# Polyhedral compilation formalism

Siddharth Bhat

# Contents

**1 Problems, Solutions, and Resources** **5**
- 1.1 Problems . . . 5
  - 1.1.1 Kannan . . . 6
- 1.2 Solutions . . . 6
  - 1.2.1 Kannan . . . 6
- 1.3 Resources . . . 6

**2 Diagonalization** **7**

**3 Hierarchy Theorems** **9**
  - 3.0.1 Proof sketch . . . 9
- 3.1 Savitch's Theorem: `NSPACE(`$f(n)$`)` $\subseteq$ `PSPACE(`$f(n)^2$`)` . . . 10
- 3.2 Cook Levin theorem . . . 10
- 3.3 `EXPSPACE` completeness - $EQ_{REG\uparrow}$ . . . 11

**4 NP** **13**
- 4.1 Cook Levin theorem . . . 13
  - 4.1.1 Proof . . . 13
- 4.2 3-SAT is NP-complete . . . 15
- 4.3 `CLIQUE` is NP-complete . . . 15

**5 `PSPACE`** **17**
- 5.1 `PSPACE`-completeness . . . 17
  - 5.1.1 TQBF - Totally quantified boolean formula . . . 17
  - 5.1.2 Winning strategies in games . . . 17
  - 5.1.3 Proof of `PSPACE`-completeness of `TQBF` . . . 17
- 5.2 Relativization – P versus NP (Baker Gill Soloway '75) . . . 18
  - 5.2.1 Intuition: program diagonalization proofs relativize . . . 18
  - 5.2.2 Proof of BGS . . . 18

**6 LogSpace** **21**
- 6.1 The languages `L` & `NL` . . . 21
  - 6.1.1 `NL`-completeness . . . 21
- 6.2 The `PATH` problem . . . 21
- 6.3 Verifiers . . . 22

6.4 `co-NP` . . . . . 22
6.5 `CLIQUE` . . . . . 22
6.5.1 `PH`-completness . . . . . 23
6.5.2 `PH` $\subset$ `PSPACE` . . . . . 23
6.6 `co-NL` . . . . . 23
6.6.1 Verifiers for `NL`, `co-NL` . . . . . 23

**7 Other flavours of complexity classes:`BPP`, `P/Poly`** **25**
7.1 Introduction . . . . . 25
7.2 `P/Poly` . . . . . 25
7.2.1 Karp-Lipton theorem . . . . . 25
7.3 `BPP` . . . . . 26
7.3.1 `BPP`$=$ `P`, the conjectured proof . . . . . 26

**8 Exploring probabilistic complexity classes** **27**
8.1 `BPP` $\subset$ `P/Poly` (Adleman's theorem) . . . . . 29
8.2 Computation is messy (Philosophy) . . . . . 30

**9 Property testing** **31**
9.1 `PARITY` . . . . . 31
9.2 `MAJORITY` . . . . . 31
9.3 Approximate decision problems . . . . . 31
9.3.1 $\epsilon -$ `MAJORITY` . . . . . 31
9.4 `SORTED` . . . . . 32
9.4.1 `SORTED` test . . . . . 32
9.4.2 Property testing on languages that are invariant under subgroups of permutations . . . . . 33
9.5 Linearity testing . . . . . 33
9.5.1 Local consistency . . . . . 33
9.5.2 Global consistency . . . . . 33
9.5.3 Proof of soundness: $\delta(f) \geq 2\epsilon(f)$ . . . . . 33
9.5.4 $\phi$ is linear . . . . . 34
9.5.5 $\delta(f,\phi) \leq 2\epsilon(f)$ . . . . . 34

**10 `IP`** **35**
10.1 The class `IP` . . . . . 35
10.2 Arithmetization . . . . . 35
10.2.1 Rewrite `#SAT`in terms of arithmetization . . . . . 36
10.2.2 Interactive proof for $k$ in terms of $Q_\phi$ . . . . . 36
10.2.3 Using this property to verify . . . . . 36
10.2.4 Completeness . . . . . 37
10.2.5 Soundness . . . . . 37
10.3 `IP` $=$ `PSPACE` . . . . . 37

# Chapter 1

# Problems, Solutions, and Resources

## 1.1 Problems

Alphabet set is finite, call it $\Sigma$. Strings must be finite length.

Given some input, and a computer that produces some output, the description could be infinite — both input and output.

However, the machine's *description* (aka, the relationship between input and output) must be finite.

So, the *total input* can be infinite, but the input chunk must be finite, and the response of the machine per *input chunk* must be finite.

So, we can just use the language $L = \{0, 1\}$ for the machine.

Problems which have yes/no as answers are called decision problems. Inputs are from $\Sigma^*$, outputs are from $\{0, 1\}$. The problem is a mapping $f : \Sigma^* \to \{0, 1\}$. This is equivalent to providing the set $\texttt{ACCEPT} \subset \Sigma^* = f^{-1}(1)$. Note that $\texttt{REJECT} = \texttt{ACCEPT}^c$. The set `ACCEPT` is called a language.

Now, we can study languages by looking at their grammars (welcome, Chomsky).

What about fractional bit problems? Is this useful? Could we exploit some properties of fractional dimension?

### Cantor set

take $S_0 = [0, 1]$ In each iteration, remove the middle one-third of each continuous interval. Therefore,

- $S_0 = [0, 1]$
- $S_1 = [0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$

In $S_\infty$, *uncountably infinite* points remain (However, this set has *measure* 0).

So now, the question is, what is the dimension? We define Haussdorf dimension, and use this to exhibit fractional dimension of the Cantor set.

`TODO: fill this up!`

The total number of problems that can exist is $powerset(\Sigma^*)$. `RE` (recursively enumerable) Is a subset of $powerset(\Sigma^*)$ which computers can handle. The annoying thing is that there are *finite length problems* which computers cannot solve.

### 1.1.1 Kannan

If the universe is a machine, then it must have infinite description.

QM is the meeting point of universes?

## 1.2 Solutions

### 1.2.1 Kannan

**Question:** We study a lot of Science — why? What is the ultimate goal of science? Equivalently, what is the theory of everything we need to find to halt on the journey of Science?

Assuming Science = God, we need to ask Science a question. Which language will we use to query Science? Or, well, which language is *enough* to query Science? If the query alphabet is $\Sigma$, we can ask $\Sigma^*$ questions. However, we can only reasonably pose questions of finite length (even though the Science oracle can answer questions of infinite length).

In this case, have we achieved te ultimate goal of science?

## 1.3 Resources

## Chapter 2

# Diagonalization

- Level 1: $\mathbb{R}$ is uncountable.
- Level 2: $\exists L, L \notin \texttt{RE}$.
- Level 3: Halting problem is undecidable.
- Level 4: Time/Space hierarchy.
- Limitations: Exists oracles A, B such that — $P^A = NP^A, P^B \neq NP^B$
- Level 5: If $P \neq NP$, $\exists L, L \notin P, L \notin NPC$ (Ladner's theorem)

Diagonalization cannot separate P, NP — If it could, then it should also separate P with any oracle, and NP with the same oracle. We know that there exists an oracle such that we can separate $P^A = NP^A$, as well as $P^B \neq NP^B$.

chapterReview of the last 3 lectures, after add-drop

- Is it easier to *pose* problems than to *solve* them?
- Can every "solvable" problem have a solution that uses finite resources?
- What problems are *interesting*?
- Are all interesting problems solved in an *interesting* way? (P v/s NP)
- Can things get more interesting? (Quantum Mechanics, Approximation, Randomness, Interactivitiy, ...)

Are there problems with infinite length input / output but can still be posed in finite time? Eg. output $\pi$ in decimal. However, we decided that both input/output should be finite. We decided this does not belong to problems we wish to solve it, since we cannot solve it in finite time. If we believe that nature is inherently noisy, or nature is quantized, or nature has finite precision, then we cannot consider problems that require infinite time as problems in this universe (since Nature / the universe itself cannot pose such a problem).

Quantum mechanics (which is a theory of quantization) is developed over infinite precision mathematics ($\mathbb{C}$). Does this really make sense? There is a way in which a quantized universe can be infinite precision: This is by using 'external help': There are infinite such quantized universes which intersect at some points, and at those points, precision will increase. (If we both have a resolution of 1 pixel but are at a gap of 1/2, my least count is now 1/2). If there are an infinite number of universes overlapping at a single point, then we can construct "infinite precision". (*I feel this is crazy. Is this really crazy?*)

Posing a question is creating a language $L \subset \Sigma^*$. (Sid: a solution is a classifier for $L$).

Kannan's view:

- Finite space $\equiv$ finite information can be stored. (Turing: finite tape alphabet. Since a cell demarcates a finite volume, we want to have a finite amount of info in this cell)
- Information travels at finite speed. If we have cells, we should not be able to store and retreive information "equally" (based on how far we are from it). Hence, all infinite memory must be sequential memory since information travels at finite speed.
- Finite program $\equiv$ finite control.

Solution to these choices is a TM.

Do all languages have a TM recognizing it? No (`RE` = solvable by TM).

The class `R` = decidable by a TM (TM halts on all inputs). Diagonalization led us to Halting problem.

We have the class $P$, and we claimed that $P$ is interesting. Given that $P$ is considered interesting because of feasibility, it is possible that there are questions that are interesting even though **solving them** is not feasible. For example, if we can actually **understand** the solution, or the proof of non-existence of solutions, then we will care. `IP = PSPACE` is one such magical case where if someone can solve with a lot more power than you have access to, you can learn things from them interactively in reasonable time.

# Chapter 3

# Hierarchy Theorems

$\exists L$, such that $\forall f : \mathbb{N} \to \mathbb{N}$, where $f$ is space/time constructible,

$$Space(f) \supsetneq Space(o(f))$$
$$Time(f) \supsetneq Time\left(\frac{o(f)}{log f}\right)$$

So, there is a Hierarchy of complexity classes in time and space.

### 3.0.1 Proof sketch

We exhibit a languag $A$, such that $A \in Space(f(n))$, and $A \notin Space(o(f(n)))$.

Let $D$ decide $A$. D's definition:

- compute $f(n)$ and mark the end of $f(n)$ cells. If the read-write head ever crosses it, `REJECT`, `HALT`. We first need $f(n)$ to use $f(n)$ cells or less to compute. This is called as **space-constructribility**. ($f : \mathbb{N} \to \mathbb{N}$ is space-constructible iff given $n$, $\exists$ TM which computes $f(n)$ using at most $f(n)$ cells). Also, we want $f(n)$ to be at least $log(n)$. Clearly, this process is in space $f(n)$.
- We now need to "seaparate" A from the smaller classes. If A can be solved in a smaller space (ie, we cannot separate $A$), then there must be a TM (say, $D'$) which decides A in space less than $f(n)$. So now, we need to choose some input such that $D'$ is different from $D$. We can use diagonalization to construct such a function.
- let the input be $x$. Let $x = M10*$ for some TM $M$. if not, `REJECT`, `HALT`.
- Let $D$ simulate $M$ on input $M$. If $M$ takes less that $f(n)$ time to run on $M$, then $M$ can decide A in time less than $f(n)$. So now, $D$ knows how much space $M(\langle M \rangle)$ requires. if $M(\langle M \rangle)$ accepts, we reject. If $M(\langle M \rangle)$ rejects, we accept (diagonalization).
- To find out whether $M(\langle M \rangle)$ rejects, note that it is space-bounded, so we can just check how many states of the configuration space it visits. If it has not halted after visiting all states in the configuration space, we can conclude that $M(\langle M \rangle)$ does not halt. The configuration space is $O(2^{f(n)})$. So we need to run $D$ for time $O(2^f(n))$, and then `REJECT` if it continues running.

**Arjun Q:** Are there examples of non-space constructible functions, which are non-trivial? Other than ones that are too-small?

Proofs of time are similar to the space separation theorem.

- compute $t(n)$ ($t(n)$ should be time-constructible). decrement a counter initialized to $t(n)$. if this hits 0, `REJECT, HALT`. We get a *log* factor due to the slowdown of keeping time. (People are trying to speed this up).
- once again, repeat the same construction used for $SPACE$.

## 3.1 Savitch's Theorem: `NSPACE(`$f(n)$`)` $\subseteq$ `PSPACE(`$f(n)^2$`)`

$NSPACE(f(n))$ – one branch of a NTM $N$ decides $L$ in space $O(f(n)$.

Configuration space is $O(\texttt{alphabet}^{f(n)}) = O(2^f(n))$ – otherwise, configurations are repeated.

Our branch depth is exponential in $f(n)$. So, we need to keep track of $O(2^{f(n)})$ data.

Given $\langle C_1 \in Config(N), C_2 \in Config(N), t \in \mathbb{N}\rangle$ if we can find whether $C_1$ goes to $C_2$ in $t$ space, then we can solve our original problem.

This can be solved by recursion by asking if there exists a $C_{mid}$, such that $C_1 \to C_{mid}$ in $t/2$ steps, similarly $C_{mid} to C_2$ in $t/2$ steps.

## 3.2 Cook Levin theorem

$L$ is `NP-complete`, if

- $L \in NP$
- $\forall L' \in NP$, there exists a Karp reduction from $L'$ to $L$: $L' \leq_p L$ (`NP-hard`)

$A \leq_p B$ if there exists a poly time computable function $f : \Sigma^* \to \Sigma^*$ such that

$$w \in A \Leftrightarrow f(a) \in B$$

Karp reduction = poly time mapping reduction.

Define `SAT`, and show that `SAT` is `NP-complete`.

We have boolean formulas $\phi$, which is given in CNF.

$$\texttt{SAT} = \{\phi \mid \phi \text{ is in CNF (product of sums), } \phi \text{ is satisfiable}\}$$

.

This is clearly decidable since we can try all possible assignments.

It is in NP since a NDTM can try to guess assignments.

To show that this is NP-complete, take any language $L'$ in `NP`. We provide a karp reduction to `SAT`. We take the poly-time checker for $L'$ into a SAT problem $\psi$, such that **iff** a solution for $\psi$ exists, then the poly time checker will accept the string, and vice versa (for reject).

## 3.3 `EXPSPACE` completeness - $EQ_{REG\uparrow}$

$r\uparrow \equiv \exists k \in \mathbb{N}. r \uparrow$ is regular if $r$ is regular. We need this operator to control input size.

Question: Check if two regular expressions are the same – We show that this $\notin$ `PSPACE`, and hence $\notin$ `PTIME`. We show that this problem is `EXPSPACE` complete.

# Chapter 4

# NP

## 4.1 Cook Levin theorem

`SAT` is `NP-hard`.

### 4.1.1 Proof

. Unfold theorem statement into: $\forall L \in \texttt{NP}, L \leq_p \texttt{SAT}$. Since this should work for all things in NP, let's just write down the definition:

there exists an `NDTM` $N$ such that $N$ accepts $w$, $\forall w \in L$, in $|w|^k$ steps.

$N$ is an `NDTM`, so $N$ accepts $w$ means that there exists a branch of N that acceeps $w$ is $|w|^k$ steps.

We should be able to construct a CNF such that $\phi(w)$ is `SAT` iff there exists an accepting branch for $N(w)$.

#### Caveats

1. the construction of $\phi$ from $N$ should make sure that $\phi$ has $poly(|w|)$ clauses — otherwise, this is no longer a poly-time reduction. We know that $\langle \texttt{AND}, \texttt{OR}, \texttt{NOT} \rangle$ is universal, so we can clearly construct any `TM` into a circuit. The problem is that the CNF we construct from the truth-table of the `TM` will be polynomial.

**Sid Q:** Proof that boolean circuits are universal?

#### Proof sketch

Consider the NDTM $N(n)$, we will now argue about its configuration.

We can cut off the turing tape after the first polynomial number of cells — since the `NDTM` can only access those many cells.

We should start will the initial state $q_{start}$.

We should get the accept state $q_{accept}$ in $n^k$ steps.

If we can pose this in terms of a CNF formula of poly-length, we are done.

**Setting up `SAT`**

**Variables - Cells of the tape** The state of the turing tape on the $i$th step at the $j$th position of the turing tape for all $s \in \mathit{alphabet}(N)$ as $x_{i,j,s}$. $x_{i,j,s} = 1$ is interpreted as "at step $i$, on cell $j$, value $s$ is written.

For this to be valid, we need each cell to have exactly one symbol.

**Formula - Validity of cells** For every $(i,j)$ for at **least one** $s$ must be 1: $\phi_{cellleast} = \bigwedge_{i,j} (\vee_s x_{i,j,s})$

For every $(i,j)$ for at **most one** $s$ must be 1. This is equivalent to saying that for every $(s,t)$, one of them must be absent. $\phi_{cellmost} = \bigwedge_{s,t,s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t})$.

$\phi_{cell} = \phi_{cellleast} \wedge \phi_{cellmost}$

**Formula - Initial state** The initial cells contain the correct letters, corresponding to the initial input $w = \langle w_1 w_2 \ldots w_n$, and the other cells must be blank. $\phi_{init} = x_{1,0,\#} \wedge x_{1,0,q_{start}} \wedge (x_{1,1,w_1} \wedge x1, 2, w_2 \wedge \cdots \wedge x_{1,n,w_n}) \wedge (x_{1,n+1,blank} \wedge x_{1,n+2,blank} \wedge \cdots \wedge x_{1,n^k,blank})$

We use the $x_{\texttt{STATE},\texttt{LOC},\texttt{STATE ALPHABET}}$ to encode the current state we are in. We adjoin this to the alphabet since we automatically get mutual exclusion. The size of the `STATE ALPHABET` is constant, so it doesn't matter. The `LOC` of this will correspond to the **location of the head**. So, by placing the state symbol at a point, we understand that the head is at $(\texttt{LOC} + 1)$ in the original tape.

**Formula - Final state** To encode a TM configuration, what we need to know is the state of the TM, the position of the head, and the letters on the tape.

$\phi_{accept} = \vee_{i,k} x_{i,j,q_{accept}}$

**Formula - Transition** Every valid move in the TM can only touch two adjacent cells, since the memory is not really 'random-access', it localizes computation.

So, every transition formula will view three adjacent cells at once. If we can view three are valid, we can "slide the window" to check the correctness of all cells.

$\phi_{move} = \bigwedge_{\text{valid adjacent triplets}}$

To identify valid adjacent triplets, let the original config be `<a | b | c>`, let the new config be `<d | e | f>`.

The total number of legal windows will be a subset of $s^9$. So, for any legal window, we can create a formula of some constant size.

$\phi_{move} = \bigwedge$ legal window.

The number of legal windows is $O(n^k)$, since we only have $n^k$ steps and each legal window adds some constant number of formulas.

**Final $\phi$** : $\phi = \phi_{cell} \wedge \phi_{init} \wedge \phi_{accept} \wedge \phi_{move}$ If the NDTM accepts, then $\phi$ will be valid and vice versa.

**Sid Q:** Check that this reduction actually takes *poly-time in n*! It is clear to me that this inflates the size in poly, but it's not clear to me that this takes only *poly-time in n*.

Hence, SAT is **NP-hard** (well, NP-complete).

## 4.2 3-SAT is NP-complete

3-SAT: Every clause has 3 literals.

We reduce 4-SAT into 3-SAT, and then we show how to reduce `n-SAT` to `(n-1)-SAT`

$$C_{\texttt{4-SAT}} = x_1 \vee x_2 \vee x_3 \vee x_4$$
$$C'_{\texttt{3-SAT}} = (x_1 \vee x_2 \vee z) \wedge (x_3 \vee x_4 \vee \neg z)$$

If $C$ is true, We can always pick an assignment for $z$ to make the **other clause true**. So, we can pick a correct $z$.

Similarly, if $C$ is false, $C'$ reduces to $C' = z \wedge \neg z$. This is `UNSAT`, so this is not possible.

Now, in general, we reduce `n-SAT` into `(n-1)-SAT`.

## 4.3 `CLIQUE` is NP-complete

Does the graph contain a $k$ clique?

Proof.

Reduce 3-SAT to `CLIQUE`

$l$ clauses, $m$ variables.

For every clause and its complement, create a collection of vertices. Within a triplet, there are no edges. All triplets are across. Connect everything to everything, unless they are contradictory.

The number of edges will be high.

Now, ask for an $l$ clique. If this exists, then we have $l$ assignments which do not contradict each other.

# Chapter 5

# PSPACE

## 5.1 `PSPACE`-completeness

L is `PSPACE`-complete if:

- $L \in \texttt{PSPACE}$
- $\forall A \in \texttt{PSPACE}, A \leq_p L$ (polytime-reduction)

### 5.1.1 TQBF - Totally quantified boolean formula

A boolean formula $\phi$. (For example: $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$). TQBF will be of the form $\exists x_1,\ \forall x_2,\ \exists x_3 \ldots,\ \phi$. SAT was $\exists x_1, \exists x_2, \ldots, \exists x_n, \phi$. TQBF allows "forall" quantification.

### 5.1.2 Winning strategies in games

Winning strategies are basically objects of the form:

$WinningStrat \equiv \exists$ a move $x_1,$ such that $\forall$ moves $x_2, \exists$ a move $x_3, \ldots, \phi(x_1, x_2, \ldots)$

$WinningStrat$ looks like the specification of a winning path in a game tree! In the literature, this game is called the `FORMULA-GAME`.

So, we choose to work on TQBF now.

### 5.1.3 Proof of `PSPACE`-completeness of `TQBF`

**Proof of `TQBF` $\in$ `PSPACE`:**

let $problem = Q_1\ x_1,\ \ Q_2 x_2,\ \ \ldots\ \ \phi$. Let the solution procedure be called $S$.

$S(Q_1\ x_1\ \ Y)$. First check $x = 0, S(Y)$, next $x = 1, S(Y)$. When we recurse for $x = 1$ from doing $x = 0$, we try to reuse space. Once we get the answer, we need to `AND` (for $\forall$) or `OR` (for $\exists$) correctly.

$Space(n) = Space(n-1) + poly(n)$to store the answer bits. We get $Space(n) = Space(n-1) + \ldots$, because we can **reuse the space in the two recursive invocations**.

If we solve the recurrence, we find that this is in `PSPACE`.

**Proof of $L \leq_p$ `TQBF`:**

We want to solve the problem $\Phi c_{init}, c_{accept}, O(2^{f(n)})$, where $\Phi_{x,y,t}$ is the predicate that links $x$ to $y$ in configuration space in $t$ steps, and $f(n)$ is the time used by the turing machine for $L$.

We are trying to solve the reachability of the initial state to the final state of the turing machine for $L$ for any given input $w$ in TQBF. So, we create the configuration graph $G_{\langle L,w\rangle}$, and we encode the path problem from $c_{init}$ to $c_{accept}$ using $\Phi$.

$\Phi_{c_1,c_2,t} = \exists c_m,\ \Phi_{c_1,c_m t/2} \wedge \Phi_{c_m,c_2,t/2}$.

But this is equivalent to: $\Phi_{c_1,c_2,t} = \exists c_m,\ \forall (c_3,c_4) \in \{(c_1,c_m),(c_m,c_2)\},\ \Phi_{c_3,c_4 t/2}$.

This does not cause a blow up in formula length - at each step, formula length increases by a constant amount! Also, this reduction is quadratic time, so the reduction happens in *poly*.

Hence, `TQBF` is `PSPACE`-complete.

## 5.2 Relativization – P versus NP (Baker Gill Soloway '75)

there exists oracles A, B such that — $P^A = NP^A, P^B \neq NP^B$. If our proof for $P \neq NP$ tries to relativize, then our proof will not work, because we have choice of oracles $A$ and $B$ which allow for both $P^A = NP^A$ and $P^B \neq NP^B$.

### 5.2.1 Intuition: program diagonalization proofs relativize

Assume that we are applying diagonalization of programs. What we are actually doing is we are talking about the execution of programs on a universal turing machine $U(M, x)$ where $M$ is the programs we diagonalizing on.

Now, if we have $M^A$, we can simply give the universal TM access to $A$, and then $U^A(M^A, x)$ can be diagonalized the exact same way. So, in some sense, program diagonalization commutes with relativization.

### 5.2.2 Proof of BGS

$\texttt{P}^{\texttt{TQBF}} = \texttt{NP}^{\texttt{TQBF}}$

$NP^{TQBF} \subset NPSPACE$, since $NPTIME \subset NPSPACE$. (replace oracle call to TQBF with actual code, everything will livein NPSPACE).

This means that $NP^{TQBF} \subset NPSPACE = PSPACE$.

Next $PSPACE \subset P^{TQBF}$, by the same argument that we had used for $NP$.

Hence,

$NP^{TQBF} \subset NPSPACE = PSPACE \subset P^{TQBF}$ (in fact, $PSPACE = P^{TQBF}$.

$\texttt{P}^{\texttt{B}} \neq \texttt{NP}^{\texttt{B}}$**:**

Let us assume we have our relativizing oracle $B$.

$U_B = \{n \in \mathbb{N} \mid \exists x \in \Sigma^*, |x| = n, x \in B\}$. Informally, $U_B$ is the set of all possible string lengths in the language $B$.

$U_B \in NP^B$, because for a given $n$, it will try to guess the string $x$ which will be a ceritificate for $n$, and will verify it using the oracle-access to $B$.

Next, we wish to show that $U_B \notin P^B$. We need to build $B$ in such a way that this happens. We build $B$ in stages.

At stage $i$, some finite number of strings would be put in $B$. Take a TM $M_i$ which runs in $O(n^i)$. We want $M_i^B \notin P$. Conider $M_i^B(k)$. Some oracle calls from $M_i$ have been made before. If the oracle call was made before, then be consistent with the answer given before.

However, if we get a **new** query, we want a setup such that $k \notin U_B$ iff $M_i^B(k) = \texttt{YES}$, and similarly $n \in U_B$ iff $M_i^B(k) = \texttt{NO}$. The way we do this is by making $k$ so large that it is not possible to check efficiently whether $k \in_? U_B$.

Make $k$ large enough that all of it cannot be seen in polytime. Now, in this case, extend $B$ such that checking if $k$ belongs to $U_B$ is outside the reac of $M_i^B(k)$.

**TODO: read this proof, this makes no sense to me**

# Chapter 6

# LogSpace

## 6.1 The languages `L` & `NL`

`L` $\equiv DPSACE(\log(n))$, `NL` $\equiv NSPACE(\log(n))$

**Open problem:** `L` **versus** `NL`.

`L` $\subset$ `P` since configuration space is $2^{O(\log(n))} = O(n)$, hence we can brute force the configuration space.

### 6.1.1 `NL`-completeness

A language $A$ is `NL`-complete if:

- $A \in$ `NL`
- $B \in$ `NL`$, B \leq_L A$

Note that $B \leq_L A$ uses a mapping reduction: $\exists f : \Sigma^* \to \Sigma^*$, $f$ converts inputs of $B$ to inputs of $A$ in log-space.

$f$ must be **implicitly computable**: That is, the problem of generating the $i$-th bit should be done in `L`.

## 6.2 The `PATH` problem

`PATH` $= \{\langle G, s, t\rangle \mid \exists s \to t$ path in $G\}$

**Theorem 1** *PATH* $\in$ *NL*

*Proof.* We perform the naive solution — We try to explore all possible path possibilities for $s \to t$.

**Theorem 2** $A \in$ *NL* $\implies A \leq_L$ *PATH*

*Proof.* Since $A \in$ `NL`, there exists an NDTM $M$ which decides $A$ by using $O(\log(n))$ space.

We try to walk through the configuration graph of $M$ (henceforth called $Config_M$, from which we can try to find the path from the initial configuration to the final configuration. We can then try to solve `PATH`$(Config_M, c_{init}, c_{final})$, which will tell us whether an accepting path exists.

## 6.3 Verifiers

A turing machine $T$ verifies language $L$ if $L = \{w \mid \exists c, V(w,c) \text{ accepts}\}$. Note that if $\forall w, |c| = 0$, then $L$ is decidable (that is, no advice is needed).

**Theorem 3** *if $L \in nptime$, then $L$ has an efficient verifier.*

*Proof.* Let $N$ be a NDTM that decides $L$. Notice that the accepting path of $L$ is poly.

We create a certificate $c$ for $V(w,c)$ to be the address of the accepting path.

**Theorem 4** *if $L$ has an efficient verifier, then $L \in$ `NP`*

*Proof.* Let $V$ be the verifier. Let $N$ be the NDTM we are constructing that decides $L$. $N$ guesses the certificate $\overline{c}$ and then runs $N(w) = V(w, \overline{c})$.

## 6.4 `co-NP`

`co-NP` $\equiv \{L^c \mid L \in \texttt{NP}\}$ Since `SAT` $\in$ `NP`, $\texttt{SAT}^c \in$ `co-NP`. How do we convince someone that something is **not** satisfiable?

Let's look at this from the verifier perspective: The certificate of all outputs is pretty useless, since that is exponential. So, we need a short certificate which allows us to decide whether a given `SAT` problem is unsolvable.

Since `P` = `co-P` (since there is no non-determinism, we can simply flip the answer), hence `P` = `co-P` $\subseteq$ `co-NP`

Let's take a closer look at the verifiers:

- $L \in \texttt{NP}, w \in L \iff \exists c, V(w,c) = 1$
- $L \in \texttt{co-NP}, w \in L \iff \forall c, V(w,c) = 0$

So, we get some sort of duality between $\forall$ and $\exists$.

## 6.5 `CLIQUE`

$$\begin{aligned}
\texttt{CLIQUE} &= \{\langle G,k\rangle \mid G \text{ has a clique of size } K\} \\
\texttt{MAXCLIQUE} &= \{\langle G,k\rangle \mid G \text{ has a clique of size } K, \text{ does not have a clique of size } K+1\} \\
\langle G,k\rangle \in \texttt{MAXCLIQUE} &\iff \exists c, V(G,k,c) = 1 \land \forall u, W(G,k+1,u) = 0 \\
\langle G,k\rangle \in \texttt{MAXCLIQUE} &\iff \exists u_1, u_2, M(G,k,u_1,u_2) = 1
\end{aligned}$$

This new class is called as $\Sigma_2^p$.

$$\begin{aligned}
L \in \Sigma_i^p &\iff \exists u_1 \forall u_2 \exists u_3 \forall u_4 \ldots, Q_i u_i, M(w,u_i) = 1 \\
L \in \Pi^p &\iff \forall u_1 \exists u_2 \forall u_3 \exists u_4 \ldots, Q_i u_i, M(w,u_i) = 1
\end{aligned}$$

$\texttt{NP} = \Sigma_1^p$, $\texttt{co-NP} = \Pi_1^p$

We can now ask question between levels of the PH –

$\Sigma_{i+1}^p \supseteq_? \Sigma_i^p$ – Intuitively, the higher the $i$, the more restricted the question!

If we can show that $\Sigma_{i+1}^p = \Sigma_i^p$, then $\forall k \in \mathbb{N}, \Sigma_{i+k}^p = \Sigma_i^p$ (ie, we can "collapse" the hierarchy)

**Sid question: why does the hierarchy collapse?** $PH = \cup_i \Sigma_i^p$

We know the existence of a $\Sigma_i^p$-complete problem for *every* $i$.

### 6.5.1 `PH`-completness

If there exists a language $L$ which is `PH`-complete, then the hierarchy collapses, ie, $\exists i, \Sigma_{i+1}^p = \Sigma_i^p$.

The current conjecture is that the polynomial hierarchy does not collapse – Hence, there does not exist a `PH`-complete problem.

### 6.5.2 `PH` ⊂ `PSPACE`

**Ramification**

We know that $TQBF$ is `PSPACE`-complete. If $TQBF \in \texttt{PH}$, then we have a `PH`-complete algorithm (which cannot exist, due to the conjecture).

**Theorem 5** *the existence of a PH-complete problem collapses PH*

*Proof.* Let $L$ be the `PH`-complete problem. We know $\exists i$, $L \in \Sigma_i^p$. Now, for any $k \in \mathbb{N}$, we can reduce problem $\Sigma_k^p$ to $L$. Hence, I can reduce all $k > i$ to $k = i$, and then solve it at level $i$. Hence, the `PH` will collapse at level $i$.

## 6.6 `co-NL`

$\overline{\texttt{PATH}} \in \texttt{co-NL}$.

What about `NL` *vs* `co-NL`?

### 6.6.1 Verifiers for `NL`, `co-NL`

Let us look at it in terms of verifiers:

Can we define `NL` in terms of verifiers? Yes, we can, if we can ensure a *read-once certificate.*

$$w \in \texttt{NL} \iff \exists u, V_{once}(w, u) = 1$$
$$w \in \texttt{co-NL} \iff \forall u, V_{once}(w, u) = 0$$

How do we give a cerificate for **no path?**

However, we now know that `NL` = `co-NL`

# Chapter 7

# Other flavours of complexity classes:`BPP`, `P/Poly`

## 7.1 Introduction

## 7.2 `P/Poly`

If `NP` $\subset$ `P/Poly`, then `PH` $= \Sigma_2^P$.

**Theorem 6** *Let* $L = \{1^n \mid n \in \mathbb{N} \wedge \ P(n)\}$*, where* $P : \mathbb{N} \to \{0, 1\}$ *is a filtering predicate. All such languages* $L \in$ *`P/Poly`.*

*Proof.* Check if $1^n \in_? L$. If $1^n \notin L$, Define $C_n \equiv$ reject all inputs. If $1^n \in L$, define $C_n \equiv$ accept all inputs.

The problem is that unary languages of this kind can be *uncomputable.*

$$\texttt{UHALT} \equiv \{1^n \mid n \text{ encodes } \langle M, w\rangle,\ M \text{ accepts}\}$$

`UHALT` $\in$ `P/Poly` since it is a unary language, but unfortunately, this language is undecidable.

The hardness is being pushed to the "construction of the circuit" — this feels pathological, so we may change this definition to allow us to fix this.

### 7.2.1 Karp-Lipton theorem

**Theorem 7** *If* `NP` $\subseteq$ `P/Poly`*, then* `PH` $= \Sigma_2^p$

*Proof.* Recall that $\Sigma_2^p = \Pi_2^p \implies PH = \Sigma_2^p$. So, we choose to show that $\Sigma_2^p = \Pi_2^p$.

The $\Pi_2^p$-complete problem is this: $\forall u \exists v, \phi(u, v) =_? 1$, where $\phi$ is a boolean formula.

We need to show that this problem is in $\Sigma_2^p$, if `NP` $\subseteq$ `P/Poly`.

If `NP` $\subseteq$ `P/Poly`, then `SAT` $\in$ `P/Poly`. So, given $\phi$, there exists a polynomial size Circuit family $\{C_n\}$, such that $C_n(\phi, u, v) \equiv \forall \phi, \forall u, \exists v, \phi(u, v) = 1$.

assume we have a `SAT` oracle. We can use it to find the satisfying assignment. Assume we have $\phi(x_1, x_2, \ldots x_n)$. If it says "no", then we say "no". If it says "yes", then we need to find the satisfying assignment. Betweem $\phi(0, x_2, \ldots, x_n)$, $\phi(1, x_2, \ldots, x_n)$, one of these must be satisfiable (as $\phi$ is satisfiable). Do this $n$ times, at which point we find the satisfying assignment.

This entire process can be written as a family of boolean circuits, $\{C'_n\}$, $C'_n(\phi, u) = v$.

$$\begin{aligned} \forall u \exists v, \phi(u, v) = 1 &\iff \\ \exists \{C'_n\}, \forall u, \phi(u, C'_n(\phi, v)) = 1 &\iff \\ \exists w, \forall u, \phi(u, \langle w \text{ interpreted as a circuit}\rangle(\phi, v)) = 1 \end{aligned}$$

But the final statement is in $\Sigma_2^p$!

## 7.3 BPP

### 7.3.1 BPP= P, the conjectured proof

If true randomness can make our algorithm run fast, then a pseudorandomness can also make our algorithm run fast, since BPPis still polynomial, but by definition, polynomial time adversary can't differentiate between PRNG and a true RNG.

So, a deterministic turing machine can try to brute force all the seeds of the PRNG.

However, for this, we need to know which seeds to look at. Somehow, a coding theory idea called **list decoding** (ambiguity-allowed decoding) will give us a polynomial number of seeds to try / brute force on.

We can brute force the list of seeds to try in poly-time, and we get poly number of seeds, so we can "de-randomize" BPP into P.

Some of the steps in this sketch is conjectured (eg, existence of PRNG). For this to happen, we need some assumptions such as P $\neq$ NP, (whose proof techniques we have been guaranteed by the end of the semester), so BPP $=$ P is something we should try to solve.

**Theorem 8** *BPP $\subset$ P/Poly: Randomization does not help us go beyond circuit complexity*

*Proof.* if $L \in$ BPP, with probability $\frac{2}{3}$ it will be correct. We can use Chernoff bounds to repeat the experiment and take majority, given polynomial number of rounds, we can take error down to **negligible (less than any polynomial function)**.

$$\forall w, P[M(w) = L(w)] \geq 1 - \frac{1}{2^{n+1}}$$

But for a length $n$, there are only $2^n$ strings.

We need to translate the definition of P/Poly from circuit complexity, to turing machines with advise.

# Chapter 8

# Exploring probabilistic complexity classes

- `BPP` $\subset \Sigma_2^p \cap \Pi_2^p$ (Sipser-Gaes theorem)
- `BPP` $\subset$ `P/Poly` (Adleman's theorem)
- Hierarchy theorem for circuit complexity — $SIZE(f(n))$

**Theorem 9** *BPP* $\subseteq \Sigma_2^p \cap \Pi_2^p$ *(Sipser-Gaes theorem)*

*Proof.* First, notice that `BPP` = `co-BPP` (flip the answer, similar to `P` = `co-P`).

Showing `BPP` $\subseteq \Sigma_2^p$, it would automatically imply that `BPP` $\subseteq \Pi_2^p$, since `BPP` = `co-BPP`. So now, all we need to show is that `BPP` $\subseteq \Sigma_2^p$.

$$\begin{aligned}&\text{Definition of } \texttt{BPP} \text{ for a language } L\text{:}\\ &\exists \text{poly time } M, \forall w \in L, P[M(w,r) = \mathit{accept}] \geq \frac{2}{3}\\ &\exists \text{poly time } M, \forall w \notin L, P[M(w,r) = \mathit{accept}] \leq \frac{1}{3}\end{aligned}$$

By repeating, we can make the probabilities:

$$\begin{aligned}&\text{Definition of } \texttt{BPP} \text{ for a language } L\text{:}\\ &\exists \text{poly time } M, \forall w \in L, P[M(w,r) = \mathit{accept}] \geq\ 1 - \frac{1}{2^n}\\ &\exists \text{poly time } M, \forall w \notin L, P[M(w,r) = \mathit{accept}] \leq \frac{1}{2^n}\end{aligned}$$

$$S_x = \{r \mid r \text{ is good for input x}\}$$

That is, given $r$ as the randomness for input $x$ on machine $M$, $M$ will accept with high probability (ie. $M(x,r)$ accepts with high probability).

There are only two cases: Either $|S_x|$ is large, or $|S_x|$ is very small, since $x \in L \vee x \notin L$. Let $|r| = m$. Now, we count the size of $|S_x|$ depending on the cases of $x \in_? L$.

$$x \in L \implies |S_x| \geq (1 - \frac{1}{2^n}) \cdot 2^m$$
$$x \notin L \implies |S_x| \leq \frac{1}{2^n} \cdot 2^m$$

We can exploit this idea to show the theorem: "every string either has a large $|S_x|$ or a small $|S_x|$ can be written as a "good" $\Sigma_2^p$ statement.

Let $S$ be any set, $|S| < 2^{m-n}$ ($|S|$ is small). Let $k = ceil(m/n) + 1$. Let us define:

$$S + u_i \equiv \{x + u_i \mid x \in S\}$$

where $+$ is bitwise-XOR.

First of all, note that $|S + u_i| = |S|$ (`blank` $+ u_i$ = XOR = bijective, hence we don't change cardinality).

$$\bigcup_{i=1}^{k} |(S + u_i)| \leq \sum_{i=1}^{k} |S + u_i| = \sum_{i=1}^{k} |S| = k|S| = k \cdot 2^{m-n}$$

$$\forall u_1, u_2, \ldots u_k \in \{0,1\}^*, |u_i| = m, \bigcup_{i=1}^{k} (S + u_i) \neq \{0,1\}^m$$

Immediate, because $|\bigcup_{i=1}^{k}(S + u_i)| \leq k \cdot 2^{m-n}$, $|\{0,1\}^m| = 2^m$. ($m = poly(n)$, hence $k = poly(n)$, now argue inequalities).

Next, we argue something similar for *large* sets. If $|S| \geq (1 - \frac{1}{2^n}) \cdot 2^m$,

$$Stmt \equiv \exists u_1, u_2, \ldots u_k \in \{0,1\}^*, |u_i| = m, \bigcup_{i=1}^{k} (S + u_i) = \{0,1\}^m$$

We prove this using the **probabilistic method**. $P[Stmt] > 0$ is what we want to show. Let us show this by considering the *converse*: $P[\neg Stmt] < 1$.

We first make some definitions to state formally:

$$B_r \equiv r \notin \bigcup (S + u_i)$$

$$B_r^i \equiv r \notin S + u_i$$

The converse is now the probability that $B_r$ does not cover $\{0,1\}^m$. We know that

$$|S + u_i| \geq (1 - \frac{1}{2^n}) \cdot 2^m$$

. Note that in this probability, we range over all $u_i$.

$$P[B_r^i] \leq \texttt{TODO: complete this} = 2^{-n}$$
$$P[B_r] \leq k \cdot 2^{-n} \leq 1$$

Hence, there must exist $\{u_i\}$.

So now, we can pose the membership query $x \in_? L$ as:

$$\exists u_1, u_2, \ldots u_k, \forall r \in \{0,1\}^m, r \in \bigcup_{i=1}^{k} (S_x + u_i)$$

If the statement had been $r \in S_x$, then this means that $M(x, r) = \texttt{ACCEPT}$. However, here we know that $r \in \bigcup_{i=1}^{k} (S_x + u_i)$. From this, we infer that $\exists i, r \in S_x + u_i$. This means that $r = r_0 + u_i$ where $M(x, r_0) = \texttt{ACCEPT}$. Hence, this means that $r + u_i$ will accept $M(x, r + u_i) = M(x, r_0 + u_i + u_i) = M(x, r_0) = \texttt{ACCEPT}$.

So, we can create a new machine $M'(x, r) = \text{run } M(x, r + u_i), \forall i \in [1, k]$. Note that $M'$ is in poly, since $M$ is poly, and $k = \mathit{ceil}(m, n) = \mathit{poly}(n)$, since $m$ is the amount of randomness the machine $M$ consumes.

**Figure out where we need the $k = \cdots + 1$ precisely (why the $+1$)**

$$\exists u_1, u_2, \ldots u_k, \forall r \in \{0,1\}^m, M'(x, r) = \texttt{ACCEPT}$$

This statement is now in $\Sigma_2^p$, since $M' \in \texttt{P}$, and we have the correct $\exists\forall$ structure.

**Intuitive picture: if $x \in L$, then the set $S_x$ is large enough that we only need a polynomial number of $\{u_i\}$ to "spread" $S_x$ around to cover all possible $r$, and the converse**.

## 8.1 BPP ⊂ P/Poly (Adleman's theorem)

$$P(M[x, r] = L(x)]) \geq 1 - \frac{1}{2^{n+1}}$$

Let the length of $r$ be $m$, and hence there are $2^m$ number of $r$. The number of bad $r$'s is $\leq \frac{2^m}{2^{n+1}}$.

There are only $2^n$ possible $x$ inputs to the machine $M$.

Let us try to count the number of bad things for *any* $x$. This will be $\leq 2^n \cdot \frac{2^m}{2^{n+1}} = \frac{2^m}{2} < 2^m$.

So, there must exist a random string $r$ that is good for **all** $x$. Give this as the advice string to derandomize. So now, we got the advice sring for `P/Poly` for every $n$.

This does not allow us to reduce to `P`, because we still need to know this special $r$ that allows us to derandomize every string of length $n$.

Note that we do need polynomial time here, so we can get exponentially close to 1.

What we are using is that for a string that belongs in the language, there are *many* certificates, and similarly for strings that don't belong, there are *very few* certificates (the "barren middle" exists for `BPP`).

## 8.2 Computation is messy (Philosophy)

Computation is in itself a ”messy” object. Small changes in input leads to huge changes in output.

In most of known mathematics, it is not as sensitive.

# Chapter 9

# Property testing

We have some language $L \subset \{0,1\}^*$. We need to decide if $x \in_? L$ by only querying a *fixed number of query locations* in $x$. the query number $q$ is fixed and is *independent* of $n$.

## 9.1 `PARITY`

$$\texttt{PARITY} \equiv \{x \in \{0,1\}^* \mid \#1\text{'s in } x \text{ is even}\}$$

parity cannot be done by querying an independent number of locations, since intuitively, an adversary can flip bits at locations that have not been read by us.

## 9.2 `MAJORITY`

$$\texttt{MAJORITY} \equiv \{x \in \{0,1\}^* \mid \#1\text{'s in } x \text{ is } \geq n/2\}$$

This is also not possible to be solved, because the adversary can flip bits of a string where $\#1's = n/2$.

## 9.3 Approximate decision problems

$x \in L$ versus $x$ which is $\epsilon$-far from $L$. ($x$ is $\epsilon$-far from $L$ if there exists no string in $L$ with hamming distance less than $\epsilon$ in L).

x is $\epsilon$-far from y if $Ham(x,y) \geq \epsilon n$. x is $\epsilon$-far from $L$ if $\forall y \in L, Ham(x,y) \geq \epsilon n$.

### 9.3.1 $\epsilon -$ `MAJORITY`

$\#1's \geq n/2$, then accept. $\#1's \leq n/4$, then reject. The first part is strings in the language. The former part is strings that are $\epsilon = 1/4$ far away from `MAJORITY`.

- sample $i_1, \ldots i_k \in [1 \ldots n]$.
- query $x_{i_1}, \ldots x_{i_k}$.
- If $x \in$ `MAJORITY`, $E[\frac{\#1}{k}] > \frac{1}{2}$

- If $\#1's < n/4$ ($x \notin$ `MAJORITY`), $E[\frac{\#1}{k}] \leq \frac{1}{4}$
- If majority among $x_{i_\alpha}$ is 1, then output YES.

#### Chernoff / Chebyshev bounds

Let $\mu \equiv \frac{\#1s}{n}$. $X_i \equiv$ number of 1s in $x_i$ is at least $\frac{1}{2} - \epsilon$. $X_i$ is a binary random variable, which takes the value 0 with probability $1 - \mu$, and 1 with probability $\mu$.

We are taking $k$ independent samples of the same random variable $X_i$, to give us $\sum_i X_i/k$.

Chernoff bounds state that:

$$Pr\left[\left|\frac{\sum_{i=1}^{k} X_i}{k} - \mu\right| \geq \frac{1}{8}\right] \leq \frac{1}{O(2^k)}$$

#### `MAJORITY`structure

`MAJORITY` is invariant under permutations. What we are actually trying to find is statistical properties. Next, we will see a non-statistical example.

## 9.4 `SORTED`

$$\texttt{SORTED} \equiv \{0^n 1^m \mid n, m \in \mathbb{N}\}$$

### 9.4.1 `SORTED` test

- First query $F_1 \equiv \{n/6, 2n/6, 3n/6, 4n/6, 5n/6, n\}$
- query $F_2 \equiv \{m \text{ random locations}\}$
- accept iff $x|F_1 \cup F_2 \in$ `SORTED`. $x|S \equiv x$ restricted to $S$ indeces.

Suppose $x$ is $n/3$-far from being sorted. So now, for rejection, we have two cases:

#### Case 1: $x|F_1$ is not sorted

We reject directly.

#### Case 2: $x|F_1$ is sorted

Consider an example string,

`STR ≡ 0 (0 1) 1 1 1`

The gap between the consecutive locations highlighted ((0 1) is $\frac{n}{6}$ Recall that $x$ is different from a sorted string at more than $\frac{n}{3}$ points. At most $\frac{n}{6}$ can fit between two consecutive locations. So, there is an "error" of $\frac{n}{6}$ to be spread around before and after this consecutive location.Call the ones before as $\mu_1$, and then ones after as $\mu_2$. Note that $\mu_1 + \mu_2 \geq \frac{n}{6}$.

$$Pr[rej] \geq 1 - \left(1 - \frac{1}{6}\right)^m$$

### 9.4.2 Property testing on languages that are invariant under subgroups of permutations

Goldreich, property testing.

## 9.5 Linearity testing

$f : G \to H$ is a function where $G, H$ are groups. We're trying to check if it's a homomorphism.

$$f \text{ is linear} \equiv f(x +_G y) = f(x) +_H f(y)$$

We want to write an algorithm that tests if a function is linear, or is not linear.

- choose $x$, $y$ ranomly from $G$.
- query $f$ at $x$, $y$, $x +_G y$.
- Accept if $f(x +_G y) =_? f(x) +_H f(y) = true$

### 9.5.1 Local consistency

$$\epsilon(f) \equiv Pr_{x,y}\Big[f(x) + f(y) \neq f(x+y)\Big]$$

### 9.5.2 Global consistency

$$\delta(f) = Ham(f, L)$$

Completeness statement: $\delta(f) = 0 \implies \epsilon(f) = 0$

Soundness statement: $\delta(f) \leq 2\epsilon(f)$. (If your distance from $L$ is large, then we reject with high probability).

### 9.5.3 Proof of soundness: $\delta(f) \geq 2\epsilon(f)$

$\phi(x) \equiv \text{plurality}_y[f(x+y) - f(y)]$

plurality $\equiv$ take value that occured maximum number of times.

- $\delta(f, \phi) \leq 2\epsilon(f)$
- $\forall x, Pr_y\big[\phi(x) = f(x+y) - f(y)\big] \geq 2/3$
- $\phi$ is linear.

We first prove that $\phi$ is linear using the other two facts. We then prove the other two facts.

### 9.5.4 $\phi$ is linear

$$\begin{aligned}
&\phi(x) = f(y) - f(y-x) \text{ for all but } \frac{1}{3} y's \\
&\phi(z) = f(y+z) - f(y) \text{ for all but } \frac{1}{3} y's \\
&\phi(x+z) = f(y+z) - f(y-x) \text{ for all but } \frac{1}{3} y's \\
&\phi(x) + \phi(z) = -f(y-x) + f(y+z) = \phi(x+z)
\end{aligned}$$

### 9.5.5 $\delta(f, \phi) \leq 2\epsilon(f)$

Let us define

$$BAD = \left\{ x \in G \;\middle|\; Pr_y\big[f(x) \neq f(x+y) - f(y)\big] \geq \frac{1}{2} \right\}$$

Note that $x \notin BAD \implies f(x) = \phi(x)$.
$BAD$ controls how bad $f$ can differ from $\phi$:

$$\delta(f, \phi) \leq \frac{|BAD|}{|G|}$$

$$\begin{aligned}
&\epsilon(f) \equiv Pr_{x,y}\Big[f(x) + f(y) \neq f(x+y)\Big] \\
&\geq Pr[x \in BAD] Pr\Big[f(x) \neq f(x+y-f(y) \;\Big|\; x \in BAD\Big] \\
&\geq \frac{|BAD|}{|G|} \frac{1}{2} \\
&\geq \frac{\delta(f, \phi)}{2}
\end{aligned}$$

# Chapter 10

# `IP`

## 10.1 The class `IP`

TODO: copy notes from the algorithms class, they're identical. Unfortunately, I missed this because I was sick.

We know that `DIP` (deterministic IP) $=$ `NP`. We also conjecture that `P` $=$ `BPP`. So, clearly, randomization and interactivity alone don't give much.

It is surprising that `NP` $\subseteq$ `IP` (since GNI is in `IP`), since we are combining two powers that are useless on their own. So, it's shocking that `IP` $=$ `PSPACE`!

## 10.2 Arithmetization

We define a new problem,

$$\texttt{\#SAT} \equiv \{(\phi, k) \mid \phi \text{ is a 3-CNF boolean formula with exactly } k \text{ satisfying clauses}\}$$

.

Note that $\overline{\texttt{3-SAT}} = \texttt{\#SAT}(0)$. Hence, `#SAT` is a generalization of a problem that is not known to be in NP ($\overline{\texttt{3-SAT}}$). We will use *arithmetization* to show that $\texttt{\#SAT} \in \texttt{IP}$.

We have a prover $P$, and a formula $\phi$, with $k$ satisfying clauses. $P$ knows $(\phi, k)$, and wants to convince the verifier $V$ of this fact.

We have a boolean formula $\phi$ with boolean variables. If we choose to work in a larger field $F_p$, $\{0,1\} \in F_p$. A boolean function can be viewed as a polynomial in a larger field, which agrees with the boolean formula on $\{0,1\}$. For instance, if $\phi = x_1 \wedge x_2$, we can create a polynomial $q(x_1, x_2) = x_1 x_2$. Clearly, $q$ and $\phi$ agree on $x_1, x_2 \in \{0,1\}$.

- $\phi(x) = \neg x \leftrightarrow q_\phi(x) = (1 - x)$
- $\phi(x_1, x_2) = x_1 \wedge x_2 \leftrightarrow q_\phi(x_1, x_2) = x_1 x_2$
- $\phi(x_1, x_2) = x_1 \vee x_2 \leftrightarrow q_\phi(x_1, x_2) = 1 - (1 - x_1)(1 - x_2)$

Now, we can express every boolean formula as a polynomial.

### 10.2.1 Rewrite #SATin terms of arithmetization

$$(\phi, k) \in \texttt{\#SAT} \leftrightarrow k = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} Q_\phi(x_1, x_2, \ldots, x_n)$$

### 10.2.2 Interactive proof for $k$ in terms of $Q_\phi$

We provide a recursive solution to verify

$$k = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} Q_\phi(x_1, x_2, \ldots, x_n)$$

$h(x_1) \equiv Q_\phi(x_1, b_2, b_3, \ldots b_n)$, where $b_i \in \{0, 1\}$ are constant. Now, $h(x_1)$ is a univariate polynomial of degree $d$.

If we consider all possibilities of $b_2 \ldots b_n$, we get $2^{n-1}$ variants of the $h(x_1)$ polynomial. We consider all of them by summing over all possiblities, and collecting all of them in $H$.

$$H(x_1) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} Q_\phi(x_1, b_2, b_3, \ldots b_n)$$

Note that $H(x_1)$ still has degree $d$, since it is the sum of many $h(x_1)$ polynomials. However, see that $H(x_1)$ is an exponential sum (it has exponential number of terms), and therefore the verifier can't simply construct $H(x_1)$ it **needs** the prover to proxy for $H(x_1)$ in some sense. This is where we need the unbounded prover.

We can now see that the original statement is the same as saying

$$k = H(0) + H(1)$$

### 10.2.3 Using this property to verify

$n = 1$

If $n = 1$, then $V$ checks that $k = Q_{phi}(x_1)$.

$n > 1$

- If $n > 1$, $V$ asks $P$ to give $H(x_1)$. $P$ gives $S(x)$.
- $V$ checks if $k = S(0) + S(1)$. If not, reject. If success, then we need to verify that $S =_? H$.
- $V$ chooses $a \in_{random} F_p$, and computes $S(a)$.
- Recursively ask for a proof that $S(a) = \sum_{b_2} \sum_{b_3} \cdots \sum_{b_n} Q_\phi(a, b_1, b_2, \ldots b_n)$. This is a recursive step since I can write this as:

$$S(a) = \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} Q_\phi(a, x_2, \ldots, x_n)$$

Compare to

$$k = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} Q_\phi(x_1, x_2, \ldots, x_n)$$

and now $Q'_\phi(x_2, \ldots x_n) \equiv Q_\phi(a, x_2, \ldots, x_n)$ is a polynomial of degree $n-1$.

### 10.2.4 Completeness

The prover $P$ that can actually construct the correct $H$ will cause the verifier to accept with probability 1.

### 10.2.5 Soundness

We are currently checking of $(H-S)(a) = 0$. We had chosen $a \in_{random} F_p$. Only if $a$ is a root of $(H-S)$, will the prover escape undetected.

There are only $d$ roots for the polynomial $(H-S)$, and $F_p$ has $p-1$ elements. So, the prover will be undetected with probability $P < \frac{d}{p-1}$.

Probability he is caught in the first round is $d/p$.

So, the probability a cheating prover will be detected will be $\big(1 - d/p\big)^n$.

## 10.3 `IP` = `PSPACE`

We show that `TQBF` $\in$ `IP`, hence `PSPACE` $\subseteq$ `IP`, since `TQBF` is a `PSPACE`-complete problem.

To show that `IP` $\subseteq$ `PSPACE`, notice that