M\rangle, " ") \text{ rejects on paths } 0^{|\langle M\rangle|^4}, \ldots, 1^{|\langle M\rangle|^4} \\
&\Leftrightarrow M(\langle M\rangle, " ") = 0
\end{aligned}
$$

Let's look at the above chain of conclusions more closely. The first bi-implication holds because of the claim that $L(M) = L(D)$. Now, $M(\langle M\rangle, " ") = 1$ implies that there exists a computational path y on which M accepts $(\langle M\rangle, " ")$. Then the second bi-implication holds because of the first 'If' condition in the definition of TM D. Then the third bi-implication again follows because M and D compute the same language. The above story continues till the second last bi-implication where we have $D(\langle M\rangle, y) = 1$ for all $y \in \Sigma^4$ but then the 'Else' part of our definition of D says that if $|y| \geq |x|^4$, then we reject if $M(x, " ")$ accepts on the computational path y. So $D(\langle M\rangle, y) = 1$ for all $y \in \Sigma^4$ implies that $M(x, " ")$ rejects on the computational path y for all $y \in \Sigma^4$ which contradicts the first bi-implication.

□

In the above theorem, we considered functions f, g such that $f(n+1)\log f(n+1) = o(g(n))$ and then we showed that $\mathsf{NTIME}(f(n)) \subsetneq \mathsf{NTIME}(g(n))$. But as we will see in problem set 1, we can simulate a non-deterministic TM using UTM without any loss of log factor in the time. So we have,

Theorem 7.4 (Non-Deterministic Time Hierarchy Theorem)**.** *For $f, g : \mathbb{N} \to \mathbb{N}$ time-constructible functions such that $f(n+1) = o(g(n))$, we have,*

$$\mathsf{NTIME}(f(n)) \subsetneq \mathsf{NTIME}(g(n)).$$

Lecture 8

Diagonalization and Ladner's theorem

Scribe: Sourav Roy

Theorem 8.1 (Ladner's theorem)**.** *If* $\mathsf{P} \neq \mathsf{NP}$*, then there exists a language* $L \in \mathsf{NP}$ *that is neither in* P *nor is* NP*-complete.*

We would like to create a language L in a manner that can be taken as "Easier SAT" to perform either to give the machine more time or, make some instances on Ladner's theorem.

Thus the obvious question that can be ask is the following:

How do we "pretend to give more time", and thus make SAT easier?

We will do this by padding.

8.1 The language SAT_H

Definition 8.2 (SAT_H)**.** *Let* $H : \mathbb{N} \to \mathbb{N}$ *be a non-decreasing function. Define the language* SAT_H *as follows:*

$$\mathrm{SAT}_H = \left\{ \varphi \# 0^{m^{H(m)}} \ : \ \varphi \in \mathrm{SAT} \text{ and } |\varphi| = m \right\}.$$

That is, the language is essentially SAT *with the* right amount of padding. ◊

8.1.1 Some basic properties

Observation 8.3 (If H grows too quickly...)**.** *If* $H(m) = m$*, then* $\mathrm{SAT}_H \in P$*.*

Proof. Given in input $\varphi \# 0^k$ of length n, let m be the size of φ. We can check quickly (in $\mathrm{poly}(n)$ time) check if $k = m^m$ and reject if not. If the padding is right, then we have that $n \geq k = m^m$ and hence $m = O(\log n)$. Thus, we can even afford a brute-force algorithm to check if the formula φ is satisfiable or not, as we have $\mathrm{poly}(n)$ time at our disposal. □

Thus if $H(m)$ "grows too quickly", then we have $\mathrm{SAT}_H \in \mathsf{P}$.

Observation 8.4 (If H is bounded by a constant...)**.** *If $H(m) \leq c$ for all m, i.e. $H(m)$ is bounded above by a constant c, then SAT_H is NP-hard.*

Proof. We have an immediate reduction from SAT to SAT_H via $\varphi \mapsto \varphi\#0^{m^{H(m)}}$. Since $H(m)$ is bounded by a constant, this reduction takes just $O(m^c)$ time (to add the extra padding) and thus is a polynomial time many-one reduction from SAT to SAT_H. □

Observation 8.5 (Sufficient condition for $\mathrm{SAT}_H \in \mathsf{NP}$)**.** *Suppose $H : \mathbb{N} \to \mathbb{N}$ is a efficiently computable (i.e., there is a deterministic Turing machine that on input 1^n outputs $1^{H(n)}$ in $\mathrm{poly}(n)$ time). Then, $\mathrm{SAT}_H \in \mathsf{NP}$.*

Proof. Given an input $\varphi\#0^k$ of length n, the machine just needs to check if the padding is correct (which can be done in deterministic $\mathrm{poly}(n)$ time certainly), and then check if $\varphi \in \mathrm{SAT}$. All of this can be done by a non-deterministic polynomial time machine. □

8.1.2 Constructing the right H

The goal right now is the following — design a function $H : \mathbb{N} \to \mathbb{N}$ such that

1. H is efficiently computable.
2. $H(m) \to \infty$ as $m \to \infty$ (assuming $\mathsf{P} \neq \mathsf{NP}$).
3. Ensure that there does not exists any reduction from SAT $\to \mathrm{SAT}_H$.

Here is a definition that satisfies the above conditions. The value of $H(m)$ is defined via the following procedure:

> $H(m)$ is defined as the smallest $i < \log\log m$ (if any exists) such that the machine M_i, when clocked at at most n^i steps, correctly solves SAT_H on all strings x of length at most $\log\log m$.
>
> If no such i exists, then $H(m)$ is defined to be $\log\log m$.

Intuitively, $H(m) = i$ (if $i < \log\log m$) is indicating that M_i is the first machine that appears to solve SAT_H correctly (for $\log\log m$-length strings).

The above definition may appear a bit circular, but note that $H(m)$ only depends on the values of H on $i \leq \log\log m$. Thus, the above definition is indeed well-defined. It is not hard to see that H is also efficiently computable.

Observation 8.6. *$H(m)$ is computable in $\mathrm{poly}(m)$ time.*

Proof. To compute $H(m)$, we can list all strings y of length at most $\log\log m$ and machines M_i source codes with $i \leq \log\log m$, and simulate each M_i each y for at most $|y|^i$ steps. This can be done in at most $2^{O(\log\log m)} \cdot (\log\log m)^{O(\log\log m)} = \mathrm{poly}(m)$.

Based on the above simulations, we can iteratively compute $H(1), H(2), \ldots$ and eventually $H(m)$. □

Thus, by Observation 8.5, we now have that $\mathrm{SAT}_H \in \mathsf{NP}$.

Theorem 8.7 (Ladner's theorem)**.** *If* $\mathsf{P} \neq \mathsf{NP}$*, then* SAT_H *is not* NP*-complete under polynomial time many-one reductions.*

Proof. Let's consider the following two cases.

Case 1: H **is bounded.** Suppose $\max H(m) = a$ and say $H(m_0) = a$. Then, we have that the machine M_{m_0} correctly solves SAT_H for all inputs in at most n^{m_0} time. In particular $\text{SAT}_H \in \mathsf{P}$.

Then, we have an immediate reduction from SAT to SAT_H that maps any formula φ of length $m > m_0$ to $\varphi \# m^a$ (and maintain an appropriate mapping for all formulas of size at most m_0). Thus, in this case we have a reduction SAT to SAT_H. Since we observed that $\text{SAT}_H \in \mathsf{P}$, we have that $\mathsf{P} = \mathsf{NP}$, contradicting our assumption. Hence, this case isn't possible.

Case 2: H **is unbounded.** This in particular means that $\text{SAT}_H \notin \mathsf{P}$ (for otherwise). We will show that SAT_H cannot be NP-complete if $\mathsf{P} \neq \mathsf{NP}$.

Let $M_1, M_2, \cdots$ be an enumeration of deterministic Turing machines computing functions $\Sigma^* \to \Sigma^*$, with M_i clocked at n^i steps. Suppose on the contrary that the machine M_c does compute a legitimate reduction $f : \Sigma^* \to \Sigma^*$ from SAT to SAT_H in n^c time. Let n_0 be the largest integer such that $h(n_0) \leq 2c$.

We now present a polynomial time algorithm for SAT. On an input formula φ of length n, let $y = f(\varphi)$. If y is not of the form $\psi \# 0^k$ or if $k \neq m^{H(m)}$ where $m = |\psi|$, the algorithm returns No. If $|\psi| \leq n_0$, we brute-force and check if $\psi \in SAT$. If $abs\psi > n_0$, note that $n^c \geq k > m^{2c}$. Hence, $|\psi| = m < \sqrt{n} = \sqrt{|\varphi|}$. Thus we have that $\varphi \in \text{SAT}$ if and only if $\psi \in \text{SAT}$ and we have reduced the length from n to $\sqrt{n}$. Recursing, we eventually reduce to a constant-sized formula in polynomial time. Thus we have a polynomial time algorithm for SAT, which contradicts the assumption that $\mathsf{P} \neq \mathsf{NP}$. Hence, we have that SAT_H cannot be NP-hard. □

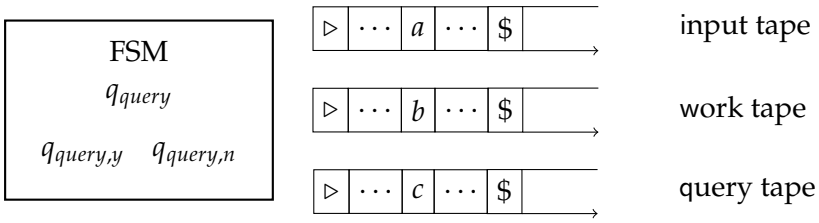
Lecture 9

Oracle Machines

Scribe: Siddharth Choudary

So far, we have covered various topics in applying "diagonalization". We will see that diagonalization also has its limitations.

9.1 Oracle Turing Machines



Definition 9.1 (Oracle Turing Machines)**.** *An Oracle Turing Machine M^L corresponding to the language L is a machine with access to an oracle which answers instantly whether the queries given by M is in L. Thus they use additional states q_{query} to query the oracle, $q_{query,y}$ when the oracle return "YES" and $q_{query,n}$ when the oracle returns "NO".* ◊

If A is some language, then P^A is the class of languages which are accepted by poly-time deterministic turing machines with access to oracle A.

Observation 9.2. $A \in \mathsf{P} \implies \mathsf{P}^A = \mathsf{P}$.

Observation 9.3. $A = \mathrm{SAT} \implies \mathsf{P}^A \supseteq \mathsf{P}, \mathsf{NP}, \mathsf{coNP}$.

Note: $\mathsf{NP}^{\mathrm{SAT}}$ is believed to be bigger than NP.

Observation 9.4 (Time hierarchy theorem with oracles)**.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ be non-decreasing time-constructible functions with $f(n) \log f(n) = o(g(n))$. Then, for any language A, we have $\mathsf{DTIME}^A(f) \subsetneq \mathsf{DTIME}^A(g)$.*

The proof is almost identical as it only involves machine simulations (the simulator machine can make the same oracle queries whenever the simulated machine intends to make one). This is a feature of a 'diagonalization' proof as we only rely on simulations and flipping

answers. However, this strength is also a weakness as it means that this technique can only prove results that remain true even in the presense of oracles.

The following theorem says that the "P vs NP" is not such an oracle-oblivious statement.

Theorem 9.5 (Baker-Gill-Solovay)**.** *There exists a language A s.t. $\mathsf{P}^A \neq \mathsf{NP}^A$ and there exists a language B s.t. $\mathsf{P}^B = \mathsf{NP}^B$.*

Proof. ii) Taking $B \in \mathsf{EXP}$-complete (under polynomial time many-one reductions), we will show that $\mathsf{P}^A = \mathsf{NP}^A = \mathsf{EXP}$.

a) $L \in \mathsf{EXP} \subseteq \mathsf{P}^B$: Since B is EXP-Complete, there is a poly-time reduction f s.t. $x \in L$ iff $f(x) \in B$.
$P^B \subseteq \mathsf{EXP}$: Solve B in EXP time at each query.
Therefore $\mathsf{P}^B = \mathsf{EXP}$.

b) $\mathsf{EXP} \subseteq \mathsf{NP}^B$: as $\mathsf{EXP} \subseteq \mathsf{P}^B \subseteq \mathsf{NP}^B$.
$\mathsf{NP}^B \subseteq \mathsf{EXP}$: Run every possible path and solve B when query is made.

i) For language A let $L_A = \{1^n \ : \$ there is a string of length n in $A\}$.

Lemma 9.6. *For any choice of A, we have that $L_A \in \mathsf{NP}^A$.*

Proof. Guess a string of length n and make oracle call to A. □

The goal is now to build the language A such that $L_A \notin P^A$.

Let $M_1, M_2, ...$ be the enumeration of all deterministic poly-time Oracle Turing Machines. We will build A in phases, where phase i will be responsible for ensuring that M_i^A does not correctly decide L_A.

Phase i:

Step 1: Clock all simulations of M_i at n^i steps. Choose a smallest n_i such that $n_i \notin \{n_1, ..., n_{i-1}\}$ and $2^{n_i} > n_i^i$.

Step 2: Simulate M_i on 1^{n_i} for n_i^i steps. Whenever M_i makes an oracle query, answer consistently if we have decided on those strings in previous phases, and default to "No" (when queried about any currently undecided string).

Step 3a: If M_i rejects 1^{n_i}, take some string of length n_i that was not queried by M_i and add it to A.

Step 3b: If M_i accepts 1^{n_i}, don't add any string of length n_i to A.

Hence, the language L_A contradicts every M_i on some input 1^{n_i}. Therefore $L_A \notin \mathsf{P}^A$.
Therefore, we have a language A s.t. $\mathsf{P}^A \subsetneq \mathsf{NP}^A$. □

Lecture 10

Mahoney's theorem and Polynomial Hierarchy

Scribe: Soumyajit Pyne

10.1 Search to decision reduction in NP

Suppose $\varphi(x_1, x_2, ..., x_n)$ is a boolean formula on n variable. We want to decide if φ is satisfiable or not. Note that $\varphi(x_1, x_2, ..., x_n)$ is satisfiable if and only if $\varphi(x_1 = 0, x_2, ..., x_n)$ or $\varphi(x_1 = 1, x_2, ..., x_n)$ is satisfiable. Thus, the task of determining whether an n-variable formula is satisfiable or not reduces to determining the satisfiability of two smaller-length formulas, which is referred to as the self-reducibility of SAT.

Definition 10.1 (Search-SAT)**.** SearchSAT *is a function* $f : \Sigma^* \to \Sigma^* \cup \{\perp\}$ *defined as follows.*

$$f(\varphi) = \begin{cases} \perp & \varphi \text{ is unsatisfiable} \\ x \in \{0,1\}^* \text{ such that } \varphi(x) = 1 & \text{otherwise} \end{cases}$$

$\Diamond$

Theorem 10.2. SearchSAT $\in FP^{SAT}$.

Proof. Now we propose a polytime algorithm which uses SAT as a subroutine.

Algorithm 1 SearchSAT$(\varphi(x_1, x_2, ..., x_n))$

if SAT$(\varphi(x_1, x_2, ..., x_n))$ is false **then**
    Return $\perp$
end if
if $n = 1$ **then**
    if $\varphi(1)$ is true **then**
        Return 1
    else
        Return 0
    end if
end if
if SAT$(\varphi(x_1 = 0, x_2, ..., x_n))$ is true **then**
    Return $0 \cdot$ SearchSAT$(\varphi(0, x_2, ..., x_n))$
else
    Return $1 \cdot$ SearchSAT$(\varphi(1, x_2, ..., x_n))$
end if

Algorithm 1 runs in poly$(|\varphi|)$ time with the help of the subroutine SAT which concludes the proof. □

10.2 Mahoney's theorems

Mahoney's theorems show that certain kinds of "simple" languages cannot be NP-hard unless P = NP. We will see a few instantiations of this.

These results follow a general template for an algorithm to solve SAT, assuming a suitable "Prune" operation.

Algorithm 2 SATSolver$(i, n, B = \{\varphi_1, \ldots, \varphi_k\})$

Input: An integer i, and a list of formulas
Output: Yes if and only if at least one of the formulas is satisfiable

if $i = n$ and any φ_i is True **then**
    Return true
end if
if $i = n$ and all φ_i are False **then**
    Return False
end if
$B' = \{\varphi(x_i = 0) \ : \ \varphi \in B\} \cup \{\varphi(x_i = 1) \ : \ \varphi \in B\}$.
$B' =$ Prune(n, B')
Return SATSolver$(i + 1, n, B')$.

In words, we just try both values for the variable x_i, thus potentially doubling the size of B, and "pruning" the size (which at the moment we haven't specified how), and repeating until we find a true formula in our list. To check if a given formula φ is satisfiable, we will just call SATSolver$(1, \{\varphi\})$.

10.2.1 Unary languages

Definition 10.3 (Unary Language)**.** *L is a unary language if $L \subseteq \{1\}^*$.* ◊

Theorem 10.4 (Mahoney's theorem for unary languages)**.** *If L is unary and* NP*-hard, then* $\mathsf{P} = \mathsf{NP}$.

Proof. We will use Algorithm 2 and appropriately specific the Prune function using L. As L is NP-hard, there is an n^c time (for some constant c) reduction $f : \Sigma^* \to \Sigma^*$ from SAT to L: i.e., $x \in \text{SAT}$ if and only if $f(x) \in L$.

We now specific the prune function.

Algorithm 3 PruneUnary$(n, B = \{\varphi_1, \ldots, \varphi_r\})$

1: **if** $|B| \leq n^c$ **then**
2: Return B
3: **end if**
4: Remove all $\varphi \in B$ such that $f(\varphi) \notin \{1\}^*$.
5: Compute $y_i = f(\varphi_i)$ for each $\varphi_i \in B$.
6: For each y, retain in B at most one φ_i such that $f(\varphi_i) = y$ and discard other collisions.
7: Return B.

We claim that Algorithm 2 instantiated with Algorithm 3 correctly solves SAT in polynomial time.

Note that f runs in n^c time, therefore $|f(\varphi)| \leq |\varphi|^c$. Algorithm 3 only discards formulas when it is guaranteed that it is not the only satisfiable formula in the set (if $f(\varphi_i) = f(\varphi_j)$, then either both are satisfiable or neither are; thus retaining just one of them is safe).

Furthermore, the set B returned by Algorithm 3 consists of formulas that under f yield distinct strings of the form $\{1\}^*$ and thus can have size at most n^c since f runs in time at most n^c. Thus, we always maintain that the set of formulas is no larger than $2 \cdot n^c$ and that there is always one satisfiable formula in this set if the original φ was satisfiable.

It is easy to see that the running time is also polynomial since the size of the set of formulas never exceeds $2n^c$ and all other operations are polynomial time operations. We conclude that $\text{SAT} \in \mathsf{P}$ which implies $\mathsf{P} = \mathsf{NP}$. □

10.2.2 Sparse languages

Definition 10.5 (Sparse Language)**.** *$L \subseteq \{0,1\}^*$ is said to be sparse if $\exists c, n_0$ such that $\forall n \geq n_0$ $|L \cap \{0,1\}^{\leq n}| \leq n^c$.* ◊

Definition 10.6 (co-Sparse Language)**.** *$L \subseteq \{0,1\}^*$ is said to be co-sparse if $\overline{L}$ is sparse.* ◊

Theorem 10.7 (Mahoney's theorem for co-sparse languages)**.** *If L is co-sparse and* NP*-hard, then* $\mathsf{P} = \mathsf{NP}$.

Proof. Once again, it suffices to specific the Prune function in this case. As $\overline{L}$ is sparse, $\exists c_1, n_0$ such that $\forall n \geq n_0$ $|\overline{L} \cap \{0,1\}^{\leq n}| \leq n^{c_1}$. As L is NP-hard, there is an n^{c_2} time reduction $f : \Sigma^* \to \Sigma^*$ such that $x \in \text{SAT}$ if and only if $f(x) \in L$.

Algorithm 4 PruneCoSparse$(n, B = \{\varphi_1, \ldots, \varphi_r\})$

1: **if** $|B| \leq n^{2c_1c_2}$ **then**
2: Return B
3: **end if**
4: Compute $y_i = f(\varphi_i)$ for each $\varphi_i \in B$.
5: **if** all y_i's are distinct **then**
6: Return {True}
7: **end if**
8: For each y, retain in B at most one φ_i such that $f(\varphi_i) = y$ and discard other collisions.
9: Return B.

The only part to justify in the above algorithm is why we can assume that the formula is satisfiable if all y_i's are distinct (Line 6). If it were the case that all the φ_i's were unsatisfiable, then $f(\varphi_i) \in \overline{L}$ for all i. But then, $|f(\varphi_i)| \leq n^{c_2}$ and $\left|\overline{L} \cap \{0,1\}^{\leq n^{c_2}}\right| \leq n^{c_1c_2}$ and therefore we do not have more $n^{c_1c_2}$ within these. Thus, we must have that at least one of $f(\varphi_i)$ must be in L and that is possible if and only if φ_i is satisfiable (since f is a valid reduction).

That concludes the theorem. □

The following statement would be given in the problem set.

Theorem 10.8 (Mahoney's theorem for sparse languages)**.** *If L is sparse and* NP*-hard, then* $\mathsf{P} = \mathsf{NP}$.

10.3 Polynomial Hierarchy

Suppose $\mathcal{C}$ is a class of languages.

Definition 10.9 ($\exists \cdot \mathcal{C}$)**.** *A language $L \in \exists \cdot \mathcal{C}$ if and only if there is a predicate $R \in \mathcal{C}$ such that $x \in L \iff \exists y$ of length poly$(|x|)$ such that $R(x, y) =$ True.* ◊

Definition 10.10 ($\forall \cdot \mathcal{C}$)**.** *A language $L \in \exists \cdot \mathcal{C}$ if and only if there is a predicate $R \in \mathcal{C}$ such that $x \in L \iff \forall y$ of length poly$(|x|)$ such that $R(x, y) =$ True.* ◊

It is immediate to check from the definition that $\exists \cdot \mathsf{P} = \mathsf{NP}$ and $\forall \cdot \mathsf{P} = \mathsf{coNP}$. It is also not hard to check that $\exists \cdot \mathsf{NP} = \mathsf{NP}$ and $\forall \cdot \mathsf{coNP} = \mathsf{coNP}$. But what about classes such that $\exists \cdot \mathsf{coNP}$ and $\forall \cdot \mathsf{NP}$? How powerful are these?

The following are not too difficult to see.

Claim 10.11. $\forall \cdot \mathsf{NP} \subseteq \mathsf{coNP}^{\mathrm{SAT}}$.

Claim 10.12. $\exists \cdot \mathsf{coNP} \subseteq \mathsf{NP}^{\mathrm{SAT}}$.

Turns out, both of the above are equalities! On the face of it, it seems like $\mathsf{NP}^{\mathrm{SAT}}$ is more powerful (as it can make adaptive queries, whereas $\exists \cdot \mathsf{coNP}$ seems like a single query). We will see that in the subsequent lectures.

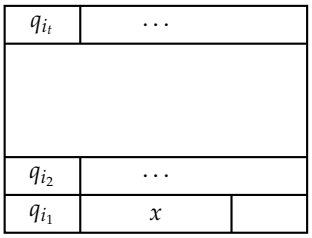
Lecture 11

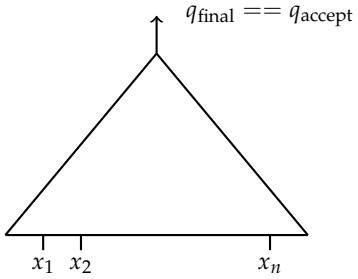
Polynomial Hierarchy

Scribe: Bikshan Chatterjee

Recall the Circuit-Eval where we are given a circuit C and an input x and want to check if $C(x)$ is true. This is morally equivalent to asking if a given deterministic Turing machine M accepts a certain input x within a pre-specific amount of time.

Given a TM M and input x (variable), the circuit (boolean formula) that encodes the statement "M accepts x" can be constructed using the computational tableau. Each row of the tableau can be computed from the previous row using basic (AND, OR, NOT) computations.





Deterministic TMs:

The length of x ($|x| = n$) has to be fixed. Then the input x to the machine can be used as the input to the circuit. The first row is given by the initial state and x and with worktapes initialied to blanks. The other rows can be computed as logical formulae depending on the first row. The final circuit would be something of the form

$$\bigwedge_{i=2}^{t} [\text{Row } i \text{ consistent with Row } i-1] \wedge \left[q_{i_t} = q_{\text{accept}}\right]$$

Thus, we have a formula $\Phi_{M,t}(x)$, of size $O(\text{poly}(t, |x|))$, that evaluates to true if and only if M accepts x within t steps.

Non-Deterministic TMs:

Both x and the certificate/computational path y can be considered inputs to the circuit. The formula would encode "M accepts x on computational path y".

If we fix x in the language and use the certificate/computational path as input, NP corresponds to the circuit being satisfiable and coNP corresponds to the circuit being tautology.

11.1 The classes $\Sigma_i^{\mathsf{P}}, \Pi_i^{\mathsf{P}}$

We first define two 'operators' to build new classes for existing classes.

Definition 11.1 (The $\exists$ and $\forall$ operators for classes)**.** *For any class $\mathcal{C}$, we define $\exists\mathcal{C}$ to consist of all languages L such that there is polynomial bounded function $\ell : \mathbb{N} \to \mathbb{N}$ and a relation $R \in \mathcal{C}$ such that for all $x \in \Sigma^*$*

$$x \in L \iff \exists y \in \Sigma^{\leq \ell(|x|)} \ : \ R(x,y) = \textit{true}.$$

On a similar vein, $\forall\mathcal{C}$ is defined via a similar requirement

$\Diamond$ $\qquad x \in L \iff \forall y \in \Sigma^{\leq \ell(|x|)} \ : \ R(x,y) = \textit{true}.$

We know $\exists\mathsf{P} = \mathsf{NP}$ and $\forall\mathsf{P} = \mathsf{coNP}$. What is $\exists\mathsf{coNP}$? For any language $L \in \exists\mathsf{coNP}$, we have $x \in L \Leftrightarrow \exists y \ : \ R(x,y) = \text{true}$, for some $R(x,y) \in \mathsf{coNP}$. That is, $\exists y \ \forall z \ R'(x,y,z) = \text{true}$, for some $R' \in \mathsf{P}$ (z represents any computational path taken by the co-non-deterministic machine computing R).

Σ_2-SAT $= \{\exists y \ \forall z \ \varphi(y,z) : \text{ it is true}\}$. This is complete for $\exists\mathsf{coNP}$ using polynomial time many one reductions.
Given a language $L \in \exists\mathsf{coNP}$, in order to check if $x \in L$, we have a predicate $R \in \mathsf{coNP}$ such that $\exists y \ : \ R(x,y) = \text{true}$. Take the co-non-deterministic machine M that computes R, and encode the statement "M accepts (x,y) on computational path z" into the formula $\varphi_{M,x}(y,z)$.

On a similar vein, Π_2-SAT $= \{\forall y \ \exists z \ \varphi(y,z) : \text{ it is true}\}$. This is complete for $\forall\mathsf{NP}$ using polynomial time many one reductions.

More generally, for any i, Σ_i-SAT $= \{\exists x_1 \ \forall x_2 \ \ldots \ \exists/\forall x_i \ \varphi(x_1, x_2, \ldots, x_i) \ : \ \text{it is true}\}$.
Π_i-SAT $= \{\forall x_1 \ \exists x_2 \ \ldots \ \exists/\forall x_i \ \varphi(x_1, x_2, \ldots, x_i) \ : \ \text{it is true}\}$.

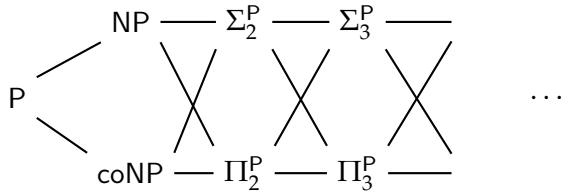
For now, we will describe classes of languages with a complete language from that class but we will have a more formal description later on.

Σ_i^{P} is the largest class of languages for which Σ_i-SAT is complete.

Π_i^{P} is the largest class of languages for which Π_i-SAT is complete.

$\Sigma_i^{\mathsf{P}} \subseteq \Sigma_{i+1}^{\mathsf{P}}$: because Σ_i-SAT can be reduced to Σ_{i+1}-SAT, by adding an unused quantifier at the end. Eg. $\exists x_1 \ \forall x_2 \ \exists x_3 \ \varphi(x_1, x_2, x_3)$ is true if and only if $\exists x_1 \ \forall x_2 \ \exists x_3 \ \forall x_4 \ \varphi(x_1, x_2, x_3)$ is true.
$\Sigma_i^{\mathsf{P}} \subseteq \Pi_{i+1}^{\mathsf{P}}$: Σ_i-SAT can be reduced to Π_{i+1}-SAT by adding an unused quantifier at the beginning.

Similarly, $\Pi_i^{\mathsf{P}} \subseteq \Sigma_{i+1}^{\mathsf{P}}$ and $\Pi_i^{\mathsf{P}} \subseteq \Pi_{i+1}^{\mathsf{P}}$.



The class PH is defined as $\mathsf{PH} = \bigcup_i \Sigma_i^P$. Recursively: $\Sigma_1^P = \mathsf{NP}$, $\Pi_1^P = \mathsf{coNP}$, $\Sigma_i^P = \exists \Pi_{i-1}^P$, $\Pi_i^P = \forall \Sigma_{i-1}^P$.

Theorem 11.2. *If $\Sigma_i^P = \Pi_i^P$, for some i, then $\mathsf{PH} = \Sigma_i^P = \Pi_i^P$.*

Proof. We will prove the above statement for $i = 1$, although the same ideas work for all i.

Recall $\Sigma_1^P = \mathsf{NP}$, $\Pi_1^P = \mathsf{coNP}$. Suppose $\Sigma_1^P = \Pi_1^P$, i.e. $\mathsf{NP} = \mathsf{coNP}$.

Let $L \in \Pi_2^P$. That is for any x,

$$\begin{aligned} x \in L &\Leftrightarrow \forall y\ R(x,y) = \text{true, where } R(x,y) \in \Sigma_1^P = \mathsf{NP} \\ &\Leftrightarrow \forall y\ \exists z\ R'(x,y,z) = \text{true, where } R'(x,y,z) \in \mathsf{P}. \end{aligned}$$

The language of the predicate R is $L' = \{(x,y)\ :\ \exists z\ R'(x,y,z) = \text{true}\}$. This language is in NP. Because $\mathsf{NP} = \mathsf{coNP}$, it is also in coNP. So there must be a predicate $R''(x,y,z) \in P$ such that

$$\begin{aligned} &L' = \{(x,y)\ :\ \forall z\ R''(x,y,z) = \text{true}\} \\ &z \text{ has to be of size poly}(|x| + |y|) \end{aligned}$$

That is, for any (x,y)

$$\exists z\ R'(x,y,z) = \text{true} \ \Leftrightarrow\ \forall z\ R''(x,y,z) = \text{true}$$

So we have

$$\begin{aligned} &x \in L \ \Leftrightarrow\ \forall y\ \forall z\ R''(x,y,z) = \text{true} \\ &\text{that is, } \forall yz\ R''(x,y,z) = \text{true, where } R''(x,y,z) \in \mathsf{P} \\ &\text{size of } y \text{ is poly}(|x|), \text{ size of } z \text{ is poly}(|x| + |y|) \end{aligned}$$

So, $L \in \mathsf{coNP}$, $\Pi_2^P \subseteq \mathsf{coNP}$. Similarly it can be shown $\Sigma_2^P \subseteq \mathsf{NP}$.

For $L \in \Sigma_2^P$, checking if $x \in L$ is equivalent to computing predicate of the form

$$\begin{aligned} &\exists y\ \underbrace{\forall z\ R(x,y,z)} \\ &\quad \text{replace with} \\ &\quad \exists z\ R'(x,y,z) \end{aligned}$$

That shows that Π_2^P is in fact equal to $\mathsf{coNP} = \mathsf{NP}$ and this can be repeated to larger (but constant!) number of quantifiers. □

11.2 The language TQBF

TQBF is the set of true quantified boolean formulas. Quantified boolean formulas are formulas of the form $\exists x_1\ \forall x_2\ \exists x_3\ \ldots\ \forall x_n\ \varphi(x_1, x_2, \ldots, x_n)$ for any n (can start or end with $\exists$ or $\forall$).

$$\text{TQBF} = \{\exists x_1\ \forall x_2\ \exists x_3\ \ldots\ \forall x_n\ \varphi(x_1, \ldots, x_n)\ :\ \text{it is true}\}$$

It is important to stress that proof of Theorem 11.2 *does not* extend to show $\mathsf{NP} = \mathsf{coNP} \Rightarrow \text{TQBF} \in \mathsf{P}$.
Removing one quantifier can increase the formula size to a polynomial of the original size. Doing this a constant number of times results in a polynomial sized formula. But doing it n times (part of input) can result in exponential sized formula.

$$\begin{aligned} &\exists x_1\ \forall x_2\ \exists x_3\ \varphi(x_1, x_2, x_3) \\ \Leftrightarrow\ &\exists x_1\ \forall x_2\ \forall x_3\ \varphi'(x_1, x_2, x_3) \end{aligned}$$

11.3 Polynomial hierarchy via oracle machines

$\Sigma_2\,\text{SAT} = \{\exists y\ \forall z\ \varphi(y, z)\ :\ \text{it is true}\}$.

This is certainly in NP^{SAT}. The oracle machine would guess y and ask the oracle if $\neg\varphi(y, z)$ is satisfiable. If the answer is yes, then for the particular y, $\forall z\ \varphi(y, z)$ is false. If the answer is no, for the particular y, $\forall z\ \varphi(y, z)$ is true.

$\Sigma_2\,\text{SAT}$ is complete for Σ_2^{P}. This shows $\Sigma_2^{\mathsf{P}} \subseteq \mathsf{NP}^{\text{SAT}}$. However, it still leaves the question: Is $\mathsf{NP}^{\text{SAT}} \subseteq \Sigma_2^{\mathsf{P}}$ true?

It seems unlikely as the oracle machine can make multiple and adaptive unsatisfiability queries (asking a SAT query and going down the "no" path), while the formula in Σ_2-SAT is restricted to a single unsatisfiability query at the end (the $\forall z\ \varphi(y, z)$ formula). (Satisfiability queries can be encoded in the $\exists y$ part).

However, turns out it is indeed the case that $\Sigma_2^{\mathsf{P}} = \mathsf{NP}^{\text{SAT}}$

Proposition 11.3. $\mathsf{NP}^{\text{SAT}} = \Sigma_2^{\mathsf{P}}$

Proof. The idea is to guess queries and answers. The NP^{SAT} machine accepting a string x would be encoded as the Σ_2-SAT statement:

$\exists$ computational path y $\exists$ SAT queries $q_1, \ldots, q_n$ $\exists$ answers $a_1, \ldots, a_n$ to the queries
$\exists$ satisfying assignment for yes answers $\sigma_1, \ldots, \sigma_m$
"on path y, queries made by $M(x)$ on path y are correct conditioned on
previous answers being correct and for yes answers σ_i are satisfying
and $\underbrace{\text{for no answers the answers are correct"}}_{\text{check unsatisfiability using } \forall \text{ formula}}$

Check if sat formulas $\varphi_1, \varphi_2, \ldots, \varphi_{n-m}$ are all unsatisfiable by the truth of

$$\forall \text{ variables in } \varphi_i \quad \neg(\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_{n-m})$$

This shows $\mathsf{NP}^{\mathrm{SAT}} = \Sigma_2^{\mathsf{P}}$. □

Thus, we can alternatively define the classes using oracle machines as follows: $\Sigma_1^{\mathsf{P}} = \mathsf{NP}$, $\Pi_1^{\mathsf{P}} = \mathsf{coNP}$. $\Sigma_2^{\mathsf{P}} = \mathsf{NP}^{\Sigma_1^{\mathsf{P}}}$, $\Pi_2^{\mathsf{P}} = \mathsf{coNP}^{\Sigma_1^{\mathsf{P}}}$

In general, $\Sigma_i^{\mathsf{P}} = \mathsf{NP}^{\Sigma_{i-1}^{\mathsf{P}}}$.

11.4 On complete problems for the polynomial hierarchy

For NP, we have seen that Circuit-SAT is complete. For Σ_2^{P}, Σ_2 SAT is the natural complete problem.

Is there a complete problem for PH?

Suppose L is a complete language for PH. $L \in \mathsf{PH} = \bigcup_i \Sigma_i^{\mathsf{P}}$ so $L \in \Sigma_i^{\mathsf{P}}$ for some i.

So every problem in PH can be reduced to a problem in Σ_i^{P}, then $\mathsf{PH} = \Sigma_i^{\mathsf{P}}$.

Thus, there are no complete problems for the entireity of PH unless the polynomial hierarchy collapses.

Lecture 12

Space Complexity

Scribe: Ashutosh Singh

In this lecture, we start considering space as a resource. So far we have been trying to classify problems based on the time it takes to solve them using some Turing machine, but now time is of no concern, we just want to consider the 'space used' by a Turing machine M to solve a particular problem. Again, we only consider halting TMs. Now recall that a Turing machine M has an input tape, k work-tapes, and an output tape. For our purposes, we will only allow the input tape to be *read only* and the output tape to be only *write once*. All the k work-tapes will have full read-write access. Then for the TM M, we define space used by M as, $S_M : \mathbb{N} \to \mathbb{N}$, such that, $S_M(n) =$ maximum number of work-tape cells accessed by TM M on any input of length n. Then similar to $\mathsf{DTIME}(T_M(n))$ and $\mathsf{NTIME}(T_M(n))$, we have $\mathsf{DSPACE}(S_M(n))$ and $\mathsf{NSPACE}(S_M(n))$, if M is a DTM or an NTM respectively and uses space $O(S_M(n))$ on any input of length n. Although technically we should write $S_M(n)$ to denote the space used by the machine on the input of length n, we drop the subscript M for brevity and also because it's clear from the context what we mean. Observe that we didn't consider the space used by the input tape in defining the space used by the TM M because for as simple a problem as PARITY, where we have a binary string as an input and we accept if and only if has an odd number of 1's. Now clearly this can be done using a constant number of work tape cells but if we were to charge for the space used by the input tape too then PARITY would belong to $\mathsf{DSPACE}(n)$.

Now consider the following language,

$$\text{Dir-ST-Conn} = \left\{ (G, s, t) \ : \ \begin{array}{c} G = (V, E) \text{ is a graph with vertices } s, t \\ \text{and there is a path from } s \text{ to } t \end{array} \right\}$$

Observe that Dir-ST-Conn $\in \mathsf{DSPACE}(n)$ because our classic graph traversal algorithm DFS would need to store vertices explored so far in the *current* path which is at most n. But if we use the power of non-determinism, then Dir-ST-Conn $\in \mathsf{NSPACE}(\log n)$, because starting from vertex s, at every step, we could just *guess* the next vertex adjacent to the current one and accept if, in the end, we reach t. We just have to be slightly careful so that we are not stuck in the loop of guessing, so we maintain a counter and reject if the counter exceeds n because, in an n vertex

graph, any path is of length at most n. Now surprisingly, Undir-ST-Conn $\in \mathsf{DSPACE}(\log n)$ as was shown by Reingold.

Similar to the time hierarchy theorems, we also have, the space hierarchy theorems, and similar to the notion of time constructible functions, here we have space constructible functions.

Definition 12.1 (Space Constructible Function)**.** *A function $f : \mathbb{N} \to \mathbb{N}$ is called space constructible if there is a TM M, such that $M(1^{(n)}) = 1^{f(n)}$ and $S_M(n) = O(f(n))$.* $\Diamond$

Theorem 12.2 (Space Hierarchy Theorem)**.** *For any two space constructible functions f and g, such that $f(n) = o(g(n))$, we have,*

$$\mathsf{DSPACE}(f(n)) \subsetneq \mathsf{DSPACE}(g(n)), \text{ and}$$
$$\mathsf{NSPACE}(f(n)) \subsetneq \mathsf{NSPACE}(g(n)).$$

Observe that, unlike the deterministic time hierarchy theorem, we don't have any logarithmic overhead in the space hierarchy theorem because UTM won't need to use any extra *space* in simulating any Turing machine.

Now, similar to P,NP etc., we have the following analogous space classes,

$$\begin{aligned}
\mathsf{PSPACE} &= \bigcup_{c \geq 0} \mathsf{DSPACE}(n^c) \\
\mathsf{NPSPACE} &= \bigcup_{c \geq 0} \mathsf{NSPACE}(n^c) \\
\mathsf{LOGSPACE} &= \mathsf{DSPACE}(O(\log n)) \\
\mathsf{NL} &= \mathsf{NSPACE}(O(\log n))
\end{aligned}$$

12.1 Relation between classes and the configuration graph

Observation 12.3. $\mathsf{NP} \subseteq \mathsf{PSPACE}$

The easiest way to see why the above observation follows is to consider SAT, and clearly, all we would need is to enumerate all possible assignments at a time and check if it is a satisfying assignment. The space can be *reused* and thus the observation.

Then we also have the following set of inclusions which are easy to see.

$$\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$$

Recall in the proof of Cook-Levin theorem how we used this notion of computational tableaus and used it to show that any language in $\mathsf{NP} \leq_{\mathsf{P}}$ Circuit-SAT. In fact, in the last lecture, we saw that any language that is decided by an NTM in some time, t, amounts to checking *satisfiability* of a circuit that is an encoding of the computational tableau of that NTM on some input. There

is a similar notion for a language that is decidable by a Turing machine using some space s, called a configuration graph. A *configuration* of a TM M on input x corresponds to all the data that one would need to store if we were to halt it at some step in its run and then continue it from that point forward sometime in the future. So as can be checked, we would need to store the following data,

1. Current state of M
2. Current head positions
3. All the work tape content

Then the next question is how many configurations are possible of a TM M on input x that uses space $f(n)$. The total number of states is $|Q|$, the total number of head positions, assuming that M is just a 2 work tape machine is $n \cdot (f(n))^2$ and finally the total different possibilities for work tape content is $\Sigma^{f(n))}$. Thus the total number of configurations is,

$$|Q| \cdot n \cdot (f(n))^2 \cdot |\Sigma|^{f(n)} = 2^{O(f(n))}$$

Now a configuration graph for machine M and input x, denoted by $G_{M,x}$ is a directed graph (V, E) where,

$$V = \{C_i \ : \ C_i \text{ is a possible configuration in the run of } M \text{ on } x\}$$

$$E = \left\{(C_i, C_j) \ : \ C_i, C_j \text{ is a configuration such that running one step of } M \text{ on } x \text{ from } C_i \text{ leads to } C_j\right\}$$

Note that if M is a DTM then every vertex in $G_{M,x}$ has out-degree of 1 except for the vertex C_{accept} that represents the accepting configuration of M on input x and if M is an NTM then every vertex except C_{accept} will have out-degree 2 and since we are considering only halting TM's $G_{M,x}$ would be acyclic. Now that we understand the notion of a configuration graph, observe that for an input x, $x \in L$ iff M accepts x using space $O(f(n))$. But equivalently we have, $x \in L$ iff there is a path from C_{start} to C_{accept} in $G_{M,x}$, where C_{start} represents the initial configuration of M on input x.

Lemma 12.4. $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{f(n)})$

Proof. Consider an $L \in \mathsf{NSPACE}(f(n))$, let M be the machine such that $L(M) = L$. Then

$$x \in L \Leftrightarrow (G_{M,x}, C_{\text{start}}, C_{\text{accept}}) \in \text{Dir-ST-Conn}$$

Since $G_{M,x}$ has potentially $2^{O(f(n))}$ vertices we can just construct $G_{M,x}$ which would take $2^{O(f(n))}$ time and then run DFS which would take linear time. □

12.2 Completeness of TQBF

In the last lecture, we were introduced to this language TQBF which we observed that it seemed more powerful than the whole polynomial hierarchy. Now we show that in fact, TQBF is PSPACE-complete.

Theorem 12.5. TQBF *is* PSPACE*-complete under polynomial-time many-one reductions.*

Proof. In the first part of the proof, we show that TQBF $\in$ PSPACE. Consider a TQBF formula $\Psi_{n,m}$ where n represents the number of quantified variables and m is the length of the expression $\varphi(x_1, \cdots, x_n)$, i.e.,

$$\Psi_{n,m} = \exists x_1 \forall x_2 \cdots \exists / \forall \varphi(x_1, \cdots, x_n).$$

Let $S(\Psi_{n,m})$ represent the space that would be needed to solve $\Psi_{n,m}$. As a base case observe that $S(\Psi_{0,m}) \leq m$ because $n = 0$ implies that there are no quantified variables and so we just need to check if φ is satisfiable or not. Then we can define $S(\Psi_{n,m})$ inductively as,

$$S(\Psi_{n,m}) = S(\Psi_{n-1,m}) + O(m) + \theta(1)$$

Because assuming that x_1 to x_n are Boolean variables we could set x_1 to 0, evaluate $\varphi(0, x_2, \cdots, x_n)$ which would take $O(m)$ space and then evaluate,

$$\forall x_2 \cdots \exists / \forall \varphi(0, x_2, \cdots, x_n),$$

which would take $S(\Psi_{n-1,m})$ space and then just store the final result which would take $\theta(1)$ space. Since x_1 is quantified by an existential quantifier we check if either setting x_1 to 0 or 1 results in TRUE for $\Psi_{n,m}$. If either assignment gives TRUE then we accept else reject. Similarly, if there were a universal quantifier we would need to check that it is satisfiable for both 0 and 1. From the above equation, it is easy to see that $S(\Psi_{n,m}) = O(nm)$ and thus TQBF $\in$ PSPACE. Next, we will prove that TQBF is PSPACE-hard. Let $L \in$ PSPACE and M be its corresponding machine that takes space $S(n)$. Then on an input x, we would like to output a TQBF instance $\Psi_{n,m}(s, t)$ such that,

$$x \in L \Leftrightarrow \Psi_{n,m}(s, t) = \text{TRUE},$$

where $\Psi_{n,m}(s, t)$ is true iff "s and t are connected by a path of length at most m in $G_{M,x}$". As can be guessed, s would correspond to C_{start} and t would correspond to C_{accept} and $m = 2^{O(S(n))}$ and if we can make sure that $\text{size}(\Psi_{n,m}(s, t)) = O(S(n))$ then we would be done. Towards that end, observe that if there is a path from s to t of length m, then there must be an intermediate vertex u such that there is a path of length $m/2$ from s to u and a path from u to t of the same length. This suggests that we can write,

$$\Psi_{n,m}(s, t) = \exists u \; \Psi_{n,m/2}(s, u) \wedge \Psi_{n,m/2}(u, t).$$

But then $\text{size}(\Psi_{n,m}(s, t)) \leq S(n) + 2 \cdot \text{size}(\Psi_{n,m/2})$, where the first term is because there are potentially $2^{O(S(n))}$ many vertices and so encoding of any vertex would require $O(S(n))$ space. This means that the above formula would lead to an exponential blow-up in the size. But observe that we haven't used the universal quantifier. And so alternatively we can write,

$$\Psi_{n,m}(s, t) = \exists u \; \forall (a, b) \in \{(s, u), (u, t)\} \; \Psi_{n,m/2}(a, b).$$

Then we have, $\text{size}(\Psi_{n,m}(s,t)) = O(S(n)) + \text{size}(\Psi_{n,m/2})$ which is clearly polynomially bounded.

□

Lecture 13

Savitch's theorem

Scribe: Yeshwant Pandit

In today's lecture notes, we will see more properties of space classes. The first result we'll see is known as Savitch's theorem. A nice consequence of this result is NPSPACE = PSPACE.

13.1 The easier observation of PSPACE = NPSPACE

Let us first try to show that NPSPACE = PSPACE using the result discussed in the last lecture. In the previous lecture, we showed that TQBF is PSPACE-complete. Recall, to show that TQBF is PSPACE-hard, we considered a language $L \in$ PSPACE, and determining whether $x \in L$ was equivalent to determining whether there is a path from the start configuration C_{start} to the accept configuration C_{accept} in the configuration graph, and this was captured using a polynomially long TQBF formula. Notice that in the proof, there was no mention of out-degree being 1 or 2 in the configuration graph. This means that the same proof can be used to show that TQBF is NPSPACE-hard. Since TQBF $\in$ PSPACE, we have NPSPACE = PSPACE.

13.2 General case of Savitch's theorem

Theorem 13.1 (Savitch's theorem)**.** *For any space-constructible $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$,* $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{SPACE}(S(n)^2)$

Proof. Let $L \in \mathsf{NSPACE}(S(n))$ be a language decided by a TM M. For every $x \in \{0,1\}^n$, determining whether $x \in L$ is equivalent to checking if there is a path from C_{start} to C_{accept} in the configuration graph $G_{M,x}$. Note that the configuration graph has at most $2^{O(S(n))}$ vertices. We describe a recursive algorithm PATHEXISTS(C_1, C_2, i) (Algorithm 5) that returns "1" if there is a path of length at most 2^i from C_1 to C_2 in the configuration graph $G_{M,x}$ and "0" otherwise. Observe that there is a path of length at most 2^i from C_1 to C_2 in $G_{M,x}$ if and only if there is a configuration C' such that the paths from C_1 to C' and from C_2 to C' are at most 2^{i-1} long.

On receiving inputs C_1, C_2, and i, the algorithm PATHEXISTS for $i \geq 1$ goes over all vertices and outputs "1" if it finds a configuration C' such that both recursive calls PATHEXISTS$(C_1, C', i-1)$ and PATHEXISTS$(C', C_2, i-1)$ return "1". Note that the recursive call PATHEXISTS$(C', C_2, i-$

1) can reuse the space already used by PATHEXISTS$(C_1, C', i-1)$. Let us find out the space complexity of PATHEXISTS(C_1, C_2, i). For this, we define $\Delta_M(i)$ to be the space required by PATHEXISTS(C_1, C_2, i) on the worst pair of configurations C_1,C_2. Note that going over all vertices requires an extra $O(S(n))$ space. Since we reuse the space for the recursive invocations, we get the following recurrence.

$$\Delta_M(i) \leq \Delta_M(i-1) + O(S(n))$$

Since $\Delta_M(0) \leq O(S(n))$, solving the recurrence we get

$$\Delta_M(i) = O(i \cdot S(n))$$

Hence, we have $\Delta_M(O(S(n))) = O(S(n)^2)$. Thus, to determine whether $x \in L$, we set $C_1 = C_{\text{start}}, C_2 = C_{\text{accept}}$ and $i = O(S(n))$ and invoke PATHEXISTS. This invocation runs in $O(S(n)^2)$ space. This concludes the proof. □

Algorithm 5 PATHEXISTS

Input: C_1, C_2, and i.
Output: "1" if there is a path of length at most 2^i from C_1 to C_2 and "0" otherwise.

```
1: if i ≤ 0 then
2:     if (C_1 = C_2) ∨ (C_1, C_2) is an edge then
3:         return 1
4:     else
5:         return 0
6:     end if
7: else
8:     for all C' ∈ V(G_{M,x}) do
9:         a = PATHEXISTS(C_1, C', i − 1)
10:        b = PATHEXISTS(C', C_2, i − 1)
11:        if a ∧ b then
12:            return 1
13:        end if
14:    end for
15:    return 0
16: end if
```

Corollary 13.2.

- NPSPACE = PSPACE
- NL $\subseteq$ $\mathsf{L}^2 = \mathsf{DSPACE}(\log^2 n)$

Lecture 14

Log-space reductions and completeness

Scribe: Sourav Roy

14.1 Log-space reductions/Log-space Transducers:

Definition 14.1 (Logspace reductions)**.** *Let us take A, B to be two languages. We will say $A \leq_{\mathsf{L}} B$ if there exists a function f computable in log-space such that $x \in A$ iff $f(x) \in B$.* $\Diamond$

14.1.1 Some of log-space reductions

We list some simple properties of log-space reductions with brief proof outlines.

1. $A \leq_{\mathsf{L}} B$ and $B \in \mathsf{L} \implies A \in \mathsf{L}$.
   Outline of proof: Suppose $f : \Sigma^* \to \Sigma^*$ is the log-space computable reduction from A to B. The main issue with the naive approach is 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 whener 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).

2. $A \leq_{\mathsf{L}} B$ and $B \leq_{\mathsf{L}} C$ implies $A \leq_{\mathsf{L}} C$. [Similar proofs]

14.2 NL-completeness

Definition 14.2 (NL-completeness under log-space reductions)**.** *A is* NL*-complete under log-space reductions if*

1. *$A \in NL$*
2. *(*NL*-hardness) For all $B \in \mathsf{NL}$, we have $B \leq_{\mathsf{L}} A$.*

(Unless otherwise stated, we'll assume that NL*-completness is under log-space reductions.)* $\Diamond$

The following problem is the most natural NL-complete language — directed reachability, or simply Dir-Path.

Definition 14.3 (Dir-Path)**.** *The language* Dir-Path *is defined as*

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

Theorem 14.4. Dir-Path *is* NL*-Complete.*

Proof sketch. By guessing one vertex at a time, we can 'guess' a path from s to t, thus showing that Dir-Path $\in$ NL.

To show hardness, let $B \in$ NL and suppose M is the NL-machine deciding it. For an input x, we can build the 'configuration graph' $G_{M,x}$ for M on input x, and simply reduce the input x to the instance $\langle G_{M,x}, C_{x,\text{start}}, C_{x,\text{accept}}\rangle$ (where $C_{x,\text{start}}, C_{x,\text{accept}}$ are the 'start' and 'accept' configurations respectively).

It is not hard to check that this can be computed in log-space. □

Other complete problems for NL include checking if a graph G has a directed cycle.

For undirected graphs, there is a surprising result of Reingold that shows that the undirected version of the above problems are actually in L.

Theorem 14.5 (Reingold, 2005)**.** Undir-Path $\in$ L *(same for finding undirected cycles).*

14.3 Certificate / witness perspective of NL

Just like we could equivalently define NP via the verifier / certificate / witness approach, a natural question is whether there is a similar definition for NL. A first attempt would be something along the following lines:

> $A \in NL$ iff (?) There exists a deterministic logspace machine M such that $x \in L$ iff $\exists w$ with $|w| = \text{poly}(|x|)$ such that M accepts (x, w).

Unfortunately, the above definition also includes SAT (we can always check if a particular assignment satisfies a formula in log-space). Thus, we need to be careful.

We fix the issue by insisting that the witness-tape is *read-once* and that turns out to be an equivalent definition of NL.

Proposition 14.6 (Certificate / witness definition of NL)**.** *The set of languages L with the property:*

> *There is some deterministic machine log-space M with a read-once witness tape such that $x \in L$ if and only if there is some w with $|w| = \text{poly}(|x|)$ such that $M(x, w)$ accepts.*

exactly co-incides with the class NL*.*

Lecture 15

Immerman-Szelepcsényi theorem, Circuit Lower Bounds

Scribe: Soumyajit Pyne

15.1 Immerman-Szelepcsényi Theorem:

One of the most surprising results related to NL is the following theorem of Immerman and Szelepsényi.

Theorem 15.1 (Immerman and Szelepsényi.)**.** NL = coNL.

We will prove this by exhibiting a NL-machine that solves a coNL-complete problem, namely the complement of the Dir-Path problem.

Definition 15.2 ($\overline{\text{Dir-Path}}$)**.**

$$\overline{\text{Dir-Path}} := \{\langle G, s, t\rangle \ : \ G \text{ is a directed graph and there is no directed path from } s \text{ to } t\}.$$

$\Diamond$

Theorem 15.1 is proved by exhibiting a *certificate* that the vertex t is not reachable from s in the graph G and this is done via a technique that is now referred to as 'inductive counting'. On a high level, we will 'certify' that t is not reachable from s by instead certifying the number m of vertices that *are* reachable from s, and giving a certificate of reachability for all $n - m$ other vertices in G.

Proof sketch. We define the following sets:

$$R_i = \{u \in G \ : \ \text{There is a path of length at most } i \text{ from } s \text{ to } t\}.$$

Clearly, s and t are disconnected if and only if $t \notin R_n$.

Certainly membership in R_n can be certified easily — just provide the path as the certificate. What about non-membership? Suppose, somehow, the verifier knew the *size* $r_n = |R_n|$. Then,

a certificate to show that $t \notin R_n$ could just be of the form $[u_1, \text{path for } u_1], \cdots, [u_{r_n}, \text{path for } u_{r_n}]$ where the u_i's are in lexicographically increasing order and none of them is t.

But how do we certify that r_n is correct? And this is the key idea: if we knew the value of r_{n-1}, we can actually give a certificate for the value of r_n. This process is called *inductive counting* and is elaborated below.

Base Case: Note that $R_0 = \{s\}$ and hence $r_0 = 1$ is trivially certifiable.

Inductive Step: Assume that we have already provided a certificate for the value of r_{i-1}. We now wish to provide a certificate for the value of r_i. The certificate will be of the following type:

$$[r_i][u_1, b_1, \text{cert. for } \in R_i \text{ or } \notin R_i] \cdots [u_n, b_n, \text{cert. for } \in R_i \text{ or } \notin R_i]$$

where b_j is the indicator for whether $u_j \in R_i$, and the u_i's are in ascending order.

Certificate for $u_j \in R_i$: Provide a path of length at most i starting from s as the certificate.

Certificate for $u_j \notin R_i$: This certificate is a little more involved. The certificate takes the following form and will infact not even depend on u_j (the verifier's check will though):

$$[w_1, \text{cert. for } w_1 \in R_{i-1}] \cdots [w_{r_{i-1}}, \text{cert. for } w_{r_{i-1}} \in R_{i-1}]$$

where the w_k's are the elements of R_{i-1} in ascending order. Once again the certificate for membership is just a path of length $i-1$ or less.

The above is enough to convince the verifier that $u_j \notin R_i$ by just checking that u_j is not one step away from any of the r_{i-1} many w's in R_{i-1}.

Putting it all together, we can build a certificate for $t \notin R_n$ that is verifiable by a log-space verifier. Thus, $\overline{\text{Dir-Path}} \in \mathsf{NL}$. □

15.2 Circuits

A Boolean circuit is simply a visual representation demonstrating the process of generating an output based on an input through a combination of fundamental Boolean operations involving OR, AND, and NOT gates (see figure 15.1).

Formally, we define a boolean circuit as follows.

Definition 15.3 (Boolean Circuits)**.** *For every $n, m \in \mathbb{N}$ a Boolean circuit C with n inputs and m outputs is a directed acyclic graph. It contains n nodes with no incoming edges; called the input nodes and m nodes with no outgoing edges, called the output nodes. All other nodes are called gates and are labeled with one of $\vee$, $\wedge$ or $\neg$. The size of C denoted by $|C|$ is the number of nodes in it.*

The depth of a Boolean circuit refers to the length of the longest path from the input node to the output node. ◊

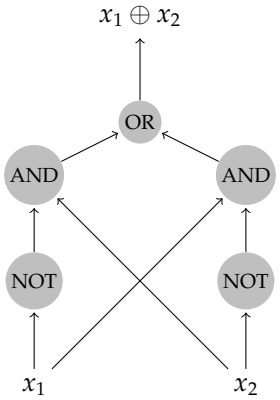


Figure 15.1: This circuit returns 1 iff $x_1 = x_2$

The boolean circuit in the above definition implements functions from $\{0,1\}^n$ to $\{0,1\}^m$. Now we will prove the following claim.

Claim 15.4. *Addition of two integers can be done with a constant depth, unbounded fanin circuit.*

Proof sketch. Let x and y be two n bit integers and $z = x + y$, where z_i, x_i, and y_i represent the i-th bit of z, x, and y, respectively. For each z_i, we will design a constant depth circuit. c_i denotes the carry generated by adding x_i and y_i. Therefore, for all $i \leq n$, we have $z_i = x_i \oplus y_i \oplus c_{i-1}$ ($c_0 = 0$) and $z_{n+1} = c_n$. Now we will build a constant depth circuit for each c_i. Note that $c_i = 1$ if and only if there exists a $j \leq i$ such that $x_j \wedge y_j = 1$ and for all $i \geq j' \geq j$, $x_{j'} \vee y_{j'} = 1$. Let $\tilde{j}$ be a Boolean variable corresponding to such j, i.e., $\tilde{j} = 1$ iff such j exists. We can build a depth 3 circuit (figure 15.2) for each $\tilde{j}$ such that $j \leq i$. Therefore, $c_i = \tilde{i} \vee \widetilde{(i-1)} \vee ... \vee \tilde{1}$ which can be realized by a constant depth circuit. Note that the size of the resulting circuit is polynomial in n.

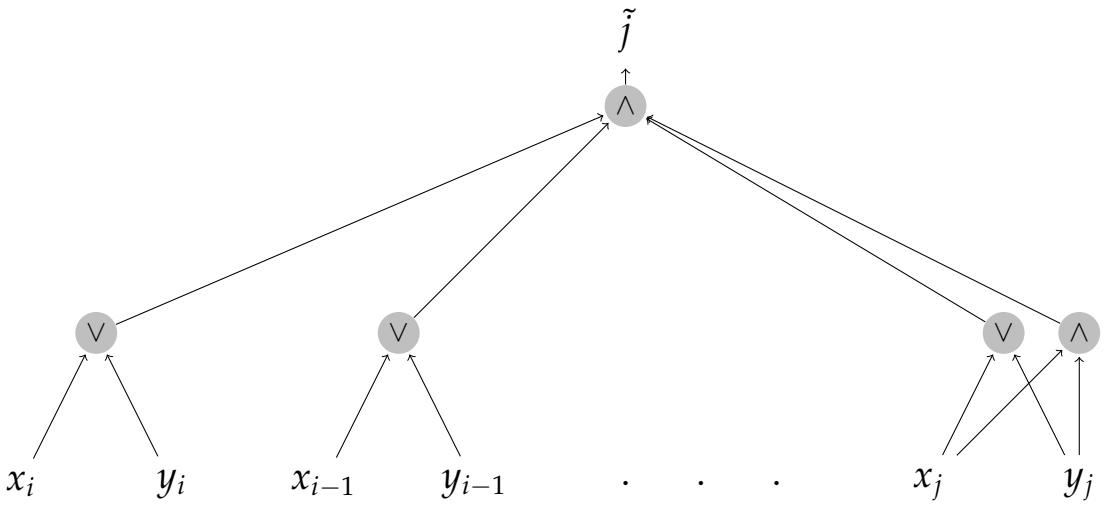


Figure 15.2: Depth 2 circuit for $\tilde{j}$

□

Now we define equivalent definition of circuits for deciding a language.

Definition 15.5 (Circuit Family and Language Recognition)**.** *Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-sized circuit family is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs and a single output, such that $|C_n| \leq T(n)$ for every n. We say that a language $L \in \mathsf{SIZE}(T(n))$ if there exists a $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every $x \in \{0,1\}^n$, $x \in L \iff C_{|x|}(x) = 1$.* ◊

Circuit families can be surprisingly powerful because there is no expectation on how long it would build the n-th circuit. In fact, here is a ridiculous language that is computable by very small circuit families.

$$L = \{1^i : M_i \text{ halts on a blank tape input}\}$$

The above language is of course undecideable. However, it is computable by a really simple circuit family.

$$C_i = \begin{cases} (x_1 \wedge ... \wedge x_i) & \text{if } 1^i \in L \\ \neg(x_1 \wedge ... \wedge x_i) & \text{otherwise} \end{cases}$$

Therefore, $L \in \text{size}(n)$.

15.2.1 Some common circuit classes

Now we define two models of computation in circuits.

Definition 15.6 (NC^i)**.** *A language is in NC^i if there exists a poly size circuit with two fanins and $\log^i n$ depth that decides it.* ◊

Definition 15.7 (AC^i)**.** *A language is in AC^i if there exists a poly size circuit with unbounded fanin and $\log^i n$ depth that decides it.* ◊

It is easy to see that $\mathsf{AC}^0 \subseteq \mathsf{NC}^1$. In the next lecture, we will show that the problem Parity $\in \mathsf{NC}^1$ but Parity $\notin \mathsf{AC}^0$.

Lecture 16

PARITY is not in AC^0

Scribe: Bikshan Chatterjee

Today's lecture will focus on the language Parity_n and the circuit class AC^0.

Parity_n: computes the parity of the n input bits.

It is a sensitive function: changing any input bit changes the output. Intuitively, it feels like a constant depth circuit made of AND / OR / NOT gates cannot be too sensitive.

Upper bounds for Parity_n Parity_n can be computed by circuits of $\log n$ depth, $O(n)$ size using divide and conquer (recursively compute parity of left half and right half, then XOR). The gates only require fan in 2. This shows $\text{Parity}_n \in \mathsf{NC}^1$ (circuits with fan in 2, $O(\log n)$ depth). It can be shown Parity_n can be computed by depth d circuits of size $O(2^{n^{1/(d-1)}})$.

What sort of lower bound can we prove for depth-d circuits computing Parity_n? Can it be computed by polynomial sized depth-d circuits? There was a long line of results that showed that Parity_n requires AC^0 circuits of very large size to compute them starting with the result of Furst-Saxe-Sipser, culminating in work of Håstad. In this lecture, we will see a proof due to Razborov and Smolensky.

Theorem 16.1 (Razborov Smolensky)**.** Parity_n *requires depth d cicruits of size* $2^{\Omega(n^{1/2d})}$.

The result relies on a notion of 'approximating a function by a polynomial' which shortly. But roughly speaking, Theorem 16.1 is proved via the following two lemmas (which is perhaps vague at the moment).

Lemma. *Any* AC^0 *circuit of depth d and size s can be "approximated" by a polynomial of degree* $O(\log s)^d$.

Lemma. *Any polynomial that "approximates"* Parity_n *must have degree* $\Omega(\sqrt{n})$.

The above two lemmas would immediately yield that $(\log s)^d = \Omega(\sqrt{n})$ which implies that $s = 2^{\Omega(n^{1/2d})}$.

We now proceed towards defining the notion of approximation.

16.1 Approximating a function by a polynomial

We will restrict ourselves to the field $\mathbb{F}_3$ consisting of three elements $\{0,1,2\}$ (or $\{0,1,-1\}$). An element $\vec{a} \in \{0,1\}^n$ can be thought of as an element of $\mathbb{F}_3^n$ as well.

Definition 16.2 (Computation by polynomials)**.** *Let $F : \{0,1\}^n \to \{0,1\}$ be a Boolean function. We will say that a polynomial $g(x_1,\ldots,x_n) \in \mathbb{F}_3[x_1,\ldots,x_n]$ computes F correctly at input $\vec{a} \in \{0,1\}^n$ if $F(\vec{a}) = g(\vec{a})$ (where we are identifying $0,1$ with the $0,1$ inside $\mathbb{F}_3$).* $\Diamond$

For example, the following are polynomials for the OR and AND functions that compute it correctly everywhere.

$$g_\wedge(\vec{x}) = x_1 x_2 \ldots x_n$$
$$g_\vee(\vec{x}) = 1 - (1-x_1)(1-x_2)\ldots(1-x_n)$$

However, both the above have degree n and we would like to ask if we can at least approximate the function using a far lower degree polynomial.

16.1.1 Weak approximations

Definition 16.3 (Weak-approximation)**.** *We will say that a polynomial $g(\vec{x}) \in \mathbb{F}_3[\vec{x}]$ ε-weakly approximates a Boolean function $F : \{0,1\}^n \to \{0,1\}$ if*

$$\Pr_{\vec{a}\in\{0,1\}^n}[g(\vec{a}) \neq F(\vec{a})] \leq \varepsilon.$$

In words, g computes F correctly on all but an ε-fraction of the inputs. $\Diamond$

Under this definition, $\wedge$ and $\vee$ of fan in n can be $1/2^n$ approximated with 0 degree polynomials

$$g_\wedge(\vec{x}) = 0 \quad \text{and } g_\vee(\vec{x}) = 1$$

where $\vec{x} = (x_1,\ldots,x_n)$.

However, if we have a circuit consisting of many AND/OR gates, it is unclear how we can 'compose' these approximating polynomials in a meaningful way. It is because of these reasons that a stronger notion of approximation is studied that is more amenable to composition.

16.1.2 Strong-approximation

Definition 16.4 (Strong-approximation by polynomials)**.** *For $\vec{x} = (x_1,\ldots,x_n) \in \mathbb{F}_3^n$ and $\vec{r} = (r_1,\ldots,r_m) \in \mathbb{F}_3^m$, a polynomial $g(\vec{x},\vec{r}) \in \mathbb{F}_3[\vec{x},\vec{r}]$ ε-approximates $F : \{0,1\}^n \to \{0,1\}$ if*

$$\forall \vec{a} \in \{0,1\}^n \;:\; \Pr_{\vec{r}\in\mathbb{F}_3^m}[g(\vec{a},\vec{r}) \neq F(\vec{a})] \geq \varepsilon.$$

That is, each $g(\vec{x},\vec{r}) \in \mathbb{F}_3[\vec{x},\vec{r}]$ gives a set of polynomials (indexed by different values of $\vec{r}$). The above notion gives a bound on the error probability over random choice of $\vec{r}$ for all inputs *instead of*

error for a randomly chosen input. ◊

16.1.3 Strong-approximation for OR

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

$$g(\vec{x}, \vec{r}) = x_1 r_1 + x_2 r_2 + \ldots + x_n r_n$$

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

If some $x_i = 1$, $\sum x_j r_j$ is a uniformly random element in $\mathbb{F}_3$.
r_i needs to be equal to $1 -$ (whatever the rest of the sum is), and r_i is independent of the rest of $\vec{r}$. So it happens with probability $1/3$.

Currently the error probability is $2/3$ (worst case; when some $x_i = 1$). This can be improved by using the polynomial

$$g'(\vec{x}, \vec{r}) = (x_1 r_1 + x_2 r_2 + \ldots + x_n r_n)^2$$

We still have for all $x_i = 0$, $g(\vec{x}, \vec{r}) = 0$ irrespective of $\vec{r}$. When some $x_i = 1$, $g'(\vec{x}, \vec{r})$ is correct whenever $g(\vec{x}, \vec{r})$ is 1 or -1. So correct with probability $2/3$, error probability $1/3$.

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

$$g_\vee(\vec{x}, \vec{r_1}, \vec{r_2}, \ldots, \vec{r_t}) = 1 - \prod_{i=1}^{t} (1 - g'(\vec{x}, \vec{r_i}))$$

(each $\vec{r_i}$ is from $\mathbb{F}_3^n$)

This is still 0 when all x_i are 0. And when some $x_i = 1$, this is 1 whenever some sample $\vec{r_i}$ makes $g'(\vec{x}, \vec{r_i}) = 1$ (which happens with probability $2/3$).
This polynomial is wrong (for some $x_i = 1$ case) only when $g'(\vec{x}, \vec{r_i}) = 0$ for all $\vec{r_i}$. The error probability is $(1/3)^t$.

Lemma 16.5 (Strong-approximation for OR)**.** *The OR function on n bits can be ε-approximated using a polynomial of degree $O(\log(1/\varepsilon))$.*

Proof. Follows by setting $(1/3)^t < \varepsilon$, and the degree of the polynomial (in $\vec{x}$) is $2t$. □

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

16.1.4 Strong-approximation for AND

$\vec{x} = (x_1, \ldots, x_n)$. Using De Morgan's law,

$$\text{AND}(x_1, \ldots, x_n) = \neg \text{OR}(\neg x_1, \ldots, \neg x_n)$$

AND can be $(1/3)^t$ approximated by the degree $2t$ polynomial

$$g_\wedge(\vec{x},\vec{r}) = 1 - g_\vee((1-x_1, 1-x_2, \ldots, 1-x_n), \vec{r})$$

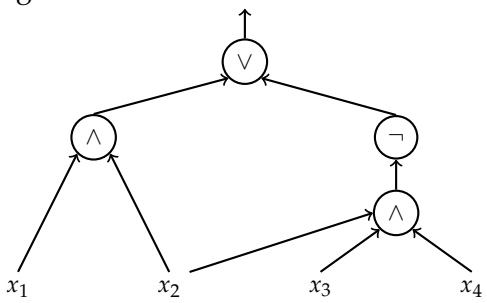
(here $\vec{r}$ is a tn element vector made of $(\vec{r_1}, \ldots, \vec{r_t})$)

16.1.5 Handling constant-depth circuits

Now a depth d circuit C can be approximated by composing the AND, OR, NOT polynomials. Let the size of the circuit be s (number of gates).

To get an ε-approximating polynomial for the entire circuit, we begin with a δ-approximating polynomial for each AND, OR and NOT gates and compose them in the obvious way. We know that the degree of the δ-approximating polynomials is $O(\log(1/\delta))$ each and thus composing them in depth d will make the degree no more than $O(\log(1/\delta))^d$.

Eg. $C =$



is approximated by

$$\begin{aligned} & f_C((x_1,x_2,x_3,x_4),(r_1,r_2,r_3)) \\ = \; & g_{\vee,2}\left(g_{\wedge,2}((x_1,x_2),r_1), 1 - g_{\wedge,2}((x_2,x_3,x_4),r_2), r_3\right) \end{aligned}$$

The only thing to check is the error probability of this composed approximating polynomial. This can be bounded by the probability of event that any of the s gates is wrong. Let us fix any input $\vec{a}$

$$\begin{aligned} & \Pr_r[\text{polynomial computes wrong value at some gate}] \\ \leq \; & \sum_{i=1}^{s} \Pr_r[\text{polynomial computes wrong value at } i] = s \cdot \delta. \end{aligned}$$

Thus, if we eventually want to get an ε-approximating polynomial, we need to set $\delta = \varepsilon/s$ and hence the degree of the approximating polynomial is therefore $O(\log(s/\varepsilon))^d$.

We summarise this as the following lemma.

Lemma 16.6 (Strong-approximation for AC^0)**.** *Any Boolean function computable by a size s depth d circuit can be $(1/4)$-approximated using a polynomial of degree at most $O(\log s)^d$*

16.2 On strong-approximations for Parity_n

The following lemma will allow us to complete the proof of Theorem 16.1.

Lemma 16.7 (Parity requires high degree to approximate)**.** *If $g(x_1, \ldots, x_n) \in \mathbb{F}_3[x_1, \ldots, x_n]$ is a $(1/4)$-weak approximation for Parity_n, i.e.*

$$\Pr_{\vec{a} \in \{0,1\}^n}[g(\vec{a} \neq \text{Parity}_n(\vec{a}))] \leq \frac{1}{4},$$

then $\deg(g) \geq \frac{\sqrt{n}}{4}$.

(The above lemma was why we chose $\mathbb{F}_3$ and not $\mathbb{F}_2$ since the Parity_n can be trivially computed over $\mathbb{F}_2$ via the polynomial $x_1 + \cdots + x_n$.)

The above lemma is *stronger* than stating that any strong-approximation for Parity_n requires degree $\Omega(\sqrt{n})$ — if $g(\vec{x}, \vec{r}) \in \mathbb{F}_3[\vec{x}, \vec{r}]$ is an $(1/4)$-strong approximation for Parity_n, then there is some setting for $\vec{r} = \vec{\alpha}$ such that $g'(\vec{x}) = g(\vec{x}, \vec{\alpha})$ is a $(1/4)$-weak approximation for Parity_n.

16.2.1 Proof of Lemma 16.7

Let us assume that $f(\vec{x})$ is a polynomial that $(1/4)$-weakly approximates Parity_n.

Switching from $\{0,1\}$ to $\{1,-1\}$: It would be convenient to switch perspective to the $\{1,-1\}$ basis since parity in this world (let's call it $P(\vec{x}) : \{-1,1\}^n \to \{-1,1\}$) is simply the monomial $x_1 \cdots x_n$.

If $f(\vec{x})$ is an approximating polynomial for Parity_n, then the polynomial

$$g(\vec{x}) = 1 + f(x_1 + 1, \ldots, x_n + 1)$$

is an approximating polynomial for $P(\vec{x})$ in the $\{-1,1\}^n$ world. This doesn't change the degree of the polynomial. [This is just using the fact that the map $x \mapsto x + 1$ maps 0 to 1 and 1 to $2 = -1 \bmod 3$.]

Let $G = \left\{\vec{a} \in \{-1,1\}^n \;:\; g(\vec{a}) = a_1 \cdots a_n\right\}$. By the hypothesis, we have that $|G| \geq (3/4) \cdot 2^n$. Let $\deg(g) = d$. Our goal is to prove that $d = \Omega(\sqrt{n})$.

Consider the space of all functions $\mathcal{F} = \{H : G \to \mathbb{F}_3\}$. Clearly, $|\mathcal{G}| = 3^{|G|}$. We will compute the site of the set differently in terms of d by showing that all of these functions have a structured polynomial $h(\vec{x})$ that computes it exactly on all of G and count the number of such structured polynomials.

Any function from $Q : \mathbb{F}_3^n \to \mathbb{F}_3$ can be computed by some polynomial $q(\vec{x}) \in \mathbb{F}_3[\vec{x}]$ using

interpolation as follows:

$$q(\vec{x}) = \sum_{\vec{a} \in \mathbb{F}_3^n} Q(\vec{a}) \cdot \prod_i \prod_{\alpha \neq a_i} \frac{x_i - \alpha}{a_i - \alpha}$$

and thus any function $H \in \mathcal{G}$ has such a polynomial $h(\vec{x})$. Furthermore, since we only care about evaluating on a subset G of $\{-1, 1\}^n$, we can make use of the fact that $x^2 = 1$ for $x = \pm 1$ and thus assume without loss of generality that $h(\vec{x})$ is in fact a *multilinear* polynomial that agrees with H on all of $\mathcal{G}$.

We will now further simplify the polynomial $h(\vec{x})$ by making use of the fact that, over G, we have $x_1 \cdots x_n = g(\vec{x})$, the purported approximating polynomial for $P(\vec{x})$. Using this, any monomial of $h(\vec{x})$ with degree $> n/2$ can be replaced as follows, without changing its evaluation on any point of G:

$$\prod_{i \in S} x_i = (\prod_{i \in \overline{S}} x_i) \cdot x_1 x_2 \ldots x_n = (\prod_{i \in \overline{S}} x_i) \cdot g(\vec{x}),$$

again using the fact that $x_i^2 = 1$ in G and $x_1 \cdots x_n = g(\vec{x})$ in G. The above process may once again introduce some x_i^2 terms that we can eliminate again. Also note that the degree of the RHS above is at most $(n/2) + d$ since $|S| \geq (n/2)$.

Therefore, we now have that any $H : G \to \mathbb{F}_3$ can be expressed as a *multilinear* polynomial $h(\vec{x}) \in \mathbb{F}_3[\vec{x}]$ of degree at most $(n/2) + d$. Hence, the size of $\mathcal{G}$ is at most the number of such structured polynomials.

Suppose the number of monomials possible monomials in such structured polynomials is at most M, then the number of such polynomials is at most 3^M (by choosing coefficients for each such monomial). Hence

$$\begin{aligned} \mathcal{G} = 3^{|G|} = 3^{0.75 \cdot 2^n} &\leq 3^M \\ \implies M &\geq 0.75 \cdot 2^n. \end{aligned}$$

On the other hand, we can bound M via

$$\begin{aligned} M &= \binom{n}{0} + \cdots + \binom{n}{\frac{n}{2} + d} \\ &= \left(\binom{n}{0} + \cdots + \binom{n}{\frac{n}{2}} \right) + \left(\binom{n}{\frac{n}{2} + 1} + \cdots + \binom{n}{\frac{n}{2} + d} \right) \\ &= 2^{n-1} + \left(\binom{n}{\frac{n}{2} + 1} + \cdots + \binom{n}{\frac{n}{2} + d} \right) \\ &\leq 2^{n-1} + d \cdot \binom{n}{\frac{n}{2}} \\ &\approx 2^{n-1} + d \cdot \frac{2^n}{\sqrt{(\pi/2) \cdot n}} \leq 2^n \cdot \left(\frac{1}{2} + \frac{d}{\sqrt{n}} \right) \end{aligned}$$

Thus, the only way $M \geq 0.75 \cdot 2^n$ is when $\frac{d}{\sqrt{n}} \geq 0.25$, thus forcing $d \geq \frac{\sqrt{n}}{4}$.

Lecture 17

Circuit Families

Scribe: Siddharth Choudary

Definition 17.1. *Circuit Families are sets of circuits* $\{C_n : n \in \mathbb{N}\}$ *where each* C_n *is a circuit on* n *inputs.* $\Diamond$

The power of these families comes from the fact that the complexity to construct each of these circuits is not taken into account. These are called "Non-Uniform Models".

Definition 17.2. $\{C_n : n \in \mathbb{N}\}$ *is a* $\mathcal{D}$*-Uniform circuit family for a language* L *if constructing* C_n *from input* 1^n *is in complexity class* $\mathcal{D}$*.*

$\Diamond$

Theorem 17.3 (Karp-Lipton)**.** *If* $\mathsf{NP} \subseteq \mathsf{SIZE}(\text{poly})$*, then* $\mathsf{PH} = \Sigma_2 \cap \Pi_2$*.*

Proof. Suppose $L \in \Pi_2$. i.e. There is a Φ s.t.
$x \in L \iff \forall y \exists z \Phi_x(y, z)$ $(\star)$
By assumption, SAT is computable by a family of poly-size circuits. i.e.
$\exists C_m \forall y$: CheckSAT&Verify$(\Phi_x(y, _), C_m)$ $(\dagger)$

Algorithm 6 CheckSAT&Verify(Φ, C)

```
if C(Φ(_)) =False then
    return False
else            (i.e. C says that Φ ∈ SAT)
    Use C to get satisfying assignment 'a' for Φ
    if any error then
        return False
    else if Φ(a) =True then
        return True
    else
        return False
    end if
end if
```

Therefore, if $(\star)$ is true, then $(\dagger)$ is true by hypothesis, and if $(\star)$ is false, then CheckSAT&Verify never returns True, thus $(\dagger)$ is also false.

Hence, $L \in \Sigma_2$.
Therefore $\Pi_2 \subseteq \Sigma_2$ and thus PH collapses to $\Pi_2 \cap \Sigma_2$. □

17.1 Hierarchies

Let us look at some properties:

1. No. of functions from $\{0,1\}^n$ to $\{0,1\}$ is 2^{2^n}.
2. No. of these functions that are computable by size s circuits is $2^{O(s \log s)}$ (using any linear space representation of a circuit and assigning gates and inputs for each node.)
3. Any function is computable by a size $n2^n$ circuit. (by using CNF or DNF form)

From these properties, we can see that there are functions which cannot be computed by size $O(2^n/n)$ circuits.

Similarly, we have that $\mathsf{SIZE}(s) \subsetneq \mathsf{SIZE}(s * 100n^2)$.

17.2 Turing Machines with advice

Definition 17.4 (Advice)**.** *An* advice *is a set* $\{z_i \in \{0,1\}^* : i \in \mathbb{N}\}$ *and its associated length function* $\ell : \mathbb{N} \to \mathbb{N}$ *defined by* $\ell(i) = |z_i|$*. A language* $L \in \mathcal{C}/\ell$ *iff there is a Turing Machine* M *in complexity class* $\mathcal{C}$ *and advice* $\{z_i : i \in \mathbb{N}\}$ *of length* $\ell(i)$ *s.t.* $x \in L \iff M(x, z_{|x|}) = 1$. ◊

In other words, for every i, there is a string z of length $l(i)$ s.t. M with advice z correctly decides membership of any length i input x.

$\mathsf{P}/\ell = \{L$ computed by det poly time TM with ℓ-length advice$\}$.

17.2.1 Class Containments

- $\mathsf{P}/\text{poly} \subseteq \mathsf{SIZE}(\text{poly})$ by Cook-Levin and freezing z_i per length.
- $\mathsf{P}/\text{poly} \supseteq \mathsf{SIZE}(\text{poly})$ by taking circuit description as advice.
- $\mathsf{coNEXP} \subseteq \mathsf{NEXP}/\text{poly}$ by taking z_n as number of strings of length n that are not in L and taking non-det guess as the sequence of pairs (x, y) for every $x \in \overline{L}$ of length n.

Lecture 18

Randomised computation

Scribe: Sourav Roy

In this module of the course, we will be dealing with randomised algorithms. Here are a few examples of interesting randomised algorithms:

1. The Miller-Rabin test for primality: a one-sided randomised algorithm to check if a given n-bit number is prime.

2. 2-approximation for Max-Cut: Given a graph G, the task is to compute a "cut" that is at least 50% as large as the best possible "cut". This problem admits a really simple randomised algorithm:

   > Pick a set S at random by adding every vertex of G to the set S with probability $1/2$ each.

   This simple randomised algorithm actually has an expected cut size of $|E|/2$, which is certainly at least 50% of the largest possible cut size.

3. Bipartite matching: Given a graph G, check if there is a bipartite matching in the graph.

   There is a simple randomised algorithm obtained by building a modified bipartite adjacency matrix $A = (a_{ij})_{i,j}$ defined as follows:

   $$a_{i,j} = \begin{cases} 0 & \text{if } u_i \text{ is not connected to } v_j \\ \text{randomly chosen from } \{0, \ldots, n^2\} & \text{if } u_i \text{ is connected to } v_j \end{cases}$$

   The algorithm returns "Yes" if $\det(A) \neq 0$, and zero otherwise.

4. Computing square-roots modulo p: Given an integer a and a prime p (both given in binary), compute a "square root" of a modulo p (if one exists). That is, output an integer b such that $b^2 = a \mod p$, if one exists.

   Surprisingly, checking if a square-root exists turns out to be easy.

   Fact 18.1. *a has a square-root modulo p if and only if $a^{(p-1)/2} = 1 \mod p$.*

Proof sketch. Follows from the fact that $\mathbb{F}_p^*$ is cyclic (with say g as a generator), and the only a's that have a square root are those of the form g^{2i}. Thus, $a^{(p-1)/2} = g^{2i\cdot(p-1)/2} = \left(g^i\right)^{p-1} = 1 \bmod p$. And, the equation $x^{(p-1)/2} - 1 = 0$ has at most $(p-1)/2$ solutions and so this equation exactly characterises the elements that have a square-root. □

Turns out, this is a problem for which we know a randomised algorithm, but we have no known deterministic algorithm that is efficient!

18.1 Modelling randomised Computation in TMs

We will model randomised computation via the same syntactics used for non-deterministic and co-non-deterministic machines. The machine has two transitions functions, δ_0 and δ_1, and we will pretend the machine tosses a fair coin to choose between the two transitions at each step. Thus, each computational path is associated with a certain probability mass. We'll refer to these as randomised Turing Machines (although syntactically they are the same as non-deterministic or co-nondeterministic machines, but have different semantics for acceptance / rejection).

The acceptance criterion for various complexity classes will be based on this probability. We will use $M(x, r)$ to denote the final accept/reject status when M runs on input x on path defined by r.

18.1.1 Randomised complexity classes

Definition 18.2 (BPP, or bounded-error probability polynomial time)**.** *For two constants $0 \leq s < c \leq 1$, define the class $\mathsf{BPP}_{s,c}$ as the set of language L such that there is some randomised polynomial time Turing machine M such that for all $x \in \Sigma^*$*

$$x \in L \implies \Pr_r[M(x,r) = \mathit{accept}] \geq c,$$
$$x \notin L \implies \Pr_r[M(x,r) = \mathit{accept}] \leq s,$$

(Typically, $c = \frac{2}{3}$ and $s = \frac{1}{3}$.) ◊

Definition 18.3 (RP and coRP)**.** *For a constant $0 < p < 1$, define the class RP_p as the set of language L such that there is some randomised polynomial time Turing machine M such that for all $x \in \Sigma^*$*

$$x \in L \implies \Pr_r[M(x,r) = \mathit{accept}] \geq (1-p),$$
$$x \notin L \implies \Pr_r[M(x,r) = \mathit{accept}] = 0.$$

Similarly, coRP_p is the set of languages L such that

$$x \in L \implies \Pr_r[M(x,r) = \mathit{accept}] = 1,$$
$$x \notin L \implies \Pr_r[M(x,r) = \mathit{accept}] \leq p.$$

(Typically, $p = \frac{1}{2}$.) $\Diamond$

In words, RP_p and coRP_p are *one-sided error* randomised algorithms, with the error probability being p. On the other hand, $\mathsf{BPP}_{s,c}$ refer to two-sided error randomised algorithms where s is the *false-positive* probability and $(1-c)$ is the *false-negative* probability.

Observation 18.4. $\mathsf{RP}, \mathsf{coRP} \subseteq \mathsf{BPP}$, *and* $\mathsf{RP} \subseteq \mathsf{NP}$ *and* $\mathsf{coRP} \subseteq \mathsf{coNP}$.

It is widely believed that $\mathsf{BPP} = \mathsf{P}$ although we are far from proving it. However, we will show that BPP is within the second level of the polynomial hierarchy in the next class.

18.2 Success amplification

18.2.1 For one-sided-error randomised algorithms

Let us compare $\mathsf{RP}_{0.5}$ with RP_δ, for $\delta < 0.5$. Obviously, $\mathsf{RP}_\delta \subseteq \mathsf{RP}_{0.5}$ as any algorithm that has an error of less than δ certainly has error less than 0.5.

Turns out, we can easily convert a one-sided randomised algorithm A with error 0.5 to another algorithm A' with error δ. We define the new algorithm A' as follows

> Compute $b_i = M(x, r_i)$ for $i = 1, \ldots, t$, where each r_i is a fresh random string.
> Accept x if any of the b_i's are accept.

One can easily see that

$$x \notin L \implies \Pr_{r'}[A'(x, r') = \text{accept}] = 0,$$
$$x \notin L \implies \Pr_{r'}[A'(x, r') \neq \text{accept}] = (\Pr_r[A(x, r) = \text{reject}])^t \leq (0.5)^t.$$

Thus, setting $t = \log_2(1/\delta)$ gives us the required randomised algorithm with error at most δ.

Note that the running time of the algorithm increases by a factor of t. As long as $t = \text{poly}(n)$, we are still within the range of polynomial time algorithms. Thus, we can tolerate $\delta = 1/2^{\text{poly}(n)}$.

Similarly, we could have also replaced 0.5 with $1 - 1/\text{poly}(n)$. Thus, we can summarise this in the following lemma.

Lemma 18.5 (Success amplification for RP)**.**

$$\mathsf{RP}_{1-\frac{1}{\text{poly}(n)}} = \mathsf{RP}_{0.5} = \mathsf{RP}_{\frac{1}{2^{\text{poly}(n)}}}$$

In words, even if we had a randomised algorithm with an inverse-polynomially small success probability, we can convert it to another randomised algorithm (with only a polynomial slow-down in time) where the error probability is inverse-exponentially small.

Thus, we will simply refer to the class as RP and ignore the constant.

18.2.2 For two-sided-error randomised algorithms.

How does the class $\mathsf{BPP}_{\frac{1}{3},\frac{2}{3}}$ compare with $\mathsf{BPP}_{0.1,0.9}$? Once again, one direction is trivial. In fact, if $c' > c$ and $s' < s$ we always have $\mathsf{BPP}_{s',c'} \subseteq \mathsf{BPP}_{s,c}$ just from the definitions.

With some modest conditions on c', s', we can prove an equivalence between them. Before that, we need the following well-known bound called the Chernoff-Hoeffding inequality.

Theorem 18.6 (Chernoff-Hoeffding inequality)**.** *Suppose $X_1, \ldots, X_n$ are n independent, identically distributed random variables with $X_i \in \{0,1\}$ and $\mathbb{E}[X_i] = \mu$. Let $X = \frac{X_1+\cdots+X_n}{n}$. Then, for any $0 < \varepsilon < 1$,*

$$\Pr[|X - \mu| > \varepsilon \cdot \mu] \leq 2e^{-\varepsilon^2 n\mu/3}.$$

Using this, we can convert a $\mathsf{BPP}_{\frac{1}{2}-\varepsilon,\frac{1}{2}+\varepsilon}$ algorithm A machine in to a $\mathsf{BPP}_{\delta,(1-\delta)}$ algorithm A' as follows:

> Set $t = O\left(\frac{1}{\varepsilon^2}\log\left(\frac{1}{\delta}\right)\right)$. Let $b_i = A(x, r_i)$ for $i = 1, \ldots, t$ where r_i is an independently chosen random string.
>
> If $\sum b_i > (t/2)$, accept. Otherwise, reject

It is easy to see that if A was a $\mathsf{BPP}_{\frac{1}{2}-\varepsilon,\frac{1}{2}+\varepsilon}$, then A' is a $\mathsf{BPP}_{\delta,1-\delta}$.

Once again, since we can tolerate $t = \mathrm{poly}(n)$, this means that ε can be as small as $\frac{1}{\mathrm{poly}(n)}$ and δ can be as small as $\frac{1}{2^{\mathrm{poly}(n)}}$. Thus, we can summarise this the following lemma.

Lemma 18.7 (Success amplification for BPP)**.**

$$\mathsf{BPP}_{\frac{1}{2}-\frac{1}{\mathrm{poly}(n)},\frac{1}{2}+\frac{1}{\mathrm{poly}(n)}} = \mathsf{BPP}_{\frac{1}{3},\frac{2}{3}} = \mathsf{BPP}_{\frac{1}{2^{\mathrm{poly}(n)}},1-\frac{1}{2^{\mathrm{poly}(n)}}}.$$

In other words, we can amplify just an inverse-polynomial gap between the yes and no instances to exponentially small error.

In fact, it is not even important that the completeness and soundness probabilities are symmetric around 0.5. The following is left as an execise:

> **Exercise:** Show that $\mathsf{BPP}_{0.6,0.7} = \mathsf{BPP}$.

Lecture 19

Relationship between BPP and other complexity classes

Scribe: Yeshwant Pandit

In this section, we'll explore the connections between BPP and other complexity classes. We will first show that $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

Theorem 19.1 (Adleman)**.** $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

Proof. To prove the result, we consider a language $L \in \mathsf{BPP}$ and then use the advice definition of $\mathsf{P/poly}$ to show that $L \in \mathsf{P/poly}$. By the definition of BPP and its error reduction procedure, we know $L \in \mathsf{BPP}$ if there exists a polynomial-time Turing machine M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$,

$$x \in L \implies \Pr_{r \in \{0,1\}^{poly(|x|)}}[M(x,r) = 1] \geq 1 - \delta$$

$$x \notin L \implies \Pr_{r \in \{0,1\}^{poly(|x|)}}[M(x,r) = 1] \leq \delta$$

where $\delta = \frac{1}{2^{poly(|x|)}}$. To show $L \in \mathsf{P/poly}$, we show the existence of an advice string $r \in \{0,1\}^{poly(n)}$ that works for all $x \in \{0,1\}^n$. That means $x \in L \implies M(x,r) = 1$ and $x \notin L \implies M(x,r) = 0$.

For simplicity, let $m = poly(n)$. Let A denote $2^n \times 2^m$ matrix whose rows correspond to all input strings of length n and columns correspond to all random bits of length m. For an input x_i (i.e., row x_i) and a random bit string r_j (i.e., column r_j), $A[x_i, r_j] = \checkmark$ if $M(x_i, r_j) = L(x_i)$, where $L(x_i) = 1$ if $x_i \in L$ and $L(x_i) = 0$ otherwise. By the definition of BPP, every row has at least $(1 - \delta) \cdot 2^m$ $\checkmark$ marks. Thus, we have at least $2^n \cdot (1 - \delta) \cdot 2^m$ $\checkmark$ marks in total.

Our goal is to show that there exists a column r_j with all $\checkmark$ marks. To the contrary assume that every column has at least one $\times$ mark. Thus, the total number of $\times$ marks is at least 2^m and the total number of $\checkmark$ marks is at most $2^n \cdot 2^m - 2^m$. To obtain a contradiction, the following

should hold.

$$\text{Total number of } \checkmark \text{ marks by assumption} < \text{Total number of } \checkmark \text{ marks by definition}$$
$$2^n \cdot 2^m - 2^m < 2^n \cdot (1-\delta) \cdot 2^m$$
$$2^n - 1 < 2^n \cdot (1-\delta)$$
$$\therefore \quad \delta < \frac{1}{2^n}$$

Since $\delta = \frac{1}{2^{poly(n)}}$, we can set $\delta = \frac{1}{2^{n+1}}$ and get a contradiction. Hence, we obtain an advice r that works for all inputs $x \in \{0,1\}^n$. □

The next result establishes a connection between BPP and the polynomial hierarchy.

Theorem 19.2 (Gács-Sipser)**.** $\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{P}} \cap \Pi_2^{\mathsf{P}}$

Proof. Since BPP is closed under complementation, it is enough to prove that $\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{P}}$.

Suppose $L \in \mathsf{BPP}$. Then by the definition of BPP and its error reduction procedure, we know $L \in \mathsf{BPP}$ if there exists a polynomial-time Turing machine M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$,

$$x \in L \implies \Pr_{r \in \{0,1\}^m}[M(x,r) = 1] \geq 1 - \delta$$
$$x \notin L \implies \Pr_{r \in \{0,1\}^m}[M(x,r) = 1] \leq \delta$$

where $\delta = \frac{1}{2^{q(|x|)}}$, $m = p(|x|)$, and p and q are polynomials. To prove $L \in \Sigma_2^{\mathsf{P}}$, we need to show $x \in L \iff \exists u \in \{0,1\}^{q(|x|)} \ \forall v \in \{0,1\}^{q(|x|)} \ \Psi(x,u,v)$, where q is a polynomial.

For $x \in \{0,1\}^n$, let $\text{Accept}(x) = \{r \in \{0,1\}^m | M(x,r) = 1\}$, i.e., the set of random strings r for which the machine M accepts the input pair (x,r). Then we know $|\text{Accept}(x)| \geq (1-\delta) \cdot 2^m$ if $x \in L$ and $|\text{Accept}(x)| \leq \delta \cdot 2^m$ otherwise.

For a set $S \subseteq \{0,1\}^m$ and string $u \in \{0,1\}^m$, let $S \oplus u$ denote the shift of the set S by u: $S \oplus u = \{x \oplus u | x \in S\}$ where $\oplus$ denotes the bitwise XOR operator.

Now we use the following idea:

- If $x \in L$, i.e., $|\text{Accept}(x)| \geq (1-\delta) \cdot 2^m$ then using few translates $u_1, \ldots u_k$ we can cover the set $\{0,1\}^m$ entirely, i.e., $\bigcup_{i=1}^k (\text{Accept}(x) \oplus u_i) = \{0,1\}^m$. This can be further interpreted as $\exists u_1, \ldots u_k \ \forall r \in \{0,1\}^m \ \exists i \in [k] \ r \in \text{Accept}(x) \oplus u_i$ which is equivalent to $\exists u_1, \ldots u_k \ \forall r \in \{0,1\}^m \ \bigvee_{i=1}^k (M(x, r \oplus u_i) = 1)$.
- If $x \notin L$, i.e., $|\text{Accept}(x)| \leq \delta \cdot 2^m$ then even after using all translates we cannot cover the set $\{0,1\}^m$ entirely, i.e., $\bigcup_{i=1}^k (\text{Accept}(x) \oplus u_i) \neq \{0,1\}^m$. This is equivalent to $\forall u_1, \ldots u_k \ \exists r \in \{0,1\}^m \ \bigwedge_{i=1}^k (M(x, r \oplus u_i) = 0)$.

Claim 19.3. *If δ, k satisfy $\delta \cdot k < 1$ then $x \notin L$ implies $\forall u_1, \ldots u_k \ \bigcup_{i=1}^k (\text{Accept}(x) \oplus u_i) \neq \{0,1\}^m$*

Proof. By the union bound we have $|\bigcup_{i=1}^k (\text{Accept}(x) \oplus u_i)| \leq k \cdot |\text{Accept}(x)| \leq k \cdot \delta \cdot 2^m < 2^m$ □

Claim 19.4. *If δ, k satisfy $2^m \cdot \delta^k < 1$ then $x \in L$ implies $\exists u_1, \ldots, u_k$, $\bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i) = \{0,1\}^m$.*

Proof. We prove this claim using probabilistic method. We show that if $u_1, \ldots, u_k$ are chosen independently at random then $\Pr[\bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i) = \{0,1\}^m] > 0$.

$$\begin{aligned}
\Pr[\bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i) = \{0,1\}^m] &= 1 - \Pr[\exists r \in \{0,1\}^m, r \notin \bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i)] \\
&> 1 - \sum_{r \in \{0,1\}^m} \Pr[r \notin \bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i)] \text{ [By union bound]} \\
&= 1 - \sum_{r \in \{0,1\}^m} Pr[\forall i \in [k], r \notin (\text{Accept}(x) \oplus u_i)]
\end{aligned}$$

Since $u_1, \ldots, u_k$ we chosen independently at random we get,

$$\begin{aligned}
Pr[\bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i) = \{0,1\}^m] &> 1 - \sum_{r \in \{0,1\}^m} (Pr[r \notin (\text{Accept}(x) \oplus u)])^k \\
&> 1 - \sum_{r \in \{0,1\}^m} \delta^k \\
&= 1 - 2^m \cdot \delta^k
\end{aligned}$$

Thus, if $2^m \cdot \delta^k < 1$ then we get that $Pr[\bigcup_{i=1}^{k}(\text{Accept}(x) \oplus u_i) = \{0,1\}^m] > 0$. □

Claims 19.3 and 19.4 require k to be such that $\frac{m}{\log \frac{1}{\delta}} < k < \frac{1}{\delta}$ holds. Observe that $k = m$ and $\delta = \frac{1}{2^n}$ would satisfy this condition. Thus, these claims allow us to infer that $L \in \Sigma_2^{\mathsf{P}}$. Hence, $\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{P}}$. □