

[CSS.203.1]: Computational Complexity (2026-I)

Summary scribes

Lectures

1	Starting with circuits	7
1.1	Introduction	7
1.1.1	An example to get comfortable with circuits	7
1.2	Counting functions, and circuits	8
1.3	Dealing with varying lengths	8
2	Size Hierarchy and Universal Circuits	10
2.1	Recap	10
2.2	Size Hierarchy	11
2.3	Circuits as Strings	11
2.4	Universal Circuits	11
2.5	Summary	12
3	Turing Machines	13
3.1	Turing Machines	13
3.2	Alphabet Reduction	14
3.3	Tape Reduction	14
3.4	Universal <i>TM</i> 's	15
3.5	Time Complexitiy classes	15
4	Time hierarchy theorem	16
4.1	Recap	16
4.2	Various complexity classes	16
4.3	Time Hierarchy theorem	17
4.4	Non-determinism	18
4.5	Summary	19
5	Reductions and Hardness	20
5.1	Non-determinism (Continued)	20
5.2	Reductions: Turing and Many-One	21
5.3	Hard and Complete Problems for a Class	23
6	NP-Completeness and the Cook-Levin Theorem	25
6.1	NP and the Prover–Verifier View	25
6.2	The First NP-Hard Language	26

6.3	The Cook–Levin Theorem	26
6.4	More NP-Complete Problems	27
6.5	Summary	28
7	Conditional implications of $P = NP$, Time Hierarchies, and Oracles	29
7.1	If $P = NP$, then $EXP = NEXP$	29
7.2	Nondeterministic Time Hierarchy	30
7.3	Oracle Turing Machines	31
8	Baker-Gill-Solovay and Mahoney’s Theorems	33
8.1	Oracle Turing Machines	33
8.2	The Baker-Gill-Solovay Theorem	34
8.3	Conditional Implications of $P = NP$	36
8.3.1	Unary NP-Complete $\Rightarrow P = NP$	36
8.3.2	Sparse NP-Complete $\Rightarrow P = NP$	37
9	Polynomial Hierarchy	39
9.1	Polynomial Hierarchy	39
9.2	Complete languages for Σ_i, Π_i	40
9.3	PH-complete languages?	41
10	Karp-Lipton-Sipser theorem	42
10.1	Recap	42
10.2	NP^{SAT} TM	42
10.3	Karp-Lipton-Sipser theorem	43
10.4	Meyer theorem	43
10.5	Summary	44
11	Introduction to Space Complexity	45
11.1	Setup and Definitions	45
11.2	Two example problems	47
11.3	Space Hierarchy Theorem	48
11.4	Configuration Graph and class containments	48
11.5	Savitch’s Theorem	49
12	PSPACE Completeness	51
12.1	Savitch’s Theorem	51
12.2	A PSPACE-Complete Problem	52
12.2.1	TQBF	52
12.3	Generalized Geography	53
13	Oracle and Space machines	54
13.1	Space-bounded oracle machines	54
13.2	Savitch’s Theorem does not relativise	55
13.3	A relativised separation	56

13.4	Logspace reductions	56
13.4.1	Logspace transducers	56
13.4.2	Properties of logspace reductions	57
13.5	NL-complete problems	57
14	Verifier Perspective for NL, and NL = coNL	59
14.1	Verifier Perspective for NL	59
14.1.1	A First Attempt	59
14.1.2	Why the converse doesn't hold	60
14.1.3	Read-Once Verifiers	60
14.1.4	Key Insight	61
14.2	Immerman-Szelepcsényi Theorem	61
14.2.1	Witness for r_{i+1}	61
15	Tree Evaluation Problem	63
15.1	Tree Evaluation Problem	63
15.2	Some Algebra	64
15.2.1	Multilinear extensions	64
15.2.2	Interpolation	64
15.3	The Cook-Mertz Procedure	65
16	Simulating time in square-root space	68
16.1	Recap	68
16.2	Reducing the required space	68
16.3	Time-Space simulation using TEP	69
16.4	Building the tree	70
17	Catalytic Computation	71
17.1	Setup and Definitions	71
17.2	Bounds on CL	72
17.3	Reversible Turing Machines	73
17.4	An Application: Cook-Li-Mertz-Pyne Theorem	74
18	Randomised Complexity Classes	76
18.1	Motivation	76
18.2	Randomised Complexity Classes	76
18.2.1	The Class $BPP_{c,s}$	76
18.2.2	RP, coRP	77
18.3	Error Reduction	77
18.3.1	Error Reduction for RP	77
18.3.2	Error Reduction for BPP	78

19	Randomised Complexity (continued)	80
19.1	Zero-error randomised computation	80
19.2	Characterising ZPP	81
19.3	Structural results for BPP	81
19.3.1	Adleman's Theorem	81
19.3.2	$BPP \subseteq \Sigma_2 \cap \Pi_2$	82
20	Lauteman's proof, Promise problems, and Introduction to IP	84
20.1	Lauteman's proof for $BPP \subseteq \Sigma_2 \cap \Pi_2$	84
20.2	Promise Problems	86
20.3	The Circuit Acceptance Probability Problem (CAP)	87
20.4	Reductions for Promise Problems	88
20.5	Introduction to Interactive Proofs	88
20.5.1	Deterministic Interactive Proofs	89
20.6	Major Result	89
21	Interactive Protocols	90
21.1	Interactive Protocols	90
21.2	Graph Non-Isomorphism (GNI)	91
21.3	Formal definition of IP	91
21.3.1	IP is in PSPACE	92
21.4	The Lund-Fortnow-Karloff-Nisan (LFKN) protocol	92
21.4.1	Arithmetisation	92
22	IP = PSPACE	94
22.1	#SAT \in IP via the Sum-Check Protocol	94
22.1.1	The Sum-Check Protocol: Attempt 1	95
22.1.2	The (sound) Sum-Check Protocol	96
22.2	TQBF \in IP and Shamir's Theorem	97
22.2.1	Quantified Boolean Formulas	97
22.2.2	Degree Reduction via Linearization	98
22.2.3	Protocol for TQBF	98
23	Public Coin Protocols, AM, MA	102
23.1	Graph Non-Isomorphism using public coins	102
23.1.1	Towards a public coin protocol	103
23.1.2	Goldwasser-Sipser Set Size Protocol	104
23.2	Classes AM and MA	105
24	AM Round Reduction	107
24.1	Definitions and Intuition	107
24.2	Operator View of AM and MA	108
24.3	Round Reduction	108
24.4	Containment: $AM \subseteq \Pi_2$	109

24.5 Application: Graph Isomorphism	110
25 TBA	111
26 MIP = NEXP	112
26.1 Multi-prover interactive proofs	112
26.1.1 Two provers are enough	113
26.2 Designing a protocol for NEXP	113
26.2.1 Succinct-SAT	113
26.2.2 Setting up for an LFKN-like prover claim	114
26.2.3 The Prover-Oracle protocol	115
26.2.4 Enforcing multilinearity of \tilde{A}	116
26.2.5 Ensuring \tilde{A} only takes 0/1 values on $\{0, 1\}^n$	116
26.3 Scaling down to NP	117

A dummy's guide

Theorem 1. *This is a cool theorem*

In fact we can refer to theorems using [Theorem 1](#).

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

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

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

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

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

$\sin\theta$ $\sin\theta$ $\sin\theta$ $\sin\theta$

And the same within an emphasised text

$\sin\theta$ $\sin\theta$ $\sin\theta$ $\sin\theta$

Here is a general guide for deciding which to use:

If you just want to use text within a math block, then use `\text`. For examples such as defining a set called 'PRIMES' (which are not used as functions or operators), you may use `\mathrm{PRIMES}`. If you are defining functions or operators, then use `\operatorname{blah}` as it adds the right spacing around it.

Using $\$blah\$$ should only be used when you are multiplying four variables called b, l, a and h .

Lecture 1

Starting with circuits

Scribe: Ramprasad Saptharishi

Topics covered in this lecture

- General introduction to the course
- Different circuits for Th_2

1.1 Introduction

Complexity is about the study of *computational tasks* that can be performed by *computational models* under *certain resource constraints*. This takes multiple avatars and we will see several of them throughout the course.

To begin with, we are going to deal with Boolean functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$. The computational model we will start with will just be Boolean circuits (made up of \wedge, \vee gates (of arbitrary fan-in) and \neg gates (of fan-in 1, obviously)).

1.1.1 An example to get comfortable with circuits

Consider the following simple function: $\text{Th}_2 : \{0, 1\}^n \rightarrow \{0, 1\}$ that is 1 whenever the input has at least two 1s in it. There are several ways we can try to compute this function via a circuit.

- **The obvious method:** Run through every pair of variables and check if both are one:

$$\bigvee_{i \neq j \in [n]} \bigwedge(x_i, x_j)$$

This circuit has $O(n^2)$ gates and wires, and depth 2.

- **Linear cuts:** Run through each $i = 2, \dots, n - 1$ and check if there is a 1 among $\{1, \dots, i - 1\}$

and $\{i, \dots, n\}$.

$$\bigvee_{i=2}^{n-1} \left(\left(\bigvee_{j=1}^{i-1} x_j \right) \wedge \left(\bigvee_{j=i}^n x_j \right) \right)$$

This has $O(n)$ gates, $O(n^2)$ wires and depth 3.

- **Hiding approach:** For each i , let x_{-i} be the $(n-1)$ -bit string obtained by removing the i -th coordinate. If the input has at least 2 ones if and only if $\bigvee(x_{-i}) = 1$ for all i .

$$\bigwedge_{i=1}^n \bigvee(x_{-i})$$

(and a few more such examples)

1.2 Counting functions, and circuits

It is easy to note that the number of functions $f : \{0,1\}^n \rightarrow \{0,1\}$ is exactly 2^{2^n} . How do we count the number of circuits of a given size (where size is say the number of gates)? For simplicity, let us just assume that the circuit has fan-in bounded by 2 (if you need larger fan-in, just split that into multiple gates of the same type of fan-in 2 each).

To count this, we will just try to describe each circuit as a string (its encoding), and we will just count the number of such strings. To describe a circuit entirely, we merely need to describe each gate — what is its type, and what are the children of that gate. If we index the gates as $1, \dots, s$ (including the literals x_1, \dots, x_n as gates as well), each gates needs 2 bits to describe its type (AND, OR or NOT), and $2 \log s$ bits to describe the two children feeding into it. Thus, every circuit of size s can be described using about $2s \log s + 2s$ gates. Thus, we have the following bound on the number of circuits of a certain size.

Observation 1.1. *The number of fan-in 2 circuits of size s is at most $2^{2s \log s + 2s}$.*

As a corollary, we immediately get the following.

Corollary 1.2 (Existence of ‘hard’ functions). *For all n large enough, there exists Boolean functions $f : \{0,1\}^n \rightarrow \{0,1\}$ that cannot be computed by circuits of size at most $2^n / 10n$.*

Proof. Set $s = 2^n / 10n$ in [Corollary 1.2](#) and we see that the number of circuits is way smaller than 2^{2^n} . □

To contrast this, note that *every* function can be computed using a circuit of about 2^n gates using the *truth-table*.

1.3 Dealing with varying lengths

One drawback of circuits is that the input length is fixed. Thus, if the idea was to compute functions $f : \{0,1\}^* \rightarrow \{0,1\}$, we cannot hope to compute it using a single circuit. Thus, we

will often be dealing with families of circuits $\{C_n : n \in \mathbb{N}\}$ such that C_n handles inputs of length n .

For such a family, we can now talk about its size as a function $s : \mathbb{N} \rightarrow \mathbb{N}$ where $s(n)$ is just the size of C_n . Now that we have such a function, we can talk about a circuit family being $O(n^2)$ size etc. to mean that $s(n) = O(n^2)$.

Thus, if we define $\text{SIZE}(s)$ as

$$\text{SIZE}(s) = \{f : \{0,1\}^* \rightarrow \{0,1\} : f \text{ is computable by a size } s \text{ circuit family}\},$$

then the discussion above can be phrased as saying that $\text{SIZE}(2^n \cdot n)$ contains *all* functions, and there *are* functions that are not in $\text{SIZE}(2^n/10n)$. In fact, we will soon see that something stronger is true — for every ‘reasonable’ s , we have that $\text{SIZE}(s) \subsetneq \text{SIZE}(10s)$. “With more gates comes more computation”

Lecture 2

Size Hierarchy and Universal Circuits

Scribe: Aindrila Rakshit

Topics covered in this lecture

- Circuit size hierarchy
- Circuits as strings
- Universal circuits

2.1 Recap

Recall that a Boolean circuit C_n computes a function $f : \{0, 1\}^n \rightarrow \{0, 1\}$. To compute functions on inputs of varying lengths, we consider *families of circuits* $\{C_n\}_{n \in \mathbb{N}}$, where C_n handles inputs of length n .

The size of a circuit family is measured by a function $s : \mathbb{N} \rightarrow \mathbb{N}$, where $s(n)$ denotes the number of gates in C_n .

From the previous lecture:

- The number of Boolean functions on n bits is 2^{2^n} .
- The number of circuits of size s is at most $2^{O(s \log s)}$.
- Hence, there exist functions requiring circuit size at least $2^n / (10n)$.
- Every Boolean function is computable by a circuit of size $O(2^n \cdot n)$.

We now show that allowing more gates strictly increases the computational power of circuits.

2.2 Size Hierarchy

Theorem 2.1 (Size Hierarchy Theorem). For any $0 \leq s \leq 2^n / (10n)$,

$$\text{SIZE}_n(s) \subsetneq \text{SIZE}_n(s + O(n)).$$

Proof. From the counting argument last lecture, we know that there exists a hard function $f^* : \{0, 1\}^n \rightarrow \{0, 1\}$ that requires circuit of size at least $2^n / (10n)$.

Order all n -bit strings lexicographically.

Define a sequence of functions $\{f^{(\bar{a})}\} : \{0, 1\}^n \rightarrow \{0, 1\}$ by

$$\{f^{\bar{a}}\}(x) = \begin{cases} f^*(x) & \text{if } x \leq a, \\ 0 & \text{otherwise.} \end{cases}$$

Observe that $\{f^{(\bar{0})}\}(x)$ is the constant zero function, while $\{f^{(\bar{1})}\}(x) = f^*(x)$ for all x . Moreover, $f^{(i)}$ and $f^{(i+1)}$ differ on exactly one input.

A circuit for $f^{(i+1)}$ can be obtained from a circuit for $f^{(i)}$ by adding a small equality test for the differing input and combining the outputs using the circuit in Q1 of the quiz. This increases the circuit size by $O(n)$ gates. So, $\text{size}((f^{(i+1)})) \leq \text{size}((f^{(i)}))$

Thus, the circuit size increases gradually from $O(1)$ to $O(2^n / (10n))$, implying that for any $s(n)$ in this range there exists a function computable with size $s(n) + O(n)$ but not with size $s(n)$. \square

2.3 Circuits as Strings

A Boolean circuit of size s can be encoded as a binary string of length $2s + 2 \log s$ by describing, each gate and type using 2 bits and its two children by $\log s$ bits each.

This allows us to treat circuits themselves as inputs to other circuits.

2.4 Universal Circuits

A universal circuit $U_{s,n}$ takes as input:

- a description $\langle C \rangle$ of a circuit C of size at most s with n inputs, and
- an input $x \in \{0, 1\}^n$,

and outputs $C(x)$.

Theorem 2.2 (Universal Circuit). For all $n, s \in \mathbb{N}$, there exists a Boolean circuit $U_{s,n}$ of size $O(s \log s)$ such that for every Boolean circuit C of size at most s with n inputs and every input $x \in \{0, 1\}^n$,

$$U_{s,n}(\langle C \rangle, x) = C(x),$$

where $\langle C \rangle$ denotes a binary description of C .

Proof-sketch. A Boolean circuit C of size s can be encoded using $2s + 2 \log s$ bits for each gate, its type and the 2 children of each gate.

The universal circuit $U_{s,n}$ simulates C gate by gate. Let g_1, g_2, \dots, g_s be the gates of C , and let G_i denote the value of gate g_i on input x . The description $\langle C \rangle$ specifies, for each gate g_i , its gate type and its 2 children.

We can imagine that we have wires g_1, \dots, g_s , which are initialised to zeroes, and wire g_i is supposed to hold the value of the gate g_i . If we wish to update the value of gate i (after having computed all the previous gates), build a circuit of the form:

$$g_i^{(\text{new})} = \text{Lookup}_2(c_{i,1}c_{i,2}, \text{AND}(g_{i_1}, g_{i_2}), \text{OR}(g_{i_1}, g_{i_2}), \text{NOT}(g_{i_1}), x_{i_1})$$

where $c_{i,1}c_{i,2}$ is the two bits that determine the type of the gate i , and i_1 and i_2 are the two children feeding in. Of course, we also need to *lookup* g_{i_1}, g_{i_2} etc. (all the parts above in blue) from the circuit description, but they are also appropriate lookup circuits.

One can check that the size of the universal circuit is $O(s^2 \log s)$ (since there are s phases that computes one gate each, and the lookups are of size $O(s \log s)$). \square

2.5 Summary

- There is a strict hierarchy of circuit size classes.
- Circuits can efficiently simulate other circuits.
- Circuit complexity is inherently non-uniform.

Lecture 3

Turing Machines

Scribe: Ananya Ranade

Topics covered in this lecture

- What are Turing Machines?
- Alphabet and Tape Reductions
- Universal Turing Machines
- Time Complexity

In this lecture we understood what are Turing Machines, and how they correctly capture the idea of computation.

3.1 Turing Machines

A Turing machine is a machine which is composed of an infinite input tape, several work tapes and a finite state machine (automaton). The input to the machine is written on the one-sided infinite input tape starting from its left most end. It is followed by the special symbol # which is followed by \sqcup all the way to infinity. The machine is allowed to make changes to the input tape once it starts running.

The tape alphabet Γ is a finite set of symbols. Here we assume it is some set containing $\{0, 1, \#, \sqcup\}$ with #, \sqcup special symbols only used to indicate end of input and space respectively.

The work tapes are initially blank one-sided infinite tape on which we can read, write and erase one symbol at a time.

The finite state machine has heads (pointers) to the input tape and each of the work tapes, and the heads can read, write or erase symbol in the current cell it is pointing at. It can also move left or right by 1 place at a time. These instructions are given by the current state of the finite state machine after reading the symbols the heads are currently reading. The finite state machine has 3 special states : $q_{start}, q_{accept}, q_{reject}$. The machine starts from initial state q_{start} with

all heads at leftmost ends of the respective tapes. It accepts the input and stops running once it reaches q_{accept} state, and it rejects the input and stops running once it reaches q_{reject} state.

Definition 3.1 (Basic aspects of Turing machines). 1. Language computed by TM M is denoted $L(M)$ which is $\{x : M \text{ on } x \text{ accepts}\}$.

2. M is a halting TM if it halts on all inputs.

3. M computes $f : \Sigma^* \rightarrow \{0, 1\}$ if $f(x) = 1 \iff x \in L(M)$.

4. Time Complexity of M : Let $T_{M,x}$ be the number of steps M took to run on input x . Then the time complexity of M denoted T_M is a function from $\mathbb{N} \rightarrow \mathbb{N}$ where $T_M(n) = \max_{x \in \Sigma^n} T_{M,x}$. Time complexity of $f : \Sigma^* \rightarrow \{0, 1\}$ is the lowest time complexity among all possible halting TM's M computing f .

◇

After defining so many things, it is a natural question to ask that does the time complexity change a lot based on the number of tapes or alphabet size, since in the definition of TM we allowed it to have any finite alphabet size and work tapes.

To address this issue we will prove the following theorems.

3.2 Alphabet Reduction

Theorem 3.2. If M is a TM with tape alphabet Γ , then there is an equivalent TM M' with tape alphabet $\{0, 1, \#, \sqcup\}$. Furthermore $T_{M'} = O(T_M)$ where the constant depends only on $|\Gamma|$.

Proof-idea. In order to compensate for a smaller alphabet size, we make the finite state machine larger. Basically we will encode all symbols in Γ using roughly $\log|\Gamma|$ size binary encoding. The states in the automaton will slowly read the $\log|\Gamma|$ size encoding one step at a time and basically simulate one move of M in around $\log|\Gamma|$ steps. Thus, we can always reduce the tape alphabet. However, in doing so, we incur $\log|\Gamma|$ factor extra time. This is a constant which depends only on the alphabet size and not on other parameters like input and hence it is not a bad blow up. □

3.3 Tape Reduction

Theorem 3.3. If M is a TM with k tapes then there is an equivalent TM M' with just 2 tapes. $T_{M'} = O(T_M \log(T_M))$.

Proof Idea : We will interleave the content on all work tapes in a single tape. We will need a symbol α' for each symbol α in the alphabet of M . This is basically to tell the i^{th} tape head pointer would have been at this position. Then, we just scan across all the k tape heads and then move. Since on input of length n it is spending atmost $T_M(n)$ time, the maximum it has to traverse to simulate 1 step of M is $T_M(n)$. So, to simulate M on length n input, it needs atmost $T_M(n)^2$ time. We can also reduce the alphabet size as done in previous part. Thus, $T_{M'} = O(T_M^2)$ where the constant depends on $k, |\Gamma|$.

We can do this in a better way. Assume that the alphabet size is $|\Gamma|^k$, and we have a 2 sided infinite work tape. From the starting head position, we split the left and right side into consecutive blocks L_1, L_2, \dots and R_1, R_2, \dots respectively, where $|L_i| = |R_i| = 2^i$. Further, at any point for all i , L_i and R_i are either full or empty and atmost one of them is full. Now, we try to simulate all k head movements by thinking of it as moving tapes instead of moving heads, and then doing the corresponding changes to the work tape. Since there are a lot of blank spaces, the amortised time to do this T_M times is $\log(T_M)$. We can convert it to smaller alphabet size and single sided infinite tape by incurring constant factor blow up. Thus, $O(T_M \cdot \log(T_M))$ time suffices.

Any TM M can be fully described by providing the description of states, transitions, tape alphabet, and number of tapes. All these can be described as binary strings using some convention. Thus, each TM has a finite encoding as a binary string. We denote it as $\langle M \rangle$. Further, we will say that any string of the form $\langle M \rangle \# \alpha$ (where $\#$ is a special symbol and α is any finite string) is also an encoding of M denoted $\langle M \rangle_\alpha$. Thus, any TM M has infinitely many encodings.

3.4 Universal TM 's

We can build a TM \mathcal{U} which takes as input $(\langle M \rangle, x)$ and does what M does on x . Furthermore, $T_{\mathcal{U}}(\langle M \rangle, x) = O(T_{M,x} \log(T_{M,x}))$, where the constant depends only on Γ, k and not the padding α . The idea is the same as in tape reduction proof.

3.5 Time Complexity classes

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function. Then, we define the following "deterministic time complexity classes":

$$DTIME(f) := \{L \subseteq \{0,1\}^* : \text{There is a det. TM } M \text{ with } L(M) = L \text{ with } T_M = O(f)\}$$

Once we have the above, we can define the class P (of polynomial time computable functions) as

$$\begin{aligned} P &:= \bigcup_{c \geq 1} DTIME(n^c) \\ &= \{L : \exists c \text{ with } L \in DTIME(n^c)\} \end{aligned}$$

Lecture 4

Time hierarchy theorem

Scribe: Chandralekha P

Topics covered in this lecture

- Defining various complexity classes
- Time hierarchy theorem
- Not-deterministic Turing machine

4.1 Recap

Recall that the number of tapes and alphabet size do not affect the running time of TM by much.

We saw how to encode a TM as a string, and that there can be arbitrarily many such encoding for a given TM based on arbitrary length padding. We also saw that there is a Universal TM U , which on given input as $\langle \langle M \rangle_{\alpha}, x \rangle$, does what M does on x .

$$T_{U,(\langle M \rangle_{\alpha}, x)} = O(T_{M,x} \log T_{M,x})$$

We defined $\text{DTIME}(f) = \{L : L \text{ is recognised by a TM } M \text{ with } T_M = O(f)\}$ where $f : \mathbb{N} \rightarrow \mathbb{N}$. We can now use the above definition to come up with different class of functions, and also see how the set of recognised languages changes based on the function we take. We will also see how things will be if we allow non determinism power to our TM.

4.2 Various complexity classes

Given the definition of $\text{DTIME}(f)$, our natural question would be to see the class of recognised languages for various class of functions. By that idea, we define the following classes:

- $P = \bigcup_{c \geq 1} \text{DTIME}(n^c)$

- $E = \bigcup_{c \geq 1} \text{DTIME}(2^{c \cdot n})$
- $\text{EXP} = \bigcup_{c \geq 1} \text{DTIME}(2^{n^c})$
- $\text{EEXP} = \bigcup_{c \geq 1} \text{DTIME}(2^{2^{n^c}})$

From the above, we can clearly see that $P \subseteq E \subseteq \text{EXP} \subseteq \text{EEXP}$.

Now, the natural question one could ask is, are these containments strict? This leads us to the Time Hierarchy theorem.

4.3 Time Hierarchy theorem

Before we see the general theorem, we will see a special case of it.

Theorem 4.1. $\text{DTIME}(n^3) \subsetneq \text{DTIME}(n^5)$

Proof. To show the strict containment, it is sufficient to find a language that is in $\text{DTIME}(n^5)$ but not in $\text{DTIME}(n^3)$. So, we want a language L that is doable in n^5 time, but no $O(n^3)$ time TM solves it correctly. Consider the following:

D : Input $x (= \langle M \rangle)$

1. $n = |x|$
2. Mark off n^5 cells on a tape.
3. Run n^5 steps of the UTM to simulate M on $\langle M \rangle$
4. If M accepts by then, we reject
5. Else, we accept.

(The idea used here is diagonalisation which is used in multiple proofs. The idea is to make sure that every $O(n^3)$ machine fails at some string, which in this case is its own encoding. This technique works only when you are simulating another machine efficiently.)

By the above, we get that

$$L_D = \{ \langle M \rangle : \text{UTM on } (\langle M \rangle, \langle M \rangle) \text{ either rejects or times out in } n^5 \text{ steps} \}$$

Hence, it is easy to see that $L_D \in \text{DTIME}(n^5)$. Now, we need to show that no $O(n^3)$ TM can recognise this language.

Claim. If M is a machine with $T_M = O(n^3)$ then $L_M \neq L_D$.

Proof. Consider the string $x = \langle M \rangle$, $|x| = n$. Then, M decides x in $O(n^3)$ steps. Hence, for the UTM to simulate M on x , $O(n^3 \log n^3)$ steps will be sufficient. Since $n^5 \gg n^3 \log n^3$, the run will be fully simulated on D . Therefore, M accepts x iff D rejects x and hence the languages differ at least at this string, and hence cannot be equal. \square

Hence, using the above, we have gotten the distinguishing language.

Now the question is, how can we be sure that n^5 time is actually sufficient? Because all of these time bounds are in asymptotics. So to make sure it works, we will take a large enough padded encoding of M to make sure that the UTM does indeed fully simulate M on x . □

Now, from the above proof, it is clear that if the two functions differ by a more than a log factor, then there will be a distinguishing language. This precisely is the Time hierarchy theorem statement.

Theorem 4.2 (Time hierarchy theorem). $f, g : \mathbb{N} \rightarrow \mathbb{N} \implies \text{DTIME}(f) \subsetneq \text{DTIME}(g)$ provided $f \log f = o(g)$ and g is time constructible (that is, there is a TM that computes $1^n \rightarrow 1^{g(n)}$ runs in atmost $O(g(n))$ time)

From the above theorem, we get that $P \subsetneq E \subsetneq \text{EXP} \subsetneq \text{EEXP}$

4.4 Non-determinism

We have seen that in the case of automata, non determinism allows us to produce a more succinct automata than the case of deterministic. In the same way, we could allow non determinism for Turing machines which could give more power to the machine.

To do this, we give the TM two transition functions δ_0 and δ_1 . At each stage of the run, the machine decides whether to use δ_0 or to use δ_1 . Hence, this gives rise to various paths based on the choice of function.

Now, the natural question is to ask when we consider the machine to be halting? We say it is halting if on every input and every path, it halts after finitely many steps.(that is, it reaches an accept or reject state)

We call the above machine a Not-deterministic machine. The time taken by it is given by:

$$T_{M,x} = \max_{\text{paths}} \{\text{time taken by } M \text{ on } x \text{ on path}\}$$

$$T_M : \mathbb{N} \rightarrow \mathbb{N}$$

$$\text{defined by } T_M(n) = \max_{x \in \{0,1\}^n} T_{M,x}$$

Now, based on different acceptance conditions, we get different TM. Two such machines are as follows.

- Non-deterministic TM: $x \in L_M \iff$ there is some path on which M accepts.
- co-non-deterministic TM: $x \in L_M \iff$ all paths accept.

Now, based on what non-deterministic machine can do, we can again define various classes of languages. $\text{NTIME}(f) = \{L : L \text{ is accepted by a non-det. TM that runs in time } O(f)\}$.

Similarly, we can also define $\text{coNTIME}(f)$.

Now, using this for various family of functions, we get $NP = \bigcup_{c \geq 1} NTIME(n^c)$. Similarly, NE, NEXP, etc. can be defined.

Now, the natural question to ask is, what problems are in these classes, and is there a hierarchy in these classes too. To see this, we will see some problems in each class.

- $CIRCUIT-EVAL = \{ \langle C, x \rangle : C(x) = 1 \} \in P$,
- $CIRCUIT-SAT = \{ \langle C \rangle : \exists x \text{ s.t. } C(x) = 1 \} \in NP$,
- $TAUT = \{ \langle C \rangle : \forall x, C(x) = 1 \} \in coNP$.

More of these problems, and their relations will be seen in next class.

4.5 Summary

- Defined various classes of languages based on time taken.
- There is a strict hierarchy of classes based on time taken (and even within the classes).
- Introduced to Non-determinism in TM.

Lecture 5

Reductions and Hardness

Scribe: Hrishikesh Saikia

Topics covered in this lecture

- Non-determinism
- Turing Reductions and Many-One Reductions
- Hard and Complete Problems for a Complexity Class
- NP Completeness

The primary goal of this lecture would be to formalize what it means to say that one problem is “harder” than another in a complexity class.

5.1 Non-determinism (Continued)

Given a not-deterministic machine M and an input x , we can label the possible computational paths of M by strings $p \in \{0,1\}^*$, where the i -th bit of p indicates the transition function applied at the i -th step. So, given a fixed p , M acts as a deterministic machine. When we are trying to design not-deterministic Turing Machines or reason about them, it is often convenient to take this deterministic perspective.

An example construction: We will show that $\text{CIRCUIT-SAT} \in \text{NP}$ and $\text{TAUT} \in \text{coNP}$. For an input $\langle C \rangle$ (a circuit description), the computational paths in a not-deterministic machine for either of the problems will correspond to the possible inputs of C . Consider a not-deterministic Turing Machine M with the following syntax:

On input $\langle C \rangle$, do the following:

1. Guess a path $x \in \{0,1\}^n$, where n is the number of inputs of C .

2. On each path, do:
 - (a) Compute $\text{CIRCUIT-EVAL}(\langle C \rangle, x) = b$ (say).
 - (b) Accept (on this path) if $b = 1$.
 - (c) Reject (on this path) if $b = 0$.

Note that $\text{CIRCUIT-EVAL} \in \text{P}$ as given any circuit description $\langle C \rangle$ and input x , we can compute the value at each gate in time polynomial in the size of C . So, if we now invoke the non-deterministic acceptance criterion on M , we get a non-deterministic polynomial time machine solving CIRCUIT-SAT (since we just need atleast one x such that $C(x) = 1$ for $\langle C \rangle$ to be in the language). So, $\text{CIRCUIT-SAT} \in \text{NTIME}(\text{poly}(n)) = \text{NP}$.

On the other hand, if we invoke the co-non-deterministic acceptance criterion on M , we get a co-non-deterministic polynomial time machine solving TAUT (since we need all x 's to satisfy $C(x) = 1$ for $\langle C \rangle$ to be in the language). So, $\text{TAUT} \in \text{coNTIME}(\text{poly}(n)) = \text{coNP}$.

Observation 5.1. For any language L , we have $L \in \text{NP} \iff \bar{L} \in \text{coNP}$.

Proof. For the forward direction, consider a non-deterministic machine M_L for L . Construct the not-deterministic machine M'_L that on each path just negates the answer. As a co-nondeterministic machine M'_L precisely accepts all those strings that are not on L .

The same argument shows the other direction as well. □

5.2 Reductions: Turing and Many-One

Motivating reductions: Consider the following languages:

1. $\text{VertexCover} = \{(G, k) : G \text{ has a vertex cover of size } \leq k\}$
2. $\text{Clique} = \{(G, k) : G \text{ has a clique of size } \geq k\}$
3. $\text{IndSet} = \{(G, k) : G \text{ has an independent set of size } \geq k\}$
4. $\text{UnSAT} = \{\langle C \rangle : C(x) = 0, \forall x\}$

Typically, when we see problems whose “yes” instances are answers to questions of the flavour “Does there exist something satisfying a certain property?”, the problems are likely in NP as the non-deterministic machine can simply guess the object (path) and check if it satisfies the property. In the above case, the first three languages are in coNP:

1. For VertexCover , the machine can guess a vertex set of size k , and then it is a polynomial time check to see if those k vertices form a vertex cover: go over all $O(n^2)$ edges of G (n is the number of vertices of G) and for each edge, check if either of the endpoints are in the chosen set.
2. For Clique , the machine can guess a set of k vertices, and then it is a polynomial time check to see if those k vertices form a clique: go over all $O(k^2)$ pairs of vertices from the set and for each pair, check if they have an edge between them.

3. For IndSet, the machine can guess an set of k vertices, and then check in polynomial time if those k vertices form an independent set: go over all $O(k^2)$ pairs of vertices from the set and for each pair, check if they have an edge between them.

On the other hand, UnSAT \in coNP as the coNP machine can simply guess the x and check in polynomial time that $C(x) = 0$, and it will accept if the check passes on all paths.

Now, note that $S \subseteq V(G)$ is a vertex cover in $G \iff \bar{S}$ is a clique in $\bar{G} \iff \bar{S}$ is an independent set in G . In particular, $(G, k) \in \text{VertexCover} \iff (\bar{G}, n - k) \in \text{Clique} \iff (G, n - k) \in \text{IndSet}$. So, we can solve an instance of any one of the problems by “reducing” it to an instance of either of the remaining two with only a polynomial blowup (the complement of a graph can be computed in polynomial time). So these problems seem to be “equivalent” in some sense. On the other hand, for any circuit $\langle C \rangle$, we have $\langle C \rangle \in \text{TAUT} \iff \langle \neg C \rangle \in \text{UnSAT} \iff \langle \neg C \rangle \notin \text{CIRCUIT-SAT}$. So these three problems also seem to be of “similar” complexity as we can solve instances of one by converting them to instances of another with only a polynomial blowup.

This motivates us to define a notion of one problem being “no harder to solve” than another, and the most natural definition we can come up with is to simply say that problem A is no harder than problem B if we can solve instances of A using instances of B with a reasonable overhead.

Definition (Turing reductions (informal)). *A language A is said to be polynomial time Turing reducible to a language B , denoted $A \leq_p^{\text{Turing}} B$, if given a subroutine for B , we can solve A with polynomial overhead.* \diamond

(The above definition is vague at places, and we will define what Turing reductions are more formally when we discuss oracles later). So according to the above definition, for any languages $A, B \in \{\text{VertexCover}, \text{Clique}, \text{IndSet}\}$, or in the set $\{\text{UnSAT}, \text{TAUT}, \text{CIRCUIT-SAT}\}$, we have $A \leq_p^{\text{Turing}} B$.

The above definition however has a drawback. By the above definition, TAUT Turing reduces to CIRCUIT-SAT. However, we believe that an NP machine for TAUT is hard to construct, and so we do not want to consider CIRCUIT-SAT and TAUT to be of “similar” complexity. The above definition does not respect that. In this light, we consider another weaker type of reductions where the “yes” instances of A will always map to “yes” instances of B and “no” instances will always map to “no” instances (in other words, unlike in Turing reductions, we cannot simply take an instance of the other problem and flip the answer).

Definition 5.2 (Many-One Reductions). *A language A is said to be polynomial time many-one reducible to another language B , denoted $A \leq_p^m B$, if there exists a deterministic polynomial time computable map $\varphi : \Sigma_A^* \rightarrow \Sigma_B^*$ mapping instances of A to instances of B such that $x \in A \iff \varphi(x) \in B$.* \diamond

As an example, note that UnSAT is many-one reducible to TAUT as we can simply take φ mapping $\langle C \rangle$ to $\langle \neg C \rangle$. Similarly, it is easy to see that VertexCover, Clique, and IndSet are all many-one reducible to each other under the stated reductions. However, the reduction from UnSAT or TAUT to CIRCUIT-SAT discussed earlier is not many-one as “yes” instances of one were getting mapped to “no” instances of the other and vice-versa.

With the definition in place, we can now easily make the following observation:

Observation 5.3. *Suppose $B \in \text{NP}$, and $A \leq_p^m B$. Then $A \in \text{NP}$.*

Proof. Given an input x for A , we can run the reduction $y = \varphi(x)$ and then run the machine for B on input y . \square

The same holds if we replace NP by any class that can accommodate a polynomial slowdown (say coNP, EXP, NEXP etc.).

5.3 Hard and Complete Problems for a Class

Once we now have a notion of one problem being no harder than another, it is natural to ask if there are problems that are harder than all other problems in a class. This motivates the following definitions (under many-one reductions):

Definition 5.4 (*C-hardness and completeness*). *A language B is C-hard for some complexity class \mathcal{C} if $A \leq_p^m B$ for every $A \in \mathcal{C}$.*

A language B is C-complete if $B \in \mathcal{C}$ and B is C-hard. \diamond

The natural question to ask now is whether there even exists hard and complete problems for any complexity class. The following theorem answers this in the affirmative:

Theorem 5.5 (Cook-Levin). *CIRCUIT-SAT is NP-complete.*

We will see the proof of the above in the next lecture. Now, we will take the theorem for granted and use it to show that the language 3-CNF-SAT is NP-complete. A 3-CNF formula is a boolean formula of the form $C_1 \wedge C_2 \wedge \dots \wedge C_m$, where each of the C_i 's (called a clause) is an OR of three literals (either a variable or its negation). The language 3-CNF-SAT then consists of all satisfiable 3-CNF formulas. We have the following claim.

Claim 5.6. *3-CNF-SAT is NP-complete.*

Proof. Note that 3-CNF-SAT is in NP as a non-deterministic machine can simply guess the satisfying assignment. We will now show that CIRCUIT-SAT reduces to 3-CNF-SAT. This will show that 3-CNF-SAT is NP-complete as any NP language reduces to CIRCUIT-SAT (we are using the transitivity of many-one reductions which is easy to verify: just compose the mappings).

Instead of 3-CNF-SAT, we will consider the more general problem 3-LOCAL-SAT, where each clause can be any boolean function of three variables. Consider an instance $\langle C \rangle$ of CIRCUIT-SAT. We need to transform C into a 3-local formula such that C is satisfiable iff the 3-local formula is. Here is the idea behind the transformation:

For each non-input gate in C , introduce a new variable (semantically, to denote the value of the gate). The clauses of our 3-LOCAL-SAT would then consist of a single clause for each non-input gate such that semantically, each clause will confirm that the corresponding gate computed the value correctly, and one output clause saying that the output gate computed value 1. The details of the construction are as follows (we repeatedly use the boolean expression for equality: $(x = y) \equiv (x \wedge y) \vee (\neg x \wedge \neg y)$):

1. *AND gate*: If $g = g_1 \wedge g_2$, then the corresponding clause would be $(g_1 \wedge g_2 \wedge g) \vee (\neg(g_1 \wedge g_2) \wedge \neg g)$.
2. *OR gate*: If $g = g_1 \vee g_2$, then the corresponding clause would be $((g_1 \vee g_2) \wedge g) \vee (\neg(g_1 \vee g_2) \wedge \neg g)$.
3. *Negation gate*: If $g = \neg h$, then the corresponding clause would be $(g \wedge \neg h) \vee (\neg g \wedge h)$.
4. *Output clause*: If g is the variable corresponding to the output gate, the output clause would simply be g .

Semantically, the 3-LOCAL formula is exactly capturing a correct computation of the circuit. So the formula is satisfiable iff the circuit is: any satisfying assignment for the circuit extends to a satisfying assignment for the formula where the clause variables take the value computed at the corresponding gates, and any satisfying assignment for the formula gives a satisfying assignment for the circuit with the corresponding node values.

Finally, to reduce 3-LOCAL-SAT to 3-CNF-SAT, write down each 3-LOCAL clause in CNF form as follows: consider the truth table of the 3-LOCAL clause. It has 8 rows. Write the corresponding indicator CNF clauses for the rows corresponding to value 0, and AND all such clauses. This is a polynomial time reduction as the size blowup is a constant factor. This completes the proof. \square

Lecture 6

NP-Completeness and the Cook-Levin Theorem

Scribe: Krishnashree J B

Topics covered in this lecture

- Prover-Verifier characterization of NP
- Cook-Levin Theorem
- Examples of NP-complete problems

6.1 NP and the Prover–Verifier View

Recall that

$$\text{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c).$$

An equivalent characterization of NP is given by the prover–verifier viewpoint.

Definition 6.1. A language L is in $\widetilde{\text{NP}}$ if there exists a deterministic polynomial-time algorithm $V(x, \pi)$ such that:

- (Completeness) If $x \in L$, then there exists a proof π with $V(x, \pi)$ accepting.
- (Soundness) If $x \notin L$, then for all proofs π , $V(x, \pi)$ rejects.

Here, the length of π is polynomially bounded in $|x|$. ◇

Intuitively, NP consists of problems whose yes-instances admit efficiently verifiable certificates.

Claim 6.2. $\text{NP} = \widetilde{\text{NP}}$.

Proof. We prove both inclusions.

(\subseteq) Let $L \in \text{NP}$. Then there exists a non-deterministic Turing machine M that decides L in polynomial time. For an input x , a certificate π can be taken to be a description of an accepting computation path of M on x . A deterministic verifier can simulate this path. Hence $L \in \widetilde{\text{NP}}$.

(\supseteq) Let $L \in \widetilde{\text{NP}}$. Then there exists a polynomial-time verifier $V(x, \pi)$ such that $x \in L$ if and only if there exists a certificate π with $V(x, \pi)$ accepting. A non-deterministic Turing machine can guess π and run V deterministically. This takes polynomial time, and hence $L \in \text{NP}$. \square

6.2 The First NP-Hard Language

Let $A \in \text{NP}$, and let M_A be a non-deterministic Turing machine deciding A in time t steps.

Define the language

$$L = \{ \langle M, x, 1^t \rangle : M \text{ accepts } x \text{ within } t \text{ steps} \}.$$

It can be shown that for every language $A \in \text{NP}$, there exists a polynomial-time many-one reduction from A to L . Fix a language $A \in \text{NP}$ and let M_A be a non-deterministic Turing machine deciding A in time n^5 without loss of generality. Define

$$\Phi_A : x \mapsto (\langle M_A \rangle, x, 1^{|x|^5}).$$

By construction, $x \in A$ if and only if M_A accepts x within $|x|^5$ steps. Hence, L is NP-complete. However, this language is not very intuitive, so other NP-complete languages are explored.

6.3 The Cook–Levin Theorem

Theorem 6.3 (Cook–Levin). $\text{CIRCUIT-SAT} = \{ \langle C \rangle : \exists x \text{ such that } C(x) = 1 \}$ is NP-complete.

Proof sketch. To show NP-hardness, let $A \in \text{NP}$ and let M_A be a non-deterministic Turing machine deciding A in t time steps. For an input x , we construct a Boolean formula Ψ_x encoding a *computational tableau* of the execution of M_A on x .

The tableau records, for each time step, the machine state, tape contents, and head positions. The formula enforces:

- a correct start configuration,
- the existence of an accepting configuration, and
- local consistency between consecutive configurations.

The formula Ψ_x is satisfiable if and only if M_A accepts x . Moreover, Ψ_x can be constructed in polynomial time. This gives a polynomial-time many-one reduction from A to CIRCUIT-SAT . \square

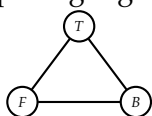
6.4 More NP-Complete Problems

Once CIRCUIT-SAT is known to be NP-complete, from previous lecture we can conclude 3CNF-SAT. Also, many other problems can be shown NP-complete via many-one reductions.

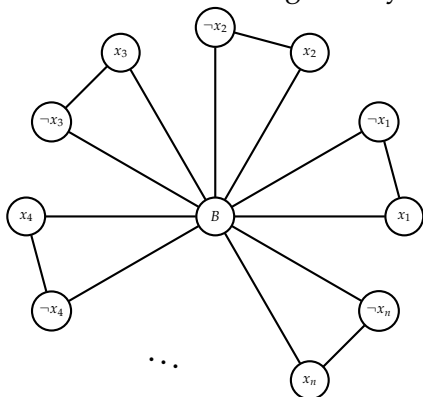
Theorem 6.4. 3-COLORING is NP-complete.

Proof sketch. For NP-hardness, we give a polynomial-time reduction from 3CNF-SAT. A gadget is constructed for a given formula such that the resulting graph is 3-colourable if and only if the formula is satisfiable. The vertices in the gadget are assigned 3 colours {True, False, Base} where Base has no significance in the respective formula, whereas the rest correspond to the boolean value given to the literal corresponding to the vertex in the gadget.

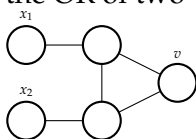
This ensures that the three vertices get three different colours, and we will use this in subsequent gadgets.



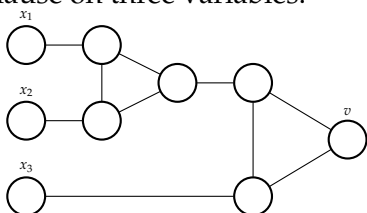
To ensure each literal gets only T or F as its colour, we do the following



And finally, we need to set up a gadget for a clause. This is set up using the following gadget for the OR of two variables.



The main observation is that if x_1 and x_2 are coloured the same, then every valid 3-colouring *must* have v coloured with the same colour. On the other hand, if x_1 and x_2 get two different colours, there is a valid colouring where v is coloured with either x_1 's or x_2 's colour. Thus, the only way v can be coloured T is if one of x_1 or x_2 are coloured T . The following extends this to a clause on three variables.



By connecting v to B and F , we can ensure that v is expected to be coloured T . □

Theorem 6.5. CLIQUE is NP-complete.

Proof sketch. For NP-hardness, we give a polynomial-time reduction from 3CNF-SAT. Given a formula $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_m$, we construct an m -partite graph with one vertex for each assignment of the literals in the clause that evaluates the clause to true, and add an edge between two vertices if the corresponding literals are consistent.

The graph contains a clique of size m if and only if one can choose a consistent true literal from each partition, which holds if and only if φ is satisfiable. \square

6.5 Summary

- NP admits an equivalent prover–verifier characterization.
- The Cook–Levin theorem establishes CIRCUIT-SAT is a NP-complete problem.
- Many other problems are NP-complete via reductions from SAT.

Lecture 7

Conditional implications of $P = NP$, Time Hierarchies, and Oracles

Scribe: Soham Ghosh

Topics covered in this lecture

- Conditional implications of $P = NP$
- Nondeterministic Time Hierarchy Theorem
- Introduction to Oracle Turing Machines

7.1 If $P = NP$, then $EXP = NEXP$

Claim 7.1. *If $P = NP$, then $EXP = NEXP$.*

Proof idea. We use padding.

Let $L \in NEXP$. Suppose $L \in NTIME(2^{n^5})$, and let M be a nondeterministic Turing machine deciding L in time 2^{n^5} .

Define the padded language

$$L_{\text{pad}} = \{x\#0^m : m = 2^{n^5}, n = |x|\}.$$

We claim that $L_{\text{pad}} \in NP$. Consider a nondeterministic TM D that works as follows:

- Given an input $x\#0^m$, first verify that $m = 2^{n^5}$ where $n = |x|$. This can be done in polynomial time since m is smaller than the total input length, and 2^{n^5} can be computed using binary exponentiation.
- If the padding length is valid, delete the padding and run M on x .

Since the computation of M takes time 2^{n^5} , which is at most the input length, D runs in non-deterministic polynomial time. Hence $L_{\text{pad}} \in NP$.

Now assume $P = NP$. Then there exists a deterministic polynomial-time TM D' deciding L_{pad} . Its running time is polynomial in

$$n + m = n + 2^{n^5}.$$

We now construct an EXP machine M' deciding L :

- On input x , pad it to obtain $x\#0^{2^{n^5}}$.
- Run D' on the padded input.

The running time of M' is exponential in n , and it decides L . Hence $L \in \text{EXP}$, proving $\text{NEXP} \subseteq \text{EXP}$. Since the reverse containment is trivial, we conclude $\text{EXP} = \text{NEXP}$. \square

7.2 Nondeterministic Time Hierarchy

Claim 7.2. $\text{NTIME}(n^3) \subsetneq \text{NTIME}(n^5)$.

Proof idea. The standard diagonalisation argument from the deterministic case does not directly work here, since an NTM may have exponentially many computation paths, and flipping acceptance requires examining all of them. Instead, we use a technique called *lazy diagonalisation*.

We construct a unary language $L \subseteq \{1\}^*$ that is accepted by an $\text{NTIME}(n^5)$ machine but not by any $\text{NTIME}(n^3)$ machine.

Let M_1, M_2, \dots be a computable enumeration of all nondeterministic Turing machines.

Partition the natural numbers into disjoint half-open, exponentially growing intervals $(a_i, b_i]$ such that $b_i = 2^{(a_i+1)^3}$. For example, let $a_1 = 0$ and $b_1 = 2^{(0+1)^3}$, and $a_2 = b_1$ and $b_2 = 2^{(a_2+1)^3}$ etc.

Define a nondeterministic TM D as follows:

- If the input is not of the form 1^n , reject immediately.
- On input 1^n , compute the interval $(a_i, b_i]$ containing n .
- If $a_i < n < b_i$, simulate M_i on input 1^{n+1} for n^5 steps. Accept if M_i accepts; if M_i rejects or does not halt, reject.
- If $n = b_i$, simulate *all* computation paths of M_i on input 1^{a_i+1} and flip the acceptance outcome on every path.

The running time of D on input 1^n is at most n^5 for all n . Note that for $n = b_i$, we have $n = 2^{(a_i+1)^3}$.

Now suppose, for contradiction, that L_D is decided by some machine M running in $\text{NTIME}(n^3)$. Let M_i be a sufficiently padded encoding of M .

Then we have:

$$D(1^{a_i+1}) = M(1^{a_i+1}), \quad D(1^{a_i+2}) = M(1^{a_i+2}), \quad \dots, \quad D(1^{b_i}) = M(1^{b_i}).$$

By construction, $D(1^n) = D(1^{n+1})$ for all $a_i < n < b_i$, but

$$D(1^{a_i+1}) \neq D(1^{b_i}),$$

a contradiction.

Thus, no $\text{NTIME}(n^3)$ machine can decide L_D , and the claim follows. \square

Theorem 7.3 (Nondeterministic Time Hierarchy Theorem). *Let $f, g : \mathbb{N} \rightarrow \mathbb{N}$ be time-constructible functions such that $f(n+1) = o(g(n))$. Then*

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

Remark. Note the absence of the logarithmic factor present in the deterministic hierarchy theorem. This is because, in the nondeterministic setting, we can bypass tape reduction. See the first assignment for details.

Corollary 7.4. $\text{NP} \subsetneq \text{NEXP}$.

7.3 Oracle Turing Machines

An *oracle Turing machine* is a Turing machine equipped with a special oracle tape and three distinguished states:

$$q_{\text{query}}, q_{\text{yes}}, q_{\text{no}}.$$

At any point during the computation, the machine may write a string on the oracle tape and enter the state q_{query} . If the string belongs to a fixed language B , the machine transitions to q_{yes} ; otherwise, it transitions to q_{no} . This oracle access is assumed to take unit time.

A machine with oracle access to B is denoted by M^B .

For example, P^{SAT} denotes polynomial-time machines with access to a SAT oracle. Clearly, every language in NP lies in P^{SAT} . Moreover, TAUT (the set of tautologies) also lies in P^{SAT} , since we may flip the oracle's answers. Hence,

$$\text{NP} \cup \text{coNP} \subseteq \text{P}^{\text{SAT}}.$$

We now give an example of a language in P^{SAT} that is not obviously in NP or coNP.

Example. Define $\text{LexLeastSATAss}(\varphi)$ to be the lexicographically smallest satisfying assignment of a Boolean formula φ .

This language lies in P^{SAT} . We determine the assignment as follows:

- Set $x_n = 1$ and query whether φ is satisfiable.
- If yes, set $x_n = 0$ and query again. If the oracle answers yes, keep $x_n = 0$; otherwise, set $x_n = 1$.
- Continue this procedure for $x_{n-1}, x_{n-2}, \dots, x_1$.

Definition. We say that $A \leq_{\text{poly}}^{\text{Tur}} B$ if $A \in P^B$.

Hierarchy with oracles. Oracle machines admit the same hierarchy arguments, since we can define universal oracle Turing machines. Consequently, for any oracle B ,

$$\text{DTIME}^B(n^3) \subsetneq \text{DTIME}^B(n^5) \quad \text{and} \quad \text{NTIME}^B(n^3) \subsetneq \text{NTIME}^B(n^5).$$

Lecture 8

Baker-Gill-Solovay and Mahoney's Theorems

Scribe: Aindrila Rakshit

Topics covered in this lecture

- Oracle Turing Machines
- Baker-Gill-Solovay Theorem
- Conditional Implications of $P = NP$

8.1 Oracle Turing Machines

An *oracle Turing machine* is a Turing machine equipped with:

- an input tape
- bunch of work tapes
- an additional oracle tape,
- three special states: $q_{\text{query}}, q_{\text{yes}}, q_{\text{no}}$.

When the machine enters q_{query} , the string currently written on the oracle tape is checked for membership in a fixed language $A \subseteq \Sigma^*$:

- If the string is in A , the machine transitions to q_{yes} .
- Otherwise, it transitions to q_{no} .

When someone gives an oracle TM, they specify the entire transition function except for one: $\Gamma(q_{\text{query}}, \dots)$, since it's unspecified whether on a given query ones move to yes state or a no state.

For a given machine M , after fixing an oracle $\mathcal{O} \subseteq \Sigma^*$, the entire transition function is specified, so now we can talk about $L(M^{\mathcal{O}}) = \{x : M^{\mathcal{O}} \text{ on } x \text{ accepts}\}$.

Oracle TM helps us understand Turing Reductions, which says given a subroutine for \mathcal{O} , we can solve the above language. It also provides guardrail for some intuition one may develop for Time Hierarchy Theorems etc., since many of these results just uses UTM simulation which will just simulate as if it's given access to the same oracle.

Observation 8.1. $UTM^A(\langle M^{\square} \rangle, x)$ which is a UTM with oracle access to A with code of an oracle machine M (A is not given, i.e. transition function is not specified) and input x simulates M^A on x with the same efficiency as $UTM(\langle M \rangle, x)$ simulates M on x .

Proof. When transitioning from state $q \neq q_{\text{query}}$, i.e. non oracle state, then do the simulation as if $UTM(\langle M \rangle, x)$ simulates M on x . If $q = q_{\text{query}}$, then UTM^A makes the same query. So the extra overload is in copying the oracle query on the oracle tape and moving the states accordingly. \square

Corollary 8.2. For any language $A \subseteq \Sigma^*$:

$$DTIME^A(f) \subsetneq DTIME^A(g), \quad NTIME^A(f) \subsetneq NTIME^A(g).$$

if f, g are time constructible and $f \ll g$. i.e. Time Hierarchy Theorems relativise.

Proof. We can construct a universal oracle Turing machine UTM^A that simulates M^A while making the same oracle queries. Diagonalisation arguments go through unchanged because they do not depend on the internal structure of the oracle. \square

Techniques such as diagonalization don't care about the presence of oracles. This phenomenon of "relativization" is such that a statement holds in any world where there are oracle access that is free.

8.2 The Baker-Gill-Solovay Theorem

Theorem 8.3 (Baker-Gill-Solovay). *There exist languages $A, B \subseteq \Sigma^*$ such that:*

$$P^A = NP^A \quad \text{and} \quad P^B \neq NP^B.$$

Consequence: Any proof technique that *relativises* (i.e., continues to hold in the presence of an arbitrary oracle) cannot resolve the P vs. NP question.

In particular, diagonalisation alone cannot prove either $P = NP$ or $P \neq NP$.

Proof. **Language A such that $P^A = NP^A$:** We want a language A such that $P^A = NP^A$ so we construct such a language by letting

$$A = \{(\langle M \rangle, x, 1^t) : M \text{ accepts } x \text{ in } 2^t \text{ time}\}.$$

Clearly, $A \in \text{EXP}$ since the UTM will just simulate M for 2^t steps and then decide whatever it's going to do in that time bound. Also, this language is EXP-complete by construction.

$P^A = \{L(M^A) : M \text{ is a deterministic poly time machine}\}$

Note that $P^A \subseteq \text{EXP}$ — we can run polynomially many queries in exponential time, and thus the overall running time is bounded by exponential.

Also, $\text{EXP} \subseteq P^A$ — using the oracle A , we can simulate any exponential-time computation in one query. Hence: $P^A = \text{EXP}$.

Infact, $\text{NP}^A = \text{EXP}$ — we build an deterministic machine that explores every path of the given NDTM and whenever an oracle query is made, run an exponential time machine to answer the query. Since the length of any path is $\leq n^c$, the total number of paths is $\leq 2^{n^c}$, the size of the query is n^c and the time take to solve A is 2^{n^d} , the total time taken is $2^{n^c} 2^{n^d}$ for some constants c, d .

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

Language B such that $P^B \neq \text{NP}^B$:

We now construct an oracle $B \subseteq \{0,1\}^*$ such that $P^B \neq \text{NP}^B$.

Clearly, $P^B \subseteq \text{NP}^B$.

Note: If class $A \subseteq$ class B , then its not necessary that for oracle $\mathcal{O}, A^{\mathcal{O}} \subseteq B^{\mathcal{O}}$, since the machines are different. We will see concrete examples when space complexity is introduced.

For any oracle B , define

$$L_B := \{1^n \mid \exists y \in B \text{ with } |y| = n\}.$$

In words, $1^n \in L_B$ iff the oracle contains at least one string of length n .

Claim 8.4. For every oracle B , $L_B \in \text{NP}^B$.

Proof. On input 1^n , a nondeterministic machine guesses a string $y \in \{0,1\}^n$ and queries whether $y \in B$. It accepts iff the oracle answers “yes”. This runs in polynomial time with one oracle query. \square

We now construct B so that $L_B \notin P^B$.

Claim 8.5. We can choose $B \subseteq \Sigma^*$, such that $L_B \notin P^B$.

Proof. (Showing limits of diagonalization via diagonalization.) Let M_1, M_2, M_3, \dots be an enumeration of all padded deterministic polynomial-time oracle Turing machines.

We will ensure that M_i^B will fail on input 1^{n_i} , for each $i \in \mathbb{N}$.

We build B in phases.

Phase i : Choose a fresh integer n_i satisfying:

- n_i is larger than all previously chosen lengths,
- n_i is sufficiently large relative to the running time bound of M_i .

Now simulate M_i^B on input 1^{n_i} .

Whenever M_i makes a query to a string whose membership in B has not yet been fixed, answer “no”.

Since M_i runs in polynomial time, it makes at most $n_i^{k_i}$ oracle queries for some constant k_i . Hence it queries only polynomially many strings.

However, $|\{0,1\}^{n_i}| = 2^{n_i}$, so exponentially many strings of length n_i remain unqueried. After the simulation halts, we act as follows.

Case 1: M_i accepts 1^{n_i} . Then M_i believes that there exists some string of length n_i in B .

To make it wrong, we ensure that *no* string of length n_i is placed into B .

Thus, $1^{n_i} \notin L_B$ but M_i accepts.

Case 2: M_i rejects 1^{n_i} . Then M_i believes that no string of length n_i is in B .

Since only polynomially many strings were queried, choose some string $y \in \{0,1\}^{n_i}$ that was not queried and insert it into B .

Then $1^{n_i} \in L_B$ but M_i rejects.

Each stage uses a fresh length n_i . Hence later stages never alter decisions made at earlier lengths, and no inconsistencies arise. For every i , the machine M_i^B fails on input 1^{n_i} . Therefore no deterministic polynomial-time oracle machine decides L_B . Hence, $L_B \notin P^B$. \square

Since $L_B \in NP^B$, we conclude: $P^B \neq NP^B$. \square

8.3 Conditional Implications of $P = NP$

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

Theorem 8.6. *If there exists a unary NP-complete language, then $P = NP$.*

Proof. Suppose $L \subseteq \{1\}^*$ is NP-complete.

Then there exists a deterministic polynomial-time many-one reduction

$$\varphi : \Sigma^* \rightarrow \{1\}^*$$

from CIRCUIT-SAT to L such that $C \in \text{CIRCUIT-SAT} \iff \varphi(C) \in L$.

Since L is unary, for every circuit C , $\varphi(C) = 1^{m(C)}$ for some integer $m(C)$ (without loss of generality; if the string is not of the form 1^m then we know $C \notin \text{CIRCUIT-SAT}$). Because φ runs in polynomial time, there exists a constant c such that $m(C) \leq |C|^c$. Hence, for circuits of size n , the reduction can produce at most n^c distinct outputs.

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

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

We can do two operations in \mathcal{B} : EXPAND or PRUNE.

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

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

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

So PRUNE(\mathcal{B}) is applied when size of \mathcal{B} is bigger, $|\mathcal{B}| > n^2 + 1$. Now let $\mathcal{B} = \{C_1, \dots, C_t\}$. Let $y_i = \varphi(C_i)$. So if any of the y_i is not of the form 1^k then PRUNE throws them away. If $y_i = y_j$ for $i \neq j$ then either both are satisfiable or unsatisfiable. So we can just discard one of them.

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

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

Algorithm 1: SAT-Generic-Algo

Input: n -variate boolean formula φ

Output: Is C satisfiable.

```

1 begin
2    $\mathcal{B} \leftarrow \{C\}$ 
3   for  $i = 1, \dots, n$  do
4      $\mathcal{B} \leftarrow \text{EXPAND}(\mathcal{B})$ 
5     if  $|\mathcal{B}| > n^2 + 1$  then
6       PRUNE( $\mathcal{B}$ )
7   if any element of  $\mathcal{B}$  is true then
8     return True
9   return False

```

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

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

Definition 8.7. A language $L \subseteq \{0, 1\}^*$ is called n^c -sparse if there exists a constant c such that

$$|L \cap \{0, 1\}^n| \leq n^c$$

for all sufficiently large n . ◇

So sparse means: At most polynomially many strings per length.

Unary means: Exactly one string per length. In particular, unary languages are 1-sparse.

Definition 8.8. A language $L \subseteq \{0, 1\}^*$ is called co- n^c -sparse if \bar{L} is n^c -sparse. ◇

Theorem 8.9 (Fortune's theorem (i.e. Mahaney's theorem for co-sparse languages)). *If there are co- n^c -sparse NP-complete languages, then $P = NP$.*

Proof. Suppose L is co- n^3 -sparse and NP-complete.

Then there exists a deterministic polynomial-time many-one reduction $\varphi : \Sigma^* \rightarrow \{0, 1\}^*$ from CIRCUIT-SAT to L such that $C \in \text{CIRCUIT-SAT} \iff \varphi(C) \in L$.

Suppose \mathcal{B} be the bag of smaller formulas. And we define the EXPAND operation like in the unary case. The PRUNE operation needs some modification since we can't throw away C if $\varphi(C)$ is not of the form 1^k . But if we do find a collision, i.e. $\varphi(C_i) = \varphi(C_j)$, we can safely throw one of them away.

So suppose $\varphi(C_i)$'s are all distinct for $C_i \in \mathcal{B}$ with $|\mathcal{B}| > n^6$. Then since L is $\text{co-}n^3$ -sparse, there are at most n^6 strings of all possible n^2 length strings in \bar{L} . So if the original formula was unsatisfiable then the bag \mathcal{B} will only be filled with unsatisfiable formulas. Thus, if the size of the bag was larger than n^6 and there is no collision, then we can safely report that the original formula is satisfiable. \square

Lecture 9

Polynomial Hierarchy

Scribe: Ananya Ranade

Topics covered in this lecture

- Polynomial Hierarchy
- Σ_i, Π_i complete languages
- PH-complete languages

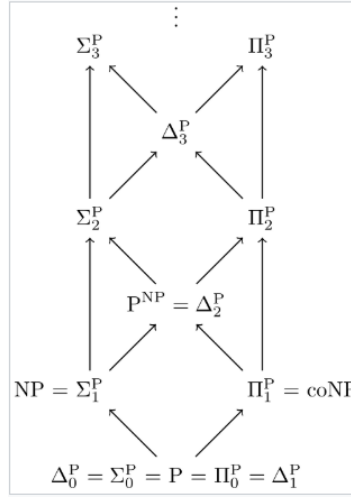
Abuse of notation : In an oracle TM, with some NP-complete oracle, it does not matter which NP-complete language is given as the oracle, since we can reduce an instance of query to one oracle to an instance of query to the other oracle in poly time. In other words $C^{\text{NP}} = C^L$ for any NP-complete language L , and complexity class C . Thus, we will now denote complexity classes by P^{NP} (poly time DTM decidable languages when NP-complete language is given as oracle), NP^{NP} (poly time NTM decidable languages when NP-complete language is given as oracle), etc.

In today's class we tried to understand the classes P^{NP} and NP^{NP} . The following are examples of languages in P^{NP} , NP^{NP} :

- $\text{Exact-VC} = \{(G, k) : \text{smallest VC is of size } k\} \in P^{\text{NP}}$.
- $\overline{\text{MinCircuit}} = \{\varphi : \varphi \text{ has a smaller equivalent circuit}\} \in NP^{\text{NP}}$.

9.1 Polynomial Hierarchy

If $P \subsetneq NP$, then NP is a strictly stronger class than P . Now, what if NP time TM was given NP (e.g. SAT) as an oracle? Can we do better? Seems like we can! Because now membership in the language of all "Tautologies", that is formulas that always evaluate to True, can be decided by a NTM machine with SAT oracle in polytime. It just queries if the complement of the given formula is satisfiable. The oracle is able to decide for both yes and no instances (without oracle



NTM machine can only certify yes instances in poly time), and thus gives more power to the machine.

Thus, using P , NP , $coNP$ and the subsequently derived classes as oracles, we can build an infinite hierarchy of complexity classes, called the polynomial hierarchy. The diagram shows the various complexity classes and their containment is shown by arrows. The classes are P , $\Sigma_1 = NP$, $\Pi_1 = coNP$, $\Delta_1 = P$ and recursively $\Sigma_{i+1} = NP^{\Sigma_i}$ (poly time NTM decidable languages when Σ_i -complete language is given as oracle), $\Delta_{i+1} = P^{\Delta_i}$ (poly time DTM decidable languages when Δ_i -complete language is given as oracle), $\Pi_{i+1} = coNP^{\Sigma_i}$ (poly time NTM decidable complement languages when Σ_i -complete language is given as oracle). The complexity class PH (polynomial hierarchy) is defined as $\bigcup_{i \geq 1} \Sigma_i = \bigcup_{i \geq 1} \Pi_i$.

9.2 Complete languages for Σ_i, Π_i

Now, let us try to understand how languages in each of these classes look like and if they even have complete languages. Let $\Sigma_i\text{-SAT} = \{ \text{"}\exists x_1 \forall x_2 \dots Q_i x_i : \varphi(x_1, \dots, x_i)\text{"} : \text{that are true} \}$. Similarly let $\Pi_i\text{-SAT} = \{ \text{"}\forall x_1 \exists x_2 \dots Q_i x_i : \varphi(x_1, \dots, x_i)\text{"} : \text{that are true} \}$.

The following are some simple observations.

- $\Sigma_i\text{-SAT} \in \Sigma_i$: The $NP^{\Sigma_{i-1}}$ machine on path $(x_1 = a)$ asks Σ_{i-1} oracle if $\neg \varphi(a, x_2, \dots, x_i) \in \Sigma_{i-1}\text{-SAT}$. If yes, reject. Else, accept.
- $co \Sigma_i = \{ \bar{L} : L \in \Sigma_i \} = \Pi_i$: Follows from definition.

All these things make sense if $\Sigma_i\text{-SAT}$ (or rather some language) is complete for Σ_i (and similar for Π_i). Let us focus on $i = 2$. The proof can be generalized for all i . $\Sigma_2\text{-SAT}$ looks like NP^{SAT} with 1 query. Thus, if we define $NP^{NP[1]}$ as the class where we can make at most 1 oracle query, $\Sigma_2\text{-SAT}$ is complete for that class. But, in general we can make poly many queries! As we will see, 1 query is equally strong as poly many queries.

Theorem 9.1. $\Sigma_2 = NP^{NP} = NP^{NP[1]}$

Proof. A path in any NP^{SAT} computation consists of some choices made non-deterministically with a few oracle queries asked in between. However, we can push the oracle queries down.

Suppose at some point we wanted to query the oracle. We just guess what the oracle would have answered and proceed assuming the answer is correct. At the end, if a path is rejected with these assumptions, we reject it. Else, we need to verify if these assumptions were correct. Wherever we thought the answer was supposed to be 'yes', we extend the path by guessing more and trying to find the satisfying assignment. There will be some path along the extension which succeeds if it was indeed a yes instance. For all queries where we guessed "no", (say the queries were ψ_1, \dots, ψ_b), we ask the SAT oracle if $\psi_1 \vee \psi_2 \vee \dots \vee \psi_b \in SAT$. If no, we accept the path. Else, we reject. Thus, we can do the computation with slight overhead (at most polynomially many queries were pushed), but with just 1 query. Thus, $NP^{NP} = NP^{NP[1]}$ and hence, Σ_2 -SAT is complete for Σ_2 . \square

Similar arguments can be used for showing Σ_i -SAT, Π_i -SAT are complete in Σ_i, Π_i respectively.

A natural question to ask is, can either of these classes be equal?

Theorem 9.2. *If $\Sigma_i = \Pi_i$, then $PH = \Sigma_i = \Pi_i$.*

Proof. We will do it for the case $i = 2$. Similar argument works for other i 's as well. Suppose $\Sigma_2 = \Pi_2$. We will show Σ_3 -SAT $\in \Sigma_2$, which is enough to show that PH collapses to the i^{th} level by induction.

Σ_3 -SAT = $\{\text{"}\exists x \forall y \exists z : \varphi(x, y, z)\text{"} : \text{that are true}\}$. Fix $x = a$. Define $L_a = \{\varphi(a, y, z) : \forall y \exists z : \varphi(a, y, z)\}$. This is in Π_2 , which by assumption implies is in Σ_2 . Thus, L_a reduces to some language of the form $\{\exists u \forall v : \Gamma_a(u, v)\}$.

Thus, $\exists x \forall y \exists z : \varphi(x, y, z) \iff \exists x (\exists u \forall v \Gamma_x(u, v))$. The new formula $\Gamma_x(u, v)$ is still poly in length. But now, with this representation, the language is in Σ_2 . Thus, Σ_3 -SAT $\in \Sigma_2$. \square

9.3 PH-complete languages?

Consider the language of true quantified boolean formulas (TQBF) :

$$TQBF := \{\text{"}Q_1x_1Q_2x_2 \dots Q_mx_m\varphi(x_1 \dots x_m)\text{"} : \text{that are true}\}$$

Clearly, TQBF is PH-hard as if any language $L \in PH$, then it is in Σ_i form for some i , and thus can be reduced to Σ_i -SAT which is a special case of a TQBF instance.

Can TQBF be PH-complete?

Suppose it was in PH. Then, it is in Σ_i for some i . But since any language in Σ_{i+1} is a special case of TQBF, it is also in Σ_i . Thus, $\Sigma_i = \Sigma_{i+1}$ and by the above theorem this implies that PH collapses to the i^{th} level.

In fact, the same argument shows that there is no PH-complete languages, unless PH collapses.

Lecture 10

Karp-Lipton-Sipser theorem

Scribe: Chandralekha P

Topics covered in this lecture

- Different approach on oracle based TM
- Karp-Lipton-Sipser theorem
- Meyer's theorem

10.1 Recap

In the previous lecture, we saw how oracle TMs work. We were introduced to Polynomial hierarchy, containments and complete problems for Σ_i and Π_i , which are Σ_i -SAT and Π_i -SAT resp.. We also saw when PH collapses and since we believe that PH would not collapse, we get potential statements that are untrue.

In this lecture, we will see a slightly different approach on the TM construction we saw for Σ_2 completeness of Σ_2 -SAT. We will also see the Karp-Lipton-Sipser theorem which says that if the class NP is such that they have poly size circuits, then the PH collapses. We will also see Meyer's theorem which says that if $\text{EXP} \subseteq \text{size}(\text{poly})$, then $\text{EXP} = \Sigma_2 \cap \Pi_2$.

10.2 NP^{SAT} TM

In last class, we saw that whatever query the machine wants to make to the oracle can be deferred to the end into a single query because the machine can guess the output of the oracle and later verify if the assumption is true.

Now, suppose the run is such that, if the oracle says yes to the query, we accept it. Then, the machine could continue and guess the potential assignment of the formula, and then accept. So this is like, we accept if there is one subpath that accepts (subpath starts from the query node with all possible assignments). Hence, this is behaving like NP machine

Instead, if the path was such that it would accept if the oracle says no to the query, then we accept it. Then, we want the machine to go through all possible assignments of this formula and make sure none of these satisfy. So, this is behaving like a coNP machine.

Hence, we want some kind of machine that can allow both these types of nodes. This is called an alternating TM where some states are existential states, and others are universal states. The existential state checks if there is some accepting path, and the universal state checks if all the paths are accepting.

Hence, this machine M on an input x and path $(p_{\exists}, p_{\forall})$ (existential and universal nodes in the path) is a deterministic computation. And M accepts x iff $\exists p_{\exists} \forall p_{\forall} M$ accepts x on p_{\exists}, p_{\forall} . Hence, this is a reduction from L to Σ_2 -SAT.

10.3 Karp-Lipton-Sipser theorem

We believe that $P \neq NP$. But what if $SAT \in size(poly)$? We will show that this will cause PH collapse, and hence unlikely.

Theorem 10.1 (Karp-Lipton-Sipser). *Suppose $NP \subseteq size(poly)$. Then, $PH = \Sigma_2 \cap \Pi_2$*

Proof-sketch. If we show that $\Pi_2 - SAT \in \Sigma_2$ then we are done. Suppose we have $\forall x \exists y : \varphi(x, y)$ (which is like a satisfiability problem for each x) then we want to reduce it to a Σ_2 -SAT statement. We are given that there is a small circuit C solving SAT. But, we cannot simply check $\exists C \forall x : C(\varphi(x, _)) = 1$ because, the trivial circuit satisfies this but it is clearly not solving SAT. Therefore, we should not blindly trust C to solve SAT correctly. Hence, given a circuit C that is claiming that $\varphi(x, _)$ is satisfiable, we find the satisfying assignment for $\varphi(x, _)$ using a sub-routine $FindSatAsst(C, \varphi(x, _))$ which literally goes and finds the assignment one variable at a time by asking C if it is satisfiable.

Now, we see the formula $\exists C \forall x : \varphi(x, FindSatAsst(C, \varphi(x, _)))$. Now, if our original formula is true, then by taking the correct circuit C , this new formula gives us true. And if the original one is false, then regardless of what our algorithm gives, it cannot satisfy the formula, and hence it has to return false. Hence, we have reduced it to a Σ_2 formula which is of poly size of original formula.

By this, we get that $\Pi_2 \subseteq \Sigma_2$ and hence $\Sigma_2 = \Pi_2$ □

10.4 Meyer theorem

Now, we will see what happens if all EXP problems have poly size circuits.

Theorem 10.2 (Meyer). *If $EXP \subseteq size(poly)$, then $EXP = \Sigma_2 \cap \Pi_2$.*

Proof-sketch. Consider a language $L \in EXP$. We want to give a Σ_2 -SAT formula for the same. Given the TM M for L . Now, on an input x , we consider the computational tableau of it, and to compute one particular cell of this tableau, it takes exponential time. Hence, we have a circuit family that does this. Now, using this we can check if the tableau is running correctly on this input x , and if it finally reached some accepting state. Hence, the formula would need to check

if there is a circuit C such that all the checks are passed and if it reaches an accepting state. This gives us a Σ_2 -SAT statement and hence, we get $EXP \subseteq \Sigma_2$ and since EXP is deterministic, we are done. \square

10.5 Summary

We have now seen about alternating TM and its behavior. We have also seen the relation between polynomial hierarchy and circuit size of different classes.

Lecture 11

Introduction to Space Complexity

Scribe: Hrishikesh Saikia

Topics covered in this lecture

- Basic definitions and the model of space-bounded computation
- Examples of problems in NL and PSPACE
- Space Hierarchy Theorem
- Configuration graphs and class containments
- Savitch's Theorem

In this lecture, we initiate the study of complexity classes defined in terms of the space used by Turing Machines.

11.1 Setup and Definitions

We will still be working with Turing Machines as our model of computation. For a start, we would like to define what it means for a deterministic machine to use a certain amount of "space". A natural attempt would be to consider the number of tape cells used. To see whether this would be a good choice, consider the following language:

$$L = \{x \in \{0,1\}^* : x \text{ has an even number of 1's}\}$$

To decide L , we can design a machine M which simply uses a single work tape cell to maintain a bit b denoting the parity of the number of 1's encountered so far: the machine will start with $b = 1$, and it will scan the input from left to right and every time it sees a 1, it will flip b . It reaches an accepting or rejecting state depending on the value of b (1 or 0 respectively) at the end of the scan. It is easy to see that $L(M) = L$. As per our attempted definition, the space

used by M is equal to $n + 1$ for an n length input. But we really had to use only one additional cell.

The input to any problem will always be given in the input tape. So under our definition, all languages that use sublinear additional space (like L) will all run in linear space. Intuitively, we do not want to categorize these problems under the same complexity class. So, it makes more sense to define the space used as the number of work tape cells accessed. But we also do not want to allow the machine to do its entire computation in the input tape itself, treating it as a work tape. The easy fix is to let the input tape be a read-only tape. This is the setup we will consider.

Given a deterministic Turing Machine M and an input x , the space used by M on x , denoted by $\text{Space}_{M,x}$, is equal to the number of work tape cells accessed by M while running on x . Note that M might not halt on x , but it is possible that M keeps looping withing the same bounded number of work tape cells. So $\text{Space}_{M,x}$ can be a finite number even if M does not halt on x .

For a deterministic TM M , we define the function $\text{Space}_M : \mathbb{N} \rightarrow \mathbb{N}$ as: for any $n \in \mathbb{N}$,

$$\text{Space}_M(n) = \max_{x \in \Sigma^n} (\text{Space}_{M,x})$$

that is, $\text{Space}_M(n)$ is the maximum space used by M on inputs of length n . We will assume that $\text{Space}_M(n) \geq \log n$, to allow the machine to be able to store pointers.

What about not-determinism?

We will define the space used by a non-deterministic machine N on an input x , denoted by $\text{NSpace}_{N,x}$, as the the maximum number of work tape cells used on a single computational path, over all computational paths. Then $\text{NSpace}_N(n)$ for $n \in \mathbb{N}$, is the maximum of $\text{NSpace}_{N,x}$ over all x of length n . We define $\text{coNSpace}_{N,x}$ and coNSpace_N similarly for co-non-deterministic machines.

Complexity Classes

For any $s : \mathbb{N} \rightarrow \mathbb{N}$, we define the following classes:

1. $\text{DSPACE}(s) = \{L \subseteq \{0,1\}^* : L \text{ can be decided by a deterministic TM } M \text{ with } \text{Space}_M = O(s)\}$
2. $\text{NSPACE}(s) = \{L \subseteq \{0,1\}^* : L \text{ can be decided by an NDTM } M \text{ with } \text{NSpace}_M = O(s)\}$
3. $\text{coNSPACE}(s) = \{L \subseteq \{0,1\}^* : L \text{ can be decided by an coNDTM } M \text{ with } \text{coNSpace}_M = O(s)\}$
4. $L = \text{DSPACE}(\log n)$
5. $NL = \text{NSPACE}(\log n)$
6. $\text{PSPACE} = \cup_{c \geq 1} \text{DSPACE}(n^c)$
7. $\text{NPSPACE} = \cup_{c \geq 1} \text{NSPACE}(n^c)$

11.2 Two example problems

Consider the following language

$$\text{DIR-ST-CONN} = \{(\langle G \rangle, s, t) : \text{there is an } s \rightarrow t \text{ path in } G\}$$

consisting of directed graphs G with vertices s and t such that there is a path from s to t in G . We will show that this language is in NL. Note that our familiar DFS or BFS does not directly work, as they require us to mark the visited vertices and we can have $n - 1$ (n is the number of vertices of G) marked vertices in the worst case. So we have to come up with a different algorithm. And the key observation is that unlike time, space can be reused. So we can be time inefficient and still solve a problem in very small space (by reusing the same space over and over again). We will exploit this idea in the following algorithm, and in general, throughout space complexity.

Claim 11.1. $\text{DIR-ST-CONN} \in \text{NL}$

Proof. Consider the following algorithm:

1. Set $curr = s$ and $i = 0$
2. While $i < n$:
 1. If $curr = t$, accept.
 2. Set $curr \leftarrow$ a non-deterministic neighbour of $curr$
 3. Set $i \leftarrow i + 1$
3. Reject

The above algorithm basically starts a non-deterministic walk from s and continues for n steps. If there indeed is an $s \rightarrow t$ path, then there must be an accepting sequence of non-deterministic guesses that reach t within n steps as the length of any path in the graph will be less than n .

Note that we are reusing the same space to store all the $curr$ one after another. At any point, we need $\log n$ cells to store the $curr$, another $\log n$ cells to store the non-deterministically guessed new $curr$, and another $\log n$ cells to store i . So the total space used is $O(\log n)$. \square

For the second example, consider the problem

$$\text{TQBF} = \{Q_1x_1 \dots Q_nx_n : \varphi(x_1, \dots, x_n) : Q_1x_1 \dots Q_nx_n : \varphi(x_1, \dots, x_n) \text{ is true}\}$$

Claim 11.2. $\text{TQBF} \in \text{PSPACE}$

Proof. The idea is to use recursion, wherein we use the same space for the recursive calls. Let

$$\psi = Q_1x_1 \dots Q_nx_n : \varphi(x_1, \dots, x_n)$$

Consider the following recursive algorithm:

TQBF-Check(ψ):

1. If $n = 0$, return $(\psi \equiv T)$
2. Set $\tilde{\psi} = Q_2x_2 \dots Q_nx_n : \varphi(0, x_2, \dots, x_n)$
3. Compute $b_0 = \text{TQBF-Check}(\tilde{\psi})$
4. Set $\tilde{\psi} = Q_2x_2 \dots Q_nx_n : \varphi(1, x_2, \dots, x_n)$
5. Compute $b_1 = \text{TQBF-Check}(\tilde{\psi})$
6. If $Q_1 = \exists$, return $b_0 \vee b_1$. Else, return $b_0 \wedge b_1$.

The correctness follows easily by induction (for the base case, note that ψ just consists of constants). Note that we will reuse the same space to compute b_0 and b_1 . And we need an additional $O(n + m)$ space to write down $\tilde{\psi}$ at the start of each recursive step (m is the size of φ). So

$$\text{Space}(n) \leq \text{Space}(n - 1) + O(n + m) \implies \text{Space}(n) = \text{poly}(n, m) \quad \square$$

11.3 Space Hierarchy Theorem

In case of space complexity, we have a universal Turing Machine \mathcal{U} that can simulate any other Turing Machine M on any input x with only a constant factor overhead in space (the idea is that \mathcal{U} will do whatever M does on x step by step). This allows us to mimic the proof of Theorem 4.1 by putting a cap on space instead of time to get a hierarchy theorem for space-bounded computation. There is a small catch here: it is possible that a machine with a space cap might run forever. To address this, we put a cap on the runtime as well (exponential in the space bound; see the section on configuration graphs).

Theorem 11.3. *For space constructible functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ satisfying $f(n) = o(g(n))$, we have*

$$\text{DSPACE}(f(n)) \subsetneq \text{DSPACE}(g(n))$$

Proof. Exercise! Also, come up with a non-deterministic space hierarchy theorem. \square

11.4 Configuration Graph and class containments

We introduce the notion of configuration graphs. Given a machine M (deterministic or non-deterministic) and input x , the configuration graph $G_{M,x}$ is a directed graph whose nodes correspond to configurations of M on x , and there is an edge from a configuration C_1 to a configuration C_2 iff C_2 is reachable from C_1 in one computational step of M on x . We further have three special nodes: C_{start} , corresponding to the start configuration, C_{accept} , corresponding to an accepting configuration of M on x , and C_{reject} , corresponding to a rejecting configuration of M on x . So, M accepts x iff there is a path from C_{start} to C_{accept} . Here, the configuration of M on x consists of the tape contents, the state, and head positions. So if M is bounded by space s , then the configuration can be specified by an $O(s)$ length binary string, thus giving us $2^{O(s)}$ many nodes in $G_{M,x}$.

Note that in case of a deterministic machine M , all nodes in $G_{M,x}$ have outdegree atmost 1, while in case of a non-deterministic machine, each node has outdegree atmost 2.

We now discuss a few class containments.

Observation 11.4. $\text{DTIME}(f) \subseteq \text{DSPACE}(f)$

Proof. Let $L \in \text{DTIME}(f)$. So there is a deterministic TM M deciding L in $O(f)$ time, or $O(f)$ computational steps. Since each computational step corresponds to one application of the transition function, so in each step, each of the pointers can move atmost one cell to the left or to the right. So in $O(f)$ steps, the pointers can move atmost $O(f)$ cells from the left in each work tape. Since there are constantly many work tapes, so M runs in $O(f)$ space. Thus $L \in \text{DSPACE}(f)$. \square

Observation 11.5. $\text{DSPACE}(s) \subseteq \text{DTIME}(2^{O(s)})$

Proof. Let $L \in \text{DSPACE}(s)$. So there exists a deterministic TM M deciding L and running in $O(s)$ space. Consider the following $2^{O(s)}$ -time machine for L : On input x , construct $G_{M,x}$ in $2^{O(s(|x|))}$ time. Now, run a DFS/BFS from C_{start} to check if there is a path to C_{accept} . If yes, accept. Else, reject. By the definition of configuration graphs, this machine decides L and hence $L \in \text{DTIME}(2^{O(s)})$. Since L was arbitrary, so $\text{DSPACE}(s) \subseteq \text{DTIME}(2^{O(s)})$. \square

Observation 11.6. $\text{NSPACE}(s) \subseteq \text{DTIME}(2^{O(s)})$

Proof. Same as the proof of Observation 11.5 \square

Observation 11.7. $\text{NP} \subseteq \text{PSPACE}$

Proof. Let $L \in \text{NP}$. So there exists a polytime TM M and a polynomial $p(\cdot)$ such that for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ such that } M(x,u) = 1$$

Consider the following PSPACE machine for L : On input x , it will go over all possible $u \in \{0,1\}^{p(|x|)}$ and check if $M(x,u) = 1$. If yes, accept. Else, reject. The idea here is that the same space will be reused for each certificate $u \in \{0,1\}^{p(|x|)}$ computation $M(x,u)$. \square

Observation 11.6 implies that $\text{NL} \subseteq \text{P}$ and $\text{NPSPACE} \subseteq \text{EXP}$. Combining everything, we get the following chain:

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE} \subseteq \text{EXP} \subseteq \text{NEXP}$$

11.5 Savitch's Theorem

Savitch's Theorem says that any language that can be solved by a $\text{NSPACE}(s)$ machine can be solved by a $\text{DSPACE}(s^2)$ machine. An immediate corollary of this is that $\text{PSPACE} = \text{NPSPACE}$, as they are both unions of $\text{DSPACE}(n^c)$'s and $\text{NSPACE}(n^c)$'s respectively. We will prove the general version in the next lecture. Here, we will show it for the DIR-ST-CONN language.

Theorem 11.8. $\text{DIR-ST-CONN} \in \text{DSPACE}(\log^2 n)$

Proof. Consider an input (G, s, t) . The $\text{DSPACE}(\log^2 n)$ algorithm builds on this simple observation: if there is a path of length $\leq n$ from s to t , then there must exist a “middle” vertex u such that there is a path of length $\leq n/2$ from s to u , and a path of length $\leq n/2$ from u to t . We build the following recursive algorithms $\text{Path}(u, v, k)$, which accepts iff there is a path from u to v of length $\leq k$:

1. If $t = 1$, check if there is an edge from u to v , and output accordingly.
2. For $w \in G$:
 1. Compute $b_1 = \text{Path}(u, w, k/2)$
 2. Compute $b_2 = \text{Path}(w, v, k/2)$
 3. If $b_1 \wedge b_2$, accept.
3. Reject

The correctness follows easily by induction. Note that we are reusing the same space for b_1 and b_2 . So if we let $f(k)$ denote the space complexity of $\text{Path}(u, v, k)$, we get $f(k) \leq O(\log n) + f(k/2) \implies f(k) = O(k \log n)$. The $O(\log n)$ comes from the space used to store w before each recursive call. To decide DIR-ST-CONN , we need to call $\text{Path}(s, t, n)$, which takes $O(\log^2 n)$ space. Thus $\text{DIR-ST-CONN} \in \text{DSPACE}(\log^2 n)$, as desired. \square

Lecture 12

PSPACE Completeness

Scribe: Krishnashree J B

Topics covered in this lecture

- Savitch's Theorem
- PSPACE-complete problems
- TQBF and Generalised Geography

12.1 Savitch's Theorem

Theorem 12.1 (Savitch). *For every space-constructible function $s(n)$,*

$$\text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s(n)^2).$$

Proof. Let M be an $\text{NSPACE}(s(n))$ machine and input x . Let the configuration graph be $G_{M,x}$. Each configuration can be encoded using $O(s)$ bits ($s(n) = s$), so $|G_{M,x}| \leq 2^{O(s)}$.

M accepts x iff there is a path from C_{start} to C_{accept} . Define $\text{Path}(C_1, C_2, k)$ to check if there is a path of length at most k . Setting $k = 2^{O(s)}$ and querying $\text{Path}(C_1, C_2, k)$ says if M accepts or not.

Algorithm 2: $\text{PATH}(C_1, C_2, k)$

Input: Configurations C_1, C_2 , integer k

for each configuration C_{mid} **do**
 $b_1 \leftarrow \text{PATH}(C_1, C_{\text{mid}}, \lfloor k/2 \rfloor)$
 $b_2 \leftarrow \text{PATH}(C_{\text{mid}}, C_2, \lfloor k/2 \rfloor)$
 if $b_1 \wedge b_2$ **then accept**
else reject

Space Analysis:

$$S(k) \leq O(s) + S(k/2)$$

$$\Rightarrow S(k) = O(s \log k) = O(s^2).$$

Thus, $\text{NSPACE}(s) \subseteq \text{DSPACE}(s^2)$. □

Corollary 12.2. $\text{PSPACE} = \text{NPSPACE} = \text{coNPSPACE}$.

Proof. $\text{NPSPACE} = \bigcup_c \text{NSPACE}(n^c) \subseteq \bigcup_c \text{DSPACE}(n^{2c}) \subseteq \text{PSPACE}$. and the other direction is trivial. □

12.2 A PSPACE-Complete Problem

Consider: $L = \{(\langle M \rangle, x, 1^s) : M \text{ accepts } x \text{ using space } \leq s\}$

The language L belongs to PSPACE since we can simulate the machine M on input x using at most s space. Moreover, L is PSPACE-hard because any computation of a polynomial-space machine can be encoded as an instance of L . Thus, reduce any problem in PSPACE to L .

12.2.1 TQBF

Theorem 12.3. TQBF is PSPACE-complete (under polynomial time many-one reductions).

Idea. Given a TQBF has the form: $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \varphi(x_1, \dots, x_n)$, where φ is in CNF. We already know $\text{TQBF} \in \text{PSPACE}$. For hardness, we should reduce any PSPACE computation to a formula. Since any PSPACE computation can be encoded as a configuration graph $G_{M,x}$.

$$\text{Path}(C_1, C_2, k) \leftrightarrow \psi_k(C_1, C_2)$$

And ψ_k can be defined recursively as below.

$$\psi_k(C_1, C_2) = \exists C_{\text{mid}} \psi_{k/2}(C_1, C_{\text{mid}}) \wedge \psi_{k/2}(C_{\text{mid}}, C_2).$$

But this could give an exponential blowup in the size of ψ_k . So make use of the universal quantifier.

$$\psi_k := \exists C_{\text{mid}} \forall b \in \{0, 1\} \psi_{k/2}(\alpha_b, \beta_b)$$

where, $\alpha_b := \text{IfThenElse}(b, C_1, C_{\text{mid}})$ and $\beta_b := \text{IfThenElse}(b, C_{\text{mid}}, C_2)$. When $b = 0$, (C_1, C_{mid}) is checked and when $b = 1$, (C_{mid}, C_2) is checked.

$$|\psi_k| \leq |\psi_{k/2}| + O(s) = O(s \log k).$$

For $k = 2^{O(s)}$, we get $|\psi_k| = O(s^2)$. Thus the reduction is polynomial. □

12.3 Generalized Geography

Problem: Two players alternate moving a token along directed edges of a graph, and a vertex may not be revisited. The player who cannot move loses.

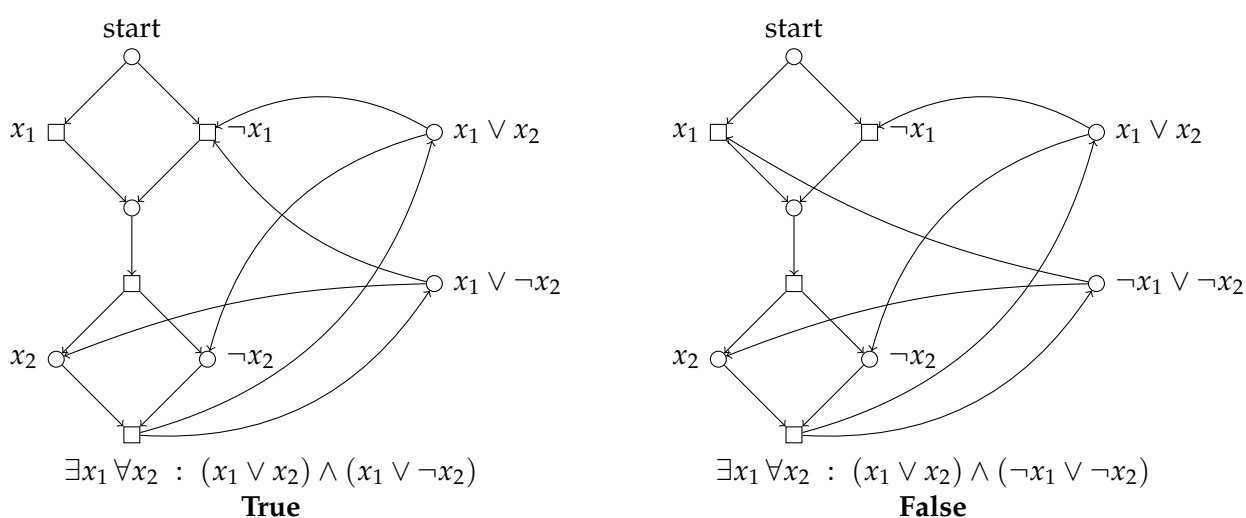
Given a graph G and a starting vertex s , does Player 1 have a winning strategy?

Theorem 12.4. *Generalised Geography is PSPACE-complete.*

Idea. Reduce from TQBF:

- Existential quantifiers correspond to Player 1 moves
- Universal quantifiers correspond to Player 2 moves
- Clauses are encoded as graph gadgets: For every move (to assign value to a variable) a player can make, a circle or square node is introduced depending on whether it is Player 1's or Player 2's turn. For every variable x , two vertices x and $\neg x$ are introduced. For every clause, a vertex c is introduced. Edges are added from each circle/square node to the corresponding x and $\neg x$ vertices. Edges from x and $\neg x$ back to the next circle/square node are added. From the last such node, edges to all vertices corresponding to clauses are added. For a vertex c corresponding to a clause, edges from c to the negations of the variables appearing in the clause are added.

The following is a pictorial representation of the reduction, where circle nodes correspond to Player 1's moves and square nodes to Player 2's moves. When the game begins, the players alternate and choose values for x_1 and x_2 . Then, the square player moves to a clause they believe is falsified by the assignment. The circle player can then move only to a literal node that was set to false — but such a node is already visited iff the assignment actually satisfied that literal, leaving the square player stuck. Thus Player 1 wins iff the formula is satisfiable.



□

Lecture 13

Oracle and Space machines

Scribe: Soham Ghosh

Topics covered in this lecture

- Savitch's Theorem does not relativise
- Logspace reductions
- NL-completeness

13.1 Space-bounded oracle machines

We first introduce the notion of *space-bounded oracle machines*. As before, we have a read-only input tape and space-bounded work tapes. In addition, there is an oracle tape.

However, there is an important nuance: to prevent "cheating" by using the oracle tape as extra workspace, we impose the following restriction.

- The oracle tape is *write-once and read-once*.
- Once a query is made, the contents of the oracle tape are erased.
- The machine immediately transitions to either q_{yes} or q_{no} depending on membership in the oracle language.

This ensures that the oracle tape cannot be reused as additional space.

Example. Let

$$A = \{\text{adjmatrix}(G) : G \text{ satisfies property } \mathcal{P}\},$$

where \mathcal{P} could be, for instance, having a clique of a certain size or containing a Hamiltonian cycle.

Consider

$$B = \{\text{adjlist}(G) : G \text{ satisfies } \mathcal{P}\}.$$

We can solve B using oracle access to A . Given an adjacency list, we can construct the adjacency matrix row-by-row on the oracle tape by scanning the input. This does not require storing the entire matrix on the work tape, and avoids quadratic space usage.

13.2 Savitch's Theorem does not relativise

Recall *Savitch's Theorem*:

$$\text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s(n)^2).$$

The proof proceeded by exploring the configuration graph. Let M be a nondeterministic machine and x an input. Let $G_{M,x}$ denote its configuration graph, with C_1 the start configuration and C_2 an accepting configuration.

We define a recursive procedure:

$$\text{Path}(C_1, C_2, k)$$

which checks if there is a path of length at most k from C_1 to C_2 .

- If $k = 1$, check if $C_1 \rightarrow C_2$ is a valid transition.
- Otherwise, for each intermediate configuration C_{mid} :

$$b_1 = \text{Path}(C_1, C_{\text{mid}}, k/2), \quad b_2 = \text{Path}(C_{\text{mid}}, C_2, k/2).$$

Accept if $b_1 \wedge b_2$.

The space usage satisfies:

$$\text{space}(k) \leq \log N + \text{space}(k/2),$$

where N is the number of configurations. Since $N = 2^{O(s)}$, we obtain:

$$\text{space}(N) = O(\log^2 N) = O(s^2).$$

Why this fails with oracles. Consider a machine M^A with oracle access. Now, configurations must also include the contents of the oracle tape.

From today's quiz, a space- s machine may generate queries of length $2^{O(s)}$. Hence, the number of possible configurations is no longer bounded by $2^{O(s)}$.

Thus, the key bound on N breaks, and the same argument does not go through.

Key takeaway. Savitch's Theorem (and many space complexity results) *do not relativise*.

13.3 A relativised separation

Theorem 13.1 (Ladner–Lynch). *There exists a language A such that*

$$\text{NL}^A \not\subseteq \text{DSPACE}^A(n).$$

Proof idea. We construct a language $L_A \in \text{NL}^A$ that is not in $\text{DSPACE}^A(n)$.

Define:

$$L_A = \{1^n : \exists y \in A \text{ with } |y| = n^2\}.$$

$L_A \in \text{NL}^A$. We maintain a counter for n^2 in binary using $O(\log n)$ space. We nondeterministically guess y bit-by-bit, writing it onto the oracle tape and incrementing the counter. Once n^2 bits are written, we query the oracle. Accept if the oracle answers yes.

$L_A \notin \text{DSPACE}^A(n)$. Any deterministic space- n oracle machine runs in time at most $2^{O(n)}$, and hence can make at most $2^{O(n)}$ oracle queries on input 1^n .

However, there are 2^{n^2} possible strings of length n^2 . Thus, the machine cannot query all possibilities. Using a diagonalisation argument (as in Baker–Gill–Soloway), we can construct A to "fool" any such machine. □

13.4 Logspace reductions

We now introduce reductions appropriate for logarithmic space-bounded computation.

Observation First note that any language in P is P-complete under polynomial-time many-one reductions, except Σ^* and \emptyset . For completeness with respect to logspace classes, we need to define the notion of a logspace reduction. Suppose $A \in \text{DSPACE}(\log n)$ and M is a $\text{DSPACE}(\log n)$ oracle machine. Then

$$L(M^A) \subseteq \text{DSPACE}(\log n),$$

since we can simulate oracle queries using a logspace machine for A . Although the query string may be long, the machine for A reads its input one bit at a time, and we can supply these bits on demand by recomputing them instead of storing the entire query. As any query has length at most $2^{O(\log n)}$, the simulation uses $O(\log n)$ space. Hence, many-one and Turing reductions coincide in logspace.

13.4.1 Logspace transducers

A *logspace transducer* is a machine with:

- a read-only input tape,

- a write-only output tape,
- a work tape using $O(\log n)$ space, where n is the length of the input.

Definition 13.2. We say $A \leq_m^{\log} B$ if there exists a logspace transducer Φ such that

$$x \in A \iff \Phi(x) \in B.$$

◇

Definition 13.3. A language A is NL-hard if for every $B \in \text{NL}$, we have $B \leq_m^{\log} A$.

If additionally $A \in \text{NL}$, then A is NL-complete.

◇

13.4.2 Properties of logspace reductions

Lemma 13.4. If $A \leq_m^{\log} B$ and $B \in \text{NL}$, then $A \in \text{NL}$.

Proof idea. Let Φ be the logspace transducer corresponding to the reduction. On input x , simulate the machine for B on $\Phi(x)$, supplying bits of $\Phi(x)$ on a need-to-know basis by recomputing them. This uses only logarithmic space as we do not write the whole string. □

Lemma 13.5. Logspace reductions are transitive.

Proof idea. If $A \leq_m^{\log} B$ via Φ_1 and $B \leq_m^{\log} C$ via Φ_2 , then we can simulate $\Phi_2(\Phi_1(x))$ by recomputing bits of $\Phi_1(x)$ on a need-to-know basis. This uses only logarithmic space, so $A \leq_m^{\log} C$. □

13.5 NL-complete problems

Theorem 13.6. Directed graph reachability is NL-complete.

Proof idea. Membership in NL follows from previous discussions.

For hardness, let M be a nondeterministic logspace machine and x an input with $|x| = n$. The configuration graph $G_{M,x}$ has at most $2^{O(\log n)} = \text{poly}(n)$ vertices, since each configuration can be described using $O(\log n)$ bits. There is a directed edge from a configuration C_1 to C_2 if there is a valid transition of M that turns the configuration C_1 to C_2 . Let s be the initial configuration and t be the (unique) accepting configuration.

We construct the adjacency matrix of $G_{M,x}$ using a logspace transducer. Given a pair of configurations (C_1, C_2) , a deterministic logspace machine can:

- verify whether C_2 is a valid next configuration of C_1 by inspecting the state space of M , and
- output the corresponding entry of the adjacency matrix (output 1 if there is a valid transition, and 0 otherwise).

Since configurations have size $O(\log n)$ and we generate each entry on the fly without storing the entire graph, the transducer uses only $O(\log n)$ space.

Thus, M accepts x if and only if there is a path from s to t in $G_{M,x}$. Hence, every language in NL reduces to directed reachability, and since directed reachability is in NL, it is NL-complete. \square

Theorem 13.7 (Reingold, 2004). *Undirected reachability is in L.*

The proof is beyond the scope of this course.

Lecture 14

Verifier Perspective for NL, and $NL = coNL$

Scribe: Aindrila Rakshit

Topics covered in this lecture

- Verifier Perspective for NL
- Immerman–Szelepcsényi Theorem

14.1 Verifier Perspective for NL

Recall the verifier characterization of NP: $A \in NP \iff$ there is a deterministic polytime algorithm V such that

$$x \in A \iff \exists w \in \{0,1\}^{\text{poly}(n)} V(x,w) = 1$$

We now ask: *Can we obtain a similar characterization for NL?*

14.1.1 A First Attempt

Suppose we define: $A \in NL' \iff$ there is a $DSPACE(\log n)$ machine V such that

$$x \in A \iff \exists w \in \{0,1\}^{\text{poly}(n)} V(x,w) = 1$$

Claim 14.1. *If $A \in NL'$, then there exists a logspace verifier.*

Proof. Take witness as the path. The verifier is the deterministic simulation on the path, and this is a $DSPACE(\log n)$ machine.

For example, in graph reachability, the certificate is a path

$$s = v_0, v_1, \dots, v_k = t.$$

The verifier checks each edge (v_i, v_{i+1}) using $O(\log n)$ space. □

14.1.2 Why the converse doesn't hold

The converse says that if I have a language for which there is a logspace verifier such that there will only be witnesses for strings in my language then it implies $A \in \text{NL}$.

Anything that is doable using logspace verifier is certainly within NP, since asking for a deterministic logspace verifier is only stronger than asking for a deterministic polytime machine.

Take a NP-complete problem, say 3-COL. The logspace verifier for 3-COL: The input is the input graph G and the witness is a specific colouring. The verifier is the following: the current worktape contains currently checked edges, names of those two end points and what these colours are. Clearly, it is verifiable in $\text{DSPACE}(\log n)$.

Fact 14.2. *This class NL' is equal to NP, in fact Cook-Levin is doable in logspace.*

Thus, if the verifier has unrestricted access to the witness, then this model becomes as powerful as NP, even though you are making a lot of non-deterministic choices, these choices have to be made on the fly (in 3-COL, you could read the witness again and again), so we need to restrict how the verifier accesses the witness, make it read once.

14.1.3 Read-Once Verifiers

We impose the following restriction:

- The witness tape is **read-once** (one-way).

Lemma 14.3. *$A \in \text{NL}$ if and only if there exists a verifier V such that:*

- $V \in \text{DSPACE}(\log n)$,
- V has read-once access to the witness,
- $x \in A \iff \exists w V(x, w) = 1$.

Proof. (\Rightarrow) Verifier reads the i^{th} bit of the witness to decide which path to take.

(\Leftarrow) Construct an $\text{NSPACE}(\log n)$ machine for that language. We have a verifier and a read once witness. Guess the witness w one bit at a time. (In NP, we guessed the witness in its entirety and checked at the end with the verifier machine.) □

Example: Graph Reachability

- Input: (G, s, t)
- Witness: path $s = v_0, v_1, \dots, v_k = t$

The verifier:

- Reads vertices one by one,
- Checks adjacency,
- Uses $O(\log n)$ space to store the current vertex.

14.1.4 Key Insight

- Logspace verifier + read-once witness \Rightarrow NL
- Logspace verifier + unrestricted witness \Rightarrow NP

14.2 Immerman-Szelepcsényi Theorem

Theorem 14.4 (Immerman-Szelepcsényi). $NL = \overline{NL}$.

Proof. We take one \overline{NL} -complete problem and construct an NL machine for it. We saw last class that $\text{GraphReachability} \in NL$, so $\overline{\text{GraphReachability}} \in \overline{NL}$.

$\overline{\text{GraphReachability}} = \{ \langle \langle G \rangle, s, t \rangle : \text{there is no path from } s \text{ to } t \}$.

We will build a read-once log verifiable certificate for this language. Here's a key insight: suppose one knew exactly r vertices that were reachable from s , can a read-once log verifiable certificate for s not having a path to t be built?

Assuming such a r is given to us, we can say the following: the input is an ordered labelling of the vertices of G . The certificate is as follows: we list the r vertices reachable from s in ascending order of the labelling, along with the path from s to those r vertices: $\{u_1 : \text{Path to } u_1, \dots, u_r : \text{Path to } u_r\}$. The verifier and prover both agree on the r that is promised to them. The verifier keeps a counter to see r vertices, the counter gets incremented everytime it sees a block: $\{u_i : \text{Path to } u_i\}$. It needs to check if the blocks are in ascending order (label of previous vertex is less than the label of current vertex), check if the path from s to u_i is legitimate by storing the current vertex and checking if there is an edge from the previous vertex in that path, and it needs to check that $t \notin \{u_1, \dots, u_r\}$. So, overall it needs $4 \log(n)$ space.

So, it suffices to figure out r in NL. We do this by inductive counting. We define a set $R_i = \{u \in G : \text{Path}(s, t, i)\}$, where $\text{Path}(s, t, i)$ means there is a path of length at most i from s to t and $r_i = |R_i|$.

Trying to figure out r_n , i.e. if at all there is a path from s to t , it should be of length at most n .

Base case: $R_0 = \{u \in G : \text{Path}(s, t, 0)\} = \{s\}$ and $r_0 = |R_0| = 1$.

Inductive case: Suppose r_i is known, how do we build read-once witness for correct r_{i+1} .

14.2.1 Witness for r_{i+1}

The witness assuming r_i has been agreed upon is, gives the following claims in ascending order:

- $u_1 \in R_{i+1} : \text{Path to } u_1$
- $u_2 \in R_{i+1} : \text{Path to } u_2$
- $u_3 \notin R_{i+1} : \text{list out all vertices in } R_i: v_1, \dots, v_{r_i} \text{ along with their paths of length at most } i$
- \vdots
- $u_{r_{i+1}} \in R_{i+1} : \text{Path to } u_{r_{i+1}}$

What should the verifier checks:

- $u_1 < u_2 < \dots < u_n$ are in ascending order
- $\#(u \in R_{i+1}) = r_{i+1}$
- For $u \in R_{i+1}$, path is correct.
- For $u \notin R_{i+1}$:
 - $v_1 < \dots < v_{r_i}$
 - their paths are correct
 - $u \notin \{v_1, \dots, v_{r_i}\}$
 - No edge from v_j to u

Space required by verifier: Keep track of what portion of the proof it is checking

- remember previous u to check current v is valid (i.e. $u < v$),
- counter to check how many items in R_{i+1}

Overall the total space used by the verifier is $10 \log(n)$.

Observation 14.5. *If r_i was correct, and above checks pass, then r_{i+1} is correct.*

Proof. If $r_{i+1}^{\text{true}} < r_{i+1}$: No way one can give valid paths for r_i many vertices.

If $r_{i+1}^{\text{true}} > r_{i+1}$: then there was a $u \in R_i$ or there is an edge to u from the vertices in R_i . That is some $u \in R_{i+1}$ must have a proof that passes, i.e. there must be some proof that the prover is trying to claim that it is not in R_{i+1} but there is no such proof. \square

So, the NL machine is such that there is an input tape with the description of the graph G , the start vertex s , and the end vertex t written on it. It has a polynomially long read once witness tape with the proofs for each block written on it and in the work tape it currently has the value of the current r_i and some space for verifying π_i which is overall $10 \log(n)$.

Thus, $\overline{\text{GraphReachability}} \in \text{NL}$. \square

Lecture 15

Tree Evaluation Problem

Scribe: Ananya Ranade

Topics covered in this lecture

- Tree Evaluation Problem
- Some Algebra
- Cook Mertz Procedure

Recap We know that $L \subseteq NL = \text{coNL} \subseteq L^2$. Further, the general belief is that $L \subsetneq NL \subsetneq P$.

15.1 Tree Evaluation Problem

A potential candidate to separate L from P is believed to be the Tree Evaluation Problem ($\text{TEP}_{\ell,h}$). The input consists of a function $f : \{0,1\}^\ell \times \{0,1\}^\ell \rightarrow \{0,1\}^\ell$. We have to evaluate the value at the root of a height h binary tree. To evaluate the value at a non leaf node, we first evaluate the values of its left and right children. Let the values be σ_L, σ_R . Then, the value at the node is $f(\sigma_L, \sigma_R)$. We are provided the values of the 2^h leaves of the tree.

Hence, the total input size is $2^{2^h} \ell + 2^h \ell = n$. We further assume that $h, \ell \approx \log n$.

It is easy to see that if we compute the value at each of the nodes from bottom to the top, then time is equal to the number of nodes. Thus, $\text{TEP}_{\ell,h} \in \text{DTIME}(2^{\mathcal{O}(\ell+h)})$ and hence $\text{TEP}_{\ell,h} \in P$.

Now, let us consider the following obvious low space algorithm for $\text{TEP}_{\ell,h}$:

1. Compute $\sigma_L = \text{TEP}_{\ell,h-1}$ on left subtree from the root.
2. Compute $\sigma_R = \text{TEP}_{\ell,h-1}$ on right subtree from the root.
3. Compute $\sigma = f(\sigma_L, \sigma_R)$.
4. Return σ .

By this algorithm, $\text{Space}_{\ell,h} \leq \ell + \text{Space}_{\ell,h-1}$. And hence, $\text{Space}_{\ell,h} = \mathcal{O}(\ell h)$. Since this feels like the best possible thing we can do, there was a conjecture that $\text{TEP}_{\ell,h}$ requires space $\Omega(\ell h)$. However, the conjecture is false! The Cook-Mertz Theorem gives a procedure to solve the $\text{TEP}_{\ell,h}$ instance in $\text{DSPACE}((h + \ell) \log \ell)$. Thus, when $\ell, h = \mathcal{O}(\log n)$, we get that $\text{TEP} \in \text{DSPACE}(\log n \log \log n)$.

15.2 Some Algebra

15.2.1 Multilinear extensions

Suppose we are given a function $g : \{0, 1\}^k \rightarrow \{0, 1\}$. If g was just the AND function, the polynomial $\tilde{g}(z_1, \dots, z_k) = z_1 z_2 \dots z_k$ coincides with g on the domain $\{0, 1\}^k$. In general for any function $g : \{0, 1\}^k \rightarrow \{0, 1\}$, we say \tilde{g} is a polynomial extension of g if \tilde{g} is a polynomial such that $g(a_1, \dots, a_k) = \tilde{g}(a_1, \dots, a_k)$ for all $a_1, \dots, a_k \in \{0, 1\}$.

Lemma 15.1 (Multilinear extensions). *Every function $g : \{0, 1\}^k \rightarrow \{0, 1\}$ has a polynomial (multilinear) extension of degree $\leq k$ in \mathbb{F}_p .*

Proof. Let $\mathbb{1}(z = a)$ be the indicator function, which evaluates to 1 if $z = a$ and is 0 otherwise. Thus, if $z = (z_1, \dots, z_k), a = (a_1, \dots, a_k)$, then $\mathbb{1}(z = a) = \prod_{i=1}^k (z_i a_i + (1 - z_i)(1 - a_i))$ is a multinomial in z which takes values equal to $\mathbb{1}(z = a)$ on $z \in \{0, 1\}^k$. Now, consider

$$\tilde{g}(z_1, \dots, z_k) = \sum_{a \in \{0,1\}^k} \mathbb{1}(z = a) \cdot g(a_1, \dots, a_k) = \sum_{a \in \{0,1\}^k} g(a_1, \dots, a_k) \cdot \prod_{i=1}^k (z_i a_i + (1 - z_i)(1 - a_i)).$$

It coincides with g on $\{0, 1\}^k$ and is a polynomial in z_1, \dots, z_k of degree $\leq k$. \square

If we use this extension, given g we can compute $\tilde{g}(\alpha_1, \alpha_2, \dots, \alpha_k)$ in space $\approx k(\log p)$. Here \tilde{g} is thought of as a polynomial in \mathbb{F}_p and hence $\alpha_1, \dots, \alpha_k \in \mathbb{F}_p$.

15.2.2 Interpolation

Let $g(x) = g_0 + g_1 x + \dots + g_d x^d$ (unknown), and suppose we are given $g(1), \dots, g(d+1)$. Then, we can compute $g(0)$, and $g(0) = \sum_{i=1}^{d+1} \beta_i g(i)$ for some constants β_i 's and for all g .

Lemma 15.2 (Interpolation). *Suppose $1, 2, \dots, d+1$ are distinct elements in the field. Then, there exists constants $\beta_1, \dots, \beta_{d+1}$ such that for any polynomial $g(x) = g_0 + g_1 x + \dots + g_d x^d$ we have*

$$g(0) = \sum_{i=1}^{d+1} \beta_i \cdot g(i).$$

Proof. If we know the values of $g(\alpha_0), \dots, g(\alpha_d)$ for some distinct $\alpha_0, \dots, \alpha_d \in \mathbb{F}_p$, then we can

find g uniquely. Consider the system,

$$\begin{bmatrix} 1 & \alpha_0 & \alpha_0^2 & \cdots & \alpha_0^d \\ 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_d & \alpha_d^2 & \cdots & \alpha_d^d \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_d \end{bmatrix} = \begin{bmatrix} g(\alpha_0) \\ g(\alpha_1) \\ \vdots \\ g(\alpha_d) \end{bmatrix}$$

Any solution $[g_0, \dots, g_d]^T$ to this system will be a valid polynomial g . The matrix is a Vandermonde Matrix and hence is invertible. Thus, there is a unique polynomial of degree d which agrees with g on $\alpha_0, \dots, \alpha_d$. Thus, $g(0) = g_0$. And we can write g_0 as a linear combination of $g(\alpha_0), \dots, g(\alpha_d)$, where the coefficients of $g(\alpha_0), \dots, g(\alpha_d)$ depend only on $\alpha_0, \dots, \alpha_d$.

Example if g is quadratic, $g(0) = 3g(1) - 3g(2) + g(3)$. □

We now have all the ingredients we need for the Cook-Mertz TEP algorithm.

15.3 The Cook-Mertz Procedure

In the tree evaluation problem, we are given a function $f : \{0, 1\}^\ell \times \{0, 1\}^\ell \rightarrow \{0, 1\}^\ell$.

We will choose a suitable field \mathbb{F}_p (for a prime p) and consider the multilinear extension $\tilde{f} : \mathbb{F}_p^{2\ell} \rightarrow \mathbb{F}_p^\ell$. Note that given the truth-table of f , we can evaluate $\tilde{f}(\bar{\alpha}, \bar{\beta})$ for any $\bar{\alpha}, \bar{\beta} \in \mathbb{F}_p^\ell$ using space $O(\ell \log p)$. Therefore, we may assume that we have the truth-table of \tilde{f} with us.

In the recursive algorithm for TEP mentioned earlier, we recursively need to compute the values at the nodes. Thus, since the recursion depth is h , we need to remember at least ℓh bits. In contrast, the Cook-Mertz procedure will just use 3 registers of length ℓ which store values in \mathbb{F}_p , and keep cycling through them to compute f .

For reasons that will soon become clear, we will attempt to always compute values in a “reversible” manner. We will say that a program P “reversibly computes $\sigma \in \mathbb{F}_p^\ell$ on register 1” to mean the following:

Assume that the three registers are initialised to some values $\tau_1, \tau_2, \tau_3 \in \mathbb{F}_p^\ell$. When program P is run with these register values, at the end of the computation the register values are $\tau_1 + \sigma, \tau_2, \tau_3$.

The above guarantee must hold for *every* possible initialisation of the three registers. To make it more clearer, we will assume that we have two programs P_1^+ and P_1^- which result in

$$\begin{aligned} P_1^+ : (\tau_1, \tau_2, \tau_3) &\mapsto (\tau_1 + \sigma_1, \tau_2, \tau_3) \\ P_1^- : (\tau_1, \tau_2, \tau_3) &\mapsto (\tau_1 - \sigma_1, \tau_2, \tau_3) \end{aligned}$$

We will try to construct the program that reversibly computes the root value recursively. Clearly, for each leaf node v , building programs that compute σ_v recursively is trivial — just read the leaf value and increment/decrement the appropriate register.

The inductive step: Suppose we want to calculate the value at some node v . Let the value that this node would get be σ . Let the value of its left and right children be σ_u, σ_v respectively. Thus, $\sigma = \tilde{f}(\sigma_u, \sigma_v)$.

Assume inductively that we already have programs P_u^\pm, P_v^\pm such that

$$P_u^\pm : (\tau_1, \tau_2, \tau_3) \mapsto (\tau_1 \pm \sigma_u, \tau_2, \tau_3)$$

$$P_v^\pm : (\tau_1, \tau_2, \tau_3) \mapsto (\tau_1, \tau_2 \pm \sigma_v, \tau_3)$$

We are trying to build:

$$P : (\tau_1, \tau_2, \tau_3) \mapsto (\tau_1, \tau_2, \tau_3 + \sigma)$$

where $\sigma = \tilde{f}(\sigma_u, \sigma_v)$.

If \tilde{f} happens to be linear: Suppose $\tilde{f}(\sigma_u, \sigma_v) = \sigma_u + \sigma_v$ for all $\sigma_u, \sigma_v \in \mathbb{F}_p^\ell$, then of course we can build such a program easily — swap the order of the entries in the tuple, then apply P_u, P_v suitably.

Suppose $\tilde{f}(\sigma_u, \sigma_v) = \sigma_u \sigma_v$: Then, we can't just apply the programs one after another and multiply the registers because it would then carry the value $\tau_3 + (\tau_1 + \sigma_u) \cdot (\tau_2 + \sigma_v)$ instead of $\tau_3 + \sigma_u \sigma_v$. But we can make use of P_u^\pm and P_v^\pm to get rid of the excess terms.

$(\tau_1, \tau_2, \tau_3) \rightarrow (\tau_1 + \sigma_u, \tau_2 + \sigma_v, \tau_3)$. Now, multiply the first 2 slots and add it to 3rd to get $\tau_3 + \sigma_u \sigma_v + \tau_1 \sigma_v + \tau_2 \sigma_u + \tau_1 \tau_2$. Now, subtract σ_u from first slot by applying P_u^- , multiply the first 2 slots and subtract from 3rd. Now, add σ_u to first, subtract σ_v from 2nd, multiply the slots and subtract the value from 3rd slot. Now, subtract σ_v from 2nd slot, multiply 1st 2 slots and subtract it from third. We get $(\tau_1, \tau_2, \tau_3 + \sigma_u \sigma_v)$. Similarly we can do subtraction.

However, there is a smarter way to do this. We know by interpolation that for any degree d polynomial g , if we know the values of the polynomial on $d + 1$ points $\alpha_0, \dots, \alpha_d$, then we can write $g(0)$ as a linear combination of $g(\alpha_0), \dots, g(\alpha_d)$, where the coefficients depend only on $\alpha_0, \dots, \alpha_d$.

If we set $\tilde{f}(c\tau_1 + \sigma_u, c\tau_2 + \sigma_v) = g(c)$, then if \tilde{f} is quadratic, $g(0) = 3g(1) - 3g(2) + g(3)$. Thus, $\tilde{f}(\sigma_u, \sigma_v) = 3\tilde{f}(\tau_1 + \sigma_u, \tau_2 + \sigma_v) - 3\tilde{f}(2\tau_1 + \sigma_u, 2\tau_2 + \sigma_v) + \tilde{f}(3\tau_1 + \sigma_u, 3\tau_2 + \sigma_v)$, and these computations are doable in a reversible manner!

Final Algorithm

Algorithm 3: Computing $\sigma = \tilde{f}(\sigma_u, \sigma_v)$ reversibly on register R_3

```

1 global memory  $R_1, R_2, R_3 \in \mathbb{F}_p^\ell$ .
2 for  $c = 1, \dots, d$  do
3    $R_1 \leftarrow c \cdot R_1$ 
4    $R_2 \leftarrow c \cdot R_2$ 
5   Run program  $P_u^+$  on  $R_1$ 
6   Run program  $P_v^+$  on  $R_2$ 
7    $R_3 \leftarrow R_3 \pm \beta_c \tilde{f}(R_1, R_2)$ 
8   Run program  $P_u^-$  on  $R_1$ 
9   Run program  $P_v^-$  on  $R_2$ 
10   $R_1 \leftarrow c^{-1} \cdot R_1$ 
11   $R_2 \leftarrow c^{-1} \cdot R_2$ 

```

Choosing the field \mathbb{F}_p : For the above algorithm, we just need \mathbb{F}_p to have enough distinct elements to be able to do the interpolation. Therefore, setting $p \approx d = O(\ell)$ is sufficient.

The algorithm recursively computes P_u^\pm and P_v^\pm using the same 3 registers. There is a global memory which takes space $3\ell \log p$, and a local memory which needs to remember $\log d$ bits to remember the loop index c . Since the recursion depth is h , total space required by this algorithm is $\leq 3\ell(\log p) + (\log d)h \leq 3\ell(\log \ell) + h \log \ell = \mathcal{O}((h + \ell) \log \ell)$.

Theorem 15.3 (Cook and Mertz). $\text{TEP}_{\ell, h} \subseteq \text{DSPACE}((\ell + h) \log \ell)$. In particular, when $h, \ell = O(\log n)$ we have $\text{TEP} \in \text{DSPACE}(\log n \log \log n)$.

Lecture 16

Simulating time in square-root space

Scribe: Chandralekha P

16.1 Recap

In previous lecture, we saw the Cook-Mertz theorem which showed that the TEP problem can be done in $DSPACE(\log n \log \log n)$.

We have also seen in earlier lectures that:

- $DTIME(t) \subseteq DSPACE(t)$
- $DSPACE(t) \subseteq DTIME(2^{O(t)})$

Our goal in this lecture will be to see, if we can simulate $t(n)$ time machine with $o(t(n))$ space machine.

16.2 Reducing the required space

There are known results where we can do it with slightly lesser space. One of them is the following given by Hopcroft-Paul-Valiant.

Theorem 16.1 (Hopcroft-Paul-Valiant '75). $DTIME(t(n)) \subseteq DSPACE(\frac{t(n)}{\log t(n)})$

For the special case of 1-tape turing machines, an even better bound has been shown which is the following:

Theorem 16.2 (Hopcroft-Ullman '68). *For 1-tape TMs, $DTIME(t(n)) \subseteq DSPACE(\sqrt{t(n)})$*

Now, we want to ask if it is possible that there is an $\epsilon > 0$ for which $DTIME(t(n)) \subseteq DSPACE(t(n)^{1-\epsilon})$

This will be the main focus of today's lecture.

16.3 Time-Space simulation using TEP

First we will see an overview of HPV '75 Consider the time steps till $t(n)$, and divide it into blocks of length B . We will now individually look at what the machine did at the i^{th} epoch(essentially, the i^{th} block).

At the end of each block, we want to give the configuration of the TM, which consists of the current state, head positions, and tape contents.

The idea of HPV is: If we are given the configuration at time step iB , then we can use at most B space to compute the configuration at time step $(i + 1)B$.

Now, the question is, do we really need to know the whole configuration at time iB . This is not necessary, because at any epoch i , the head can move atmost B steps before and after the current head position. Hence, if we are just given this, then it is sufficient to run the whole computation.

Now, for convenience, we will assume that the TM is block respecting(any machine can be made into block respecting with only a constant factor increase in the running time). We will use the following notations for ease of use.

- $\text{TapeBlock}(h, i)$: It denotes what block in tape h is the head in during epoch i .
- $\text{Contents}(h, i)$: Contents of the above tape block after epoch i .
- $\text{LocalConfig}(i)$: This will denote the state at the end of epoch i , the contents+head positions of $\text{TapeBlock}(*, i)$

To compute $\text{LocalConfig}(i)$, we want to know what the $\text{LocalConfig}(i - 1)$ was, and then we can simulate it. But the issue would come when the tape h changed block between epoch $i - 1$ and i , and hence we will need to know when that block was last accessed in. So, we somehow need to store this information as well. Hence,

$\text{LocalConfig}(i)$ depends on:

- $\text{LocalConfig}(i - 1)$
- $\text{LocalConfig}(j_h(i))$ where $h \in [\text{tapes}]$ where $j_h(i)$ tells what the most recent time this block was worked on.

Hence, when we are given all the required data, then we can compute the local configurations like a tree. Hence, this is a TEP instance with height $h = O(\frac{t}{B})$, arity $a = \text{number of tapes} + 1$, and the leaf label size ℓ is $O(B)$

Hence, by using the Cook-Mertz theorem, we can get that this can be done in $O(\sqrt{t} \log t)$ (the bound can be made better but we will not do that here)

Now, the only thing is to build the tree. Only if we know the tree, we can run the whole algorithm.

16.4 Building the tree

Corresponding to each tape of the TM, we will have a sequence $m_{h,i}$ which is 0 (if head h stays in same block at end of epoch), 1 (moves right), -1 (moves left).

Using this, we can see what the current block is, and when the previous time the tape was accessed. These can just be done by looking at the sum of the sequence.

Now, the question is, how will we get the sequence. But, this we will just guess and verify. Hence, for this guessed sequence, we can build the tree, and if at some point the sequence seems to not satisfy the transitions, then we abort and guess another sequence. Hence, using this we get the tree, and now we can directly run our TEP algorithm.

The total space required for this is $O(t/B) + O(t/B) + O(B) + O((B + t/B) \log B)$. If we take $B = \sqrt{t}$, then we get that $O(\sqrt{t} \log t)$ space is required.

Possible open problem extending from this is whether we can improve this bound and remove the log factor.

The above is the proof sketch of Williams proof.

Lecture 17

Catalytic Computation

Scribe: Hrishikesh Saikia

Topics covered in this lecture

- Catalytic Turing Machines and Complexity Classes
- Bounds on Catalytic Logspace
- Reversible Turing Machines
- Cook-Li-Mertz-Pyne Theorem

In this lecture, we introduce the catalytic model of computation and study complexity classes defined in terms of the resources used by these machines.

17.1 Setup and Definitions

We consider our usual Turing Machine for space computation which consists of a single read-only input tape and a bunch of read-write work tapes, and add to it an extra read-write tape, called the “catalytic” tape, which will be initialized to an arbitrary string. We require that the machine halts (accepts or rejects the input) with the catalytic tape content returned to its initial string. This model of computation is called a catalytic Turing Machine and we will be working with this model in this chapter.

Definition 17.1. Let $s, c : \mathbb{N} \rightarrow \mathbb{N}$ be non-decreasing functions. We say that a language L is decidable by a catalytic TM M in space $s(n)$ and catalytic space $c(n)$ if for every input string x and arbitrary string y of length $c(|x|)$ written on the catalytic tape, M halts with y written on its catalytic tape and correctly decides whether x is in L or not by using at most $s(|x|)$ work tape cells and $c(|x|)$ catalytic tape cells corresponding to the cells of y . \diamond

We can now define the catalytic complexity class:

$$\text{CSPACE}(s(n), c(n)) = \{L : L \text{ is decidable by a catalytic TM in space } s(n) \text{ and catalytic space } c(n)\}$$

As a shorthand, we let $\text{CSPACE}(s(n)) = \text{CSPACE}(s(n), 2^{O(s(n))})$. We then define

$$\text{CL} = \text{CSPACE}(\log n) = \bigcup_{c \geq 1} \text{CSPACE}(\log n, n^c)$$

17.2 Bounds on CL

Observation 17.2. $L \subseteq \text{CL} \subseteq \text{PSPACE}$

Proof. Consider $L \in L$. Consider a logspace machine M deciding L . We can construct a CL machine M' for L by simply adding a blank catalytic tape to M and simulating the behaviour of M exactly. So $L \in \text{CL} \implies L \subseteq \text{CL}$.

Let $L \in \text{CL}$. Consider a CL machine M for L . We can construct a PSPACE machine M' for L by simply considering the catalytic tape of M as a work tape and simulating M exactly. Since the catalytic tape takes up polynomial space (and other work tapes take up logspace), so $L \in \text{PSPACE} \implies \text{CL} \subseteq \text{PSPACE}$. \square

Theorem 17.3 (Pyne). $\text{NL} \subseteq \text{CL}$

Proof. It suffices to construct a CL machine for DIR-ST-CONN. Consider an instance (G, s, t) , where $G = (V, E)$ and $s, t \in V$. Wlog (why?), suppose G is layered. The catalytic tape C will consist of $|V| = n$ (say) cells marked with the vertices of G and initiated with arbitrary values. We then define the following two subroutines:

Fwd-Pass:

1. For layers $i = 1$ to n :
 1. For $u \in \text{layer } i$:
 1. For v such that there is an edge from v to u :
 1. Update $C[u] = C[u] + C[v]$.

Fwd-Pass-Inverse:

1. For layers $i = n$ to 1:
 1. For $u \in \text{layer } i$:
 1. For v such that there is an edge from v to u :
 1. Update $C[u] = C[u] - C[v]$.

Suppose v_t is the value of $C[t]$ after one call of Fwd-Pass and let v'_t be the value of $C[t]$ after one call of Fwd-Pass with $C[s]$ incremented by 1. Then $v'_t - v_t$ is the number of directed paths from s to t in G (why?). So we can store the values of v_t and v'_t in the work tapes and check if their difference is non-zero to decide whether G has a directed $s \rightarrow t$ path or not. Note that the Fwd-Pass-Inverse subroutine lets us undo and get back the catalytic tape contents after a Fwd-Pass.

However, note that the number of $s \rightarrow t$ paths might be exponential in n . So storing the values of v_t and v'_t will take $O(n)$ bits which is more than what is allowed by the work tapes. As a fix, we will instead compare v_t and v'_t bit by bit. In the i -th iteration, we will only store and compare the i -th bits of v_t and v'_t . So our final algorithm is as follows: Suppose our catalytic tape has $m = O(n)$ bits per vertex. Let $M = 2^m$.

1. For $i = 1$ to m :
 1. Run Fwd-Pass (mod M).
 2. Let $b = i$ -th bit of $C[t]$.
 3. Run Fwd-Pass-Inverse.
 4. Set $C[s]_+ = 1$.
 5. Run Fwd-Pass (mod M).
 6. Let $b' = i$ -th bit of $C[t]$.
 7. Run Fwd-Pass-Inverse.
 8. If $b \neq b'$, return "YES".
2. Return "NO".

□

So CL seems to do way more than L. Infact, we have the following containments:

$$L \subseteq NL \subseteq \text{LOGCFL} \subseteq \text{TC}^1 \subseteq \text{CL}$$

Exercise 1 (Problem set 3). *Prove that $\text{CL} \subseteq \text{ZPP}$.*

Open Problem 1. *Is $\text{CL} \subseteq \text{P}$?*

17.3 Reversible Turing Machines

Definition 17.4. *Given a machine M , a reversible simulation M' of M is a machine equipped with two bijective functions Fwd and $Back$ on the set of configurations of M' such that Fwd and $Back$ are inverses of each other, and there is a start \rightarrow accept path in the configuration graph of M' for input x using Fwd operations iff $x \in L(M)$. \diamond*

Theorem 17.5 (Lange-McKenzie-Tapp). *Any $\text{DSPACE}(\log n)$ machine has a reversible simulation using a logspace machine.*

Proof. Let M be a $\text{DSPACE}(\log n)$ machine. Consider its configuration graph with respect to an input x . At each vertex there will be a bunch of edges feeding in and exactly one edge going out (since M is deterministic). Label the outgoing edge by 1 and the remaining edges by 2, 3, etc. in anti-clockwise direction. So each edge gets two labels corresponding to the two endpoints. The configurations of M' then consists of tuples of the form (C, i) where C is a

configuration of M and i is an integer (corresponding to an edge label). The start configuration would be $(start_M, 1)$. Then the *Fwd* and *Back* operations are defined as follows:

Fwd: On input (C, i) :

1. Let $Rot(C, i) = (C', j)$
2. Return $(C', j + 1)$

Back: On input (C', k) :

1. Let $Rot(C', k - 1) = (C, i)$
2. Return (C, i)

Here, $Rot(u, i) = (v, j)$, where v is the i -th neighbour of u and u is the j -th neighbour of v . So using these operations, we can basically perform an Eulerian tour of the configuration graph of M by storing only $O(\log n)$ many bits at each step (a couple of configurations and edge labels). This gives us a logspace reversible simulation of M . \square

Observation 17.6 (Dulek). *The above holds for a CL machine as well: just tag along the catalytic tape.*

17.4 An Application: Cook-Li-Mertz-Pyne Theorem

We will now see an application of the above observation. Define the class CLP as the set of languages that can be decided by a CL-machine that runs in polytime. Since any CLP is also a CL-machine and also a P-machine, so $CLP \subseteq CL \cap P$.

Theorem 17.7 (Cook-Li-Mertz-Pyne). $CLP = CL \cap P$

Proof Idea: We need to show that $CL \cap P \subseteq CLP$. Let $L \in CL \cap P$. So there exists a CL-machine M_{CL} and a P-machine M_P for L . For convenience, suppose M_{CL} is a $CSPACE(2 \log n, n^3)$ -machine and M_P is an n^5 -time machine. We will use these two machines to construct a CLP-machine M' . The catalytic tape of M' will be $n^5 \log n + n^3 + 3 \log n$ cells long (three blocks). The first block of $n^5 \log n$ cells (initialized to some arbitrary string) will serve as a "buffer" space (as we will see later). Suppose τ is the initial content of the second block of n^3 cells, while the final block of $3 \log n$ cells contain l (an integer at most n^3).

Suppose we run M_{CL} with catalytic tape content τ for l steps. There are two cases:

Case 1: M_{CL} terminates in these many steps. Then M' outputs whatever M_{CL} outputs.

Case 2: M_{CL} hasn't terminated yet. Suppose the catalytic tape content is τ' and work tape content is a $2 \log n$ bit string w . We claim that we can recover (τ, l) from (τ', w) . To see why, we use the fact that M_{CL} has a reversible simulation. We repeatedly use the *Back* operation until we hit the start state and get τ . To compute l , we can keep a counter

along the way to count the number of times we used *Back*. So, we basically managed to come up with a mechanism to compress an $n^3 + 3 \log n$ bit string to an $n^3 + 2 \log n$ bit string.

We will do the above repeatedly to free up the buffer space and then use it to run M_P . More precisely, suppose the second and third block contents were τ_1 and l_1 respectively. We run M_{CL} for l_1 steps with catalytic tape τ_1 and end up with τ_2 on the second block and l_2 ($2 \log n$ bits long) on the third block. As seen above, we can get back (τ_1, l_1) from (τ_2, l_2) . Since we have a free space of $\log n$ bits in the third block, we transport a chunk of $\log n$ bits from the buffer block, thus freeing up $\log n$ bits in the buffer block. We continue this for at most n^5 steps. If at any time M_{CL} terminates, M' outputs whatever M_{CL} output and reverses the content of the tape to its original string. Else, we are left with $n^5 \log n$ bits free space on the first block which we can use to run M_P and decide L . M' clearly runs in polynomial time and uses logarithmic work space (corresponding to M_{CL}) and polynomial catalytic space. Thus $L \in \text{CLP}$, as desired. \square

Lecture 18

Randomised Complexity Classes

Scribe: Krishnashree J B

Topics covered in these lectures

- Randomised Turing Machines and Acceptance Criteria
- The Classes BPP, RP, coRP.
- Error Reduction via Repetition and Chernoff Bounds.

18.1 Motivation

Recall that the main computation model studied so far are deterministic machines which either accepts an input x or rejects it; non-deterministic machines which accept when at least one witness(or path) accepts; and non-deterministic machines which accept when all witnesses accept. A natural intermediate model asks: what if the witness w is a random guess and when $x \in L$ has good number of witnesses. The acceptance criterion then depends on $\Pr_w [M(x, w) = \text{acc}]$.

Note that, in the nondeterministic model, there are two subtrees rooted at $w_1 = 0$ and $w_1 = 1$ respectively and it might happen that $w_1 = 0$ is itself an immediate accept, whereas the other subtree has to compute all possible paths before deciding. In such cases, assume that the probability is considered over $2^{|w|}$ many witnesses and not just $2^{|w|-1} + 1$ strings.

18.2 Randomised Complexity Classes

18.2.1 The Class $\text{BPP}_{c,s}$

The class $\text{BPP}_{c,s}$ is called *Bounded-Error Probabilistic Polynomial Time* with completeness c and soundness s .

Definition 18.1 ($\text{BPP}_{c,s}$). Let $1 \geq c > s \geq 0$, a language L is in $\text{BPP}_{c,s}$ if there exists a nondeterministic polynomial-time machine M such that, for every input x :

Completeness. $x \in L \implies \Pr_r[M(x, r) = \text{acc}] \geq c$.

Soundness. $x \notin L \implies \Pr_r[M(x, r) = \text{acc}] \leq s$.

Here r is a uniformly random string of polynomial length. \diamond

The default parameters are $c = \frac{2}{3}$ and $s = \frac{1}{3}$, giving a two-sided error of at most $\frac{1}{3}$. We write BPP for $\text{BPP}_{2/3, 1/3}$.

18.2.2 RP, coRP

Definition 18.2 (RP_c (randomised polynomial time with one-sided error c)). A language $L \in \text{RP}_c$ if there is a polynomial-time nondeterministic machine M with

$$x \in L \implies \Pr_r[M(x, r) = \text{acc}] \geq c, \quad x \notin L \implies \Pr_r[M(x, r) = \text{acc}] = 0.$$

\diamond

RP_c is $\text{BPP}_{c, 0}$.

Definition 18.3 (coRP). A language $L \in \text{coRP}_s$ if there is a polynomial-time nondeterministic machine M with

$$x \in L \implies \Pr_r[M(x, r) = \text{acc}] = 1, \quad x \notin L \implies \Pr_r[M(x, r) = \text{acc}] \leq s.$$

\diamond

$\text{coRP}_s = \text{BPP}_{1, s}$.

Remark 18.4. These are semantic classes. That is, given an encoding of a machine its difficult to say if it is a randomised complexity class or not. This makes it hard to find natural complete problems for BPP, RP, or coRP. \diamond

Observation 18.5. $\text{RP} \subseteq \text{NP}$ and $\text{coRP} \subseteq \text{coNP}$.

Proof. Suppose $L \in \text{RP}$ via a machine M . If $x \in L$, then $\Pr_r[M(x, r) = \text{acc}] \geq \frac{2}{3} > 0$, so at least one accepting path exists; this path serves as an NP witness. If $x \notin L$, the acceptance probability is exactly 0, so no witness exists. Hence $L \in \text{NP}$. The argument for $\text{coRP} \subseteq \text{coNP}$ is symmetric. \square

18.3 Error Reduction

A key feature of these classes is that the specific numerical thresholds do not matter (as long as there is a gap between completeness and soundness).

18.3.1 Error Reduction for RP

Claim 18.6. $\text{RP}_{2/3} = \text{RP}_{0.99}$. More generally, $\text{RP}_{2/3} = \text{RP}_{1-\varepsilon}$ for every $\varepsilon > 0$, using $k = O(\log \frac{1}{\varepsilon})$ repetitions.

Proof. The inclusion $\text{RP}_{0.99} \subseteq \text{RP}_{2/3}$ is immediate. For the other direction, let M be an $\text{RP}_{2/3}$ machine for L . Construct a new machine M' that, on input x , samples k independent random strings r_1, \dots, r_k and accepts if *any* of the runs $M(x, r_i)$ accepts.

Completeness. If $x \in L$, then M' accepts whenever at least one r_i is a good witness, which happens with probability $1 - \Pr[\text{all } k \text{ runs reject}]$.

Soundness. If $x \notin L$, then $\Pr_r[M(x, r) = \text{acc}] = 0$ for every r , so M' rejects with probability 1.

For the completeness bound, since the r_i are independent,

$$\Pr_{r_1, \dots, r_k} [M'(x, r_1, \dots, r_k) = \text{rej}] = \prod_{i=1}^k \Pr_{r_i} [M(x, r_i) = \text{rej}] \leq \left(\frac{1}{3}\right)^k.$$

Choosing $k = \lceil \log_3(1/\varepsilon) \rceil = O(\log \frac{1}{\varepsilon})$ makes this at most ε . The runtime of M' is k times that of M , which remains polynomial.

In particular, taking $k \approx 10$ gives error at most $(1/3)^{10} < 0.01$, so $\text{RP}_{2/3} \subseteq \text{RP}_{0.99}$. \square

Claim 18.7. $\text{RP}_{2/3} = \text{RP}_\delta$ for a small δ .

Idea. Similar to previous proof, it suffices to show $\text{RP}_\delta \subseteq \text{RP}_{2/3}$. The same idea of k repetition works here too. While bounding the $\prod_{i=1}^k \Pr_{r_i} [M(x, r_i) = \text{rej}] \leq (1 - \delta)^k$ and we want this to be less than $\frac{1}{3}$. To achieve this, $k = O(\frac{1}{\delta})$. \diamond

We want the number of repetitions to be polynomial. So, ε can be inverse exponential and δ can be inverse polynomial.

Corollary 18.8. $\text{RP}_{1/\text{poly}(n)} = \text{RP} = \text{RP}_{1-1/2^{\text{poly}(n)}}$.

18.3.2 Error Reduction for BPP

For two-sided error, the reduction is more involved: we run M many times and take a *majority vote*.

Claim 18.9. $\text{BPP}_{2/3, 1/3} = \text{BPP}_{1-\delta, \delta}$ for any $\delta > 0$.

Proof. $\text{BPP}_{1-\delta, \delta} \subseteq \text{BPP}_{2/3, 1/3}$ is immediate. For other direction, let M be a $\text{BPP}_{2/3, 1/3}$ machine. Construct M' that samples k independent strings r_1, \dots, r_k and accepts iff a *strict majority* of the $M(x, r_i)$ accept.

Fix any x . Define indicator random variables $Y_i = \mathbf{1}[M(x, r_i) = \text{rej}]$, and let $Y = \sum_{i=1}^k Y_i$. Then M' rejects iff $Y > k/2$.

- If $x \in L$: $\Pr[Y_i = 1] \leq \frac{1}{3}$, so $\mu := \mathbb{E}[Y] \leq k/3$.

We use the following inequality.

Lemma 18.10 (Chernoff Bound). Let Y_1, \dots, Y_k be independent identically distributed Bernoulli random variables with $Y = \sum Y_i$ and $\mu = \mathbb{E}[Y]$. Then for every $\varepsilon > 0$,

$$\Pr[|Y - \mu| \geq \varepsilon \mu] \leq 2e^{-\varepsilon^2 \mu / 3}.$$

For $x \in L$: $\mu \leq k/3$. The event M' rejects is $\{Y > k/2\}$. Setting ε so that $(1 + \varepsilon)\mu = k/2$ and applying Chernoff gives

$$\Pr[M' \text{ rejects}] \leq \Pr[|Y - \mu| \geq \varepsilon\mu] \leq 2e^{-\varepsilon^2 k/9}.$$

Choosing $k = O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta}\right)$ makes this at most δ . The case $x \notin L$ is symmetric. Since k is polynomial in $1/\delta$ and $\log 1/\varepsilon$, the runtime of M' is still polynomial. \square

Corollary 18.11.

$$\text{BPP}_{1/2+1/\text{poly}(n), 1/2-1/\text{poly}(n)} = \text{BPP} = \text{BPP}_{1-1/2^{\text{poly}(n)}, 1/2^{\text{poly}(n)}}.$$

Even a tiny gap of $1/\text{poly}(n)$ between completeness and soundness is sufficient to bring a language into BPP.

Lecture 19

Randomised Complexity (continued)

Scribe: Soham Ghosh

Topics covered in this lecture

- ZPP
- Structural results for BPP

19.1 Zero-error randomised computation

We have already seen randomised classes such as BPP, RP, and coRP. We now consider a different type of randomised algorithm: one that makes *no errors*, but whose running time is a random variable with polynomial expectation.

A canonical example is randomised quicksort. Such algorithms were termed *Las Vegas algorithms* by Babai, in contrast to BPP, RP, and coRP, which are called *Monte Carlo algorithms*.

Definition 19.1. ZPP (Zero-error Probabilistic Polynomial Time) is the class of languages L such that there exists a randomised Turing machine M satisfying:

- If $x \in L$, then $M(x, r)$ either accepts or does not halt.
- If $x \notin L$, then $M(x, r)$ either rejects or does not halt.

Further, the expected running time satisfies

$$\mathbb{E}[\text{time}_M(x)] \leq |x|^c$$

for some constant c . ◇

Since $\mathbb{E}[\text{time}_M(x)] \leq |x|^c$, this immediately implies that the probability that M does not halt on an input is zero (although this is not stating that *all* computational paths halt. There could be a measure-zero set of paths that do not halt).

Claim 19.2. $CL \subseteq ZPP$.

Proof idea. Let M_{CL} be the catalytic machine. Initialise the catalytic tape with random bits. Showing that this yields a ZPP algorithm will appear as a problem in the next problem set. \square

19.2 Characterising ZPP

Theorem 19.3.

$$\text{ZPP} = \text{RP} \cap \text{coRP}.$$

Proof. We first show $\text{RP} \cap \text{coRP} \subseteq \text{ZPP}$.

Let $L \in \text{RP} \cap \text{coRP}$, with corresponding machines M_1 and M_2 , each with error probability at most $1/3$.

Our ZPP algorithm is:

On input x , run $M_1(x)$ and $M_2(x)$ with fresh randomness. If M_1 accepts, accept. If M_2 rejects, reject. Otherwise, repeat.

This algorithm never errs. For the running time, if $x \in L$, we repeat only if M_1 rejects, which happens with probability at most $1/3$. Thus,

$$\Pr[k \text{ iterations}] \leq (1/3)^k,$$

and hence the expected number of iterations is

$$\sum_{k \geq 1} k(1/3)^k = O(1).$$

A similar argument applies if $x \notin L$ using M_2 . Hence the expected running time is polynomial.

For the other direction, since ZPP is closed under complement, it suffices to show $\text{ZPP} \subseteq \text{RP}$.

Let M be a ZPP machine with expected running time n^5 . Define an RP machine M' that runs M for $10n^5$ steps and rejects if M has not accepted by then.

If $x \notin L$, M never accepts, so M' never accepts. If $x \in L$, by Markov's inequality,

$$\Pr[M \text{ runs for more than } 10n^5 \text{ steps}] \leq \frac{1}{10}.$$

Thus M' accepts with high probability. \square

19.3 Structural results for BPP

19.3.1 Adleman's Theorem

Theorem 19.4 (Adleman).

$$\text{BPP} \subseteq \text{SIZE}(\text{poly}).$$

Proof idea. Let $L \in \text{BPP}$ and fix an input length n . Suppose there exists a random string r_n such that $M(x, r_n)$ is correct for all $x \in \{0, 1\}^n$. Then we can hard-code r_n into a circuit to obtain a polynomial-size circuit for L .

We show such an r_n exists by amplification and counting. Amplify M so that its error probability is at most 2^{-n^2} .

Construct a matrix with rows indexed by inputs $x \in \{0, 1\}^n$ and columns indexed by random strings r . Mark an entry if $M(x, r)$ errs. Each row has at most a 2^{-n^2} fraction of marked entries.

Hence,

$$\#(\text{marked entries}) \leq 2^{-n^2} \cdot (\#\text{rows}) \cdot (\#\text{columns}).$$

Since $\#\text{rows} = 2^n$, we have

$$2^{-n^2} \cdot \#\text{rows} < 1,$$

and therefore

$$\#(\text{marked entries}) < \#\text{columns}.$$

Thus, there exists a column with no marked entries, giving a random string r_n on which M never errs.

19.3.2 $\text{BPP} \subseteq \Sigma_2 \cap \Pi_2$

Theorem 19.5 (Gacs-Sipser, Lautemann).

$$\text{BPP} \subseteq \Sigma_2 \cap \Pi_2.$$

Proof idea. Let $L \in \text{BPP}$ and M be a $\text{BPP}_{1-\delta, \delta}$ machine. Fix input x , and let R be the set of all possible random strings that can be used by M . Define

$$A(x) = \{r \in R : M(x, r) = 1\}.$$

If $x \in L$, then $|A(x)| \geq (1 - \delta)|R|$, and if $x \notin L$, then $|A(x)| \leq \delta|R|$.

Gacs' idea was to use hash functions: if $x \in L$, then $R \setminus A(x)$ is small, so there exists a hash function that maps all elements of $R \setminus A(x)$ such that there are no collisions.

Lautemann's argument uses translates. We claim there exist a_1, \dots, a_k such that for all $r \in R$,

$$r + a_1 \in A(x) \vee \dots \vee r + a_k \in A(x),$$

and we want this to hold iff $x \in L$.

If $x \notin L$, then $|A(x)| \leq \delta|R|$, and the total number of strings covered by these translates is at most $k|A(x)| \leq k\delta|R|$. Thus, if $k\delta < 1$, we cannot cover all of R .

If $x \in L$, we'll use a probabilistic argument; choose a_1, \dots, a_k independently and uniformly at random. For any fixed r , $\Pr[r \notin A(x) + a_i] \leq \delta$, so $\Pr[r \text{ not covered}] \leq \delta^k$ as the a_i 's are chosen independently and uniformly at random. By a union bound over all $r \in R$, the probability some r is left uncovered is at most $\delta^k |R|$.

Thus, if $\delta^k |R| < 1$, such a_1, \dots, a_k exist with which we can cover all of R .

Showing that $k\delta < 1$ and $\delta^k |R| < 1$ can be satisfied simultaneously is left as an exercise. We will revisit this proof in more detail in the next lecture.

Lecture 20

Lauteman's proof, Promise problems, and Introduction to IP

Scribe: Aindrila Rakshit

Topics covered in this lecture

- Lauteman's proof for $BPP \subseteq \Sigma_2 \cap \Pi_2$
- Promise problems
- Introduction to IP

20.1 Lauteman's proof for $BPP \subseteq \Sigma_2 \cap \Pi_2$

Theorem 20.1 (Gacs-Sipser, Lautemann). $BPP \subseteq \Sigma_2 \cap \Pi_2$.

Proof. Let $L \in BPP$ and M is the machine for it and let R be the all possible set of random strings that M uses, R could be $\{0, 1\}^*$.

For each $x \in \{0, 1\}^*$ define:

$$A_x := \{r \in \{0, 1\}^m : M(x, r) = 1\},$$

the set of random strings that cause M to accept x .

Then, after error reduction:

$$x \in L \implies |A_x| \geq (1 - \delta)|R|,$$

$$x \notin L \implies |A_x| \leq \delta|R|,$$

for $\delta = 2^{-\Omega(n)}$.

We construct the following Σ_2 sentence:

$$\psi(x) := \exists a_1, \dots, a_k \in \{0, 1\}^m \forall y \in \{0, 1\}^m : \bigvee_{i=1}^k M(x, y \oplus a_i) = 1.$$

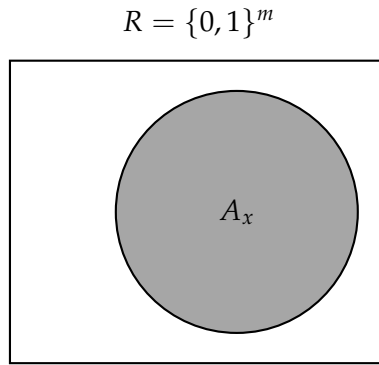


Figure 20.1: A_x as a subset of $R = \{0, 1\}^m$

Equivalently:

$$\psi(x) := \exists a_1, \dots, a_k \in \{0, 1\}^m \quad \forall y \in \{0, 1\}^m : \bigcup_{i=1}^k (A_x \oplus a_i) = \{0, 1\}^m.$$

where \oplus is the bitwise XOR.

This is definitely a Σ_2 sentence since it's of the form \exists, \forall followed by a polynomially computable circuit for small enough k .

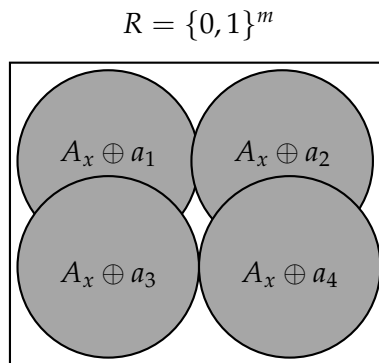


Figure 20.2: If A_x is large, small enough additive shifts of A_x cover the entire space

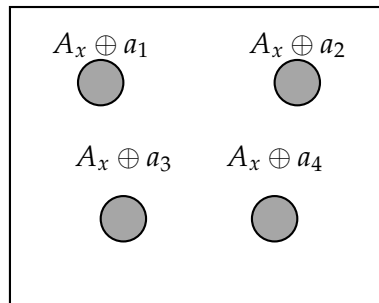


Figure 20.3: If A_x is tiny, small enough additive shifts of A_x will fail to cover the entire space

Claim 20.2. *If $\delta k \leq 1$, then $x \notin L \implies \psi$ is false.*

Proof. Each shift covers at most $\delta|R|$ elements, so:

$$\left| \bigcup_{i=1}^k (A_x \oplus a_i) \right| \leq k\delta|R|.$$

If $k\delta < 1$, then this is strictly less than $|R|$, so $x \notin L \implies \psi(x)$ is false. \square

Claim 20.3. *If $\delta^k|R| \leq 1$, then $x \in L \implies \psi$ is true.*

Proof. Pick a_1, \dots, a_k independently and uniformly at random.

Fix any $y \in \{0, 1\}^m$. Then:

$$\mathbb{P}_{a_1}[M(x, y - a_1) = 0] = \mathbb{P}_{a_1}[y - a_1 \notin A_x] \leq \delta.$$

Since a_1, \dots, a_k are independently chosen:

$$\mathbb{P}_{a_1, \dots, a_k}[M(x, y - a_1) = 0, \dots, M(x, y - a_k) = 0] \leq \delta^k.$$

By union bound over all y :

$$\mathbb{P}_{a_1, \dots, a_k}[\exists y : M(x, y - a_1) = 0, \dots, M(x, y - a_k) = 0] \leq \delta^k|R| \leq 1$$

So, via probabilistic method, then there exists a choice of shifts covering all y with positive probability, so $x \in L \implies \psi(x)$ is true. \square

\therefore If $\delta^k|R| \leq 1$ and $\delta k \leq 1$, we are done.

So, if we started with a BPP machine for L with error $\leq 1/3$ and l random bits, we can boost the success probability to $\geq 1 - \delta$, so that:

$$\delta = 2^{-\ell^2}, \quad m = \ell^3, \quad |R| = 2^{\ell^3}$$

Choosing $k = O(\ell)$ ensures:

$$k\delta < 1 \quad \text{and} \quad |R|\delta^k < 1.$$

$\therefore L \in \Sigma_2^P$.

So, $\text{BPP} \subseteq \Sigma_2$. Since BPP is closed under complementation, $\text{BPP} \subseteq \Sigma_2 \cap \Pi_2$. \square

Remark 20.4. *These are semantic classes. There are no known complete problems.* \diamond

20.2 Promise Problems

A *language* is a pair $(L_{\text{yes}}, L_{\text{no}})$ such that $L_{\text{yes}}, L_{\text{no}} \subseteq \{0, 1\}^*$ and

$$L_{\text{yes}} \cup L_{\text{no}} = \Sigma^*$$

$$L_{\text{yes}} \cap L_{\text{no}} = \emptyset.$$

Definition 20.5. A promise problem is a pair $(L_{\text{yes}}, L_{\text{no}})$ such that $L_{\text{yes}}, L_{\text{no}} \subseteq \{0, 1\}^*$ and

$$L_{\text{yes}} \cap L_{\text{no}} = \emptyset.$$

◇

So, L_{yes} is the set of *yes* instances, L_{no} is the set of *no* instances, and anything outside $L_{\text{yes}} \cup L_{\text{no}}$ is a *don't care* instance.

Definition 20.6. A machine M solves $(L_{\text{yes}}, L_{\text{no}})$ if:

$$x \in L_{\text{yes}} \implies M(x) = 1,$$

$$x \in L_{\text{no}} \implies M(x) = 0,$$

Else, ... *don't care*

◇

20.3 The Circuit Acceptance Probability Problem (CAP)

Definition 20.7. Fix $\varepsilon > 0$. Define CAP_ε :

$$L_{\text{yes}} = \{\langle C \rangle : \Pr_r[C(r) = 1] \geq \frac{1}{2} + \varepsilon\},$$

$$L_{\text{no}} = \{\langle C \rangle : \Pr_r[C(r) = 1] \leq \frac{1}{2} - \varepsilon\}.$$

◇

Theorem 20.8. $\text{CAP}_\varepsilon \in \text{PromiseBPP}$.

Proof. We construct a machine M which takes the encoding of $\langle C \rangle$ and does the following:

Sample r_1, \dots, r_k independently and compute $C(r_i)$.

Accept iff a strict majority accept.

Use similar Chernoff bounds type arguments used in boosting $|BPP$ machines.

Set $k = O(\varepsilon^{-2} \log(1/\delta))$.

Then:

- If $C \in L_{\text{yes}}$, accept w.p. $\geq 1 - \delta$
- If $C \in L_{\text{no}}$, accept w.p. $\leq \delta$

$\therefore C \in \text{PromiseBPP}$.

□

20.4 Reductions for Promise Problems

Definition 20.9. A reduction $(A_{\text{yes}}, A_{\text{no}}) \leq_p (B_{\text{yes}}, B_{\text{no}})$ is a polytime computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that:

$$\forall x \in A_{\text{yes}} \implies f(x) \in B_{\text{yes}},$$

$$\forall x \in A_{\text{no}} \implies f(x) \in B_{\text{no}}.$$

◇

Theorem 20.10. $\text{CAP}_{1/3}$ is PromiseBPP-complete.

Proof. Let $(A_{\text{yes}}, A_{\text{no}}) \in \text{PromiseBPP}$.

So, there is some M such that:

$$x \in A_{\text{yes}} \implies \mathbb{P}_r[M(x, r) = 1] \geq \frac{2}{3},$$

$$x \in A_{\text{no}} \implies \mathbb{P}_r[M(x, r) = 1] \leq \frac{1}{3}.$$

Let $f : \Sigma_A^* \rightarrow \Sigma_{\text{CAP}}^*$ be a polytime computable function.

Thus, $x \mapsto C_x$ is a valid reduction to $\text{CAP}_{1/3}$, where C_x is the encoding of the machine $M(x, r)$ and takes as input the random bits r .

□

20.5 Introduction to Interactive Proofs

Recall the prover-verifier definition of NP:

A language $L \in \text{NP}$ if there exists a polytime verifier V such that

$$x \in L \implies \exists w \in \{0, 1\}^{\text{poly}(n)} : V(x, w) = 1$$

$$x \notin L \implies \forall w \in \{0, 1\}^{\text{poly}(n)} : V(x, w) = 0$$

So, we can think of this as some sort of a half round conversation between a prover P and a verifier V where to check wheter an input $x \in L$ a prover sends a message w and the verifier checks if its a valid witness:

$$P \xrightarrow{x} V$$

$$x \in L \implies \exists \text{Prover} : V \leftarrow P(x) = 1$$

$$x \notin L \implies \forall \text{Prover} : V \leftarrow P(x) = 0$$

An *interactive proof* allows a prover and a verifier to exchange messages in order to convince the verifier of the validity of a statement.

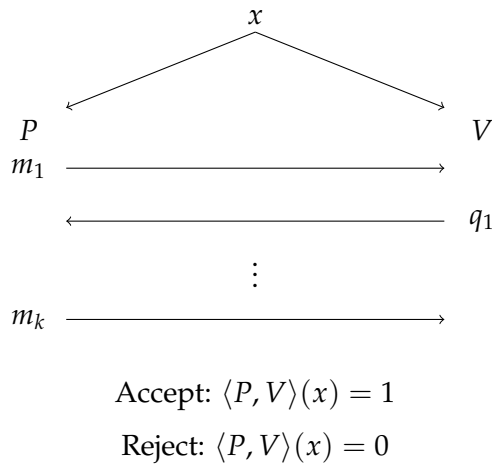


Figure 20.4: An interactive proof protocol between prover P and verifier V

20.5.1 Deterministic Interactive Proofs

Theorem 20.11. $\text{DIP} = \text{NP}$.

Proof. If the verifier is deterministic, the prover can send the entire transcript.

The verifier checks consistency in polynomial time, so this is exactly NP. □

Definition 20.12 (IP). A language L is in IP if there exists a randomized polytime verifier V such that:

$$\begin{aligned}
 x \in L &\implies \exists P : \Pr[V \leftrightarrow P(x) = 1] \geq \frac{2}{3}, \\
 x \notin L &\implies \forall P : \Pr[V \leftrightarrow P(x) = 1] \leq \frac{1}{3}.
 \end{aligned}$$

◇

20.6 Major Result

Theorem 20.13 (Shamir). $\text{IP} = \text{PSPACE}$.

Theorem 20.14 (BFL). $\text{MIP} = \text{NEXP}$

Lecture 21

Interactive Protocols

Scribe: Ananya Ranade

Topics covered in this lecture

- Interactive Protocol (IP)
- Graph Non-Isomorphism
- $IP \subseteq PSPACE$
- Introduction to LFKN Protocol

21.1 Interactive Protocols

An interactive protocol between a prover and verifier is a sequence of messages sent back and forth, at the end of which the verifier either accepts or rejects the input. We would want the verifier to be polytime but the prover can be all powerful, that is has no restriction on computational resources like time, space, randomness, non-determinism, etc. We would also require that the number of rounds and each message is also polynomial in the size of the input, so that the total time taken by verifier is polynomial.

At any point, the partial transcript, that is the set of messages exchanged so far is visible to both, the verifier and prover.

Also, if a language L has an interactive protocol, we would want the verifier to make an error with probability atmost $\frac{1}{3}$.

Deterministic interactive protocols: What if we restrict the verifier to be deterministic ? Such protocols are called deterministic interactive protocols. Last time we saw that in that case $DIP = NP$ as the prover can just send the entire transcript in one go.

If we allow verifier to be randomized, many more languages can have interactive protocols as we will see next.

21.2 Graph Non-Isomorphism (GNI)

Definition 21.1 (Graph isomorphism). *Two graphs G_1, G_2 are said to be isomorphic (denoted by $G_1 \equiv G_2$) if they both have the same number of vertices (say n) and there exists a bijection $\sigma : [n] \rightarrow [n]$ such that $(i, j) \in E(G_1)$ if and only if $(\sigma(i), \sigma(j)) \in E(G_2)$.* \diamond

Let $\text{GRAPHISO} = \{(G_1, G_2) : G_1 \equiv G_2\}$, and let GRAPHNONISO be the complement language.

Lemma 21.2. $\text{GRAPHNONISO} \in \text{IP}$.

Proof. (This is often called the coke vs pepsi test.) The protocol proceeds as follows. The verifier randomly picks i from $\{1, 2\}$ and also a random permutation $\sigma \in S_n$. Let $H = \sigma(G_i)$ be the permuted graph. The verifier sends this graph H to the prover (without revealing i or σ). The prover is expected to correctly recognize the graph and return $i' \in \{1, 2\}$. The verifier accepts if $i = i'$.

To improve the success probability, we could repeat this test a few times and if and only if the prover correctly identifies the graph every time.

So, if we have an instance in GRAPHNONISO , prover can always figure out which graph was permuted by going through all the permutations and answer correctly. If it is not in GRAPHNONISO , the graph H is equally likely to have been generated from G_1 and G_2 . Thus, the best the prover can do is guess randomly. (This is because if σ is some permutation such that $\sigma(G_1) = H$, and $\tau(G_1) = G_2$, then $\sigma \circ \tau(G_2) = H$ and vice versa. So, the number of permutations which make the graph isomorphic to H is same for both G_1 and G_2 .) So, P is correct with probability atmost $\frac{1}{2}$. \square

21.3 Formal definition of IP

Definition 21.3 (The class IP). *A language $L \in \text{IP}$ if there is a probabilistic polynomial time verifier V such that:*

$$\begin{aligned} x \in L &\iff \exists P : \Pr[V \leftrightarrow P(x) = \text{accept}] \geq \frac{2}{3} \\ x \notin L &\iff \forall P : \Pr[V \leftrightarrow P(x) = \text{accept}] \leq \frac{1}{3} \end{aligned}$$

\diamond

Note : Verifier coin tosses may be private or public.

Prover, wlog, is deterministic: Suppose the prover was randomised. So, at any point if \tilde{P} is the partial transcript, and r_p is the randomised string based on which P answers, then the prover needs to ensure that the expected probability of V accepting over all choices of r_p is atleast $\frac{2}{3}$. So, there is some r_p for which it is atleast $\frac{2}{3}$. So, P will just answer according to \tilde{P} and r_p . So, it is not necessary for P to be randomised.

21.3.1 IP is in PSPACE

A language $L \in \text{IP}$ if there is a probabilistic polynomial time verifier V such that $\Pr_{r_v}[P \leftrightarrow V(x) = \text{accept}] \geq \frac{2}{3}$ for all $x \in L$ and provers P , and $\Pr_{r_v}[P \leftrightarrow V(x) = \text{accept}] \leq \frac{1}{3}$ for all $x \notin L$ and provers P . So, if we can find $\max_P \Pr_{r_v}[P \leftrightarrow V(x) = \text{accept}]$, where P is the set of all possible provers, then we can decide if $x \in L$. Computing this value is in PSPACE.

Suppose we have at most r rounds protocol for inputs of length n . We can assume it is of r rounds across all choices, as it can just go till r rounds even if the verifier knows whether to accept or reject earlier and accept or reject only after r rounds. Let the transcript be $q_1, a_1, q_2, a_2, \dots, q_r, a_r$, where q_i is the i^{th} message of V and a_i is the i^{th} message of P .

We can represent all possible transcripts as a tree, where each path represents the possible transcript. We will do a bottom up traversal of this. For a particular path q_1, a_1, \dots, q_r , there are various choices of a_r , each with some acceptance probability. We only remember the maximum value of it. Then, we calculate the minimum of these maximum values of a_r for each possible answer a_r along the path q_1, \dots, q_r , and only remember that. And so on, we continue to get the maximum. This is like the pebbling problem and since r is $\text{poly}(n)$, it can be done in PSPACE.

Corollary 21.4. $\text{IP} \subseteq \text{PSPACE}$

21.4 The Lund-Fortnow-Karloff-Nisan (LFKN) protocol

Now, we will show that $\text{P}^{\#\text{P}} \subseteq \text{IP}$. $\#\text{P}$ asks for the number of solutions or accepting paths for a polynomial-time non-deterministic Turing machine, and $\text{P}^{\#\text{P}}$ is the class of all languages for which this can be found in poly time. By Toda's theorem ($\text{PH} \in \text{P}^{\#\text{P}}$) we can conclude $\text{PH} \subseteq \text{IP}$.

Consider the following language (which we will call $\#\text{SAT}$ although historically that is a not a language):

$$\#\text{SAT} = \{(\Phi, k) : \Phi \text{ is a 3-CNF and has exactly } k \text{ sat. assignments}\}$$

We will work with a finite field \mathbb{F} (soon to be seen why) and try to express algebraically the claim that Φ has exactly k satisfying assignments.

21.4.1 Arithmetisation

For a clause of the form $C_i = x_1 \vee x_2 \vee \overline{x_3}$, we can associate the polynomial $F_i(x_1, x_2, x_3) = 1 - (1 - x_1)(1 - x_2)x_3$. Note that for all $\{0, 1\}$ values given to x_1, x_2, x_3 , the polynomial F_i agrees with the boolean value of C_i . The advantage of arithmetisation is that we have the ability to evaluate F_i at non-boolean points also.

For a CNF $\Phi = C_1 \wedge \dots \wedge C_m$, we can define $F(x_1, \dots, x_n) = F_1 \dots F_m$. Thus, for any $a \in \{0, 1\}^n$, we have $F(a_1, \dots, a_n) = \mathbb{1}[a \text{ satisfies } \Phi]$. Thus, the Prover's claim that Φ has exactly k satisfying assignments can be equivalently expressed as

$$\sum_{a_1=0,1} \dots \sum_{a_n=0,1} F(a_1, \dots, a_n) = k.$$

In the next class we will see the complete protocol.

Lecture 22

IP = PSPACE

Scribe: Chandralekha P

Recap: In the previous lecture, we saw what IP is:

Definition 22.1 (IP). A language $L \in \text{IP}$ if there exists a probabilistic polynomial-time verifier V such that:

- **Completeness:** $x \in L \Rightarrow \exists P : \Pr_r[P \leftrightarrow V(x) = \text{acc}] \geq \frac{2}{3}$
- **Soundness:** $x \notin L \Rightarrow \forall P : \Pr_r[P \leftrightarrow V(x) = \text{acc}] \leq \frac{1}{3}$

◇

We also saw the following inclusions:

- $\text{NP} \subseteq \text{IP}$
- $\text{GNI} \in \text{IP}$ because of the Coke-Pepsi test.

In this lecture, we will show that IP is precisely the class PSPACE.

1. $\#\text{SAT} \in \text{IP}$ [Lund–Fortnow–Karloff–Nisan]

2. $\text{TQBF} \in \text{IP}$ [Shamir]

$\Rightarrow \text{IP} = \text{PSPACE}$

22.1 #SAT \in IP via the Sum-Check Protocol

The language #SAT is a counting language, but we will look at it in the following way and show the existence in IP

$$L = \{(\Phi, k) : \Phi \text{ is a 3-CNF with exactly } k \text{ satisfying assignments}\}.$$

Write $\Phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ over n variables. Last class, we saw the *arithmetisation* of Φ . To recap, this is how we arithmetize a clause:

$$C_1 = x_1 \vee x_2 \vee x_3 \quad \rightsquigarrow \quad \tilde{C}_1 = 1 - (1 - x_1)(1 - x_2)(1 - x_3).$$

Define the polynomial

$$F(x_1, \dots, x_n) = \prod_{i=1}^m \tilde{C}_i(\bar{x}).$$

This is an n -variable polynomial of degree $\leq 3m$. This polynomial is such that, a valuation (a_1, \dots, a_n) evaluates to 1 if and only if it is a satisfying assignment. It evaluates to 0 otherwise.

Remark. *The Verifier cannot expand the whole polynomial since it may have exponentially many terms. But the Verifier can, however, evaluate F at any chosen $(r_1, \dots, r_n) \in \mathbb{F}^n$ from the above expression.* \diamond

So, we want to do something to check the number of satisfying assignments. An equivalent claim is the following:

$$\sum_{a_1=0}^1 \sum_{a_2=0}^1 \dots \sum_{a_n=0}^1 F(a_1, \dots, a_n) = k.$$

Because, the above sum precisely tells the number of satisfying assignments. Now, we want to construct an interactive protocol to convince the verifier of such a claim.

22.1.1 The Sum-Check Protocol: Attempt 1

The idea is to reduce the n -dimensional sum to lower-dimensional checks one variable at a time.

$$\text{Current claim: } \sum_{a_1=0,1} \sum_{a_2=0,1} \dots \sum_{a_n=0,1} F(a_1, \dots, a_n) = k_0$$

- Prover sends a polynomial $F_1(x_1)$ (which is supposed to be $\sum_{a_2} \dots \sum_{a_n} F(x_1, a_2, \dots, a_n)$).
- Verifier rejects if $F_1(0) + F_1(1) \neq k_0$. Else, verifier chooses a random $b_1 \in \{0, 1\}$ and sends that to prover. The prover now has to prove the claim

$$\sum_{a_2=0,1} \dots \sum_{a_n=0,1} F(b_1, a_2, \dots, a_n) = k_1$$

$$\text{where } k_1 = F_1(b_1).$$

and we repeat until we get to $F_n(b_n) = F(b_1, \dots, b_n)$. This is something that the verifier can evaluate themselves and check if it indeed equal to the claimed k_n .

Completeness If $(\Phi, k) \in L$, the honest prover sends the correct polynomials, so

$$\Pr_r[P \leftrightarrow V(x) = acc] = 1.$$

Soundness Analysis If $(\Phi, k) \notin L$ (say the number of satisfying assignments of $\Phi = k' \neq k$), then if the prover wants to make the verifier accept, he *has* to lie and send a different polynomial $\tilde{F}_1(x_1)$ in the first message. (This is because, the correct polynomial $F_1(x_1)$ satisfies $F_1(0) + F_1(1) = k' \neq k$; the Verifier will reject immediately if the prover sends $F_1(x_1)$.)

The verifier then chooses a random $b_1 \in \{0, 1\}$. If it so happens that $F_1(b_1) = \tilde{F}_1(b_1)$ then note that we are now left with the claim

$$\sum_{a_2=0}^1 \cdots \sum_{a_n=0}^1 F(b_1, a_2, \dots, a_n) = F_1(b_1) = \tilde{F}_1(b_1)$$

which is actually a true claim! Thus the provers can now start answering honestly and make the verifier accept.

Therefore, the only way the prover gets caught is if somehow the verifier choose a b_i that witnesses the wrong value at every single round. The probability of this is atmost $1/2^n$, which is way too small for the soundness guarantee of IP.

The idea is to fix this by modifying the protocol, and choosing $b_i \in_R \mathbb{F}_p$ rather than just from $\{0, 1\}$.

To fix this, instead of working with just $\{0, 1\}$, we can work with $b_i \in_R \mathbb{F}_p$.

22.1.2 The (sound) Sum-Check Protocol

Current claim: $\sum_{a_1=0,1} \sum_{a_2=0,1} \cdots \sum_{a_n=0,1} F(a_1, \dots, a_n) = k_0$

- Prover sends a polynomial $F_1(x_1)$ (which is supposed to be $\sum_{a_2} \cdots \sum_{a_n} F(x_1, a_2, \dots, a_n)$).
- Verifier rejects if $F_1(0) + F_1(1) \neq k_0$. Else, verifier chooses a random $b_1 \in \mathbb{F}_p$ and sends that to prover. The prover now has to prove the claim

$$\sum_{a_2=0,1} \cdots \sum_{a_n=0,1} F(b_1, a_2, \dots, a_n) = k_1$$

where $k_1 = F_1(b_1)$.

and we repeat until we get to $F_n(b_n) = F(b_1, \dots, b_n)$. This is something that the verifier can evaluate themselves and check if it indeed equal to the claimed k_n .

By doing this, the correctness argument does not change. Now, we should see what happens for the soundness.

Lemma 22.2. *If $(\Phi, k) \notin L$, then for all provers P ,*

$$\Pr_r[P \leftrightarrow V(x) = \text{rej}] \geq \left(1 - \frac{3m}{p}\right)^n.$$

Proof. We will do this by induction on n .

Base case ($n = 1$): Since for this the verifier itself can do the computation, there is no way for the prover to lie and hence rejection probability is 1.

Inductive step: The prover has to send some $\tilde{F}_1 \neq F_1$. The verifier chooses $b_1 \in_R \mathbb{F}_p$. We expand:

$$\begin{aligned} \Pr[V \text{ rejects}] &= \Pr_{b_1}[\tilde{F}_1(b_1) = F_1(b_1)] \cdot \Pr[V \text{ rej} \mid \tilde{F}_1(b_1) = F_1(b_1)] \\ &\quad + \Pr_{b_1}[\tilde{F}_1(b_1) \neq F_1(b_1)] \cdot \Pr[V \text{ rej} \mid \tilde{F}_1(b_1) \neq F_1(b_1)] \\ &\geq \Pr_{b_1}[\tilde{F}_1(b_1) \neq F_1(b_1)] \cdot \left(1 - \frac{3m}{p}\right)^{n-1}. \end{aligned}$$

Since $\tilde{F}_1 - F_1$ is a nonzero polynomial of degree $\leq 3m$ over \mathbb{F}_p , it has at most $3m$ roots, so

$$\Pr_{b_1}[\tilde{F}_1(b_1) \neq F_1(b_1)] \geq 1 - \frac{3m}{p}.$$

Therefore,

$$\Pr[V \text{ rejects}] \geq \left(1 - \frac{3m}{p}\right)^n$$

□

Using the approximation $(1 - \frac{3m}{p})^n \approx 1 - \frac{3mn}{p}$ (when $3mn \leq p$), if we choose $p > 9mn$ (a prime of this size) then we ensure soundness error $\leq \frac{1}{3}$.

$$\Pr[P \leftrightarrow V(x) = acc] \leq \frac{1}{3} \quad \text{if } p > 9mn.$$

The above is precisely what we wanted, and hence this protocol is indeed sound.

Some consequences

There is a theorem by Toda that says the following:

Theorem 22.3 (Toda). $P^{\#\text{SAT}} \supseteq \text{PH}$.

From the above theorem, we get the following corollary from our protocol:

Corollary 22.4. $\text{PH} \subseteq \text{IP}$.

22.2 TQBF \in IP and Shamir's Theorem

22.2.1 Quantified Boolean Formulas

Can we handle fully quantified expressions similarly? Consider:

$$\exists x_1 \forall x_2 \exists x_3 \cdots \exists x_n \Phi(x_1, \dots, x_n).$$

This is equivalent (after arithmetization) to:

$$\sum_{a_1} \prod_{a_2} \cdots \sum_{a_n} F(a_1, \dots, a_n) \geq 1.$$

Problem: If we naively ask the prover for

$$F_1(x_1) = \prod_{a_2} \sum_{a_3} \cdots \sum_{a_n} F(x_1, a_2, \dots, a_n),$$

the degree blows up: $\deg F_1 \leq 2^{n/2} \cdot 3m$. The prover cannot send $F_1(x_1)$ efficiently.

22.2.2 Degree Reduction via Linearization

Key idea: We only evaluate at $\{0, 1\}$ values, so $x_i^2 = x_i$ on Boolean inputs. Replace any occurrence of x_i^2 with x_i , effectively “linearizing” in x_i .

Define the *linearization operator* L_i :

$$L_i G(x_1, \dots, x_n) = x_i \cdot G(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) + (1 - x_i) \cdot G(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n).$$

The resulting $\tilde{G}(x_1, \dots, x_n)$ satisfies:

- $\deg_{x_j} \tilde{G} \leq \deg_{x_j} G$ for $j \neq i$,
- $\deg_{x_i} \tilde{G} \leq 1$.

To keep the notation similar, we will define the *summation operator* Σ_i and the *product operator* Π_i :

$$\Sigma_i: g(x_1, \dots, x_n) \mapsto g(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n) + g(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n),$$

$$\Pi_i: g(x_1, \dots, x_n) \mapsto g(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n) \cdot g(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n).$$

Example:

$$\begin{aligned} F(x_1, x_2, x_3) &= x_1^2 x_2 + x_1 x_3^2 + x_2 + x_3^3 + x_2 x_3 \\ \Pi_3 F &= (x_1^2 x_2 + x_2) \cdot (x_1^2 x_2 + x_1 + 2x_2 + 1) \\ &= x_1^4 x_2^2 + x_1^3 x_2 + 3x_1^2 x_2^2 + x_1^2 x_2 + x_1 x_2 + 2x_2^2 + x_2 \\ L_2 \Pi_3 F &= x_1^4 x_2 + x_1^3 x_2 + 4x_1^2 x_2 + x_1 x_2 + 3x_2 \\ L_1 L_2 \Pi_3 F &= 7x_1 x_2 + 3x_2 \\ \Sigma_2 L_1 L_2 \Pi_3 F &= 7x_1 + 3 \end{aligned}$$

22.2.3 Protocol for TQBF

The main idea is to start with an expression of the form

$$\Sigma_1 \Pi_2 \Sigma_3 \Pi_4 F(x_1, x_2, x_3, x_4) = k > 0$$

and convert this to

$$\Sigma_1 L_1 \Pi_2 L_1 L_2 \Sigma_3 L_1 L_2 L_3 \Pi_4 L_1 L_2 L_3 L_4 F(x_1, x_2, x_3, x_4) = k' > 0$$

The linearisation operations are to ensure that the degrees are brought back down to $O(n)$ by enforcing multilinearity on all variables the polynomial currently depends on. Note that if the QBF was indeed true, then the RHS even with the linearisation operators would be some positive value. This is going to be the Prover's claim.

A specific example

Let us consider the QBF $\exists x_1 \forall x_2 \Phi(x_1, x_2)$. With the linearisations, we convert this to the expression

$$\Sigma_1 L_1 \Pi_2 L_1 L_2 F(x_1, x_2)$$

The prover is going to claim that the above expression equals some integer $k > 0$.

Here is the full trace for the above example.

Initial claim: $\Sigma_1 L_1 \Pi_2 L_1 L_2 F(x_1, x_2) = k_0$

1. (The quantifier being addressed is Σ_1) Prover sends $S_1(x_1)$ (purportedly equal to $L_1 \Pi_2 L_1 L_2 F$) to the prover.
2. Verifier checks if $\Sigma_1 S_1(x) = S_1(0) + S_1(1) = k_0$ and rejects if not. Else, the verifier chooses a random $\beta_1 \in \mathbb{F}_p$ as an evaluation for x_1 .

Prover now has to assert that $L_1 \Pi_2 L_1 L_2 F(x_1, x_2)$ at $x_1 = \beta_1$ equals $k_1 = S_1(\beta_1)$.

3. (The quantifier being addressed is L_1) Prover sends $S_2(x_1)$ (purportedly equal to $\Pi_2 L_1 L_2 F$) to the verifier.
- (Note that we already have assigned $x_1 = \beta_1$. Nevertheless, the Prover sends the polynomial with x_1 remaining unset.)

4. Verifier checks if $L_1 S_2(x_1)$ at $x_1 = \beta_1$ equals k_1 . That is, checks if

$$L_2 S_2(x_1) |_{x_1=\beta_1} = \beta_1 S_2(1) + (1 - \beta_1) S_2(0) \stackrel{?}{=} k_1.$$

If the test fails, the Verifier rejects immediately. Else, the verifier sends a random $\beta'_1 \in_R \mathbb{F}_p$ and assigns that to x_1 .

The prover now has to assert that $\Pi_2 L_1 L_2 F(x_1, x_2)$ at $x_1 = \beta'_1$ equals $S_2(\beta'_1) = k_2$.

5. (The quantifier being addressed is Π_2) Prover sends $S_3(x_2)$ (purportedly equal to $L_1 L_2 F$ at $x_1 = \beta'_1$) to the verifier.
6. Verifier checks if $\Pi_2 S_3(x_2) = S_3(0) \cdot S_3(1) = k_2$ and rejects if not. Else, the verifier chooses a random $\beta_2 \in \mathbb{F}_p$ and assigns that $x_2 = \beta_2$.

The prover now has to assert that $L_1 L_2 F(x_1, x_2)$ at $x_1 = \beta'_1, x_2 = \beta_2$ equals $S_3(\beta_2) = k_3$.

7. (The quantifier being addressed is L_1) Prover sends $S_4(x_1)$ (purportedly equal to L_2F at $x_2 = \beta_2$) to the verifier.

(Note that we already have assigned $x_1 = \beta'_1$. Nevertheless, the Prover sends the polynomial with x_1 remaining unset.)

8. Verifier checks if $L_1S_4(x_1)$ at $x_1 = \beta'_1$ equals k_3 . That is,

$$L_1S_4(x_1) |_{x_1=\beta'_1} = \beta'_1S_4(1) + (1 - \beta'_1)S_4(0) \stackrel{?}{=} k_3.$$

and rejects if not. Else, the verifier chooses a random $\beta''_1 \in \mathbb{F}_p$ and assigns that $x_1 = \beta''_1$.

The prover now has to assert that $L_2F(x_1, x_2)$ at $x_1 = \beta''_1, x_2 = \beta_2$ equals $S_4(\beta''_1) = k_4$.

9. (The quantifier being addressed is L_2) Prover sends $S_5(x_2)$ (purportedly equal to F at $x_1 = \beta''_1$) to the verifier.

(Note that we already have assigned $x_2 = \beta_2$. Nevertheless, the Prover sends the polynomial with x_2 remaining unset.)

10. Verifier checks if $L_2S_5(x_2)$ at $x_2 = \beta_2$ equals k_4 . That is,

$$L_2S_5(x_2) |_{x_2=\beta_2} = \beta_2S_5(1) + (1 - \beta_2)S_5(0) \stackrel{?}{=} k_4.$$

and rejects if not. Else, the verifier chooses a random $\beta'_2 \in \mathbb{F}_p$ and assigns that $x_2 = \beta'_2$.

The prover now has to assert that $F(x_1, x_2)$ at $x_1 = \beta''_1, x_2 = \beta'_2$ equals $S_5(\beta'_2) = k_5$.

11. At this point, the Verifier can check themselves if $F(\beta''_1, \beta'_2) = k_5$. The verifier accepts if this is indeed the case and rejects otherwise.

Thus, the quantifier elimination happens exactly as earlier with one minor modification for L_i quantifiers (as this is a quantifier that leaves the variable x_i live, unlike Σ_i and Π_i). Since the original claim has the RHS as a number, whenever we encounter an L_i quantifier we must have already set a certain value for x_i . Each round the verifier does the consistency check with the previous polynomial, and chooses a new value for the variable in focus (and in case of L_i , this results in choosing a new value for that variable).

Soundness analysis: The soundness argument is exactly the same. All the polynomials involved have degree at most $\max(3mn, 2n)$. If the prover has a wrong claim to prove, then the prover *has* to send a wrong polynomial in each round. Any the probability of the verifier choosing a value where the wrong polynomial agrees with the true polynomial is once again bounded by the Schwartz-Zippel lemma.

By an analogous inductive argument:

$$\Pr[\text{Verifier rejects}] \geq \frac{2}{3} \quad \text{if } p \approx 9mn.$$

Hence, we get the following theorem.

Theorem 22.5 (Shamir, 1992).

$$\text{IP} = \text{PSPACE}.$$

Lecture 23

Public Coin Protocols, AM, MA

Scribe: Hrishikesh Saikia

Topics covered in this lecture

- A public coin protocol for Graph Non-Isomorphism
- The Goldwasser-Sipser Set-Size protocol
- Classes AM and MA
- $MA \subseteq AM$

In this lecture, we will see a public coin protocol for Graph Non-Isomorphism and introduce the classes AM and MA.

23.1 Graph Non-Isomorphism using public coins

Recall the Graph Non-Isomorphism protocol we saw earlier: On input (G_1, G_2) :

1. Verifier (V): Samples $\sigma \in_R S_n$, $i \in_R \{1, 2\}$, computes $H = \sigma(G_i)$, and sends H to the prover P .
2. Prover (P): Sends $i' \in \{1, 2\}$ to V .
3. Verifier (V): Accepts if $i = i'$, and rejects otherwise.

Note that the random coins tossed by the verifier are all private to the verifier and never revealed to the prover. If either i or σ is revealed, then the prover can cheat easily and soundness breaks. We will now construct a protocol for the same language in which all the random coins tossed by the verifier are revealed to the prover. Infact, the verifier's messages are only going to be random coins.

23.1.1 Towards a public coin protocol

Define the sets

$$\mathcal{M} = \{H : \exists \sigma \in S_n, i \in \{1, 2\}, H = \sigma(G_i)\}$$

$$\mathcal{M}_G = \{H : \exists \sigma \in S_n, H = \sigma(G)\}$$

Note that if $G_1 \cong G_2$, then $\mathcal{M} = \mathcal{M}_{G_1} = \mathcal{M}_{G_2}$ as any two isomorphic graphs are related by a permutation of the vertices. On the other hand, if $G_1 \not\cong G_2$, then $\mathcal{M} = \mathcal{M}_{G_1} \sqcup \mathcal{M}_{G_2}$ as \mathcal{M}_{G_1} and \mathcal{M}_{G_2} are disjoint (there is no permutation mapping G_1 and G_2), and every element of \mathcal{M} is in at least one of \mathcal{M}_{G_1} or \mathcal{M}_{G_2} . So \mathcal{M} is probably “much larger” when $G_1 \not\cong G_2$ and we can possibly build up on this...or can we?

Consider a concrete example, where $G_1 = C_4$ (cycle on 4 vertices) and $G_2 = K_4$ (complete graph on 4 vertices). Then we can verify that \mathcal{M}_{G_1} has size 3, while \mathcal{M}_{G_2} has size 1. Extending this, if we take G_1 to be, say, k copies of C_4 and G_2 to be $k - 1$ copies of C_4 and 1 copy of K_4 , then \mathcal{M}_{G_1} and \mathcal{M}_{G_2} aren't really “much” different in size. To fix this, we consider the set of automorphisms of the graphs. For a graph G (on n vertices), the set of automorphisms of G is defined as

$$\text{Aut}(G) = \{\sigma \in S_n : \sigma(G) = G\}$$

Observation 23.1. $|\mathcal{M}_G| \cdot |\text{Aut}(G)| = n!$

Proof. For each $\sigma \in S_n$, define the set $T_\sigma = \{\pi \in S_n : \pi(G) = \sigma(G)\}$. Note that $\pi \in T_\sigma \iff \pi(G) = \sigma(G) \iff \pi^{-1}\sigma \in \text{Aut}(G)$. So $|T_\sigma| = |\text{Aut}(G)|$.

Consider $\sigma_1, \sigma_2 \in S_n$. If $\sigma \in T_{\sigma_1} \cap T_{\sigma_2}$, then $\sigma(G) = \sigma_1(G) = \sigma_2(G)$. So $\pi \in T_{\sigma_1} \iff \pi(G) = \sigma_1(G) \iff \pi(G) = \sigma_2(G) \iff \pi \in T_{\sigma_2}$. Thus $T_{\sigma_1} = T_{\sigma_2}$. So the distinct T_σ partition S_n into disjoint sets each of size $|\text{Aut}(G)|$. Moreover, from the definition of \mathcal{M}_G , it is easy to see that these partitions are in one-to-one correspondence with elements of \mathcal{M}_G (in particular, $H \in \mathcal{M}_G$ maps to T_σ such that $\sigma(G) = H$). Thus $|\mathcal{M}_G| \cdot |\text{Aut}(G)| = |S_n| = n!$. \square

Define $S_G = \{(H, \tau) : H \in \mathcal{M}_G \text{ and } \tau \in \text{Aut}(H)\}$ and $S(G_1, G_2) = S_{G_1} \cup S_{G_2}$.

Observation 23.2. $|S_G| = n!$

Proof. Consider $H \in \mathcal{M}_G$. Let σ be such that $\sigma(G) = H$. Note that $\pi \in \text{Aut}(H) \iff \pi(H) = H \iff (\pi\sigma)(G) = H \iff \pi\sigma \in T_\sigma$ (where T_σ is defined as in the proof of Observation 23.1). Thus $|\text{Aut}(H)| = |T_\sigma| = |\text{Aut}(G)|$. So $|S_G| = |\mathcal{M}_G| \cdot |\text{Aut}(G)| = n!$. \square

Observation 23.3. $|S(G_1, G_2)| = \begin{cases} n! & \text{if } G_1 \cong G_2 \\ 2n! & \text{if } G_1 \not\cong G_2 \end{cases}$

Proof. If $G_1 \cong G_2$, then $S_{G_1} = S_{G_2} \implies |S(G_1, G_2)| = |S(G_1)| = n!$ (from Observation 23.2). If $G_1 \not\cong G_2$, then $S_{G_1} \cap S_{G_2} = \emptyset \implies |S(G_1, G_2)| = |S_{G_1}| + |S_{G_2}| = 2n!$ \square

Thus to convince the verifier that G_1 and G_2 are non-isomorphic, the prover just needs to prove to the verifier that $|S(G_1, G_2)|$ is equal to $2n!$. For this, we have the Goldwasser-Sipser Set Size Protocol.

23.1.2 Goldwasser-Sipser Set Size Protocol

Let S be a set such that membership in S is decidable in NP. Both the parties know a number K with the promise that $|S|$ is either K or at most $K/2$. The prover wants to prove to the verifier that $|S|$ is atleast K and the verifier should reject if $|S| \leq K/2$. For the GNI setting, we have $S = S(G_1, G_2)$ and $K = 2n!$. Note that $(H, \tau) \in S$ is efficiently verifiable given σ and i such that $\sigma(G_i) = H$ and $\tau(H) = H$.

The protocol goes as follows (here $S = \{0, 1\}^m$):

1. Let l be such that $2^{l-2} < K \leq 2^{l-1}$.
2. Verifier (V): Picks $h : \{0, 1\}^m \rightarrow \{0, 1\}^l$ at random, and $z \in_R \{0, 1\}^l$ and sends h and z to the prover.
3. Prover (P): Sends $y \in S$ such that $h(y) = z$ and a certificate w certifying $y \in S$.

Let $\mu = K/2^l$.

Claim 23.4. If $|S| \leq K/2$, then $\Pr_{h,z}[z \in h(S)] \leq \mu/2$.

Proof. $|h(S)| \leq |S| \leq K/2 \implies \Pr_{z,h}[z \in h(S)] = |h(S)|/2^l \leq \mu/2$. □

Claim 23.5. If $|S| \geq K$, then $\Pr_{h,z}[z \in h(S)] \geq \frac{3}{4}\mu$.

Proof. We have

$$\begin{aligned} \Pr_{h,z}[z \in h(S)] &= \Pr_{h,z}[\exists y \in S : h(y) = z] \\ &\geq \sum_{y \in S} \Pr[h(y) = z] - \sum_{y \neq y' \in S} \Pr[h(y) = h(y') = z] \\ &= \frac{|S|}{2^l} - \binom{|S|}{2} \cdot \frac{1}{2^{2l}} \\ &\geq \frac{|S|}{2^l} \left(1 - \frac{|S|}{2} \cdot \frac{1}{2^l}\right) \\ &\geq \mu(1 - \mu/2) \geq \frac{3}{4}\mu \end{aligned}$$

The last inequality follows as $\mu \leq 1/2$. □

So we have a probability gap of $\mu/4$ between the “yes” and the “no” case. So we can repeat this in parallel t times and accept if $\frac{2}{3}\mu t$ of them succeed.

Thus $\text{GNI} \in \text{IP}[V \rightarrow P \rightarrow V, \text{public coins}]$.

One major catch: To specify h , we need $l \cdot 2^m$ bits! So the verifier cannot send h . However, note that we require only two properties of h :

1. $\Pr_h[h(y) = z] = \frac{1}{2^l}$.
2. For $y \neq y'$, $\Pr_h[h(y) = z, h(y') = z] = \frac{1}{2^{2l}}$.

We know that pairwise independent hash function families satisfy them! One such family (as seen in the quiz) is given by $\mathcal{H} = \{h_{A,b} : y \mapsto Ay + b \mid A \in \mathbb{F}_2^{l \times m}, b \in \mathbb{F}_2^l\}$. So the verifier can send a random A and b using only $lm + l$ bits.

23.2 Classes AM and MA

AM is the class of languages which have an IP protocol with the verifier (Arthur) sending a random string r as the first message, the prover (Merlin) responding with a message y , and the verifier then decides to accept or reject by applying a deterministic function on the transcript. On the other hand, MA is the class of languages which have an IP protocol in which the prover (Merlin) sends the first message y and the verifier tosses some random coins r and decides to accept or reject. The completeness and soundness conditions are given below:

1. *Completeness and soundness for AM:*

$$x \in L \implies \Pr_r[\exists y, V(x, y, r) = \text{acc}] \geq 2/3$$

$$x \notin L \implies \Pr_r[\exists y, V(x, y, r) = \text{acc}] \leq 1/3$$

2. *Completeness and soundness for MA:*

$$x \in L \implies \exists y, \Pr_r[V(x, y, r) = \text{acc}] \geq 2/3$$

$$x \notin L \implies \forall y, \Pr_r[V(x, y, r) = \text{acc}] \leq 1/3$$

Using our usual techniques, we can amplify these success probabilities to “large” values.

Useful intuition: Consider a 2×2 binary grid G where the rows correspond to all possible values of r , the columns correspond to all possible values of y , and the (r, y) -th entry is 1 iff $V(x, y, r) = \text{acc}$. Then:

1. An “yes” instance of AM corresponds to “most” rows of G having a 1.
2. A “no” instance of AM corresponds to “very few” rows of G having a 1.
3. An “yes” instance of MA corresponds to there being a column in G with many 1’s.
4. A “no” instance of MA corresponds to all columns of G having “very few” ones.

Theorem 23.6. $MA \subseteq AM$.

Proof. Consider $L \in MA$. Amplify the success probability to $1 - \delta$ (we will fix δ later appropriately). The MA completeness and soundness conditions then imply

$$x \in L \implies \exists y, \Pr_r[V(x, y, r) = 1] \geq 1 - \delta$$

$$x \notin L \implies \forall y, \Pr_r[V(x, y, r) = 1] \leq \delta$$

So if we consider the grid G as defined earlier, then when $x \in L$, G has a column with at least $1 - \delta$ fraction of the entries being 1’s \implies at least $1 - \delta \geq 2/3$ (for a suitable δ , say $\delta \leq 1/3$) fraction of the rows have a 1, which is the AM “yes” condition. On the other hand, if $x \notin L$, then each column has at most δ fraction of the entries as 1’s. If R is the number of rows,

then the total number of 1's in G is at most $2^m \cdot \delta \cdot R$. Taking $\delta = 1/(3 \cdot 2^m)$, we get that at most 1/3-rds of the rows have a 1, which is the "no" condition for AM. Thus $L \in \text{AM}$, as desired. \square

Lecture 24

AM Round Reduction

Scribe: Krishnashree J B

Topics covered in this lecture

- Operator view of MA and AM
- The BP operator and promise problems
- Swapping \exists and BP
- Round reduction: $AM[k] = AM$
- Containment $AM \subseteq \Pi_2$
- Application: Its is highly unlikely that Graph Isomorphism is NP-complete.

24.1 Definitions and Intuition

Definition 24.1. A language L is in MA if there exists a polynomial-time verifier V such that:

- $x \in L \Rightarrow \exists y \Pr_r[V(x, y, r) = 1] \geq \frac{2}{3}$
- $x \notin L \Rightarrow \forall y \Pr_r[V(x, y, r) = 1] \leq \frac{1}{3}$ ◇

Definition 24.2. A language L is in AM if there exists a polynomial-time verifier V such that:

- $x \in L \Rightarrow \Pr_r[\exists y V(x, y, r) = 1] \geq \frac{2}{3}$
- $x \notin L \Rightarrow \Pr_r[\exists y V(x, y, r) = 1] \leq \frac{1}{3}$ ◇

Matrix View Intuition. Think of a matrix with rows indexed by randomness r and columns by witnesses y . We will put a check-mark at (r, y) if $V(x, y, r) = \text{accept}$.

- MA:
 - $x \in L$: there exists a column with many check marks
 - $x \notin L$: all columns have very few check marks

- AM:

$x \in L$: Many rows have a check mark in it

$x \notin L$: Most rows have no check mark in it

24.2 Operator View of AM and MA

We define two operators \exists and BP that map promise-problems to promise-problems.

Definition 24.3. Let $L = (L_{yes}, L_{no})$ be a promise problem. Then, $L' := \exists \cdot L$ is defined as follows:

$$L'_{yes} = \{x : \exists y \text{ s.t. } (x, y) \in L_{yes}\}$$

$$L'_{no} = \{x : \forall y \text{ s.t. } (x, y) \in L_{no}\}$$

If L is actually a language (i.e., there are no 'do not care' strings), then so is $\exists \cdot L$. ◇

A good example to have in mind is that $\exists \cdot \text{CIRCUIT-EVAL} = \text{CIRCUIT-SAT}$.

Definition 24.4 (BP). Let $L = (L_{yes}, L_{no})$ be a promise problem. We define the promise problem $L' = \text{BP} \cdot L$ as follows:

$$L'_{yes} = \left\{x : \Pr_r[(x, r) \in L_{yes}] \geq \frac{2}{3}\right\}$$

$$L'_{no} = \left\{x : \Pr_r[(x, r) \in L_{no}] \geq \frac{2}{3}\right\}$$

Note that even if L was a language, there may be strings $x \in \Sigma^*$ such that $\frac{1}{3} < \Pr_r[(x, r) \in L] < \frac{2}{3}$; these would be the set of don't care instances of $\text{BP} \cdot L$. ◇

We can extend the above operator to complexity classes of promise problems to denote $\exists \cdot \mathcal{C} = \{\exists \cdot L : L \in \mathcal{C}\}$ and similarly $\text{BP} \cdot \mathcal{C}$.

MA and AM with the above intuition: With the above, we can write $\text{PROMISEMA} = \exists \cdot \text{BP} \cdot P$, and $\text{PROMISEAM} = \text{BP} \cdot \exists \cdot P$. The languages in $\text{BP} \cdot \exists P$ is precisely AM and the languages in $\exists \cdot \text{BP} \cdot P$ is precisely MA.

Interpretation MA: Merlin sends y , then randomness is applied. **AM:** randomness r is chosen first, then Merlin responds.

24.3 Round Reduction

Lemma 24.5. For any class \mathcal{C} of promise problems, we have

$$\exists \cdot \text{BP} \cdot \mathcal{C} \subseteq \text{BP} \cdot \exists \cdot \mathcal{C}.$$

Proof. Let $L' \in \exists \cdot \text{BP} \cdot L$ for some $L \in \mathcal{C}$. Then, we have

$$x \in L'_{yes} \implies \exists y : \Pr_r[(x, y, r) \in L_{yes}] \geq 2/3$$

$$x \in L'_{no} \implies \forall y : \Pr_r[(x, y, r) \in L_{no}] \geq 2/3$$

By resampling more r 's, we can assume without loss of generality that we have the following stronger guarantee (for a soon-to-be-chosen δ):

$$\begin{aligned} x \in L'_{\text{yes}} &\implies \exists y : \Pr_r[(x, y, r) \in L_{\text{yes}}] \geq 1 - \delta \\ x \in L'_{\text{no}} &\implies \forall y : \Pr_r[(x, y, r) \in L_{\text{no}}] \geq 1 - \delta. \end{aligned}$$

Let us consider a matrix where rows are indexed by r 's, columns indexed by y 's, and we put a check mark whenever $(x, y, r) \in L_{\text{yes}}$ and a cross mark whenever $(x, y, r) \in L_{\text{no}}$.

If $x \in L'_{\text{yes}}$ then the above states that there is a column with at least $(1 - \delta)$ fraction of entries being check marks. This immediately implies that most rows have a check mark. Therefore,

$$x \in L'_{\text{yes}} \implies \Pr_r[\exists y : (x, y, r) \in L_{\text{yes}}] \geq 1 - \delta \geq \frac{2}{3}.$$

If $x \in L'_{\text{no}}$, then the above states that all columns have at least $(1 - \delta)$ fraction of cross marks. Thus, the total number of cross marks in the matrix is at least $(1 - \delta) \cdot |\text{Rows}| \cdot |\text{Columns}|$. If $\delta < \frac{1}{3 \cdot |\text{Columns}|}$, we have at least $\#\text{Entries} - \frac{1}{3} |\text{Rows}|$ many cross marks in the matrix. There at least $2/3$ fraction of the rows are entirely filled with cross marks. That is,

$$x \in L'_{\text{no}} \implies \Pr_r[\forall y : (x, y, r) \in L_{\text{no}}] \geq 2/3.$$

The above are exactly the guarantees required for showing $L' \in \text{BP} \cdot \exists \mathcal{C}$. Hence we have that $\exists \text{BP} \mathcal{C} \subseteq \text{BP} \exists \mathcal{C}$. □

Corollary 24.6. $\text{AMA} \subseteq \text{AM}$

Proof. $\text{AMA} = \text{BP} \exists \text{BP} \text{P} \subseteq \text{BP} \text{BP} \exists \text{P} = \text{BP} \exists \text{P} = \text{AM}$. □

Corollary 24.7. $\text{AM}[k] = \text{AM}$ for any constant k

Proof. Repeated alternations of randomness and existential quantifiers can be collapsed by repeatedly applying the swapping lemma to move all existential quantifiers inside the randomness. □

24.4 Containment: $\text{AM} \subseteq \Pi_2$

Theorem 24.8. $\text{AM} \subseteq \Pi_2$

Proof. Let $L \in \text{AM}$. Then there exists a probabilistic polynomial-time verifier M such that for some $\delta < \frac{1}{2}$:

- $x \in L \implies \Pr_r[\exists y M(x, y, r) = \text{acc}] \geq 1 - \delta$
- $x \notin L \implies \Pr_r[\exists y M(x, y, r) = \text{acc}] \leq \delta$

Consider the complement language \bar{L} . Then:

- $x \in \bar{L} \implies \Pr_r[\forall y M(x, y, r) \neq \text{acc}] \geq 1 - \delta$

- $x \notin \bar{L} \Rightarrow \Pr_r[\forall y M(x, y, r) \neq \text{acc}] \leq \delta$

Define $A_x = \{r : \forall y M(x, y, r) \neq \text{acc}\}$. Then:

- if $x \in \bar{L}$, $|A_x| \geq (1 - \delta)|R|$
- if $x \notin \bar{L}$, $|A_x| \leq \delta|R|$

By Lautemann's argument (that $\text{BPP} \subseteq \Sigma_2 \cap \Pi_2$), there exists a polynomial $t(n)$ such that for every set $A_x \subseteq R$ of size at least $(1 - \delta)|R|$, there exist shifts a_1, \dots, a_t satisfying:

$$\forall r \in R, \exists i \in [t] \text{ such that } r + a_i \in A_x.$$

Applying this, for $x \in \bar{L}$ we obtain:

$$\exists a_1, \dots, a_t \forall r \exists i \forall y : M(x, y, r + a_i) \neq \text{acc}.$$

This is a Σ_2 statement for \bar{L} , hence $\bar{L} \in \Sigma_2$ and therefore $L \in \Pi_2$. □

In a similar vein, we can also show the following.

Theorem 24.9. $\text{MA} \subseteq \Sigma_2$.

24.5 Application: Graph Isomorphism

Theorem 24.10. *If GRAPHISO is NP-complete, then the polynomial hierarchy collapses.*

Proof. Assume that GRAPHISO is NP-complete. Then its complement, GRAPHNONISO, is coNP-complete. We saw in the last lecture that GRAPHNONISO \in AM.

Suppose $L \in \Sigma_2$. Then $L = \{x : \exists y \forall z R(x, y, z) = 1\}$. Define $L' = \{(x, y) : \forall z R(x, y, z) = 1\}$. Then $L' \in \text{coNP}$ by the definition above. Since we have assume that GRAPHNONISO is coNP-complete, there is a reduction Φ from L' to GRAPHNONISO such that $\Phi((x, y)) = (G_{(x,y)}, H_{(x,y)})$ such that $(x, y) \in L'$ if and only if $G_{(x,y)} \not\cong H_{(x,y)}$.

We can now construct the following MAM protocol for L : Merlin sends y , and they now run the AM protocol for checking if $G_{(x,y)} \not\cong H_{(x,y)}$.

But we have seen that $\text{MAM} = \text{AM} \subseteq \Pi_2$. Thus, $\Sigma_2 \subseteq \Pi_2$. Hence, the polynomial hierarchy collapses to the second level. □

Remark 24.11. *If any coNP-complete language has a constant-round public-coin interactive proof, then the polynomial hierarchy collapses.* ◇

Lecture 25

TBA

Scribe: Soham Ghosh

Lecture 26

MIP = NEXP

Scribe: Ramprasad Satharishi

In this lecture, we will see a sketch of the following result by Babai, Fortnow and Lund.

Theorem 26.1 (Babai-Fortnow-Lund). $MIP = NEXP$.

That is, the set of languages that admits a multi-prover interactive protocol coincides with the class NEXP.

Perhaps we first need to define what multi-prover interactive proofs are.

26.1 Multi-prover interactive proofs

The setting for multi-prover interactive proofs is very similar to IP; here there is more than one prover that the verifier is communicating with. A k -prover interactive proof is one where the verifier V (still a randomised polynomial time machine) is communicating with k provers to eventually decide to accept / reject.

- The provers can decide on a strategy before hand (which may also involve shared randomness),
- The verifier, based on responses received so far, can decide to ask one of the provers a question and that provers provides a response,
- Once the protocol begins, the provers are not allowed to communicate with each other.

Definition 26.2 (MIP). A language L is said to be in the class $MIP(k)$ if there is a polynomial time randomised verifier V communicating with k provers such that the following guarantees hold:

$$[\text{completeness}] x \in L \implies \exists(P_1, \dots, P_k) : \Pr[(P_1, \dots, P_k) \leftrightarrow V(x) = \text{acc}] \geq 2/3$$

$$[\text{soundness}] x \notin L \implies \forall(P_1, \dots, P_k) : \Pr[(P_1, \dots, P_k) \leftrightarrow V(x) = \text{acc}] \leq 1/3$$

The protocol consists of polynomially many rounds, and each message is at most polynomially long. \diamond

The following is not-too-hard to show.

Lemma 26.3. $MIP \subseteq NEXP$.

The other direction is the more interesting one.

26.1.1 Two provers are enough

Lemma 26.4 (Two provers are enough). *For any $k \geq 2$, we have $\text{MIP}(k) = \text{MIP}(2)$.*

Proof. Consider an arbitrary $\text{MIP}(k)$ protocol for a language L , and say the total number of messages is at most n^5 . Here is a different two-prover protocol for the same language. We will use (P_1, \dots, P_k) to denote the k provers, and (Q_1, Q_2) to denote the 2 provers in the new protocol.

1. Repeat this for n^6 times.
 - (a) Simulate the original protocol by asking Q_1 to play the role of all the k provers P_1, \dots, P_k .
 - (b) Pick one of the n^5 messages at random, which say was for prover j . Give Q_2 just the transcript with P_j and ask Q_2 to provide an answer. If this is inconsistent with Q_1 's answer, reject the protocol.
2. Accept all of these runs accepted.

If Q_1 was not answering independently, we have a $(1 - 1/n^5)$ chance of catching that lie in any one of those n^6 attempts. Thus, the probability that (Q_1, Q_2) manage to fool the verifier is at most $(1 - 1/n^5)^{n^6} \approx e^{-n}$. □

An important observation here is that the prover Q_2 is basically forced to be non-adaptive since the prover only gets a single question per round (and each round is independent anyway).

Corollary 26.5 (2-prover is equivalent to (prover, oracle)). *If $L \in \text{MIP}$, then there is a two-prover protocol where one of the provers may be assumed to be a non-adaptive oracle.* □

26.2 Designing a protocol for NEXP

We first need to identify an NEXP-complete problem for which we will design a multi-prover interactive protocol. That problem is going to be a succinct-version of 3CNF-SAT.

26.2.1 Succinct-SAT

The language `SUCCINCT3SAT` deals with satisfiability of a HUGE 3-CNF instance.

Consider some function $D : \{0, 1\}^m \rightarrow \{0, 1\}^{3n+3}$. We will use D to describe a HUGE 3CNF, consisting of $M = 2^m$ clauses and $N = 2^n$ variables, as follows:

$$D(i) = (t_1, t_2, t_3, b_1, b_2, b_3) \quad \text{where } t_1, t_2, t_3 \in \{0, 1\}^n, \quad b_1, b_2, b_3 \in \{0, 1\}$$

denotes that clause $i \in [M]$ consists of variables $x_{t_1}, x_{t_2}, x_{t_3}$ with b_1, b_2, b_3 being the negation flags for the three literals.

Thus, by providing D via a $(3n+3)$ boolean formulas of size s each (one for each output gate), we have a way of using D to succinctly describe a formula Φ_D with 2^m clauses on 2^n variables.

$$\text{SUCCINCT3SAT} = \left\{ \langle D \rangle : D : \{0,1\}^m \rightarrow \{0,1\}^{3n+3} \text{ describes a satisfiable } \Phi_D \right\}$$

Turns out, SUCCINCT3SAT is NEXP-complete (under polynomial time many-one reductions). We will just assume this fact and move on.

The rest of the proof is coming up with a 2-prover protocol for SUCCINCT3SAT.

26.2.2 Setting up for an LFKN-like prover claim

Like earlier, we will fix a field \mathbb{F}_p for a large enough prime P , and deal with various arithmetisations. To begin with, we can extend $D : \{0,1\}^m \rightarrow \{0,1\}^{3n+3}$ to a polynomial $\tilde{D} : \mathbb{F}^n \rightarrow \mathbb{F}^{3n+3}$ by arithmetising each gate of the formula:

$$\begin{aligned} f = g_1 \wedge g_2 &\implies \tilde{f} := \tilde{g}_1 \cdot \tilde{g}_2, \\ f = g_1 \vee g_2 &\implies \tilde{f} := 1 - (1 - \tilde{g}_1) \cdot (1 - \tilde{g}_2), \\ f = \neg g &\implies \tilde{f} := 1 - \tilde{g}. \end{aligned}$$

Since D is a formula, it can be readily seen that the degree of the arithmetisation is at most the size of the formula.

Thus, the verifier V holds a low-degree arithmetisation $\tilde{D} : \mathbb{F}^m \rightarrow \mathbb{F}^{3n+3}$ of D . In particular, we may assume that $t_1(x_1, \dots, x_m), t_2(x_1, \dots, x_m), t_3(x_1, \dots, x_m) \in (\mathbb{F}[x_1, \dots, x_m])^n$ and $b_1(x_1, \dots, x_m), b_2(x_1, \dots, x_m), b_3(x_1, \dots, x_m) \in \mathbb{F}[x_1, \dots, x_n]$ are polynomials that the verifier can evaluate at any chosen point.

Assignments for Φ_D : Since Φ_D is a formula on $N = 2^n$ variables, any assignment is given by a function $A : \{0,1\}^n \rightarrow \{0,1\}$. The provers will be expected to work with the *unique multilinear* polynomial $\tilde{A} : \mathbb{F}^n \rightarrow \mathbb{F}$ that extends the $\{0,1\}$ behaviour of A . (Of course, the provers may cheat by instead working with a crazy polynomial; we have to find a way to somehow catch them if they do.)

With some access to an assignment, we now want to express what it means for the assignment to satisfy a clause.

Truth-value of a literal: Let $L_1(i), L_2(i), L_3(i)$ (which will be functions of an assignment $A : \{0,1\}^n \rightarrow \{0,1\}$) be the truth-value of the first, second and third literal in the i -th clause. That is,

$$L_1(i) = (1 - b_1(i)) \cdot A(t_1(i)) + b_1(i) \cdot (1 - A(t_1(i))),$$

and similarly $L_2(i), L_3(i)$. In words, the above expression takes values $A(t_1(i))$ if $b_1(i) = 0$ (i.e., the variable was not negated in clause i), and takes value $(1 - A(t_1(i)))$ if $b_1(i) = 1$.

Satisfiability of a clause: With the above, satisfiability of the i -th clause can be encoded as

$$\text{SAT}(i) = 1 - (1 - L_1(i))(1 - L_2(i))(1 - L_3(i)).$$

We extend these to polynomials $L_1(x_1, \dots, x_m), L_2(x_1, \dots, x_m), L_3(x_1, \dots, x_m)$ given by

$$L_j(x_1, \dots, x_n) = (1 - b_j(x_1, \dots, x_n)) \cdot \tilde{A}(t_j(x_1, \dots, x_n)) + b_j(x_1, \dots, x_n) \cdot (1 - \tilde{A}(t_j(x_1, \dots, x_n))).$$

Similarly,

$$\text{SAT}(x_1, \dots, x_m) = 1 - (1 - L_1(x_1, \dots, x_m))(1 - L_2(x_1, \dots, x_m))(1 - L_3(x_1, \dots, x_m)).$$

Remark. An important point here is that the verifier cannot actually evaluate SAT at any chosen (r_1, \dots, r_m) . However, from (r_1, \dots, r_m) , the verifier can compute three values $\tau_1 = t_1(r_1, \dots, r_m), \tau_2 = t_2(r_1, \dots, r_m), \tau_3 = t_3(r_1, \dots, r_m)$ such that $\text{SAT}(r_1, \dots, r_m)$ can be computed if the verifier is given the values $\tilde{A}(\tau_1), \tilde{A}(\tau_2), \tilde{A}(\tau_3)$. \diamond

The provers' claim: To convince the verifier that Φ_D is indeed satisfiable, the prover is going to provide an oracle $\mathcal{O} : \mathbb{F}^n \rightarrow \mathbb{F}$ purported to hold the evaluations of a multilinear polynomial $\tilde{A} : \mathbb{F}^n \rightarrow \mathbb{F}$, and will assert that

$$\sum_{x_1=0,1} \sum_{x_2=0,1} \cdots \sum_{x_m=0,1} \text{SAT}_{\tilde{A}}(x_1, \dots, x_m) = 2^m$$

(i.e., the assignment \tilde{A} satisfies every clause of Φ_D).

26.2.3 The Prover-Oracle protocol

We now have an oracle $\mathcal{O} : \mathbb{F}^n \rightarrow \mathbb{F}$ that the prover claims is the evaluations of a multilinear polynomial $\tilde{A}(y_1, \dots, y_n)$ (corresponding to an assignment) and claims

$$\sum_{x_1=0,1} \sum_{x_2=0,1} \cdots \sum_{x_m=0,1} \text{SAT}_{\tilde{A}}(x_1, \dots, x_m) = k_0 = 2^m.$$

Round 1: Prover sends $S_1(x_1)$ (purportedly equal to $\sum_{x_2} \cdots \sum_{x_m} \text{SAT}_{\tilde{A}}(x_1, \dots, x_m)$).

Verifier checks if $S_1(0) + S_1(1) = k_0$ and rejects if not. The verifier picks $r_1 \in_R \mathbb{F}$ and sends r_1 to the prover.

The prover must now assert that $\sum_{x_2=0,1} \cdots \sum_{x_m=0,1} \text{SAT}_{\tilde{A}}(r_1, x_2, \dots, x_m) = k_1 = S_1(r_1)$.

\vdots

Round m . Prover sends $S_m(x_m)$ (purportedly equal to $\text{SAT}_{\tilde{A}}(r_1, r_2, \dots, r_{m-1}, x_m)$).

Verifier checks if $S_m(0) + S_m(1) = k_{m-1}$ and rejects if not. The verifier picks $r_m \in_R \mathbb{F}$ and sends r_m to the prover.

Now, the verifier needs to check that $\text{SAT}_{\tilde{A}}(r_1, \dots, r_m) = k_m = S_{m-1}(r_m)$. As mentioned earlier, the verifier can compute $\text{SAT}_{\tilde{A}}(r_1, \dots, r_m)$ if provided the values $\tilde{A}(\tau_1), \tilde{A}(\tau_2), \tilde{A}(\tau_3)$ where $\tau_i = t_i(r_1, \dots, r_m)$. The Verifier will query the oracle \mathcal{O} for these three values and check if it indeed matches the prover's claim that $\text{SAT}_{\tilde{A}}(r_1, \dots, r_m) = k_m$.

In the above protocol, nowhere have we checked that \mathcal{O} actually is the evaluations of a multilinear polynomial $\tilde{A}(y_1, \dots, y_n)$. In fact, without this check, the prover may be able to instantiate \mathcal{O} to be a some high-degree polynomial such that the true $S_1(x_1)$ agrees with a fake low-degree $S_1^{(\text{bogus})}(x_1)$ at many points in \mathbb{F}_p (the worst case being if $S_1(x_1) - S_1^{(\text{bogus})}(x_1)$ is divisible by $x_1^p - x_1$, and hence they agree on all points in \mathbb{F}_p).

Thus, the verifier needs some way to ensure that

- \mathcal{O} corresponds to the evaluations of a multilinear-polynomial $\tilde{A}(y_1, \dots, y_n)$,
- the polynomial $\tilde{A}(y_1, \dots, y_n)$ only takes 0/1 values on $\{0, 1\}^n$.

26.2.4 Enforcing multilinearity of \tilde{A}

The multilinearity test is the most technical component of Babai-Fortnow-Lund and involves deep theorems in the low-degree testing literature. Roughly speaking, the multilinearity test proceeds by picking random axis parallel lines (i.e., $(a_1, a_2, \dots, a_{i-1}, *, a_i, \dots, a_n)$ for some fixed a_i 's), querying points along this line and checking if it looks like a degree 1 polynomial in the variable.

Lemma (Multilinearity test (Informal)). *Let $\mathcal{O} : \mathbb{F}^n \rightarrow \mathbb{F}$ be an arbitrary oracle. Consider the following test:*

- For each of the n directions:
 - Pick m_1 random lines in this direction.
 - On each of these lines, query m_2 points on this line.
 - Check if these evaluations match a degree ≤ 1 univariate polynomial

If the above test succeeds with high probability, then \mathcal{O} agrees (at $(1 - \varepsilon)$ fraction of locations) with the evaluations of some multilinear polynomial $\tilde{A}(y_1, \dots, y_n)$.

26.2.5 Ensuring \tilde{A} only takes 0/1 values on $\{0, 1\}^n$

The polynomial \tilde{A} takes 0/1 values on $\{0, 1\}^n$ if and only if $G(y_1, \dots, y_n) = \tilde{A}^2 - \tilde{A}$ vanishes on $\{0, 1\}^n$. This is sometimes called the *zero-on-subcube* test. Let us assume that $|\mathbb{F}| \geq 2^{n+1}$.

Let us consider the following univariate polynomial

$$p(x) = \sum_{w \in \{0, 1\}^n} G(w) x^{[w]}$$

where $[w] := w_1 + 2w_2 + 2^2w_3 + \dots + 2^{n-1}w_n$. If G was zero on the entire subcube, this polynomial is supposed to be the zero polynomial. If not, then at a random point $\zeta \in \mathbb{F}_p$ we would see $p(\zeta) \neq 0$ with probability at least $1/2$ (since p has degree at most 2^n and $|\mathbb{F}| \geq 2^{n+1}$).

$$\begin{aligned} p(\zeta) &= \sum_{w_1, \dots, w_n \in \{0,1\}} G(w) \cdot \zeta^{[w]} \\ &= \sum_{w_1, \dots, w_n \in \{0,1\}} G(w) \cdot \prod_{i=1}^n \zeta^{2^{i-1}w_i} \\ &= \sum_{w_1, \dots, w_n \in \{0,1\}} G(w) \cdot \prod_{i=1}^n (1 + (\zeta^{2^{i-1}} - 1)w_i) \end{aligned}$$

(the last equality uses the fact that $\zeta^{2^{i-1}w_i} = \zeta^{2^{i-1}}$ when $w_i = 1$ and 1 otherwise).

Thus, if we think of $H(w_1, \dots, w_n) = G(w) \cdot \prod_{i=1}^n (1 + (\zeta^{2^{i-1}} - 1)w_i)$, note that $H(r_1, \dots, r_n)$ can be evaluated by the verifier at any choice of $r_1, \dots, r_n \in \mathbb{F}$. Then, the assertion that $p(\zeta) \neq 0$ is precisely the same as asserting that

$$\sum_{w_1=0,1} \dots \sum_{w_n=0,1} H(w_1, \dots, w_n) = 0$$

Thus, we can run the LFKN protocol to confirm this.

26.3 Scaling down to NP

Lemma 26.4 can be extended to show that an MIP for a language L can be equivalently phrased as a probabilistic oracle machines.

Definition 26.6. A language L is said to have an interactive oracle protocol recognising it if there is a probabilistic polynomial time oracle machine M such that

$$\begin{aligned} x \in L &\implies \exists \mathcal{O} : \Pr[V^{\mathcal{O}} = \text{accept}] \geq 1 - \frac{1}{\text{poly}(n)} \\ x \notin L &\implies \forall \mathcal{O} : \Pr[V^{\mathcal{O}} = \text{accept}] \leq \frac{1}{\text{poly}(n)}. \end{aligned}$$

◇

That is, the prover can provide an oracle \mathcal{O} that will convince the verifier with overwhelming statistical significance to accept the input string. The oracle is just an exponentially long table that the verifier can has random access to. Since the verifier is a polynomial time algorithm, the verifier just queries this table at $\text{poly}(n)$ locations before deciding to accept / reject.

What if we scale this down? That is, suppose $L \in \text{NP}$, do we then have a $\text{poly}(n)$ long table that the verifier can query at a few locations before deciding to accept or reject with high probability? This leads us what are now called *probabilistically checkable proofs (PCPs)*

Definition 26.7 (Probabilistically checkable proofs). A language L is said to have a probabilistic checkable proof with proof length $p(n)$ and query length $q(n)$ there is a probabilistic polynomial time

verifier V such that

$$x \in L \implies \exists \pi \in \Sigma^{p(n)} : \Pr[V^\pi = \text{accept}] \geq 2/3$$

$$x \notin L \implies \forall \pi \in \Sigma^{p(n)} : \Pr[V^\pi = \text{accept}] \leq 1/3$$

The verifier V only makes $q(n)$ queries to the proof π before deciding to accept / reject. ◇

What sort of PCPs can we have for NP? After a long sequence of results, this eventually culminated in the following phenomenal result Arora, Lund, Motwani, Sudan and Szegedy.

Theorem 26.8 (The PCP Theorem). *Every language $L \in \text{NP}$ has a PCP where the proof lengths are $\text{poly}(n)$ and the number of queries is $O(1)$.*

Thus, every language in NP can be certified with a proof that the verifier can check by querying just a constantly many locations!

There are several amazing consequences of this result in proving several hardness of approximation results, and we now even have near-linear PCPs.