Computational Complexity

Valerio Venanzi

Abstract

This document contains notes taken during the *Computational Complexity* course, held by Massimo Lauria during the academic year 2022-23. Page of the course at this link (if the link is broken it might be that too much time has passed and the link has been updated for another academic year).

These notes are not meant to be comprehensive, but rather a sort of compendium of the most important definitions and aspects of the course. There are many theorems, proof and examples missing here; so this is really more for helping memorize the key concepts.

As of right now there are no citations to bibliographic sources. I intend to put them in the future, but since, as mentioned above, this document alone is not enough to study the subject, I recommend checking professor Lauria's page of the course, which well documents the most important citations to the books that he used during the course.

Contents

1 Turing Machines

Definition 1.1 (Turing Machine (TM)). *Let $M = (\Gamma, Q, \delta)$ be a TM:*

- Γ *is a finite alphabet*
- Q *is a finite set of states*
- δ *is a transition function*

Γ often contains:

- D, start of the tape
- $\sqcup$, blank
- 0 and 1, or maybe all the integer from 0 to 9

M contains k tapes:

- the first contains the *input* and is *read-only*
- the last contains the *output* at the end of the computation
- the rest are work tapes

What does it mean for a TM to compute a function?
Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. A TM calculates/computes $f(x) = y$ if it starts with x on the input tape, and, if at some point it halts, than it has y on the output tape.

Definition 1.2 (Running time). *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function, and let $T : \mathbb{N} \to \mathbb{N}$ be a time function.*

A TM M computes f in time T if, whenever M has x on its input tape and starts the computation $M(x)$ it stops within $T(|x|)$ steps, and has $f(x)$ on the output tape.

Definition 1.3 (Time constructible). *A function $T : \mathbb{N} \to \mathbb{N}$ is time constructible if:*

- $T(n) \geq n$ *(not necessary but useful)*
- *there is a TM M_T s.t. $M_T(x) =$ "binary encoding of $T(|x|)$"*

Definition 1.4 (Oblivious TM). *A TM is oblivious if, given an input x, the position of the heads at the i-th step only depends on $|x|$ and i.*

1.1 TM Encodings

Any kind of TM rely on finite resources, thus also the transition function is finite. So any TM can be encoded with a finite binary string.

Let M be a TM; $M \mapsto X_M \in B^*$, where X_M is the binary encoding of M. We can now also read an encoding: we *decode* $\alpha \in \{0,1\}^*$ and get the TM M_α encoded by α.

We assume that is always possible to decode an encoding. If a binary string is rubbish, then we can decode it into a trivial TM that, for instance, always rejects.

Also, any TM can be encoded by infinitely many strings, simply by padding a valid encoding with rubbish bits; similarly to how we can add comments to a program and write infinitely many source codes that compute the same thing.

1.2 Universal TM

Let U be the universal TM. Given the description of a TM and an input, U can run/simulate the given TM with the given input. That is, $U(\alpha, x)$ runs $M_\alpha(x)$.

We can also decide to stop the simulation after t steps; in this case we write $U(\alpha, x, t)$.

Simulating comes with a time loss: from time $T(n)$ to $T(n) \cdot log(T(n))$.

We can always make a simulation *oblivious*.

2 Introduction to complexity classes

2.1 Uncomputable functions

There are functions that can't be computed. They are also said to be *undecidable*.

2.1.1 UC

Consider the following function:

$$UC(\alpha) = \begin{cases} 0 & \text{if } M_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

Through *diagonalization* we can prove that UC is uncomputable.

2.1.2 $HALT$

$$HALT(\alpha, x) = \begin{cases} 1 & \text{if } M_\alpha(x) \text{ terminates in finite steps} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

We can prove that $HALT$ is uncomputable by *reduction*: if $HALT$ is computable, then we can compute UC. For the definition of *polynomial reductions* see Karp reduction.

2.2 Time classes

Definition 2.1 ($DTIME$)**.** *Let $f : \{0,1\}^* \to \{0,1\}$ be a function and $T : \mathbb{N} \to \mathbb{N}$ be a time function. Consider the set $L_f = \{x \mid f(x) = 1\}$.*

$L \in DTIME[T]$ if there is a TM M s.t.

- *M runs in time $O(T(n))$*
- *M decides L*

2.2.1 P

Definition 2.2 (**P**).

$$P = \bigcup_{c>0} DTIME[n^c]$$

We will consider any set in ***P*** *to be efficiently computable.*

2.2.2 NP

Problems in **NP** capture the idea of puzzles. Given a puzzle it is hard to solve it, but given a possible solution to the puzzle, it is easy to check if such solution is correct.

Definition 2.3 (**NP**). *Let L be a language. $L \in NP$ if there is a machine $V(\cdot,\cdot)$ and a polynomial p s.t.*

1. *$\forall x, y\ V(x,y)$ runs in time $p(|x|)$ (**efficienct**)*
2. *$x \in L$, $\exists y$ s.t. $|y| \leq p(|x|)$ and $V(x,y) = 1$ (**completeness**)*
3. *$x \notin L$, $\forall y\ V(x,y) = 0$ (**soundness**)*

x is an input for an **NP** problem, and y is a **witness**, a possible solution to the problem. V is a *verifier machine*, that verifies if y is a *valid witness* for x; V runs in polynomial time *w.r.t.* $|x|$ and can't be fooled (i.e. soundness).

We know that $P \subseteq NP$: given a **P** problem we can just give it a witness, ignore the witness and solve the problem directly in polynomial time.

2.2.3 EXP

Definition 2.4 (**EXP**).

$$EXP = \bigcup_{c>0} DTIME[2^{n^c}]$$

It is clear that

$$P \subseteq NP \subseteq EXP$$

We also know that $P \subsetneq EXP$, which means that there are problems strictly not solvable in polytime, and require exponential time.

2.3 Reductions

Problem A **reduces** to problem B when: given an efficient algorithm for B, we can use it to build an efficient algorithm to solve A.

We say that A is "easier" than B.

Definition 2.5 (Karp reduction). *Such reductions are often called just reduction or polytime (computable) reductions.*

A language $L \subseteq \{0,1\}^*$ is polytime (Karp) reducible to a language $L' \subseteq \{0,1\}^*$ if there exists a polytime computable function $f : \{0,1\}^* \to \{0,1\}^*$ s.t.

$$x \in L \iff f(x) \in L'$$

So, if L is reducible to L' we write $L \leq_p L'$.

2.4 NP-Hardness

A language S is NP-Hard if $\forall L \in \mathbf{NP} L \leq_p S$. So an NP-Hard problem is harder than any other problem in **NP**.

2.5 NP-Completeness

A problem L is **NP-Complete** if:

- L is **NP**-Hard
- $L \in$ **NP**

Some properties:

- if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$
- if L is **NP**-Hard and $L \in P$, then **P**= **NP**
- if L is **NP**-Complete then $L \in$ **P** iff **P**= **NP**

Theorem 2.1 (Cook-Levin)**.** *The following are **NP**-Complete problems:*

1. *SAT*
2. *3-SAT*
3. *Circuit-SAT*

2.6 On NP

NP stands for **Nondeterministic Polynomial**, because a problem in **NP** follows the computation of a **NonDeterministic Tugin Machine** (**NDTM**).

In a normal TM the computation follows a chain, it is, indeed, deterministic. IN a NDTM computation branches out, in a binary tree fashion; e.g. given n boolean variables, a NDTM, for each variable, tries all possible computations with such variable both set as 0 as well as 1, so at the end it tried all possible 2^n combinations.

A NDTM M runs in time $T(n)$ if all paths of the computation of $M(x)$, for any input x, terminate within time $T(|x|)$.

If a language L is decided by a NDTM M in time $T(n)$, then:

- M runs in time $T(n)$
- $x \in L$, then there is a path s.t. $M(x) = 1$ (i.e. it accepts)
- $x \notin L$, then for all paths $M(x) = 0$ (i.e. always rejects)

Definition 2.6 (*NTIME*)**.**

$$NTIME[T(n)] = \{L \mid L \text{ is decided by NDTM in time } T(n)\}$$

So we can give the following alternative definition for **NP**:

$$\mathbf{NP} = \bigcup_{c>0} NTIME[n^c]$$

3 Some problems

3.1 SAT and variations of it

Let φ be a boolean formula. The following problems can be defined for any kind of boolean formula, albeit it is often the case that we consider φ to be in CNF form.

We often denote an assignment of the variables of a formula with the letter α.

Definition 3.1 (SAT)**.**

$$SAT = \{\varphi \mid \textit{there is an assignment } \alpha \textit{ s.t. } \varphi(\alpha) = 1\}$$

Definition 3.2 (k-SAT)**.** *Fix k, we consider φ to be a CNF boolean formula s.t. each clause contains exactly k literals.*

$$k\text{-}SAT = \{\varphi \mid \textit{there is an assignment } \alpha \textit{ s.t. } \varphi(\alpha) = 1\}$$

3.2 Problems on graphs

For the following subsection the letter G will denote a graph.

3.2.1 Independent Set

Definition 3.3 (Independent Set)**.** $S \subseteq V(G)$ *is an independent set if* $\forall x, y \in S$ $\{x, y\} \notin E(G)$.

So we define the decision problem of finding if a graph has an independent set as follows:

$$IndSet = \{(G, k) \mid G \text{ has an independent set of size } \geq k\}$$

Theorem 3.1. *3-SAT $\leq_p$ IndSet*

Also, it is easy to see that IndSet is in **NP**, so it is **NP**-Complete.

3.2.2 Clique

Let k be an integer.

Definition 3.4 (Clique)**.**

$$Clique = \{(G, k) \mid G \textit{ has a clique of size } k\}$$

Observe that $(G, k) \in Clique \leftrightarrow (\bar{G}, k) \in IndSet$. So $Clique$ is **NP**-Complete.

3.2.3 Vertex Cover

Definition 3.5 (Vertex cover)**.** *$C \subseteq V(G)$ is a vertex cover if $\forall \{x, y\} \in E(G)$ x or y are in C, and C covers all the edges; i.e. each edge of the graph has at least one node in the cover.*

So we define the decision problems as follows:

$$VC = \{(G, t) \mid G \text{ has a vertex cover of size } \leq t\}$$

It is known that the complement of an *independent set* is always a *vertex cover*. So we can prove that $IndSet \leq_p VC$ via the following mapping: $(G, k) \mapsto (G, |V(G)| - k)$. This proves that VC is **NP**-Complete.

3.2.4 Eulerian graph

Definition 3.6 (Eulerian walk)**.** *A walk on a graph is said to be eulerian is it touches all edges exactly once.*

A graph is said to be *eulerian* if it contains an eulerian walk.

This problem is actually solvable in polynomial time; i.e. is in **P**.

To check if a graph is eulerian we can do as follows. For each vertex calculate $d = indegree - outdegree$; if one of the two following conditions hold then the graph is eulerian:

- $d = 0$ for each vertex
- one vertex has $d = 1$, one vertex has $d = -1$ and all other vertices have $d = 0$

3.2.5 Hamiltonian path

Definition 3.7 (Hamiltonian path)**.** *A walk on a graph is said to be Hamiltonian if it touches all nodes exactly once.*

The statement is similar to the one for Eulerian walks, but this problem is actually in **NP**.

4 Search vs Decision

Consider φ a boolean formula.

$$Decision(\varphi) := \begin{cases} SAT \\ \neg SAT \end{cases} \tag{3}$$

$$Search(\varphi) := \begin{cases} \alpha & \text{a SAT assignment of } \varphi \\ \perp & \text{if } \varphi \text{ is UNSAT} \end{cases} \tag{4}$$

Solving Search we can easily also solve Decision. Can we go in the opposite direction? We can, but it requires a bit more thought.

Let φ have n variables. Starting from φ with no variables fixed, we branch out all possible assignments. Each time we branch we fix the value of a variable and we ask the Decision procedure if the formula, with the fixed values, is satisfiable. In this way we also solve the Search problem.

5 coNP

5.1 Definition

By definition, **coNP** is the following set:

$$coNP = \{L \mid \bar{L} \in NP\}$$

This definition can be deceiving, beacause it might look like that $coNP = \bar{NP}$, but we know that this is not true (i.e. $coNP \neq \bar{NP}$). So we give an alternative definition.

Definition 5.1 (**coNP**). *A language L is in **coNP** if there is a TM $M(\cdot, \cdot)$ s.t.*

1. *$M(x, y)$ runs in polytime w.r.t. $|x|$ (x is the input and y the witness)*
2. *$x \in L$, then for all y $M(x, y) = 1$*
3. *$x \notin L$, there exists y s.t. $M(x, y) = 0$*

Recall the definition of NP; the consequences of a string being in a language are swapped for **NP** and **coNP**. That is, if a problem is in **NP** we need a single witness to verify that an instance is in the problem, while in **coNP** we need that all witnesses verify that an instance is in the problem. The opposite goes for an instance being not in the language.

5.2 Some problems

Some **NP** problems:

- SAT
- FALSIFIABLE (set of boolean formulas that can be falsified)

Some **coNP** problems (negations of the above **NP** problems):

- UNSAT
- TAUT (tautologies)

Claim 5.0.1. *$TAUT$ is **coNP**-Complete.*

Proof. Let $L \in coNP$. We want to prove that $L \leq_p TAUT$.

Surely $\bar{L} \in NP$, so $\bar{L} \leq_p SAT$, via some reduction f (that outputs a CNF).

$$x \in \bar{L} \longleftrightarrow f(x) \in SAT$$
$$x \in L \longleftrightarrow f(x) \notin SAT$$
$$x \in L \longleftrightarrow f(x) \in UNSAT$$
$$x \in L \longleftrightarrow \neg f(x) \in TAUT$$

□

6 Time Hierarchy

6.1 Diagonalization

It is a powerful technique that can be used to prove lower bounds on TMs. Powerful, albeit not enough to prove $P = NP$.

Things needed for diagonalization:

- representation of TMs as strings (i.e. encodings)
- universal TM that simulates the encoded machine efficiently
- every TM needs to have infinitely many encodings
- every string must represent a TM

Encoding wise we can have either:

- $x \in \{0,1\}^* \mapsto M_x$ (i.e. x is the string that encodes a machine)
- $i \in \mathbb{N} \mapsto M_i$ (i.e. we enumerate all TMs and i is the index of some TM)

We also might want a timer s.t. we can stop the computation of a TM after t steps.

Note: this subsection doesn't explain at all how Diagonalization works

6.2 Time Hierarchy Theorem

Theorem 6.1. $DTIME[n] \subsetneq DTIME[n^{1.5}]$

Proof. The goal is to build a language that is in $DTIME[n^{1.5}]$ and not in $DTIME[n]$.

Let D be a machine that diagonalizes. It works as follows.

$D(x)$:

- run $M_x(x)$ for $|x|^{1.4}$ steps
- if $M_x(x)$ outputs $b \in \{0,1\}$
  - output $1 - b$
- else
  - output 0

D basically flips the bits on the diagonal, if possible, otherwise it rejcts. It computes a language $L_D : \{0,1\}^* \to \{0,1\}$. The running time of D is $O(n^{1.5})$.

We want to show that $L_D \notin DTIME[n]$. We suppose by contradiction that $L_D \in DTIME[n]$.

So, there is a TM M that computes L_D in time $O(n)$.

For any α s.t. $M_\alpha = M$, $U(\alpha, x)$ runs in time $c \cdot n \cdot log(n)$. This means that $\exists n_0$ s.t. $\forall n > n_0\ c \cdot n \cdot log(n) < n^{1.4}$.

Pick $\hat{\alpha}$ s.t.

- $|\hat{\alpha}| > n_0$
- $M_{\hat{\alpha}} = M$ (my notes say $M_\alpha = M$, but I should check it, I'm not sure)

$U(\hat{\alpha}, \hat{\alpha})$ ends within time $|\hat{\alpha}|^{1.4}$.

So, $D(x)$ can obtain the output of $M(x)$ within $|x|^{1.4}$. but by definition of D, $D(x) = 1 - M(x) \neq M(x)$. Contradiction! We can't decide L_D in time $O(n)$. □

What this theorem proves, informally, is that if we give a TM more time then it will be able to compute more things. How much more time? In above theorem we've seen that going from $O(n)$ to $O(n^{1.5})$ is enough, but there is a more general version of this theorem.

Theorem 6.2 (General Time Hierarchy Theorem)**.** *If f and g are Time constructible functions, and $f(n) \cdot \log f(n) = o(g(n))$, then*

$$DTIME[f(n)] \subsetneq DTIME[g(n)]$$

Theorem 6.3. *If f and g are Time constructible functions, and $f(n+1) = o(g(n))$, then*

$$NTIME[f(n)] \subsetneq NTIME[g(n)]$$

6.3 On $P \neq EXP$

How do we prove that some problems have exponential lower bounds? Using the Time Hierarchy Theorem:

$$P \subseteq DTIME[2^{\sqrt{n}}] \subsetneq DTIME[2^{100\sqrt{n}}] \subseteq DTIME[2^n] \subseteq EXP$$

6.4 NP-Intermediate problems

Theorem 6.4 (Ladner)**.** *If $P \neq NP$, then there is a language L s.t.*

- *$L \in NP \setminus P$*
- *L is not NP-Complete*

These problems are so called **NP**-Intermediate: harder than P, easier than NP. They lay somewhere in the middle.

7 Machines with Oracles

7.1 Introduction

An oracle is some function $O : \{0,1\}^* \to \{0,1\}^*$. The purpose of an oracle is to give an answer to some problem.

Suppose a TM M has access to some oracle O. During the computation, M can ask O to solve a problem, to which O will answer in a single step (i.e. in time $O(1)$). We don't know how O works, we just query it and get answers.

If a TM M has access to an oracle O, then we denote it as M^O.

Oracles are "stronger" than reductions, they directly solve a problem for us. So we define classes of problems decidable given a certain oracle. For instance, P^{SAT} is the set of problems decidable in polynomial time, given an oracle for solving SAT.

We can also define classes like P^{NP}, where we have access to all oracle for NP. Note that having access to all oracles for NP is not more powerful than having access to an oracle for an NP-Complete problem.

7.2 $EXPCOM$

$$EXPCOM = \{\langle M, x, 1^n\rangle \mid M(x) \text{ outputs 1 in } \leq 2^n \text{ steps}\}$$

1^n is the unary representation of n: is it the concatenation of the number 1 n times.

How do we prove that $EXP \subseteq P^{EXPCOM}$? From $2^{n^{10}}$ we can build in polytime $\langle M, x, 1^{n^{10}}\rangle$ and the oracle for $EXPCOM$ to solve it.

8 Space complexity

8.1 Definitions

Definition 8.1 (Space)**.** *It is the number of locations in the work tapes ever visited during a computations. The input tape is read-only and not counted; i.e. if an algorithm only reads from the input tape then it uses $O(1)$ space.*

8.2 Space constructible functions

Definition 8.2 (Space constructible function)**.** *A function $s : \mathbb{N} \to \mathbb{N}$ is space constructible if there is a machine that, given input x, outputs $s(|x|)$ in space $s(|x|)$.*

8.3 Classes

Definition 8.3 ($SPACE$ class)**.** *Given a function $s : \mathbb{N} \to \mathbb{N}$ and a language $L \subseteq \{0,1\}^*$, $L \in SPACE[s(n)]$ if L is decided by a TM that uses $O(s(n))$ space on all inputs of length n.*

We can also define $NSPACE$ analogously to $SPACE$, except that we consider NDTMs instead of TMs.

Lemma 8.1.

$$DTIME[s(n)] \subseteq SPACE[s(n)] \subseteq NSPACE[s(n)] \subseteq DTIME[2^{O(s(n))}]$$

Following the most important classes for space complexity.

$$PSPACE = \bigcup_{c>1} SPACE[n^c]$$

$$NPSPACE = \bigcup_{c>1} NSPACE[n^c]$$

$$L = SPACE[\log n]$$
$$NL = NSPACE[\log n]$$

Let A and B be two languages s.t. $A \in PSPACE$ and $B \in P$. If $A \in P$, then $A \leq_p B$.

8.4 Space Hierarchy Theorem

Theorem 8.2 (Space Hierarchy Theorem)**.** *If f and g are Space constructible function, and $f(n) = o(g(n))$, then $SPACE[f(n)] \subsetneq SPACE[g(n)]$.*

8.5 Quantified boolean formulas

Definition 8.4 (Quantified Boolean Formula)**.** *A quantified boolean formula (**QBF**) is a boolean formula with the following structure. For each i, let $Q_i \in \{\exists, \forall\}$.*

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \varphi(\vec{x})$$

Such formula contains no free variables.

Consider the following problem:

$$TBQF = \{\varphi \mid \varphi \text{ is a true } \mathbf{QBF}\}$$

Theorem 8.3. *$TBQF$ is PSPACE-Hard.*

Proof. Let $L \in PSPACE$ be decided by a TM M in space $s(n)$. Given $x \in B^n$, the goal is it build a **QBF** Ψ of size $O(s^2(n))$, true iff M accepts x.

Consider the graph of configurations $G_{M,x}$; it can be proved that each configuration has polynomial size. We now build the formula Ψ recursively, as follows:

$$\Psi_{i+1}(A, B) = \exists X \forall D_1 \forall D_2((D_1 = A \wedge D_2 = X) \vee (D_1 = X \wedge D_2 = B)) \to \Psi(D_1, D_2)$$

$|\Psi_i| \leq 100 s(n) + |\Psi_{i-1}|$. So $|\Psi_{s(n)}| = O(s^2(n))$. □

Observe that above proof doesn't require the graph to have out-degree 1 on vertices, so it actually proves a stronger statement: $TBQF$ is **NPSPACE**-Hard. So, considering that $TQBF \in PSPACE \subseteq NPSPACE$ and $TBQF$ is **NPSPACE**-Hard, we get the following.

Corollary 8.3.1. ***PSPACE* = *NPSPACE***

Theorem 8.4 (Savitch)**.** *If $s : \mathbb{N} \to \mathbb{N}$ is a Space constructible function and $s(n) \geq \log n$, then $NSPACE[s(n)] \subseteq SPACE[s^2(n)]$.*

Proof. Let $L \in NSPACE[s(n)]$ be decided by a NDTM M. Consider the configuration graph $G_{M,x}$, for $x \in \{0,1\}^n$, that has $\leq 2^{O(s(n))}$ vertices.

We are now going to define a recursive procedure that return "yes" if we can go from a node u to a node v in $\leq 2^i$ steps, for some i, and "no" otherwise.

Consider $Reach(u, v, i) =$ "$u \rightsquigarrow v$ in $\leq 2^i$ steps".

$Reach(u, v, i + 1)$:

- for z in $V(G_{M,x})$:
  - $b = Reach(u, z, i)$
  - if $b = 1$ then continue (my notes say $b = 0$, but I think it's an error, it's correct $b = 1$)
  - $b = Reach(z, v, i)$
  - if $b = 1$ then return true
- return false

How much space does step i take?

- $R(0) = O(s(n))$
- $R(i + 1) = s(n) + R(i)$
- $R(s(n)) = O(s^2(n))$

□

8.6 Game on QBF

Let F be a boolean formula. In a game there are two players:

- $\forall$, who tries to make F false
- $\exists$, who tries to make F true

Following a description of how a game is played. We iterate on the quantified variables. If the quantifier for the variable is $\forall$ then we ask the $\forall$ players to fix an assignment for that variable; else if the quantifier is an $\exists$ then we ask the $\exists$ player to fix the assignment. If, at the end, F is true, then $\exists$ wins, otherwise $\forall$ wins.

F is true iff $\exists$ has a winning strategy.

8.7 LogSpace

Definition 8.5 (Logspace reduction)**.** *Let A and B be problems, and let $f : \{0,1\}^* \to \{0,1\}^*$ be a reduction from B to C s.t. $|f(x)| \leq |x|^c$.*

We write $B \leq_l C$ if f is logspace computable.

Some properties:

- $A \leq_l B$ and $B \leq_l C$, then $A \leq_l C$
- $A \leq_l B$ and $B \in L$, then $A \in L$

Theorem 8.5. *$PATH = \{\langle G, s, t\rangle \mid s \rightsquigarrow t$ in $G\}$ is **NL**-Hard w.r.t. $\leq_l$.*

Proof. Let $A \in NL$ be decided by machine M in space $O(\log n)$. Consider the graph of configuration $G_{M,x}$ where s is the starting state and t is the accepting

configuration (this is WLOG, reference). The graph is s.t. we can go from s to t iff M accepts x.

Note also that there are polynomially many vertices.

Given any two configurations, we can compute in space $O(\log |x|)$ if they are connected in the graph; in this way we can explore the graph to find out if $s \rightsquigarrow t$. □

Theorem 8.6. *NL = coNL*

9 Randomness in computation

Doing random computation is like having an additional tape such that, instead of being a guess, it contains random bits.

One of the simplest random choices that we can make is, for instance, the following:

$$x = \begin{cases} 0 & \text{w.p. } \frac{1}{2} \\ 1 & \text{w.p. } \frac{1}{2} \end{cases} \tag{5}$$

9.1 Prime example

We want to decide if a number N is prime. So we want the following decision procedure:

$$Prime(N) = \begin{cases} 1 & \text{if } N \text{ is prime} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$Prime(N)$:

- if $N = 2$, then return *prime*
- pick $A \in_R \{1, \ldots, N-1\}$ ($\in_R$ means "at random")
- if $\gcd(A, N) > 1$, then return *composite*
- if $\left(\frac{N}{A}\right) \neq A^{\frac{N-1}{2}}$, then return *composite*
- return *prime*

$\left(\frac{N}{A}\right)$ is the *Jacobi symbol*.

If N is odd and composite, then $Pr[\left(\frac{N}{A}\right) \neq A^{\frac{N-1}{2}} \mid \gcd(A, N) = 1] \geq \frac{1}{2}$.

If N is prime, then $Pr[\left(\frac{N}{A}\right) \neq A^{\frac{N-1}{2}}] = 0$

9.2 One sided error

9.2.1 RP

Definition 9.1 (RP)**.** *A language L is in **RP** if there is a polytime TM $M(\cdot, \cdot)$ and a polynomial q s.t.*

- *if $x \in L$, then $Pr_{r \in \{0,1\}^{q(|x|)}}[M(x, r) = 1] \geq \frac{1}{2}$*

- *if* $x \notin L$, *then* $Pr_{r \in \{0,1\}^{q(|x|)}}[M(x,r) = 0] = 1$

9.2.2 coRP

Definition 9.2 (coRP)**.** *A language* L *is in* ***coRP*** *if there is a polytime TM* $M(\cdot,\cdot)$ *and a polynomial* q *s.t.*

- *if* $x \in L$, *then* $Pr_{r \in \{0,1\}^{q(|x|)}}[M(x,r) = 1] = 1$
- *if* $x \notin L$, *then* $Pr_{r \in \{0,1\}^{q(|x|)}}[M(x,r) = 0] \geq \frac{1}{2}$

9.3 Two sided error

9.3.1 BPP

Definition 9.3 (BPP)**.** *A language* L *is in* ***BPP*** *if there is a polytime TM* $M(\cdot,\cdot)$ *and a polynomial* q *s.t.*

- *For all* x $Pr_{r \in \{0,1\}^{q(|x|)}}[M(x,r) = L(x)] \geq \frac{2}{3}$

$M(x,r) = L(x)$ means that M correctly recognizes if x is in the language or not.

What **BPP** captures is that if a TM accepts a string (i.e. a "yes" instance) then w.p. $\geq \frac{2}{3}$ we are sure that it is correct; instead if a TM rejects (i.e. a "no" instance) then w.p. $\leq \frac{1}{3}$ we are that that it is correct.

9.4 Probability theory

9.4.1 Chernoff bound

Let $X_1, X_2, \ldots, X_n$ be i.i.d. variables. Let $\mu = \mathbb{E}[\sum_i X_i] = \sum_i \mathbb{E}[X_i]$.

$$Pr[|\sum_i X_i - \mu| > c \cdot \mu] < 2 \cdot e^{-\min(\frac{c^2}{4}, \frac{c}{2}) \cdot \mu}$$

9.4.2 Markov Inequality

Let $X \geq 0$ be a random variable.

$$Pr[X \geq c] \leq \frac{\mathbb{E}[X]}{c}$$

9.5 Zero sided error

9.5.1 ZPP

We see two equivalent definitions of the class **ZPP**.

Definition 9.4 ($\mathbf{ZPP}_1$)**.** *Let* M *be a TM.*

- $x \in L$, *then* $Pr[M(x) = 1] \geq \frac{1}{2}$, *with* $M(x) \in \{yes, no\}$
- $x \notin L$, *then* $Pr[M(x) = 0] \geq \frac{1}{2}$, *with* $M(x) \in \{yes, ?\}$

Definition 9.5 ($\mathbf{ZPP}_2$)**.** *A language* $L \in$ ***ZPP***$_2$ *if there is a probabilistic TM* M *that outputs either "yes" or "no"; so* $x \in L$ *iff* $M(x) = yes$. *Let* $T(|x|)$ *be the running time of* $M(x)$, $\mathbb{E}[T(|x|)]$ *is polynomial in* $|x|$.

In $\mathbf{ZPP}_1$ we claim that a TM always runs in polytime, but makes errore and it can says "?" (i.e. "I don't know"). In $\mathbf{ZPP}_2$ we always get a correct answer and we have an expected running time polynomial in the size of the input, but worst case scenario the running time could be exponential.

Lemma 9.1. $\boldsymbol{ZPP_1 \to ZPP_2}$

Proof. Given a **ZPP** algorithm, run it multiple times until we get an answer. So, by iterating the algorithm, we can an answer w.p. $1n^c + \frac{1}{2}n^c + \frac{1}{4}n^c + \dots$.

$\mathbb{E}[T(|x|)] = \sum_{i=0}^{\infty} \frac{n^c}{2^i} = 2 \cdot n^c$ □

Lemma 9.2. $\boldsymbol{ZPP_2 \to ZPP_1}$

Proof. Let a **ZPP** algorithm run for $100 \cdot \mathbb{E}[T(|x|)]$ steps.

By Markov Inequality $Pr\left[T(|x|) \geq 100 \cdot \mathbb{E}[T(|x|)]\right] \leq \dfrac{\mathbb{E}[T(|x|)]}{100 \cdot \mathbb{E}[T(|x|)]} = \frac{1}{100}$ □

Theorem 9.3 ($\mathbf{ZPP}_1 = \mathbf{ZPP}_2$)**.** *Definitions* $\boldsymbol{ZPP_1}$ *and* $\boldsymbol{ZPP_2}$ *are equivalent.*

Proof. The proof is the combination of Lemma 9.1 and Lemma 9.2. □

Lemma 9.4. $\boldsymbol{ZPP \subseteq RP \cap coRP}$

Proof. **ZPP** $\subseteq$ **RP** and **ZPP** $\subseteq$ **coRP** basically by definition. □

Lemma 9.5. $\boldsymbol{RP \cap coRP \subseteq ZPP}$

Proof. Let M_r be a TM for **RP** and let M_c be a TM for **coRP**.

Run both machines on the same input:

- if the result is the same for both machines, then output it
- if the results differ, then output "?"

□

Theorem 9.6. $\boldsymbol{ZPP = RP \cap coRP}$

Proof. The proof is the combination of Lemma 9.4 and Lemma 9.5. □

9.6 Polynomial Identity Testing

In the **PIT** problem we are asked, given two polynomial p and q, to determine whether $p \equiv q$. Or, equivalently, given a polynomial p, we want to determine if $p \equiv 0$ (i.e. is the *zero* polynomial).

Lemma 9.7 (Schwartz-Zippel)**.** *Let p be a non-zero polynomial, on n variables, of degree d, defined over a field $\mathbb{F}$. Suppose $S \subseteq \mathbb{F}$.*

Pick $a_1, \ldots, a_n \in_R S$.

$$Pr[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}$$

We don't know how to solve **PIT** in polynomial time, if not by randomizing the algorithm; i.e. we don't know how to derandomize this problem in polytime.

10 Interactive proofs

10.1 Introduction

We have two players:

- **Verifier** (V)
- **Prover** (P)

Given a language L and a string x, **P** wants to convince **V** that $x \in L$.

V starts by sending a message q_1 to **P**, to which **P** replies with message a_1. To keep sending messages q_i, a_i for the duration of the whole protocol.

At the end **V** either accepts or not.

- $x \in L$, there must be a **P** that makes **V** accept
- $x \notin L$, **V** should never accept, regardless of **P**

The protocol is polynomial w.r.t. to $|x|$:

- $|x|^{O(1)}$ many messages
- $\forall i \ |q_i|, |a_i| \leq |x|^{O(1)}$ (i.e. polynomially long messages)

V is polynomially bounded; it take polytime w.r.t. $|x|$ to:

- send any message
- accept/reject

The messages sent by **V** are *randomized*. We can have both *public coin* protocols (i.e. **P** knows how **V** randomizes messages) or *private coin* protocols.

P is unbounded in time, it can be any function $f : \{0,1\}^* \to \{0,1\}^*$.

Interaction	Randomness	Class
X	X	**P**
V	X	**NP**
X	V	**BPP**
V	V	**PSPACE**

10.2 IP

Given $k : \mathbb{N} \to \mathbb{N}$, with $k(n) \geq 1$, we define $\mathbf{IP}[k]$ as the class of problems with a k-rounds interactive proof.

A language $L \in \mathbf{IP}[k]$ if there is a probabilistic polynomial time (ppt) $\mathbf{V}$ that can have a k-rounds interaction with a prover $\mathbf{P} : \{0,1\}^* \to \{0,1\}^*$.

- $x \in L$, $\exists \mathbf{P}\ Pr[V \text{ accepts}] \geq \frac{2}{3}$
- $x \notin L$, $\forall \mathbf{P}\ Pr[V \text{ accepts}] \leq \frac{1}{3}$

Definition 10.1 (**IP**)**.**

$$\boldsymbol{IP} = \bigcup_{c \geq 1} \boldsymbol{IP}[n^c]$$

Theorem 10.1. $\boldsymbol{IP} = \boldsymbol{PSPACE}$

To show that $\mathbf{PSPACE} \subseteq \mathbf{IP}$ it is sufficient to show that TQBF $\in \mathbf{IP}$ (see Quantified boolean formulas).

10.3 Arthur Merlin Protocols

These are particular types of **IP** protocols where the two parties are called, in literature, Arthur (**A**) and Merlin (**M**), respectively *Verifier* and *Prover*.

In this kind of protocol we always have *public coin* tosses; that is, **M** can see **A**'s coin flips. So **A** sends randomized messages, but deterministically verifies the interaction.

Because **M** can see **A**'s coins, then **M** can also predict **A**'s messages, so it makes sense for **A** to just send completely random messages, in order to see what **M** says.

This kind of protocol is interesting for small numbers of rounds. A few examples are:

- $\mathbf{AM}[2] = \mathbf{AM}$
- $\mathbf{AM}[2] = \mathbf{AMA}$ (**A** sends another message to probabilistically verify the interaction)
- **MA** ($\mathbf{NP} \subseteq \mathbf{MA}$)

11 Circuits

A circuit can be seen as a DAG that computes a function. Each circuit has a fixed number of **input gates** (i.e. fixed length input) and can have multiple **output gates**.

So, in general, we can see a circuit as a function $f : \{0,1\}^n \to \{0,1\}^k$. We are intereste in circuits s.t. $k = 1$; circuit solving *decision* problems.

How do we decide a language L? In general, $L \subseteq \{0,1\}^*$, so it can contain words of different length, but a circuit C can only decide $L \cap \{0,1\}^n$ (i.e. only the words in L of length n). So we pick a family of circuits $\{C_0, \dots, C_n, \dots\}$ s.t. for each i, $C_i : \{0,1\}^i \to \{0,1\}$. Such family decides L iff $\forall n \forall x \in \{0,1\}^n$ $C_n(x) = L(x)$.

Definition 11.1 (Circuit size)**.** *The **size** of a circuit is the number of gates of the circuit.*

Definition 11.2.

$$size(s(n)) = \{L \mid \textit{there exists a family } \{C_0, \dots, C_n, \dots\} \textit{ that decides } L, \textit{ and } |C_n| \leq s(n)\}$$

Definition 11.3 (Class **P/poly**)**.**

$$P/poly = \bigcup_{k \geq 1} size(n^k)$$

P/poly is the class of languages decided by a polysize family of circuits. Observe that $\mathbf{P} \subseteq \mathbf{P/poly}$ and $\mathbf{NP} \subseteq \mathbf{P/poly}$.

In a Turing Machine inputs of different length are usually solved in the same way; we have an algorithm that extends the same idea to all input lengths. Circuits however are different. Because we need a dedicated circuit for each input length, if we put no constraints in the construction of the circuit family, then each circuit can work entirely differently compared to (all) other circuits in the family. Because of this we say that the circuit model is **non-uniform**.

We can however make a uniform family in the following way.

Definition 11.4. *A circuit family $\{C_0, \dots, C_n, \dots\}$ is **P-uniform** if each C_i in the family is built by a TM in polynomial time w.r.t. i.*

Theorem 11.1. *If a language L is decidable in **P/poly** via a P-uniform circuit family, then $L \in \mathbf{P}$.*

Theorem 11.2. ***BPP** $\subseteq$ **P/poly***

Proof. If $L \in \mathbf{BPP}$, then there exists a probabilistic TM M s.t. $Pr_r[M(x,r) \neq L(x)] \leq \frac{1}{2^{n+1}}$.

We can build a matrix where rows represent the possible choices of $x \in \{0,1\}^n$ and the columns represent the choices of $r \in \{0,1\}^m$. Each column might contain an error w.p. $\frac{1}{2^{n+1}}$. If we take m large enough (i.e. $m > n$) then there are enough random strings s.t. a column cointains no errors.

So $\exists \vec{r} \forall x \in \{0,1\}^n$ $M(\vec{r}, x) = L(x)$. □

12 Polynomial Hierarchy

12.1 Hierarchy

The *polynomial hierarchy* is a generalization of **P**, **NP** and **coNP**.

Definition 12.1 (Σ_2^p)**.** *We define the following class for problems with formulas starting with $\exists\forall$ alternation of existentials.*

$\Sigma_2^p = \{L \mid \exists M, q, x \in L \exists y \in \{0,1\}^{q(|x|)} \forall z \in \{0,1\}^{q(|x|)} M(x,y,z) = 1\}$

A couple of equivalent definitions of Σ_2^p:

- $\boldsymbol{NP^{NP}}$
- $\{L \mid L \leq_p \Sigma_2 - SAT\}$, where $\Sigma_2 - SAT$ is a complete problem for Σ_2^p that will be defined later on

Observation 12.0.1. *Σ_2^p is a superset of the following two classes:*

- $\boldsymbol{NP} \subseteq \Sigma_2^p$
- $\boldsymbol{coNP} \subseteq \Sigma_2^p$

Definition 12.2 (Π_2^p)**.** *We define the following class for problems with formulas starting with $\forall\exists$ alternation of existentials.*

$\Pi_2^p = \{L \mid \exists M, q, x \in L \forall y \in \{0,1\}^{q(|x|)} \exists z \in \{0,1\}^{q(|x|)} M(x,y,z) = 1\}$

Observation 12.0.2. $L \in \Sigma_2^p \leftrightarrow \bar{L} \in \Pi_2^p$

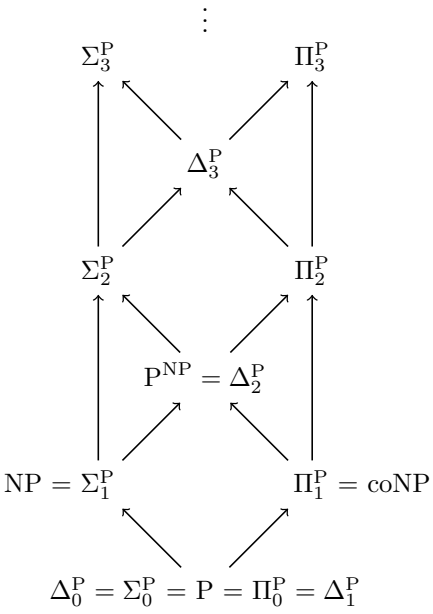
We can actually generalize these classes in order to account for any number of alternation of quantifiers:

- Σ_i^p: $\exists \vec{x_1} \forall \vec{x_2} \exists \vec{x_3} \ldots Q\vec{x_i} M(x, x_1, x_2, x_3, \ldots, x_i) = 1$
- Π_i^p: $\forall \vec{x_1} \exists \vec{x_2} \forall \vec{x_3} \ldots Q\vec{x_i} M(x, x_1, x_2, x_3, \ldots, x_i) = 1$

Obviously the last quantifier (i.e. $Q\vec{x_i}$) can be either $\exists$ or $\forall$ depending on the parity of i.

So there comes out the following *polynomial hierarchy* (for better readability, an arrows means $\subseteq$)[1]:

[1] source code of the graph here



It is the number of alternation of quantifiers that is important, not the number of quantifiers itself.

We know that this hierarchy exists, but we don't know much else; it could even be that everything is equal to **P**.

Definition 12.3 (**PH**)**.**

$$\boldsymbol{PH} = \bigcup_i \Sigma_i^p = \bigcup_i \Pi_i^p$$

It is unknown if the hierarchy stops at some point, that is, if $\Sigma_i^p = \Pi_i^p$ for some i. It that were the case, two things would happen:

- $\forall j \geq i \ \Sigma_j^p = \Pi_j^p$
- $\mathbf{PH} = \Sigma_i^p = \Pi_i^p$

We say that the hierarchy collapses at the i-th level.

12.2 Complete problems

Let $\Sigma_i - SAT$ be the decision problem asking whether a formula of the form $\exists \vec{x_1} \forall \vec{x_2}, \ldots Q \vec{x_i} F(x_1, x_2, \ldots, x_i)$ is satisfiable. Such problem is complete for Σ_i^p.

For Π_i^p we have equivalent complete problems, except the alternation of quantifiers is swapped.

Is there a complete problem for **PH**? Suppose there exists a problem L complete for **PH**. Surely L is in the hierarchy; i.e. $L \in \bigcup_i \Sigma_i^p$. So there is some i^* s.t. $L \in \Sigma_{i^*}^p$. Because L is complete for **PH**, then $\mathbf{PH} \leq_p \Sigma_{i^*}^p$, which implies that $\mathbf{PH} = \Sigma_{i^*}^p$. The hierarchy would collap to the i^*-th level.

Theorem 12.1 (Karp-Lipton)**.** $\boldsymbol{NP \subseteq P/poly \rightarrow PH = \Sigma_2^p}$

Theorem 12.2. $\boldsymbol{BPP \subseteq \Pi_2^p \cap \Sigma_2^p}$

13 Decision Trees

A decision tree (**DT**) computes a function $f : \{0,1\}^* \rightarrow \{0,1\}$.

$size^{\mathbf{DT}}(f) =$ min # of nodes of a tree that computes f

What is really important is the number of queries to the data required to get the result.

Definition 13.1 (Tree cost)**.** *Let t be a tree and x an input for t. The cost of t on x is:*

$$cost(t, x) = \#\ bits\ of\ x\ examined\ by\ t$$

Definition 13.2 (Decision tree complexity)**.** *Let f by a function.*

$$D(f) = \min_t(\max_x(cost(t, x)))$$

Where:

- min *is over all trees t that compute function f*
- max *is over all $x \in \{0,1\}^n$*

For any function f, $D(f) \leq n$; i.e. at most we need to read all the data.

Definition 13.3 (Certificate complexity)**.** *Let f be a function.*

- $C_0(f) = \#$ *of bits required to certificate* 0
- $C_1(f) = \#$ *of bits required to certificate* 1

So the certificate complexity is:

$$C(f) = \max(C_0(f), C_1(f))$$

We know that the above complexity measures are polynomially related in the following way:

$$C(f) \leq D(f) \leq C_0(f)C_1(f) \leq C^2(f)$$

13.1 Randomized DTs

In a randomized **DT** a (**RDT**) node ha finitely many descendants and chooses one at random.

Let $\mathcal{D}$ be a probability distribution over trees that compute a function f.

$$cost(\mathcal{D}, x) = \mathbb{E}_{t \in \mathcal{D}} cost(t, x)$$

Definition 13.4 (Cost of an **RDT**)**.** *Let f be a function.*

$$R(f) = \min_{\mathcal{D}}(\max_x(cost(\mathcal{D}, x)))$$

It is known that:

$$C(f) \leq R(f) \leq D(f) \leq C^2(f)$$

13.2 Lower bounds for DTs

Given a function $f : \{0,1\}^n \to \{0,1\}$, we consider the following measures to lowerbound the complexity of any **DT** that computes f.

- Sensitivity
- Block sensitivity
- Degree

13.2.1 Sensitivity

Given $X = x_1, \ldots, x_n$, we define $X^i = x_1, \ldots, x_{i-1}, \bar{x_i}, x_{i+1}, \ldots, x_n$.

Let $S \subseteq [n]$, then

$$X_i^S = \begin{cases} x_i & \text{if } i \in S \\ \bar{x_i} & \text{if } i \notin S \end{cases} \tag{7}$$

Given f and X as defined above, the **sensitivity** of f over X is the number of bits i s.t. $f(X) \neq f(X^i)$.

Definition 13.5 (Sensitivity)**.** *The sensitivity of a function f is*

$$S(f) = \max_x(\textit{sensitivity of } f \textit{ over } x)$$

$$S(f) \leq D(f)$$

13.2.2 Block sensitivity

It is a generalization of *sensitivity*

Given a function f and an input x, the **block sensitivity** of f over x. denoted as $bs_x(f)$ is the maxmimum number of disjoint blocks $B_1, \ldots, B_s \subseteq [n]$ s.t. $\forall i$ $f(x) \neq f(x^{B_i})$.

Definition 13.6 (Block sensitivity)**.** *Let f be a function, the block sensitivity of f is:*

$$bs(f) = \max_x(bs_x(f))$$

13.2.3 Degree

Definition 13.7 (Degree)**.** *Let f be a function. We can map f into a polynomial p_f.*

$$degree(f) = degree(p_f)$$

Each function $f : \{0,1\}^n \to \mathbb{F}$ can be uniquelt represented as a multilinear polynomial.

14 Communication complexity

We have two parties, Alice and Bob, that want to compute a function collectively. Alice has input $x \in \{0,1\}^n$ and Bob has input $y \in \{0,1\}^n$, and together they want to compute a function $f(x,y)$.

How many bits do they have to exchange in order to do so?

We assume that both Alice and Bob have unbounded computational power, in fact we are only interested to study the number of bits shared.

Note that this is not an adversarial scheme. We assume that the channel is secure and Alice and Bob are both trusted. There is no reason, for instance, to design private schemes as in other frameworks.

What we want is for Alice and Bob to have a deterministic protocol to compute $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. Observe that we suppose that both inputs have the same length.

So in the protocol they start exchanging messages until the function is computed. At each step i, we can think that there is a function $P_i : \{0,1\}^* \to \{0,1\}^*$ that lets us compute the next message.

So the communication, messages-wise, goes as follows:

- $m_1 = P_1(x)$
- $m_2 = P_2(y, m_1)$
- $m_3 = P_3(x, m_1, m_2)$
- $\vdots$
- $m_t = f(x,y)$

We can say that for $t' > t$ $m_{t'} =$ ""; i.e. they stop talking.

Definition 14.1 (Cost of a protocol)**.** *Let* Π *be a protocol. The cost of* Π *is:*

$$C(\Pi) = \max_{x,y} \left(\sum_i |m_i| \right)$$

Definition 14.2 (Communication complexity)**.** *The communication complexity of a function* f *is:*

$$C(f) = \min_{\Pi}(C(\Pi))$$

14.1 Lower bound methods

We now see three methods used to give lower bounds the communication complexity of functions:

- Fooling set
- Tiling
- Rank

14.1.1 Fooling set

Definition 14.3 (Combinatorial rectangle)**.** *Let M be a matrix with $\{0,1\}^n$ columns and $\{0,1\}^n$ rows.*

$N \subseteq \{0,1\}^n \times \{0,1\}^n$ is a combinatorial rectangle iff $\forall (x_1,y_1),(x_2,y_2) \in N$, $(x_1,y_2) \in N$ and $(x_2,y_1) \in N$

A combinatorial rectangle is a generalization of a geometric rectangle. Given a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ we represent it as a matrix M (the codomain of the function is actually not important, but let us stick with $\{0,1\}$ which is usually what we care about). Let us pick a geometric rectangle. We can permute both the columns and the rows of M, and the resulting matrix would be equivalent to the original one, meaning that it still represents the same function, except that the results are represent in a different order. Once we have applied the permutation, the original geometric rectangle that we had picked got split up into smaller rectangles. These smaller rectnagles are the combinatorial rectangle.

Definition 14.4 (Fooling set)**.** *A set of pairs $(x_1,y_1),\ldots,(x_t,y_t)$ is a fooling set for a function f if:*

- $\forall i \ f(x_i,y_i) = b$
- $\forall i \neq j \ f(x_i,y_j) \neq b$ *or* $f(x_j,y_i) \neq b$

Theorem 14.1. *If a function f has a fooling set of t pairs, then $C(f) \geq \log_2(t)$.*

14.1.2 Tiling

Definition 14.5 (Tailing)**.** *Let M be the matrix of a function f. A **tailing** of M is a partition of M in monochromatic rectangles.*

$$\chi(f) = \min(\#\ \textit{rectangles in a tailing of } M)$$

Theorem 14.2. *Let f be a function.*

$$\log(\chi(f)) \leq C(f) \leq \log^2(\chi(f))$$

14.1.3 Rank

Definition 14.6 (Rank)**.** *Let $M_f = \sum_i B_i$ for some function f, where, for each i, B_i is a matrix of rank* 1*.*

$$rank(M_f) = \min_t : M_f = \sum_{i=1}^{t} B_i$$

14.2 Randomized communication complexity

How do we randomize communication?

- each party flips random coins (private coin model)
- each party flips random coins in public (public coin model)

- distributions over (x, y) (i.e. the inputs)

Let Π be a random protocol. Its signature is $\Pi(R, R_A, R_B, x, y)$, where R represents public coins and R_A and R_B represent the private coins of Alice and Bob respectively. Note that we could have just used public coins, this signature is defined as is just to make it as general as possible.

The cost of Π is

$$\max_{R,R_A,R_B,x,y}(\text{cost of running } \Pi(R, R_A, R_B, x, y))$$

Error in the *worst case* is:

$$\max_{x,y} Pr_{R,R_A,R_B}[\Pi(R, R_A, R_B, x, y) \neq f(x, y)]$$

Suppose x and y come from a distribution $\mathcal{D}$. Error in the *average case* is:

$$Pr_{R,R_A,R_B,x,y}[\Pi(R, R_A, R_B, x, y) \neq f(x, y)]$$

15 Proof complexity

The aim of this field is to study the length of proofs.

e.g. We want to prove that a CNF φ is SAT. The length of the proof is the lengh of an assignment that satisfies φ.

Definition 15.1 (Proof system for UNSAT)**.** *A proof system is a polytime machine $P(\cdot, \cdot)$ with input:*

- *CNF φ*
- *candidate proof $\pi \in \{0, 1\}^*$ that φ is UNSAT*

Let φ be a formula:

- if φ is UNSAT, then $\exists \pi \ P(\varphi, \pi) = 1$
- if φ is not UNSAT, then $\forall \pi \ P(\varphi, \pi) = 0$

15.1 Decision Trees

Decision trees can be used as proof systems. Given a formula $F = \bigwedge_{i=1}^{m} C_i$, whose variables are $x_1, \ldots, x_n$, we can build a DT, querying the variables, and stopping a branch as soon as it falsies any of the clauses. If no branch falsifies F, then the DT is a *refutation*; i.e. a proof that F is UNSAT.

A DT as defined above always exists for any formula.

Any kind of lower bound of DTs also apply here.

15.2 Resolution

Resolution is a deductive proof system.

Let $F = \bigwedge_{i=1}^{m} C_i$.

$$\text{(axioms)} \ \frac{}{C_i} \tag{8}$$

$$\text{(weakening)} \ \frac{A}{A \vee B} \tag{9}$$

$$\text{(resolution)} \ \frac{A \vee x \quad \bar{x} \vee B}{A \vee B} \tag{10}$$

Some proof are **Tree**-like, and some proofs are **DAG**-like. Tree-like refutations are going to be called *TL*-Resolution refutations, while DAG like refutation just refutation in Resolution. That is, we are only going to specify the type of refutation when it's Tree-like.

15.3 Polynomial simulation

Definition 15.2 (p-simulation)**.** *A proof system A **p-simulates** a proof system B if every tautology has proofs in A of size at most polynomially larger than in B.*

That is, if A p-simulates B, then A is at least as strong as B, because it proves tautologies at least as efficiently as B.

Definition 15.3 (p-equivalence)**.** *Two proof systems A and B are p-equivalent is they p-simulate each other; i.e. A p-simulates B and B p-simulates A.*

15.4 Ordering principle

The *ordering principle* states that a non-empty set of n integers contains a least element. We will denote a formula stating this principle as OP_n.

The variables of a formula OP_n are of the form:

$$x_{ij} = \begin{cases} 1 & \text{if } "i < j", \forall i, j \in [n] \text{ s.t. } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

The clauses of the formula are going to contain the following disjunctions:

- $\bar{x_{ij}} \vee \bar{x_{ji}}$ (no 2-cycles)
- $\bar{x_{ij}} \vee \bar{x_{jk}} \vee x_{ik} \equiv x_{ij} \wedge x_{jk} \rightarrow x_{ik}$, $\forall i, j, k \in [n]$
- $\bigvee_{j \neq i} x_{ji}$, $\forall i \in [n]$

Such formulas has:

- refutations in resolution of length n^2
- TL-refutations in resolution of length 2^n

15.5 Pigeonhole principle

The *pigeonhole principle* states, informally, that if we have n pigeons that fly into $n-1$ holes, then at least one hole has to contain more than one pigeon. We refer to this problem as PHP^n_{n-1}.

The problem can of course be defined in a more general state, saying that there are m pigeons and n holes, with $m > n$. Sometimes it is also stated as the opposite: if we have more holes than pigeons, then some holes will remain empty.

The "pigeon variables" are defined as follows:

$$p_{ij} = \begin{cases} 1 & \text{if "pigeon } i \text{ flies into hole } j\text{"} \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

For $i \in [n]$ and $j \in [n-1]$.

The clauses of a formula stating the principle contain the following disjunctions:

- $\bigvee_j p_{ij}$, $\forall i \in [n]$ (pigeon i flies in some hole)
- $\bar{p_{i_1 j}} \vee \bar{p_{i_2 j}}$, $\forall i_1 \neq i_2$ (no collision between pigeon i_1 and i_2 in hole j)

Such formula has:

- refutations in resolution of length $2^{O(n)}$; actually also $2^{\Omega(n)}$
- TL-refutations in resolution of length $2^{\Omega(n \log n)} \approx n!$