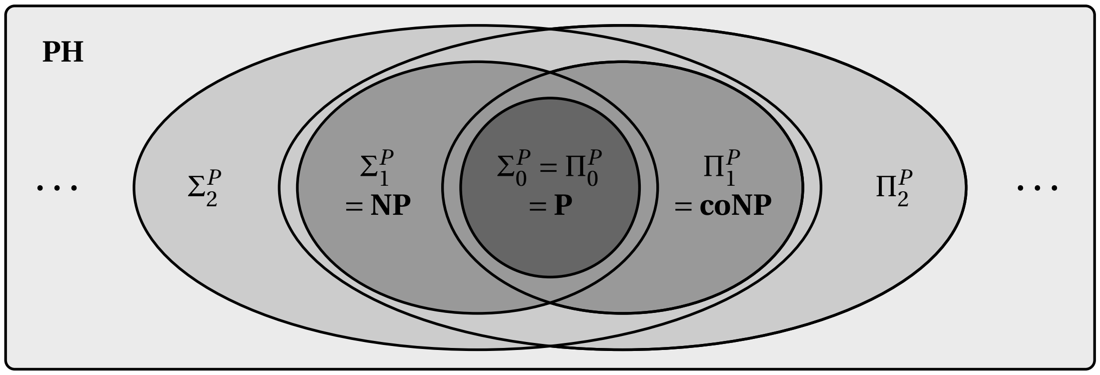
Computational Complexity I

CSE 531 — Winter 2024

Thomas Rothvoss


PH
. . .
Σ_2^P
Σ_1^P $= \mathbf{NP}$
$\Sigma_0^P = \Pi_0^P$ $= \mathbf{P}$
Π_1^P $= \mathbf{coNP}$
Π_2^P
. . .

UNIVERSITY *of* WASHINGTON

Last changes: February 28, 2024

Contents

Chapter 1

Turing machines

In this course we consider mathematical aspects of computation. The driving questions will be what functions can be computed at all and which ones can be computed efficiently. First, we need a mathematical model of what a computer actually is and what it can do. We use the following (somewhat lengthy) definition:

Definition 1.1. For $k \in \mathbb{N}$, a *k-tape Turing machine* is described by a tuple $M = (\Gamma, Q, \delta)$ containing:

- A finite set of *symbols* Γ (called the *alphabet*) where we assume that "$\Box$" (blank) and ">" (start) are in Γ.
- A finite set of *states* Q with designated states $q_{\text{start}}, q_{\text{halt}} \in Q$
- A *transition function* $\delta : Q \times \Gamma^{k+1} \to Q \times \Gamma^{k+1} \times \{L, S, R\}^{k+2}$

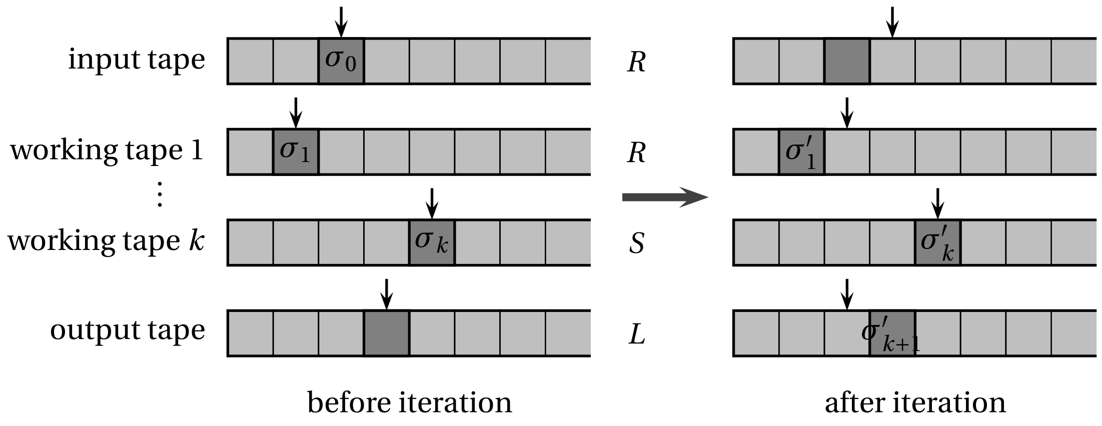
The Turing machine has

- a read only input tape that is initialized with $>, x_1, \ldots, x_n$
- k read+write working tapes initialized with $>, \Box, \Box, \ldots$
- a write-only output tape initialized with blanks.

All the tapes are infinite in one direction. Each tape has an individual *head position* that can move. The Turing machines starts in state q_{start}. In one iteration, the Turing machine does the following: Say the Turing machine is in state $q \in Q$ and the symbols at the current positions (indicated by the heads) on the input tape and k working tapes are $\sigma_0, \ldots, \sigma_k \in \Gamma$. Moreover suppose the corresponding entry in the transition function is $\delta(q, \sigma_0, \ldots \sigma_k) = (q', \sigma'_1, \ldots, \sigma'_{k+1}, z)$

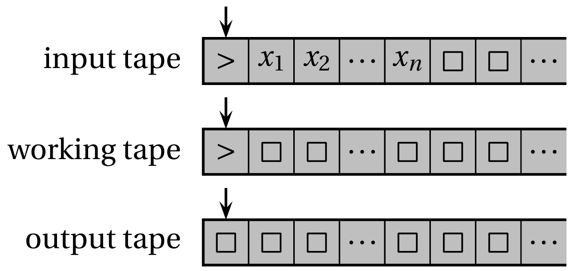
with $z \in \{L, S, R\}^{k+2}$. Then for $i \in \{1, \ldots, k+1\}$ the machine writes σ'_i on the current position of the ith tape (where we denote the output tape by index $k+1$). Moreover for $i \in \{0, \ldots, k+1\}$,

- if $z_i = L$, then the ith head moves one position to the left
- if $z_i = S$, then the ith head stays put
- if $z_i = R$, then the ith head moves one position to the right



Then the Turing machine moves into state q' and the iteration is concluded. Then we iterate until at some point the Turing machine reaches state q_{halt}, then it has halted (i.e. terminated). Note that it is also possible that the Turing machine never reaches q_{halt}.

Maybe more intuitively, one may think of a Turing machine as a computer that has a finite number of states (say finite RAM), an infinite amount of storage (one may think of hard drive) and it is is given some input $x_1, \ldots, x_n$. Then the Turing machine has a hard coded program that processes the input. Here the computation in each step is very *local* as only $O(1)$ symbols are being processed (and k, $|\Gamma|$ and $|Q|$ are constants that do not depend on the input).


input tape
working tape
output tape
$>$ x_1 x_2 $\cdots$ x_n $\square$ $\square$ $\cdots$
$>$ $\square$ $\square$ $\cdots$ $\square$ $\square$ $\square$ $\cdots$
$\square$ $\square$ $\square$ $\cdots$ $\square$ $\square$ $\square$ $\cdots$

Initial state of a Turing machine with $k = 1$ working tapes

Recall that $\{0,1\}^* = \bigcup_{k\geq 0}\{0,1\}^k$ denotes the set of binary strings of any length. We write $|x|$ as the *length* of a string.

Definition 1.2. We say that a Turing machine M *computes* a function $f : \{0,1\}^* \to \{0,1\}^*$, if on every input $x \in \{0,1\}^*$, M halts with the string $f(x)$ written on the output tape. We additionally say that the Turing machine M *computes f in time* $T(n)$ if for every input $x \in \{0,1\}^*$, M halts after at most $T(|x|)$ iterations.

We also use the notation $M(x)$ for the content of the output tape when M halts on input x. It might seem that the Turing machine model is rather restrictive, but as an exercise the reader is encouraged to verify that there are Turing machines that can

- add / multiply two numbers given in binary encoding
- sort n strings lexicographically
- compute the prime factorization of a number given in binary encoding length

Describing details of a Turing machine program is tedious and usually not very enlightening. In the arguments that we give, we will often just provide a sketch and omit many details (that could be filled in if needed).

In many cases we are interested in computing functions $f : \{0,1\}^* \to \{0,1\}$, meaning that for each input x the Turing machine only needs to make a decision between *accepting* or *rejecting* the input. In those cases, we agree that the Turing machine M does not have an output tape and instead of q_{halt} it has the two distinguished halting states q_{accept} and q_{reject}. We define the function computed by M as

$$M(x) = \begin{cases} 1 & \text{if on input } x, M \text{ halts in state } q_{\text{accept}} \\ 0 & \text{if on input } x, M \text{ halts in state } q_{\text{reject}} \end{cases}$$

assuming the Turing machine halts on every input.

1.1 Robustness of the model

In the literature one may find many different variations of how the concept of Turing machines is defined. We want to explain that the exact details did not matter for its expressive power.

Definition 1.3. A function $T : \mathbb{N} \to \mathbb{N}$ is called *time-constructable* if $T(n) \geq n$ and there is a Turing machine M that on input x, computes the binary encoding of $T(|x|)$ in time $T(|x|)$.

Here the condition $T(n) \geq n$ avoids some oddities; note that any Turing machine that at least reads the input fully must have running time $T(n) \geq n$. All typical running times that one may encounter, like $\lceil n \log(n) \rceil$, $\lceil n^c \rceil$ for some constant $c \in \mathbb{Q}_{\geq 1}$, 2^n, 2^{2^n} are time constructable.

First, we want to argue that using a larger alphabet cannot increase the set of computable functions:

Theorem 1.4. *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function computed by a k-tape Turing machine M in time-constructable time $T(n)$ using alphabet Γ. Then there is a k-tape Turing machine $\tilde{M}$ with alphabet $\{0,1,\square,>\}$ computing f in time $O(T(n) \cdot \log|\Gamma|)$.*

Sketch. We replace each cell on a working tape of M by $\log|\Gamma|$ bits in $\tilde{M}$. Then each original operation of M takes time $O(\log|\Gamma|)$ in $\tilde{M}$. $\square$

For example the book of Sipser uses a different definition of a Turing machine where there is a single tape that serves as input, working and output tape. We can verify that this model is not different in terms of its expressive power.

Theorem 1.5. *Suppose that $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time-constructable $T(n)$ time by a k-tape Turing machine M. Then there is a Turing machine $\tilde{M}$ with a single combined input/working/output tape and that computes f in time $O(T(n)^2)$.*

Proof. Suppose M uses alphabet Γ. Then $\tilde{M}$ uses the alphabet $\tilde{\Gamma} := \{0,1\} \cup (\Gamma^{k+2} \times \{\texttt{HEAD}, \texttt{NO HEAD}\}^{k+2})$. Using this larger alphabet, each cell of $\tilde{M}$'s tape encodes the content of all the $k+2$ cells at the same position in M's tapes. Moreover we also encode the position of M's heads using the larger alphabet. The only technical problem in now simulating M's computation lies in the fact that the $k+2$ heads of M may be at different positions, while $\tilde{M}$ only has a single head. But each iteration of M can be simulated by scanning from position 1 to $T(|x|)$ and back. Overall this results in a running time of $O(T(|x|)^2)$. $\square$

1.2 The Universal Turing machine

Each given string $x \in \{0,1\}^*$ can be interpreted as a description of a Turing machine that we denote by M_x (i.e. a binary encoding of the transition function δ). It does not matter how exactly the transition function is being encoded. In reverse, if M is a Turing machine, then we write $[M] \in \{0,1\}^*$ as the binary encoding of it. Occasionally it will be useful to denote M_i as the Turing machine corresponding to the binary encoding of the number $i \in \mathbb{N}$. This way we will be able to talk about a list $M_1, M_2, \ldots$ of all possible Turing machines. But we get back to that later.

First, we want to discuss what appears to be a major drawback of our Turing machines: their functionality is hard coded and each of them can only solve a single problem. This is very different from our understanding of a modern computer which clearly can run arbitrary programs. But astonishingly, there is not actually a distinction.

Definition 1.6. A *universal Turing machine* is a Turing machine $\mathcal{U}$ that receives (x, α) as input with $x, \alpha \in \{0,1\}^*$ and computes $\mathcal{U}(x, \alpha) = M_\alpha(x)$. Moreover the Turing machines halts on (x, α) whenever M_α halt on x.

We will prove that there exists a universal Turing machine. In fact, the universal Turing machine that we construct will have an additional property:

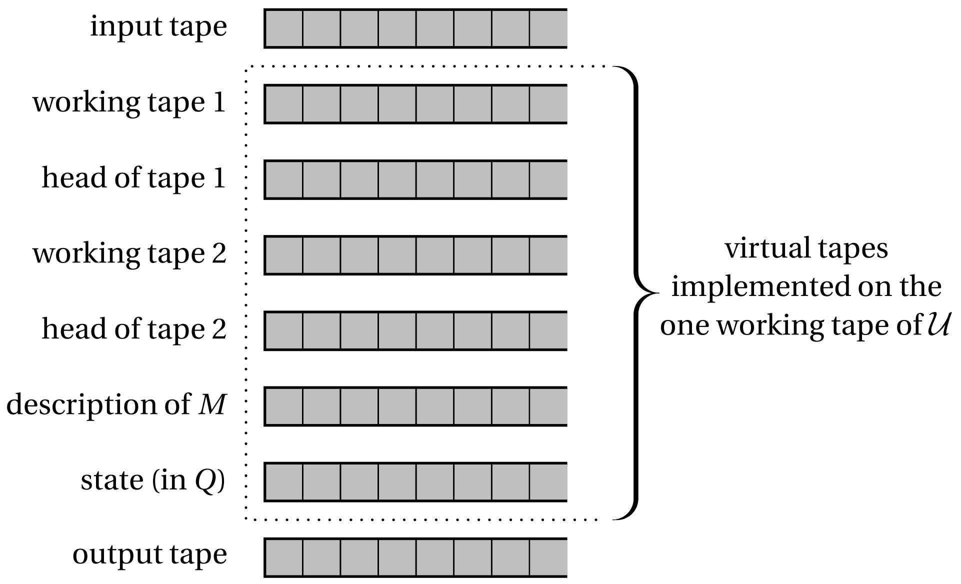
Definition 1.7. A Turing machine M is called *oblivious* if for each tape the position of the head only depends on (a) the length of the input $|x|$ and (b) the time step t.

It will often be easier to reason about a Turing machine that is oblivious.

Theorem 1.8. *There is an oblivious 2-tape universal Turing machine $\mathcal{U}$. In particular for each Turing machine M and all $T : \mathbb{N} \to \mathbb{N}$ with $T(n) \geq n$ so that M halts on input x after $T(|x|)$ steps one has: (i) $\mathcal{U}(x, [M]) = M(x)$ and (ii) on input of $(x, [M])$, $\mathcal{U}$ halts after at most $CT(|x|)\log_2 T(|x|)$ steps where $C := C_M > 0$.*

Proof. First, we explain the construction of a simple $O_M(T(|x|)^2)$-time universal Turing machine that uses one working tape. Suppose $(x, [M])$ is the input for $\mathcal{U}$ and $M = (\Gamma, Q, \delta)$ has k working tapes. Then $\mathcal{U}$ creates "virtual tapes" (on its single working tape) which are

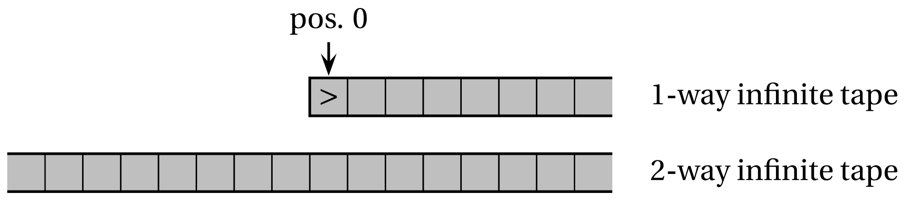
- the k working tapes with alphabet Γ
- a tape with the head position of each of the k working tapes
- a tape with the description of M
- a tape recording the current state of Q



simulation of a TM M with $k = 2$ tapes

Since $\mathcal{U}$'s alphabet size is fixed, the encoding is done in binary and the cells of the virtual tapes are listed alternatingly on the "physical" tape. Then a single computation of M can be simulated by $O_M(1)$ many left to right sweeps where the memory needed in every moment is not dependent on M. Each sweep may cover a length of $T(|x|)$ cells which results a $O_M(T(|x|)^2)$ time. We do note that (if properly implemented) this Turing machine is oblivious.

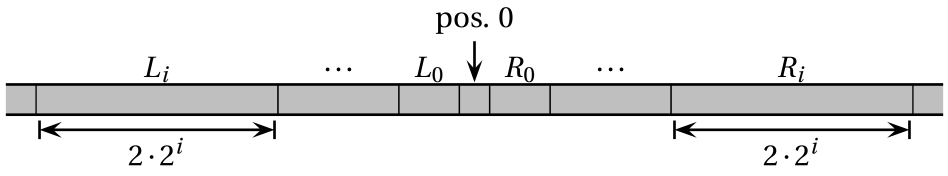
Before we discuss how to improve the running time of the simulation, we want to give an alternative way to obtain the same $O_M(T(|x|)^2)$ running time. Instead of 1-way infinite tapes we use *2-way infinite* tapes.



In fact, a Turing machine with 2-way infinite tapes can be simulated by a Turing machine with the same number of 1-way infinite tapes with a constant overhead. We use the same representation with virtual tapes. But now, if for working tape i in M, the head is supposed to move to the right (or left), then instead we shift the whole virtual tape one position to the left (or right, resp). This way, we can keep the heads always at position 0. On the other hand, shifting a tape by one position can take time $O_M(T(|x|))$ so we end up at with the same $O_M(T(|x|)^2)$ running time.

Now we describe how to improve the simulation to obtain the promised $O_M(T(|x|)\cdot\log(T(|x|)))$ running time. We add a special symbol ⊠ to our alphabet which we use as spaceholder on the virtual tapes. More precisely we allow ⊠ to be inserted at arbitrary positions without changing the semantics. For example that means the tape content 0⊠⊠10⊠1 is equivalent to 0101. But on the other hand, when shifting a virtual tape we do not have to shift all of the $T(|x|)$ positions as we can make use of the spaceholders and get away with shifting only part of the tape.

We partition each virtual tape into *zones*. Starting at position 0 we have zones $R_0, R_1, R_2, \ldots$ when going to the right and zones $L_0, L_1, L_2, \ldots$ when going to the left. Here the zones contain $|L_i| = |R_i| = 2\cdot 2^i$ many cells.



We initialize the virtual tapes so that in each zone L_i and R_i half the cells contain the placeholder ⊠. We say that the zones are *half-full*. In contrast, we call a zone *full* if all cells contain data (i.e. no ⊠) and *empty* if it contains no data (i.e. all cells are filled with ⊠). Position 0 is not part of a zone and always contains a data symbol (i.e. not ⊠). During the simulation we will maintain two invariants:

(i) Each zone L_i and R_i is either empty, half-full (i.e. exactly 2^i cells contain ⊠) or full.

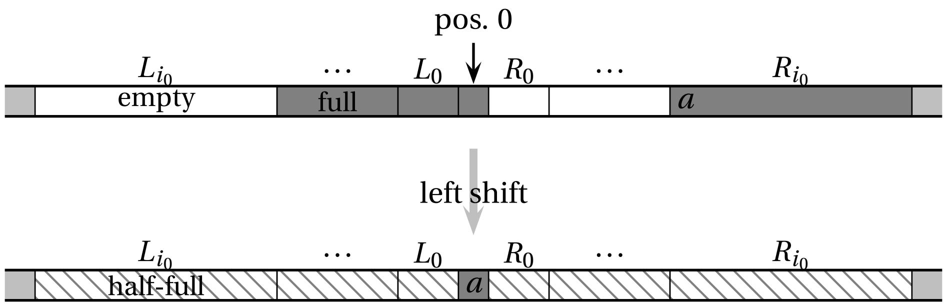
(ii) In $L_i \cup R_i$, exactly half the cells contain data, the other half contains ⊠.

Now we describe one iteration of the simulation: say that we need to simulate that the head moves to the right, so all the data needs to be shifted one position to the left. Find the minimal index i_0 so that R_{i_0} is not empty and let $a \in \Gamma$ be the first non-⊠ symbol in R_{i_0}. In particular $R_0, \ldots, R_{i_0-1}$ are empty and by invariant $(i)+(ii)$ we know that $L_0, \ldots, L_{i_0-1}$ are full. We redistribute the cells in $L_{i_0} \cup \ldots \cup L_0 \cup \text{pos } 0 \cup R_0 \cup \ldots \cup R_{i_0}$ so that the symbol a moves to position 0 and all the inner zones $L_{i_0-1}, \ldots, L_0, R_0, \ldots, R_{i_0-1}$ become half full. There are two possibilities as to how L_{i_0} and R_{i_0} are treated:

(A) If R_{i_0} was full, then L_{i_0} was empty and after the shift both are half-full.

(B) If R_{i_0} was half-full, then also L_{i_0} was half full and after the shift R_{i_0} is empty and L_{i_0} is full.

A visualization of case (A) is as follows:



We leave it to the reader to verify that because $|L_i| = |R_i| = 2 \cdot 2^i$, this is indeed possible. We also need to analyze the time that it takes to perform this single shifting operation. Naively, one could shift the $O(2^{i_0})$ cells in question one by one in time $O_M(2^{i_0})^2$ — but then we would not improve over the quadratic time we had already proven earlier. Instead we make use of the second physical tape that the Turing machine $\mathcal{U}$ possesses to implement the shift in time $O(2^{i_0})$.

One can prove that only every 2^{i_0} many left shifts we may have a shift that goes until index i_0. That means amortized (i.e. averaged) over all iterations we pay $O_M(1)$ time per zone and per time unit. As there are at most $O(\log T(|x|))$ many zones, the simulation can be implemented in total time $O_M(T(|x|) \cdot \log(T(|x|)))$ as claimed. $\square$

1.3 The class P

So far we said that Turing machines are computing *functions*. But if M computes a binary function of the form $f : \{0,1\}^* \to \{0,1\}$ then we also say that it *decides* the *language*

$$L = \{x \in \{0,1\}^* : f(x) = 1\}$$

Obviously this is an equivalent interpretation and we will use those notions interchangeably. We want to pay more attention to the running time that it takes to decide a language.

Definition 1.9. For a function $T : \mathbb{N} \to \mathbb{N}$ we define **DTIME**$(T(n))$ as all the languages L for which there is a constant $c > 0$ and a Turing machine deciding L that runs in time $c \cdot T(n)$.

That means **DTIME**$(T(n))$ is the *complexity class* of functions/boolean functions that can be computed in time $O(T(n))$. Here the D in **DTIME** stands for *deterministic.* A key concept in complexity theory are the languages that can be decided in (deterministic) polynomial time.

Definition 1.10. We define $\mathbf{P} := \bigcup_{c>0} \mathbf{DTIME}(n^c)$.

We also want to justify that the concept of a Turing machine was the right one to make:

> **Church Turing Thesis.** *Every physically realizable device can be simulated with a Turing machine.*

Note that this is not a proven theorem but more of a *law of nature* that is widely believed and still stands after almost a century. There is also a stronger form:

> **Strong Church Turing Thesis.** *Every physically realizable device can be simulated with a Turing machine with polynomial overhead.*

Quantum computers — if they ever materialize — would likely violate this thesis. So there is less of a consensus on this stronger form.

1.4 The Halting problem

One might be tempted to believe that any function could be computed once we give it enough time. But that is not true.

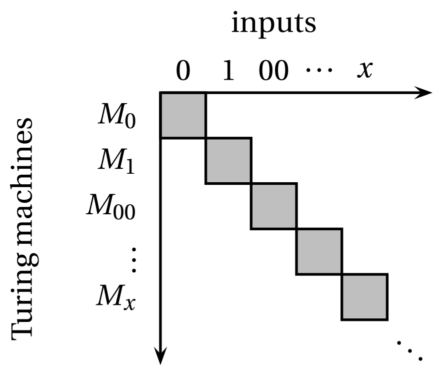
Theorem 1.11. *There is a function* $\mathtt{UC} : \{0,1\}^* \to \{0,1\}$ *that is not computable by a Turing machine.*

Proof. For $x \in \{0,1\}^*$, we define

$$\mathtt{UC}(x) := \begin{cases} 1 & \text{if } M_x \text{ halts on input } x \text{ with } M_x(x) = 0 \\ 0 & \text{if } M_x \text{ halts on input } x \text{ with } M_x(x) = 1 \\ 1 & \text{if } M_x \text{ does not halt on input } x \end{cases}$$

Now suppose there is a Turing machine M that computes $\mathtt{UC}$. In particular M always halts. Consider the input $x := [M]$ that corresponds to the encoding of the Turing machine. Then $\mathtt{UC}(x) \neq M_x(x)$ by construction. ☐

From an abstract point of view, there is an infinite list of Turing machines M_x and we explicitly constructed the function $\mathtt{UC}$ so that for every of those Turing machines M_x the function $\mathtt{UC}$ disagrees on at least one input. It was convenient to have M_x and $\mathtt{UC}$ disagree on diagonal entries, which is the reason that the technique is usually called *diagonalization.*



In Chapter 3 we will see more complex arguments where the entries are chosen rather differently — in particular in those applications the diagonal entries will not suffice. We should also mention that diagonalization has been used before the invention of computing by Cantor.

Theorem 1.12 (Cantor 1891)**.** *For all sets S, there is no surjective function $f : S \to 2^S$.*

A set S is called is called *countable* if there is an surjective map $f : \mathbb{N} \to S$. In other words, if S is countable then the elements of S can be "listed" in the form $f(1), f(2), f(3), \ldots$. For example $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ and $\mathbb{Z}^k$ for any fixed k are countable. Also the set of Turing machines is countable as they can be listed as $M_1, M_2, M_3, \ldots$. On the other hand, the set of functions $f : \{0,1\}^* \to \{0,1\}$ is not countable by Cantor's Theorem[1]. That means Theorem 1.11 already follows from Cantor's Theorem. But the simple explicit choice of UC is useful too.

We define a function

$$\texttt{HALT}(x, \alpha) = \begin{cases} 1 & \text{if Turing machine } M_\alpha \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

Theorem 1.13. *There is no Turing machine computing* HALT*.*

Proof. Suppose for the sake of contradiction that there is a Turing machine $M_{\texttt{HALT}}$ deciding HALT. We will show that we can decide UC on any input x. Recall that

$$\texttt{UC}(x) := \begin{cases} 1 & \text{if } M_x(x) = 0 \\ 0 & \text{if } M_x(x) = 1 \\ 1 & \text{if } M_x \text{ does not halt on input } x \end{cases}$$

[1]Already the restriction to functions $g : \{1^n : n \in \mathbb{N}\} \to \{0,1\}$ is not countable by Cantor.

We run $M_{\texttt{HALT}}(x,x)$ and can decide whether we are in the last case. If not, then we run the universal Turing machine on input (x,x) (which then must halt eventually) and reverse its output. $\square$

We will revisit the technique of diagonalization in more detail in Chapter 3.

Chapter 2

NP and NP completeness

The complexity class next to **P** that is most important for us is the following:

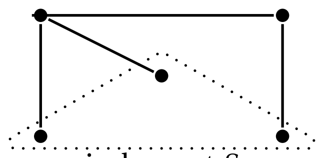
Definition 2.1. A language $L \subseteq \{0,1\}^*$ is in **NP** if there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time Turing maching M so that

$$L = \left\{x \in \{0,1\}^* \mid \exists u \in \{0,1\}^{p(|x|)} \text{ with } M(x,u) = 1\right\}$$

Here **NP** stands for *non-deterministic polynomial-time*. The Turing machine M is also called the *verifier* and the string u is called the *certificate*. Intuitively, a language is in **NP** if solutions can be *verified* efficiently. Note that $\mathbf{P} \subseteq \mathbf{NP}$. The reason why **NP** is so important is that so many relevant problems fall into it.

We want to give a few examples:

- *Independent Set.* Let $G = (V,E)$ be an undirected graph. Then a set $S \subseteq V$ of vertices is called an *independent set* if for all distinct $u, v \in S$ one has $\{u,v\} \notin E$.

  Then we define a language

  $$\texttt{INDSET} = \{(G,k) : G \text{ contains an independent of size } k\}$$

  Then $\texttt{INDSET} \in \mathbf{NP}$ where the certificate is simply the size-k independent set S and the verifier checks whether for all distinct $u, v \in S$ one has $\{u,v\} \notin E$.

- *Satisfiability*. A *SAT formula* over variables $x_1, \dots, x_n$ is of the form $\bigwedge_{i=1,\dots,m}(\bigvee_{\ell=1,\dots,k_i} u_{i,\ell})$ where each $u_{i,\ell} \in \{x_1, \neg x_1, \dots, x_n, \neg x_n\}$ is a *literal*. For example

  $$(x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

  is a SAT formula. A SAT formula is called *satisfiable* if there exists at least one satisfying assignment that makes all clauses true. We set

  $$\texttt{SAT} = \{\psi \mid \psi \text{ is a satisfiable formula}\}$$

  We claim that $\texttt{SAT} \in \mathbf{NP}$. The certificate is the assignment $x_1, \dots, x_n \in \{0,1\}$ that makes the formula true and the verifier simply checks for all clauses whether the assignment satisfies it.

- *Composite*. We define the problem

  $$\texttt{COMPOSITE} = \{n \in \mathbb{N} \mid \exists p, q \in \mathbb{Z}_{\geq 2} : n = p \cdot q\}$$

  Then $\texttt{COMPOSITE} \in \mathbf{NP}$ where the certificate are the numbers p, q (or just one of them). Actually it is even true that $\texttt{COMPOSITE} \in \mathbf{P}$, but proving that is much harder and requires a substantial amount of number theory.

It is believed that **NP** is strictly more powerful than **P**:

Conjecture 1. $\mathbf{P} \neq \mathbf{NP}$.

In all the decades of complexity research, no approach came every close to proving this and we will explain some difficulties and obstacles that any proof will face. We define **EXP** as the class of languages that can be solved in exponential time.

Definition 2.2. $\mathbf{EXP} := \bigcup_{c>0} \mathbf{DTIME}(2^{n^c})$.

At least we can give one simple inclusion:

Lemma 2.3. *One has* $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Proof. The first inclusion is clear. Next, let $L \in \mathbf{NP}$, i.e. there is a deterministic polynomial time TM M so that

$$L = \left\{x \in \{0,1\}^* : \exists u \in \{0,1\}^{p(|x|)} : M(x,u) = 1\right\}$$

for some polynomial p. Then we can try out all the $2^{p(|x|)}$ choices for u and compute $M(x,u)$ which together can be done in exponential time $2^{|x|^{O(1)}}$. $\square$

2.1 Non-deterministic Turing machines

Let us go back to the definition of Turing machines for decision problems, i.e. Turing machines that have no output tape but rather distinguished states q_{accept} and q_{reject}. Recall that the transition function in that case is of the form $\delta : Q \times \Gamma^{k+1} \to Q \times \Gamma^k \times \{L,S,R\}^{k+1}$. A *non-deterministic* Turing machine M has the same components, just that the transition function now gives a *set* of possible transitions, i.e. it is of the form

$$\delta : Q \times \Gamma^{k+1} \to \mathcal{P}(Q \times \Gamma^k \times \{L,S,R\}^{k+1})$$

and in each single iteration M may pick any possible transition. Here for a set A, $\mathcal{P}(A) := \{B \mid B \subseteq A\}$ is the *power set* which in the finite case has cardinality $|\mathcal{P}(A)| = 2^{|A|}$. That means on the same input $x \in \{0,1\}^*$ there are many *computation paths* that the non-deterministic Turing machine could possibly take. Some of these paths may end up in q_{accept} and some of the paths may be "dead ends" in the sense that they end in a state $(q, \sigma_0, \ldots, \sigma_k)$ with $\delta(q, \sigma_0, \ldots, \sigma_k) = \emptyset$[1].

Definition 2.4. We say that a *non-deterministic Turing machine (NTM) M accepts* an input x if at least one possible computation path leads to q_{accept} . The *running time* of M on input x is defined as the maximum length of any computation path.

We should note that while non-determinism is a highly useful concept in theory there does not seem to be a physical computer that it would represent. We define an analogue to **DTIME**:

Definition 2.5. For a function $T : \mathbb{N} \to \mathbb{N}$ we set

$$\mathbf{NTIME}(T(n)) := \left\{ f : \{0,1\}^* \to \{0,1\} \mid \text{there is a NTM } M \text{ computing } f \text{ in time } O(T(n)) \right\}$$

In our definition of **NP** we could have alternatively used NTMs.

Theorem 2.6. *One has* $\mathbf{NP} = \bigcup_{c>0} \mathbf{NTIME}(n^c)$.

Proof. Let $L \in \mathbf{NP}$. We can write $L = \{x \in \{0,1\}^* \mid \exists u \in \{0,1\}^{p(|x|)} : M(x,u) = 1\}$ where M is a (deterministic) Turing machine with some polynomial running time $q(n)$. We design a NTM that on input x first writes a sequence of bits $u_1, \ldots, u_{p(|x|)} \in \{0,1\}$ on a separate tape where we use the non-determinism to decide whether a bit should be 0 or 1. Then we run the TM M in time $q(|x| + p(|x|))$ which again is a polynomial.

[1] We do not actually need the rejecting state q_{reject} here — we can simple use a dead end for the same purpose.

Let $L \in \bigcup_{c>0} \mathbf{NTIME}(n^c)$, meaning there is some k-tape NTM M deciding L in polynomial time $p(|x|)$. In each step, M has the choice between at most $B := |Q \times \Gamma^k \times \{L, S, R\}^{k+1}|$ many options, where B is a constant. We choose the witness $u \in (Q \times \Gamma^k \times \{L, S, R\}^{k+1})^{p(n)}$ (encoded in $\log(B) \cdot p(|x|)$ bits). Now, we rebuild the NTM so that instead of making a non-deterministic move in iteration i, it uses the choice encoded in u_i. Clearly there is a way to reach q_{accept} using the certificate if and only if there was one using non-determinism. ☐

The second direction makes an argument that is often helpful: a NTM M could make all the non-deterministic choices at the very beginning and after that only make deterministic decisions.

2.2 Reductions and the Cook-Levin Theorem

Ideally we would like to be able to prove that for example SAT $\notin$ **P**, but we have no current approach to do that. Then the second best option is to at least show that certain groups of problems are equally hard — so they are all either in **P** or all not in **P**. This is done via reductions.

Definition 2.7 (Karp Reduction)**.** For languages $A, B \subseteq \{0,1\}^*$ we write $A \leq_p B$ if there is a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ so that

$$\forall x \in \{0,1\}^* : \quad \big(x \in A \Leftrightarrow f(x) \in B\big)$$

We can summarize a few properties of this reduction:

Lemma 2.8. *The following holds for languages A, B, C:*

(1) If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.
(2) If $A \leq_p B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$.

Proof. (1). Suppose f is the reduction from A to B computable in time $p(n)$ and suppose g is the reduction from B to C computable in time $q(n)$. Then $x \in A \Leftrightarrow g(f(x)) \in C$ and the composition $g(f(x))$ is computable in time $O(p(|x|) + q(p(|x|)))$ which is again a polynomial.
(2). Suppose f is the reduction from A to B. Then on input x we first compute $f(x)$ and then test whether $f(x) \in B$. Both operations take polynomial time. ☐

Note that we have just proven that $\leq_p$ is a partial order. Next, we are wondering whether there are some problems in **NP** that are "hardest".

Definition 2.9. A language $L \subseteq \{0,1\}^*$ is called **NP**-*hard* if $L' \leq_p L$ for every $L' \in \mathbf{NP}$. A language $L \in \{0,1\}^*$ is called **NP**-*complete* if it is **NP**-hard and $L \in \mathbf{NP}$.

Note that it is not obvious that **NP** contains **NP**-complete problems. One could have imagined that there are problems in **NP** that have no common upper bound (called *join*) w.r.t. $\le_p$ or maybe there could have been an infinite chain of harder and harder problems. But fortunately enough, there are indeed very natural hardest problems in **NP**:

Theorem 2.10 (Cook-Levin)**.** `SAT` *is* **NP**-*complete.*

Proof. We already proved that `SAT` $\in$ **NP**, hence only the hardness is missing. We prove an auxiliary result first.

Claim I. *Every boolean function* $f : \{0,1\}^k \to \{0,1\}$ *can be written as a CNF with at most* 2^k *clauses.*

Proof of Claim I. First, for any $v \in \{0,1\}^k$ the clause

$$C_v(x) := \Big(\bigvee_{i:v_i=1} \neg x_i\Big) \vee \Big(\bigvee_{i:v_i=0} x_i\Big)$$

has $C_v(v) = 0$ and $C_v(x) = 1$ for all $x \neq v$. Then

$$f(x) = \bigwedge_{v\in\{0,1\}^k : f(v)=0} C_v(x)$$

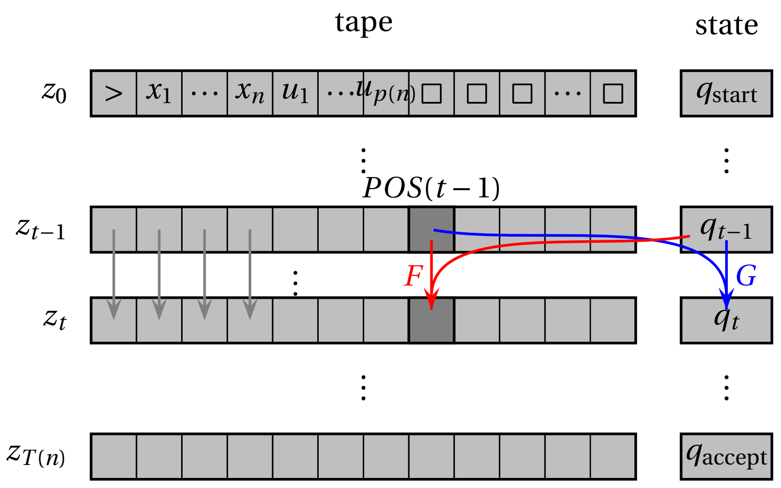
does the job. $\square$

Note that in Claim I the CNF size is exponential in the number of variables — but we will only use this fact for $k \le O(1)$. Now to the main part of the proof. Let $L \in$ **NP** be any problem in **NP**. We can write $L = \{x \in \{0,1\}^* \mid \exists u \in \{0,1\}^{p(|x|)} : M(x,u) = 1\}$ where p is a polynomial. Following the result of Theorem 1.5 we may assume that M is an *oblivious* Turing machine with polynomial running time q and with states Q that has a single combined input and working tape[2] using alphabet Γ. We fix an input $x \in \{0,1\}^*$ of length $n := |x|$. Let $POS(t)$ be the position of the head at time t (which otherwise only depends on n). Note that we may assume that $T(n) := q(n + p(n))$ is the running time and also the maximum number of cells on the tape that is being used. We want to encode the working of the Turing machine M. We index the iterations as $t = 0, \dots, T(n)$ and denote $z_t = (z_t(1), \dots, z_t(T(n))) \in \Gamma^{T(n)}$ as the content of the tape in the tth iteration. Moreover, we denote $q_t \in Q$ as the state of the Turing machine in iteration t. Then for some functions $F : Q \times \Gamma \to \Gamma$ and $G : Q \times \Gamma \to Q$ implied by the transition

[2]This restriction merely makes the notation easier and we can afford the restriction as we do not care about a polynomial blowup here.

function δ, the Turing machine can be encoded as

$$\begin{aligned} z_0 &= (>, x_1, \ldots, x_n, u_1, \ldots, u_{p(n)}, \Box, \ldots, \Box) \qquad (2.1)\\ z_t(i) &= z_{t-1}(i) \quad \forall t \in [T(n)] \text{ and } i \text{ with } i \neq POS(t-1)\\ z_t(POS(t-1)) &= F(q_{t-1}, z_{t-1}(POS(t-1))) \quad \forall t \in [T(n)]\\ q_t &= G(q_{t-1}, z_{t-1}(POS(t-1))) \quad \forall t \in [T(n)]\\ q_0 &= q_{\text{start}}\\ q_{T(n)} &= q_{\text{accept}} \end{aligned}$$



Then $x \in L$ if and only if there are $u_1, \ldots, u_{p(n)} \in \{0,1\}$ so that the equations in (2.1) hold. Note that the variables that we have used are not boolean as $z_t(i) \in \Gamma$ and $q_t \in Q$. But by using the binary encoding of symbols in Q and Γ each can be replaced by a constant number of boolean variables. Each equation in (2.1) then transforms into the form that a boolean variable equals a boolean function depending only on a *constant* number of other boolean variables. Hence by Claim I, each such equation can be transformed into a CNF of constant size. Then taking the $\bigwedge$ of all obtained CNFs gives a (larger) CNF ψ. That SAT formula contains variables $u_1, \ldots, u_{p(n)} \in \{0,1\}$ plus a polynomial number of auxiliary boolean variables y and it is satisfied by an assignment (u, y) if and only if $M(x,u) = 1$. That concludes the proof. $\Box$

2.3 More NP-complete problems

We have a few other natural problems for which we will demonstrate how to prove **NP**-completeness. Again, the harder part here will be proving the **NP**-hardness. It may seem that we would need to modify the Cook-Levin Theorem each time that we have a new target language. While this could in principle be done, it will be much easier to make use of the following observation:

Lemma 2.11. *Let A and B be any languages. If A is* **NP***-hard and $A \leq_p B$, then B is* **NP***-hard.*

Proof. Let $L \in$ **NP**. Then $L \leq_p A \leq_p B$ and hence $L \leq_p B$ by transitivity. □

We define 3SAT as the language of all satisfiable SAT formulas ψ where each clause contains at most 3 literals.

Lemma 2.12. 3SAT *is* **NP***-complete.*

Proof. One has 3SAT $\in$ **NP** by the same argument as for SAT. By the Cook-Levin Theorem we know that SAT is **NP**-hard, hence it suffices to show that SAT $\leq_p$ 3SAT. Consider any SAT instance ψ with variables $x_1, \dots, x_n$. Let $u_1 \vee u_2 \vee \dots \vee u_k$ be a clause C with $k \geq 4$ where each u_i is a literal (i.e. it is either a variable x_j or its negation $\neg x_j$). Then we introduce $k-3$ extra variables $z_{C,1}, \dots, z_{C,k-3}$ and replace the clause C by the 3SAT formula ϕ_C which is

$$(u_1 \vee u_2 \vee z_{C,1}) \wedge (\neg z_{C,1} \vee u_3 \vee z_{C,2}) \wedge \ldots \wedge (\neg z_{C,k-4} \vee u_{k-2} \vee z_{C,k-3}) \ldots (\neg z_{C,k-3} \vee u_{k-1} \vee u_k)$$

Note that each auxiliary variable $z_{C,i}$ is contained exactly twice, once positively and once negated. For any clause C with at most 3 variables we simply copy $\phi_C := C$. Then one can verify that $f(\psi) := \bigwedge_C \phi_C$ is a 3SAT formula that is satisfiable if and only if ψ is satisfiable. Moreover f is computable in polynomial time and so SAT $\leq_p$ 3SAT. □

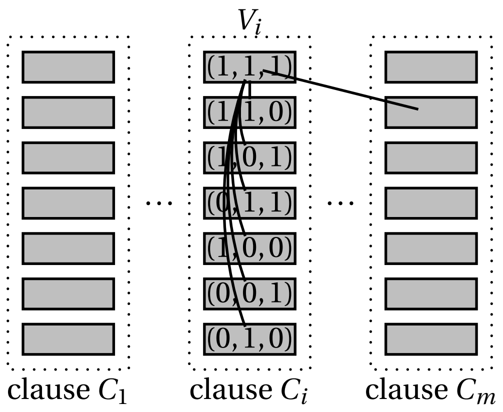
The problems of SAT and 3SAT are so similar that it did not actual come as a surprise that there is a reduction between them. But we can also show a reduction between very differently structured problems:

Lemma 2.13. INDSET *is* **NP***-complete.*

Proof. We already argued that INDSET $\in$ **NP**. To show that INDSET is **NP**-hard, we prove that 3SAT $\leq_p$ INDSET. Consider a 3SAT formula $\psi = \bigwedge_{i=1,\dots,m} C_i$ where each clause is of the form[3] where each u_{ij} is a literal and we have variables $x_1, \dots, x_n$.

[3]Well, we have defined 3SAT instance ψ as having *at most* 3 literals per clause. But in a reduction it will be notationally convenient to assume *exactly* 3 literals from distinct variables per clause. We can replace a clause C with 2 variables by two clauses $C \wedge z_1$, $C \wedge \neg z_1$. Similarly a clause C with 1 variable can be replaced by 4 clauses with all combinations of z_1, z_2. Here z_1, z_2 are newly introduced variables.

We create an undirected graph $G = (V, E)$ with $7m$ vertices where for each clause C_i, we introduce $|V_i| = 7$ nodes that corresponds to the 7 possible partial assignments $u_{C,1} = a_1, u_{C,2} = a_2, u_{C,3} = a_3$ with $a \in \{0,1\}^3 \setminus \{(0,0,0)\}$. Then $V = \bigcup_{i=1}^m V_i$. We insert an edge between two vertices if their partial assignments are *inconsistent*. In particular the vertices belonging to the same clause are inconsistent, i.e. each set V_i corresponds to a clique. But also nodes belonging to different clauses C_i and $C_{i'}$ are inconsistent if their variables overlap and are set to different values[4].



We claim that $\psi \in \texttt{3SAT} \Leftrightarrow (G, m) \in \texttt{INDSET}$ and want to verify both directions.
Claim I. $\psi \in \texttt{3SAT} \Rightarrow (G, m) \in \texttt{INDSET}$.
Proof of Claim I. Let $x \in \{0,1\}^n$ be a satisfying assignment for ψ. For each clause C_i, exactly one of the 7 partial assignments is consistent with x. We let $S \subseteq V$ be the $|S| = m$ nodes corresponding to those partial assignments. As we picked only one vertex per V_i and for different clauses only vertices that are consistent, there is no edge running inside S. That means S is an independent set and so $(G, m) \in \texttt{INDSET}$.
Claim II. $(G, m) \in \texttt{INDSET} \Rightarrow \psi \in \texttt{3SAT}$.
Proof of Claim II. Consider an independent set $S \subseteq V$ of size m. Since each V_i corresponds to a clique we know that $|S \cap V_i| = 1$. For each clause C_i we select the partial assignment in $S \cap V_i$ and use it to define 3 variables in an assignment $x \in \{0,1\}^n$. Since there are no edges running inside of S, we definitions will be consistent (i.e. is we set $x_j := 1$ due to one clause then this will also be the assignment from any other clause that contains the variable). By construction this assignment x satisfies all clauses[5]. $\square$

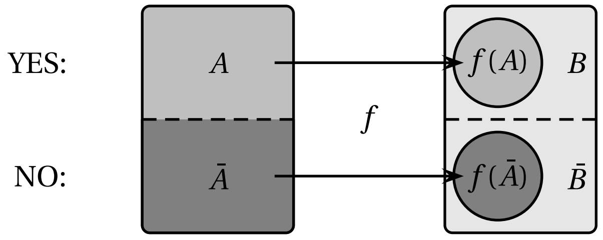
In the homework we will see more **NP**-completeness proofs. Note that as

[4]For example the partial assignments $\neg x_1 = 1, x_2 = 1, x_3 = 1$ and $x_1 = 1, x_4 = 1, x_5 = 1$ are inconsistent since the variable x_1 is once set to 0 and once to 1.

[5]It could happen that some variable did not appear in any clause, then the corresponding variable will be undefined — but then it's value is irrelevant anyway.

the number of known **NP**-complete problems grows, it becomes easier to prove **NP**-hardness for any new problem since we may pick the closest and most convenient problem for a reduction.

We also want provide a comment on the nature of the Karp reduction. If $A \leq_p B$ with a map f, then YES instances of A are mapped to YES instance of B and NO instances of A are mapped to NO instances of B. But in general f is not surjective and the instances that f produces can be rather specific.



For example all the instances (G, k) produced by the reduction in Lemma 2.13 have the special property that the graph can be partitioned into k cliques and so we know that `INDSET` is still **NP**-hard if restricted to those instances.

2.4 The structure of NP

Out of the thousands of problems that are known to be in **NP** essentially all are also known to be either **NP**-complete or to be in **P**. There are only a few arcane examples such as factoring integers where the status is open. So one might be tempted to believe that $\mathbf{P} \neq \mathbf{NP}$ and each problem in **NP** is indeed either **NP**-complete or in **P**. But that is provably not the case.

Theorem 2.14 (Ladner's Theorem)**.** *If* $\mathbf{P} \neq \mathbf{NP}$ *then there are problems in* $\mathbf{NP} \setminus \mathbf{P}$ *that are not* **NP***-complete.*

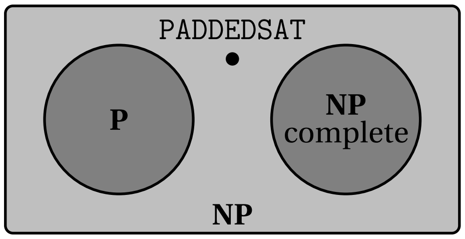
The proof has a quite delicate component, so we want to prove a weaker statement to convey the idea behind the argument. Here the choice for the function $\log\log n$ that we make is arbitrary and any growing function would have worked.

Lemma 2.15. *Assume* $\mathbf{NP} \not\subseteq \mathbf{DTIME}(n^{O(\log\log n)})$. *Then there is a language* $L \in \mathbf{NP}$ *that is neither in* **P** *nor* **NP***-complete.*

Proof. Consider the language

$$\texttt{PADDEDSAT} := \left\{(\psi, 1^{n^{\log\log n}}) : |\psi| = n \text{ and } \psi \in \texttt{SAT}\right\}$$

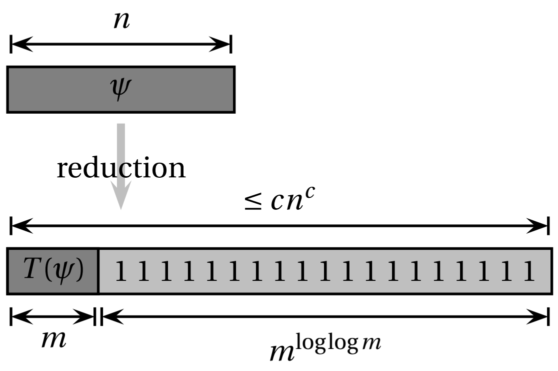
of feasible SAT formulas padded with a slightly super polynomial number of ones. We claim the following picture:



Because `SAT` $\in$ **NP**, also `PADDEDSAT` $\in$ **NP**.
Claim I. `PADDEDSAT` $\notin$ **P**.
Proof of Claim I. Now suppose for the sake of contradiction that `PADDEDSAT` $\in$ **P** which means there is an algorithm deciding length-m inputs for `PADDEDSAT` in time cm^c. Then for a length-n SAT formula ψ we can use that algorithm to test feasiblity in time $c(n + n^{\log\log n})^c \leq n^{2c\log\log n}$ for n large enough. That is a contradiction to the assumption.
Claim II. `PADDEDSAT` *is not* **NP**-*complete*.
Proof of CLaim II. Suppose for the sake of contradiction that `PADDEDSAT` is indeed **NP**-complete. That means there is a function T that maps a length-n SAT formula ψ to $(T(\psi), 1^{m^{\log\log m}})$ where $m := |T(\psi)|$ where

$$\psi \in \texttt{SAT} \quad \Leftrightarrow \quad (T(\psi), 1^{m^{\log\log m}}) \in \texttt{PADDEDSAT} \quad \Leftrightarrow \quad T(\psi) \in \texttt{SAT}$$

By assumption, the function T can be computed in time cn^c for some constant $c > 0$. Then $m^{\log\log m} \leq cn^c \Rightarrow m \leq cn^{c/\log\log n} \leq \frac{n}{2}$ for all n large enough. That means SAT is self-reducible to a shorter SAT formula. Iterating the reduction $\log(n)$ times we reach a constant size SAT formula that we can solve trivially. This gives a polynomial time algorithm to decide whether $\psi \in$ `SAT` which is a contradiction to the assumption.



$\square$

2.5 coNP and NEXP

For a language $L \subseteq \{0,1\}^*$ we define its *complement* as $\bar{L} := \{0,1\}^* \setminus L$. We make the following definition:

Definition 2.16. $\mathbf{coNP} := \{\bar{L} : L \in \mathbf{NP}\}$.

By taking the complement in the original definition of **NP** one could have also defined **coNP** as follows:

Definition 2.17 (Alternative **coNP** definition)**.** We say that a language $L \subseteq \{0,1\}^*$ is in **coNP** if there is a polynomial p and a polynomial time Turing machine M so that

$$L = \left\{x \in \{0,1\}^* \mid \forall u \in \{0,1\}^{p(|x|)} : M(x,u) = 1\right\}$$

Recall that for $L \in \mathbf{NP}$ and $x \in L$ there is an efficiently verifiable certificate for the fact that $x \in L$. In contrast, for $L \in \mathbf{coNP}$ and $x \notin L$, there is an efficiently verifiable certificate for the fact that $x \notin L$. Note that $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.

We can give an example of a problem in **coNP**. A boolean formula is in *disjunctive normal form (DNF)* if it is of the form $\bigvee_{i=1,\dots,m}(\bigwedge_{\ell=1,\dots,k_i} u_{i,\ell})$ where $u_{i,\ell}$ are literals of boolean variables.

Example 2.18. Consider the language

$$\texttt{TAUTOLOGY} = \left\{\psi \mid \psi \text{ is a DNF that is a tautology}^{6}\right\}$$

Then $\texttt{TAUTOLOGY} \in \mathbf{coNP}$ and in fact `TAUTOLOGY` is **coNP**-complete.

If $\mathbf{NP} = \mathbf{coNP}$, then for example one could efficiently certify that a SAT formula (i.e. a CNF) is not satisfiable which does not seem very plausible. So we make the following conjecture (which in particular implies that $\mathbf{P} \neq \mathbf{NP}$):

Conjecture 2. $\mathbf{NP} \neq \mathbf{coNP}$.

We can also define a non-deterministic analogue to **EXP**.

Definition 2.19. $\mathbf{NEXP} := \bigcup_{c>0} \mathbf{NTIME}(2^{n^c})$.

Clearly $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$ while in the only separations that are known are $\mathbf{P} \neq \mathbf{EXP}$ and $\mathbf{NP} \neq \mathbf{NEXP}$. We will prove those separations later in Chapter 3. It is a little harder to motivate the study of classes like **EXP** and **NEXP**, but their relationship has implications for **P** and **NP** as well:

[6] Recall that a boolean formula is a *tautology* if it is satisfied by every assignment.

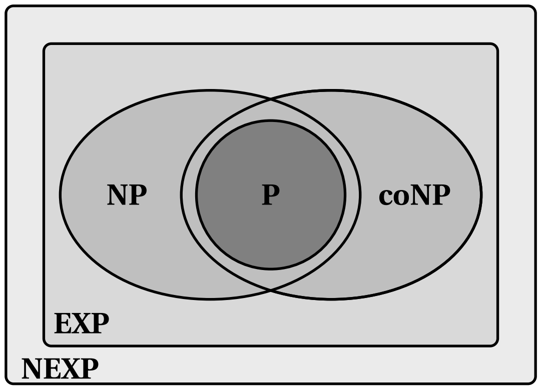
Theorem 2.20. $(\mathbf{EXP} \neq \mathbf{NEXP}) \Rightarrow (\mathbf{P} \neq \mathbf{NP})$.

Proof. It is easier to reason on the contrapositive which is $(\mathbf{P} = \mathbf{NP}) \Rightarrow (\mathbf{EXP} = \mathbf{NEXP})$. So we assume $\mathbf{P} = \mathbf{NP}$ and consider a language $L \in \mathbf{NTIME}(2^{n^c})$ for some constant $c > 0$ that is decided by some $O(n^{2^c})$-time NTM M. Define the padded version of L as

$$L_{\text{pad}} := \left\{(x, 1^{2^{|x|^c}}) : x \in L\right\}$$

Then the non-deterministic computation of $M(x)$ takes time $O(2^{|x|^c})$ which is polynomial in the length of $(x, 1^{2^{|x|^c}})$. Hence $L_{\text{pad}} \in \mathbf{NP} = \mathbf{P}$. But if $x \overset{?}{\in} L$ can be decided deterministically in time polynomial in $|x| + 2^{|x|^c}$, then this means that $L \in \mathbf{EXP}$. $\square$

We conclude with the conjectured picture of the complexity relations:



Chapter 3

Diagonalization

In this chapter, we revisit the technique of diagonalization that we briefly discussed in Chapter 1.4 and we will use it to derive some of the most striking results in complexity.

If we have a language $L \in \mathbf{DTIME}(T(n))$ computed by a Turing machine M, then all we know is that for some constant $C := C_M > 0$, M takes at most $C_M T(n)$ iterations. But that constant C_M may depend on M and in order to separate $\mathbf{DTIME}(T(n))$ from other classes we need to consider inputs that are long enough to make C_M irrelevant. There is a small trick that will be helpful:

Lemma 3.1 (The Recurrent Turing Machine Sequence)**.** *There is a sequence $M_1, M_2, M_3, \ldots$ of Turing machines that contains every Turing machine infinitely often. Moreover given $n \in \mathbb{N}$, computing $[M_n]$ takes time $O(\log n)$.*

Proof. Write the binary encoding of n as $x01^k$ where $x \in \{0,1\}^*$. Then use $M_n := M_x$. □

3.1 The Time Hierarchy Theorems

We can (completely unconditionally) prove that with a little more running time we can solve strictly more problems.

Theorem 3.2 (Time Hierarchy Theorem 1965)**.** *For any time-constructable functions $f, g : \mathbb{N} \to \mathbb{N}$ with $f(n)\log(f(n)) \leq o(g(n))$ one has*

$$\mathbf{DTIME}(f(n)) \subset \mathbf{DTIME}(g(n))$$

Proof. In order to keep the argument readable we prove the special case that for any rational numbers $c \geq 1$ and $\varepsilon > 0$ one has $\mathbf{DTIME}(n^c) \subset \mathbf{DTIME}(n^{c+\varepsilon})$. So we need to construct a function that is in $\mathbf{DTIME}(n^{c+\varepsilon})$ and that disagrees with every n^c-time Turing machine on at least one input. We do that by constructing a Turing machine D. Note that we only define D on inputs of the form 1^n and we do not care what it does for other inputs. In the following, we write M_i as the ith Turing machine in the recurrent Turing machine sequence from Lemma 3.1.

> *Turing machine D:* On input 1^n, run the universal Turing machine $\mathcal{U}$ for $n^{c+\varepsilon}$ iterations[1] on input $([M_n], 1^n)$. If the simulation of M_n did not terminate, output 0. If the simulation of M_n did terminate, reverse the output.

First, the running time of D on an input of length n is bounded by a constant times $n^{c+\varepsilon}$ and hence the language L that D computes is in $\mathbf{DTIME}(n^{c+\varepsilon})$. Now, for the sake of contradiction assume that there is a Turing machine M taking $O(n^c)$ iterations that also computes L. Then by Theorem 1.8 there is a constant $C_M > 0$ (depending on M) so that the simulation of M takes time $C_M n^c \log(n^c)$. Then there is some large enough n so that $M = M_n$ and $C_M n^c \log(n^c) < n^{c+\varepsilon}$. For this choice of n the simulation will be completed and D will have returned the reversed output of M, i.e. $D(1^n) \neq M(1^n)$. ☐

In particular this gives us a separation of two classes that we have introduced earlier

Corollary 3.3. P ≠ EXP.

We can also prove a non-deterministic analogue of Theorem 3.2.

Theorem 3.4 (Nondeterministic Time Hierarchy Theorem)**.** *If $f, g : \mathbb{N} \to \mathbb{N}$ are time-constructable functions satisfying $f(n+1) = o(g(n))$ then*

$$\mathbf{NTIME}(f(n)) \subset \mathbf{NTIME}(g(n))$$

The reader should note that the natural adaptation of the proof of Theorem 3.2 will not work because $\mathbf{NTIME}(f(n))$ is asymmetric. In particular if $L \in \mathbf{NTIME}(f(n))$ then it is not clear why one would have $\bar{L} \in \mathbf{NTIME}(f(n))$. This makes flipping the output highly problematic.

Proof. Again we show the simpler statement of $\mathbf{NTIME}(n^c) \subset \mathbf{NTIME}(n^{c+\varepsilon})$ for some rational constant $c \geq 1$ and $\varepsilon > 0$. We define an enourmously growing function h with $h(1) = 2$ and $h(i+1) := 2^{(h(i)+1)^{2c}}$. We define a non-deterministic Turing machine D. Again we only specify the behavior on inputs of the form $x = 1^n$

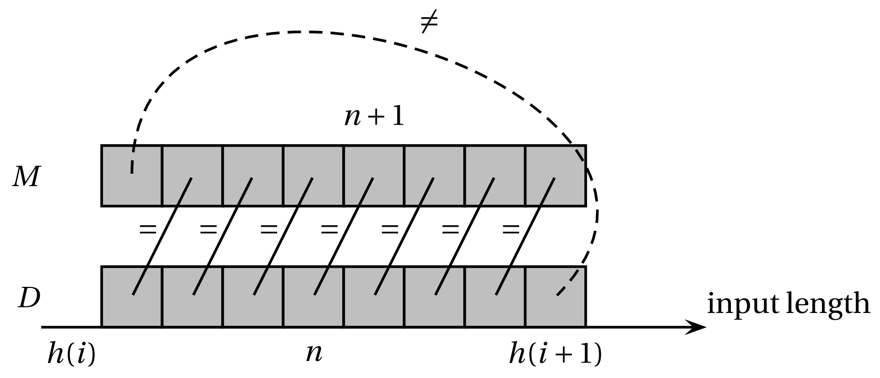
[1]This refers to the running time of $\mathcal{U}$; *not* to the (smaller) number of simulated iterations of M_n.

— the machine may do whatever on other inputs. As before, we denote M_i as the ith Turing machine in the recurrent Turing machine sequence (see Lemma 3.1).

> *Turing machine D*. On input 1^n:
>
> (1) Compute i with $h(i) < n \le h(i+1)$.
> (2) If $h(i) < n < h(i+1)$ then simulate M_i non-deterministically on input 1^{n+1} for time $n^{c+\varepsilon}$ and output its answer
> (3) If $n = h(i+1)$ then deterministically simulate M on input $1^{h(i)+1}$ (by brute force) for time $n^{c+\varepsilon}$ and flip the output.

Now suppose for the sake of contradiction that there is a Turing machine M corresponding to **NTIME**(n^c) that computes the same function as D. Consider a large enough index i with $M = M_i$ (and hence also n will be large enough). Then in time (2) completing the non-deterministic simulation takes time $C_M(n+1)^c \le n^{c+\varepsilon}$. In (3) we have to simulate a non-deterministic n^c-time Turing machine on an input of length $h(i)+1$. By trying out all possible choices of the non-determinism, this can be done in time $(C'_M)^{(h(i)+1)^c} \le h(i+1)$ where C'_M is the maximum number of choices that M makes in any iteration and we used the assumption that i is large enough. Then all simulations will complete in time and the output will satisfy

$$D(1^n) = M(1^{n+1}) \text{ for all } h(i) < n < h(i+1) \text{ and } D(1^{h(i+1)}) \ne M(1^{h(i)+1})$$



But if M and D compute the same function then also $D(1^n) = M(1^n)$ for all n — that is implossible! □

The intuition behind the proof is that flipping the input of a non-determinstic machine has to be done by trying out all non-deterministic choices which takes exponential time. But if this is only done for an input that is vastly shorter than D's input, then this can be tolerated.

3.2 Oracle Turing machines and Limits of Diagonalization

As we have seen, diagonalization appears to be the most effective proof technique for separating complexity classes. Naturally one might want to prove $\mathbf{P} \neq \mathbf{NP}$ using diagonalization as well. But there are arguments that this might not be easy (or maybe it is impossible).

Definition 3.5. For a language $O \subseteq \{0,1\}^*$, an *oracle Turing machine M^O with oracle O* is a Turing machine that has an extra tape (called *oracle tape*) and 3 special states $q_{\text{query}}, q_{\text{yes}}, q_{\text{no}}$. The Turing machine can write a string z on the oracle tape and enter q_{query}. In the next iteration the Turing machine is in q_{yes} iff $z \in O$ and in q_{no} otherwise.

Intuitively this means we give the Turing machine M^O access to queries of the form $z \overset{?}{\in} O$ at cost of 1 iteration.

Definition 3.6. For a language $O \subseteq \{0,1\}^*$ we define $\mathbf{P}^O$ as the set of languages decided by deterministic polynomial time oracle Turing machines with access to an oracle for O. Similar we define $\mathbf{NP}^O$ and $\mathbf{DTIME}^O(T(n))$.

Proofs by diagonalization tend to *relativize*, i.e. they also work relative to oracles. Here is an example:

Corollary 3.7. *For any language $A \subseteq \{0,1\}^*$ and rational $c \geq 1$, $\varepsilon > 0$ one has* $\mathbf{DTIME}^A(n^c) \subset \mathbf{DTIME}^A(n^{c+\varepsilon})$.

The reader may verify that the exact same argument as in Theorem 3.2 of simulating a machine in $\mathbf{DTIME}^O(n^c)$ and flipping its output works also in the presence of an oracle. Also Theorem 3.4 relativizes. Next, we prove a result that explains a major obstacle towards proving that $\mathbf{P} \neq \mathbf{NP}$.

Theorem 3.8. *There are languages $A, B \subseteq \{0,1\}^*$ so that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$.*

For the first part, we select A as a language that is so powerful that the non-determinism of $\mathbf{NP}$ does help. We define

$$\texttt{EXPCOM} := \left\{(M, x, 1^n) : \text{TM } M \text{ outputs 1 on input } x \text{ in time } 2^n\right\}$$

which essentially is the naturally **EXP**-complete language. To show that for $A :=$ `EXPCOM` one has $\mathbf{P}^A = \mathbf{NP}^A$ it suffices to show:

Lemma 3.9. *One has* $\mathbf{EXP} \overset{(*)}{\subseteq} \mathbf{P}^{\texttt{EXPCOM}} \subseteq \mathbf{NP}^{\texttt{EXPCOM}} \overset{(**)}{\subseteq} \mathbf{EXP}$.

Proof. For $(*)$, let $L \in \mathbf{EXP}$ and let M be a Turing machine deciding L in time $c2^{n^c}$ for some $c > 0$. Then on input x we ask the oracle whether $(M, x, 1^{c|x|^c}) \in \texttt{EXPCOM}$ which takes polynomial time[2].

Next, consider $(**)$. Let M be a non-deterministic polynomial time Turing machine with oracle access to $\texttt{EXPCOM}$. Say that M halts after $p(n)$ iterations for some polynomial p and the maximum number of non-deterministic choices in the transition funtion of M is $C > 0$. Then we can try out the at most $C^{p(n)}$ many candidate choices for the non-determinism. Moreover we can decide each $\texttt{EXPCOM}$ oracle call on input $(\tilde{M}, \tilde{x}, 1^{\tilde{n}})$ in deterministic time $\text{poly}(|\tilde{x}|+|\tilde{M}|)\cdot\log(2^{\tilde{n}})\cdot 2^{\tilde{n}}$ by a simulation with the universal Turing machine using that $|[\tilde{M}]| + |\tilde{x}| + \tilde{n} \leq p(n)$. Overall the total running time is $O(2^{n^{O(1)}})$. □

Next, we need to construct a language B so that $\mathbf{P}^B \neq \mathbf{NP}^B$. The idea is that a deterministic algorithm can only query a polynomial number of strings in B so we make sure that the queries are not useful. On the other hand, a non-deterministic algorithm can just guess the right string. This argument crucially uses that B is a black box only accessible through queries and it does not seem that such an argument would make sense for $\mathbf{P} \neq \mathbf{NP}$.

Lemma 3.10. *There is a language* $B \in \{0,1\}^*$ *so that the language* $L(B) := \{1^n \mid \exists y \in B : |y| = n\}$ *satisfies* $L(B) \in \mathbf{NP}^B$ *and* $L(B) \notin \mathbf{P}^B$.

Proof. First, for any choice of B, one has that $L(B) \in \mathbf{NP}^B$. To see this, consider an input 1^n. Then we guess $y \in \{0,1\}^n$ and use the oracle access to test whether $y \in B$.

The subtle point lies in how to construct B so that $L(B) \notin \mathbf{P}$. As before, we consider a recurrent sequence of oracle Turing machines $M_1, M_2, \ldots$ (see Lemma 3.1). The set B will be the result of an infinite iterative process. For $i = 1, 2, \ldots$ we do the following: set $n_1 := 2$ and $n_{i+1} := \frac{1}{2} \cdot 2^{n_i} + 1$ which is an enourmously fast growing function. Run $M_i^{B_{i-1}}$ on input 1^{n_i} for $\frac{1}{2} \cdot 2^{n_i}$ iterations. There must be some string y of length n_i that was not queried by M_i. We set $B_0 := \infty$ and for $i \geq 1$,

$$B_i := \begin{cases} B_{i-1} \cup \{y\} & \text{if } M_i^{B_{i-1}}(1^{n_i}) = 0 \\ B_{i-1} & \text{if } M_i^{B_{i-1}}(1^{n_i}) = 1 \end{cases}$$

(in the case that the TM does not terminate, pick any output). We define $B := \bigcup_{i \geq 0} B_i$ which is also the limit of the sequence $B_0 \subseteq B_1 \subseteq \ldots$.

[2]Recall that we do not pay any running time for the oracle computation itself.

Now we prove that $L(B) \notin \mathbf{P}$. Suppose for the sake of contradiction that there is a Turing machine M that decides $L(B)$ in polynomial time. There is an i so that $M = M_i$ and i and n_i are large enough so that $M^{B_{i-1}}$ is able to terminate on input 1^{n_i} in time $2^{n_i/2}$. Note that for $j > i$, n_j is exponentially larger and hence all strings that are added later are so long that $M^{B_{i-1}}$ will not have been able to query them and y is the string that is not queried anyway. That means $M^{B_{i-1}}(1^{n_i}) = M^B(1^{n_i})$. Then

$$M^B(1^{n_i}) = 0 \quad\Longrightarrow\quad M^{B_{i-1}}(1^{n_i}) = 0 \quad\Longrightarrow\quad y \in B \quad\Longrightarrow\quad 1^{n_i} \in L(B)$$

and analogously

$$M^B(1^{n_i}) = 1 \quad\Longrightarrow\quad M^{B_{i-1}}(1^{n_i}) = 1 \quad\Longrightarrow\quad y \notin B \quad\Longrightarrow\quad 1^{n_i} \notin L(B)$$

Hence M_i^B does not compute $L(B)$. □

Chapter 4

Space Complexity

So far we have focused on the *time* that a Turing machine takes to decide a language. But it also makes sense to bound the *space* that is needed.

4.1 Introduction to space complexity

We make the following definitions

Definition 4.1. We set

$$\begin{aligned} \mathbf{DSPACE}(S(n)) &:= \left\{ L \subseteq \{0,1\}^* \mid \begin{array}{c} \exists \text{TM } M \text{ that decides } L \text{ using at most } O(S(|x|)) \\ \text{cells on any work tape and any input } x \end{array} \right\} \\ \mathbf{NSPACE}(S(n)) &:= \left\{ L \subseteq \{0,1\}^* \mid \begin{array}{c} \exists \text{NTM } M \text{ that decides } L \text{ using at most } O(S(|x|)) \\ \text{cells on any work tape on any} \\ \text{computation path on any input } x \end{array} \right\} \end{aligned}$$

We note that **DSPACE**(0) and **DSPACE**(1) correspond to the *regular languages*[1]. For technical reasons we will limit the function $S(n)$ that we consider[2].

Definition 4.2. A function $S : \mathbb{N} \to \mathbb{N}$ is *space-constructable* if $S(n) \geq \log(n)$ and there is a deterministic TM M using space $O(S(n))$ that takes x as input and produces the binary encoding of the number $S(|x|)$.

We also want to define a few specific classes that take the centerstage in space complexity:

[1]Having $O(1)$ space is not helpful since the TM can also remember the content of its working tape in its state. Then **DSPACE**(0) corresponds to automata that can go back and forth when reading the input. But one can prove that this is not more powerful than reading the input once.

[2]Basically we are saying that the TM M needs to be able to produce a binary encoding of $S(n)$ on the space that it has.

Definition 4.3. We define

$$\begin{array}{rcll} \mathbf{PSPACE} & := & \bigcup_{c>0} \mathbf{DSPACE}(n^c) & \text{polynomial space} \\ \mathbf{NPSPACE} & := & \bigcup_{c>0} \mathbf{NSPACE}(n^c) & \text{non-deterministic polynomial space} \\ \mathbf{L} & := & \mathbf{DSPACE}(\log n) & \text{log space} \\ \mathbf{NL} & := & \mathbf{NSPACE}(\log n) & \text{non-deterministic log space} \end{array}$$

It may not be obvious what can be computed in space less than n. But for example one can multiply (or at least *verify* a multiplication) in logarithmic space.

Integer multiplication. For a bit string $a \in \{0,1\}^*$ let $N(a) \in \mathbb{Z}_{\geq 0}$ be the natural number with bit encoding a. Consider the language

$$\texttt{MULT} := \left\{(a,b,c) \mid a,b,c \in \{0,1\}^*, N(c) = N(a)\cdot N(b)\right\}$$

Then $\texttt{MULT} \in \mathbf{L}$. In fact, one can construct a Turing machine that runs the standard multiplication algorithm using pointers to positions in a, b, c while keeping track of the carry-over. This can all be done in space $O(\log n)$ where $n := |a| + |b| + |c|$.

Paths in directed graphs. Consider the following problem

$$\texttt{PATH} := \left\{(G,s,t) \mid \begin{array}{c} G=(V,E) \text{ is a directed graph with } s,t \in V \\ \text{containing an } s\text{-}t \text{ path} \end{array}\right\}$$

We claim that $\texttt{PATH} \in \mathbf{NL}$. Let $n := |V|$ be the number of vertices in the graph. A NTM can take a non-deterministic walk starting at s and guessing in each step the next edge to take while accepting if t is reached. The NTM maintains a counter and it stops the walk after n steps. This can be implemented in $O(\log n)$ space and there is an accepting computation path if and only if there is an s-t path. It is unknown whether $\texttt{PATH} \in \mathbf{L}$. However the variant for undirected graph is indeed in $\mathbf{L}$ which was proven in 2005 by Reingold.

4.2 Configuration graphs

We introduce the key concept when dealing with space bounded Turing machines:

Definition 4.4. Let $M = (\Gamma, Q, \delta)$ be a (either deterministic or non-deterministic) $S(n)$-space k-tape Turing machine and let $x \in \{0,1\}^*$. We define the *configuration graph* $G_{M,x} = (V,E)$ as the directed graph with vertices

$$V = \underbrace{\{x\} \times (\Gamma^k)^{S(|x|)}}_{\text{tape content}} \times \underbrace{\{1,\dots,|x|\} \times \{1,\dots,S(|x|)\}^k}_{\text{head pos.}} \times \underbrace{Q}_{\text{state}} \tag{4.1}$$

and edges

$$E := \left\{ (C_1, C_2) \in V \times V \mid \begin{array}{c} \text{TM } M \text{ can transition in one iteration} \\ \text{from } C_1 \text{ to } C_2 \text{ according to transition fct. } \delta \end{array} \right\}$$

The nodes in V are also called *configurations*.

In other words, each of the configurations gives a complete description of the state and "memory content" of the Turing machine in one iteration.

Note that we added x to the description of the nodes to be consistent with the textbook — though it is redundant. The graph $G_{M,x}$ contains a distinguished node

$$C_{\text{start}} = \left(x, \square^{kS(|x|)}, 1^{k+1}, q_{\text{start}}\right)$$

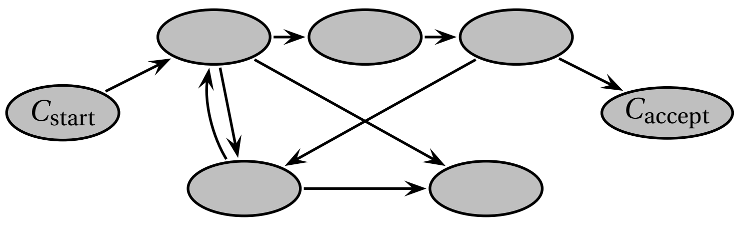
that corresponds to the first configuration of M on input x. Since we are interested only in space restrictions, we can make the convention that once a Turing machine is about to accept an input, it writes blanks on all used cells in the working tapes, moves all the heads to the left and moves finally into state q_{accept}. Then there is a unique configuration

$$C_{\text{accept}} = \left(x, \square^{kS(|x|)}, 1^{k+1}, q_{\text{accept}}\right)$$

that corresponds to accepting an input. If M is deterministic then the out-degree of every configuration is 1. If M is non-deterministic then the configurations in $G_{M,x}$ can have any out-degree in $\{0, \ldots, B\}$ where B is a constant depending on M. In both cases (M deterministic or not), for all $x \in \{0,1\}^*$ one has

$$M \text{ accepts } x \quad \Longleftrightarrow \quad \exists \text{ path from } C_{\text{start}} \text{ to } C_{\text{accept}}$$

Of course in the deterministic case, one can simply follow the single outgoing edge from C_{start} to determine that path.



graph $G_{M,x}$ for NTM M and $M(x) = 1$

Lemma 4.5. *Let M be a TM using $S(n)$-space. Then $G_{M,x} = (V, E)$ with $n := |x|$ satisfies the following*

(a) *$G_{M,x}$ has $|V| \leq n2^{O(S(n))}$ many vertices.*

(b) *For a vertex $C \in V$, let $[C] \in \{0,1\}^{O(S(n))}$ be the corresponding binary encoding. Then there is a CNF formula $\varphi_{M,x}$ of size $O(S(n) + \log(n))$ so that*

$$\forall C_1, C_2 \in V: \ \varphi_{M,x}([C_1],[C_2]) = 1 \Leftrightarrow (C_1, C_2) \in E$$

Proof. (a) follows from the definition of V in (4.1); note that $|\Gamma|, |Q|, k \leq O(1)$. We will not provide details, but (b) can be proven using arguments as in the Cook-Levin Theorem 2.10. □

We can use the configuration graphs to obtain a simple chain of inclusions:

Theorem 4.6. *For all space-constructive functions $S(n)$ one has*

$$\mathbf{NTIME}(S(n)) \subseteq \mathbf{DSPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$$

Proof. The first two inclusions are clear and we only prove the last one. Let $L \in \mathbf{NSPACE}(S(n))$ and let M be the corresponding $O(S(n))$-space bounded NTM. We construct the graph $G_{M,x}$ (which has size $2^{O(S(|x|))}$) and use standard graph algorithm to determine whether there is a path from C_{start} to C_{accept} which can be done in (deterministic) time $2^{O(S(|x|))}$. □

In particular this implies the following:

Corollary 4.7. $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{NPSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$

One can prove some separation analogous to the Time Hierarchy Theorems in Section 3.1.

Theorem 4.8. *Let f, g be space constructable with $f \leq o(g(n))$. Then*

$$\mathbf{DSPACE}(f(n)) \subset \mathbf{DSPACE}(g(n)) \quad \textit{and} \quad \mathbf{NSPACE}(f(n)) \subset \mathbf{NSPACE}(g(n))$$

The proof at least for $\mathbf{DSPACE}(f(n)) \subset \mathbf{DSPACE}(g(n))$ works again by simulating the $O(f(n))$-time machine and flipping the output. When counting space, a constant factor overhead suffices for the simulation. Theorem 4.8 also implies that $\mathbf{L} \neq \mathbf{PSPACE}$ and $\mathbf{NL} \neq \mathbf{NPSPACE}$. But the following conjectures are still open.

Conjecture 3. $\mathbf{P} \neq \mathbf{PSPACE}$.

Conjecture 4. $\mathbf{L} \neq \mathbf{NP}$.

4.3 PSPACE-completeness

Recall that **PSPACE** is the set of languages that can be decided in polynomial space. We can define a notion of completeness that is analogous to **NP**-completness (see Sec 2.2). Here we use the same Karp-reductions $\leq_p$.

Definition 4.9. A language A is **PSPACE**-*hard* if $B \leq_p A$ for all $B \in$ **PSPACE**. A language A is **PSPACE**-*complete* if $A \in$ **PSPACE** and A is **PSPACE**-hard.

The class **PSPACE** seems massive and it is not clear that there would be any natural **PSPACE**-complete language. But curiously there is. Let φ be any boolean formula depending on variables $x_1, \dots, x_n \in \{0,1\}$. We do *not* restrict φ to be a CNF or DNF.

Definition 4.10. A *quantified boolean formula (QBF)* is of the form

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \, \varphi(x_1, \dots, x_n)$$

where $Q_i \in \{\exists, \forall\}$ and φ is an arbitrary boolean formula with variables $x_1, \dots, x_n \in \{0,1\}$.

For example

$$\exists x_1 \exists x_2 \forall x_3 \, (\neg x_1 \wedge \neg x_2) \vee \neg(x_3 \wedge x_1)$$

is a QBF. Since each variable x_i appears exactly once with a quantifier in front, each QBF either evaluates to true or to false. We define the language of tautological QBFs

$$\texttt{TQBF} := \{\Phi : \Phi \text{ is a QBF that evaluates to true}\}$$

For example if φ is a CNF in variables $x_1, \dots, x_n$, then

$$\varphi \in \texttt{SAT} \quad \Longleftrightarrow \quad \exists x_1 \exists x_2 \dots \exists x_n \varphi \in \texttt{TQBF}$$

Hence $\texttt{SAT} \leq_p \texttt{TQBF}$ and TQBF is at least **NP**-hard. But in fact, one can prove the following stronger result:

Theorem 4.11. TQBF *is* **PSPACE**-*complete.*

Proof. First we prove that $\texttt{TQBF} \in$ **PSPACE**. Let Φ be the input QBF. We give a recursive algorithm to decide whether $\Phi \in \texttt{TQBF}$ that requires only linear space. Suppose for symmetry reasons that the first quantifier in Φ is an $\exists$ and so we write

$$\Phi = \exists x_1 \underbrace{Q_2 x_2 \dots Q_n x_n \, \varphi(x_1, \dots, x_n)}_{=:F(x_1)}$$

We recursively compute $F(0)$ and $F(1)$ sequentially, reusing the space, only keeping the two bits $F(0), F(1) \in \{0,1\}$. The terms $F(0)$ and $F(1)$ are again each a QBF with $n-1$ variables. Then we output $F(0) \vee F(1)$. This algorithm indeed takes space at most $O(|\Phi|)$.

Next, we prove that TQBF is **PSPACE**-hard. Consider a language $L \in$ **PSPACE** and let M be a Turing Machine that uses space $S(n)$ where $S(n)$ is a polynomial in n. For an input x with $n := |x|$, we consider the configuration graph $G_{M,x}$. That graph has size $2^{O(S(n))}$ which does not seem to be helpful. But $G_{M,x}$ is actually very structured and we know that (as $S(n) \geq \log(n)$) there is an $O(S(n))$-size CNF $\varphi_{M,x}$ that encodes all the edges, i.e.

$$\forall C, D \in V: \ \varphi_{M,x}(C,D) = 1 \Leftrightarrow (C,D) \in E$$

We will use this fact to recursively construct a more general formula ψ_i for $i \geq 0$, satisfying

$$\psi_i(C,D) = 1 \Leftrightarrow \exists \text{ path of length } \leq 2^i \text{ in } G_{M,x} \text{ from } C \text{ to } D$$

The base case is

$$\psi_0(C,D) := \underbrace{(C = D)}_{\text{dist. } 0} \vee \underbrace{\varphi_{M,x}(C,D)}_{\text{dist. } 1}$$

For the recursive step, we observe that there is a path of length at most 2^i from C to D if and only if there is some configuration F "in the middle" so that it takes at most 2^{i-1} steps from C to F and at most 2^{i-1} steps from F to D. Then the natural choice would be to set

$$\psi_i(C,D) := \exists F: \big(\psi_{i-1}(C,F) \wedge \psi_{i-1}(F,D)\big) \tag{4.2}$$

for $i \geq 1$. There is a problem however: using this definition, the size of the constructed formula would double and hence grow proportional to 2^i. A more efficient way is to define instead

$$\psi_i(C,D) := \exists F \forall C' \forall C'' \Big(\big(((C',C'') = (C,F)) \vee ((C',C'') = (F,D))\big) \Rightarrow \psi_{i-1}(C',C'')\Big) \tag{4.3}$$

We encourage the reader to verify that (4.2) and (4.3) are indeed equivalent. Next, we analyze the size of the constructed QBF. We note that $|\psi_0| \leq O(S(n))$ and the recursion gives $|\psi_i| \leq |\psi_{i-1}| + O(S(n))$ which resolves to $|\psi_i| \leq O((i+1) \cdot S(n))$ for all $i \geq 0$. Finally, the Turing machine M takes at most $2^{O(S(n))}$ iterations on input x, hence setting $i := O(S(n))$ gives

$$x \in L \Leftrightarrow \psi_{O(S(n))} \in \texttt{TQBF}$$

The final QBF has size $|\psi_{O(S(n))}| \leq O(S(n)^2)$ which concludes the claim. □

We note that the argument to prove Theorem 4.11 did not actually use the that Turing machine M is deterministic and would have worked as well for a non-deterministic one. That means `TQBF` is actually **NPSPACE**-complete and so **PSPACE** = **NPSPACE**. We can refine this claim as follows[3]:

Theorem 4.12 (Savich 1970)**.** *For any space-constructable function*[4] $S(n)$ *one has*

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

Proof. Let $L \in \mathbf{NSPACE}(S(n))$ and let M be the non-deterministic Turing machine deciding L in $O(S(n))$ space. Fix an input $x \in \{0,1\}^*$. As we have seen in the proof of Theorem 4.11, there is an efficiently constructable QBF ψ of size $O(S(|x|)^2)$ so that $x \in L$ if and only if $\psi \in$ `TQBF`. We also have seen in Theorem 4.11 that one can decide in space $O(|\psi|)$ whether $\psi \in$ `TQBF`. That gives the claim. □

We want to comment on the fact that we allowed QBFs to be arbitrary boolean formulas rather than in standard form of DNF or CNF. And in fact, there are for example $O(n)$-size DNFs in n boolean variables for which every equivalent CNF has size $2^{\Omega(n)}$. So one cannot easily convert boolean formulas — as long as one keeps the set of variables fixed. Now consider a QBF of the form

$$\psi = Q_1 x_1 Q_2 x_2 \dots Q_n x_n \, \varphi(x_1,\dots,x_n)$$

Then there is a polynomial time Turing machine that on input of φ and values $x_1,\dots,x_n$, evaluates $\varphi(x_1,\dots,x_n)$. Then by the Cook-Levin Theorem there is a CNF ϕ' of size at most poly($|\varphi|$) containing the old variables x as well as new variables y so that for all $x \in \{0,1\}^n$ one has

$$\varphi(x) = \exists y \in \{0,1\}^{\text{poly}(|\varphi|)} : \phi'(x,y)$$

That means one could actually transform any QBF in polynomial time into an equivalent one where the boolean formula is indeed a CNF. But this comes at the cost of additional variables.

4.4 More on logarithmic space

Next, we want to introduce a notion of completeness for **NL**. Note that **NL** $\subseteq$ **P** and likely this inclusion is strict. We cannot use a reduction like the Karp reduction $\leq_p$ that allows arbitrary polynomial time computation in the reduction

[3]We make it look like that Savitch's Theorem is just a trivial observation. But the reader may note that the result Savich preceeds the one of **PSPACE**-hardness for `TQBF`.

[4]With some extra work one can drop the assumption that the function $S(n)$ is space-constructable but we skip this here.

because then the reduction itself is (likely) more powerful than the class **NL** itself. In order to draw meaningful conclusions from the existence of a reduction we should restrict it to logarithmic space as well.

Definition 4.13. For languages $A, B \subseteq \{0,1\}^*$ we write $A \leq_\ell B$ (we say A is *logspace reducible* to B) if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ that is computable in space $O(\log n)$ so that

$$x \in A \Leftrightarrow f(x) \in B \quad \forall x \in \{0,1\}^*$$

In the definition, the Turing machine computing f has a read-only input tape containing x, a write-only output tape for writing $f(x)$ and $k = O(1)$ working tapes with space $O(\log n)$.

Lemma 4.14. *For languages $A, B, C \subseteq \{0,1\}^*$ one has*

(a) If $A \leq_\ell B$ and $B \leq_\ell C$ then $A \leq_\ell C$
(b) If $A \leq_\ell B$ and $B \in \mathbf{L}$ then $A \in \mathbf{L}$
(c) If $A \leq_\ell B$ and $B \in \mathbf{NL}$ then $A \in \mathbf{NL}$

Proof. We prove only (a); (b) and (c) use similar arguments. Let f be the function in the reduction $A \leq_\ell B$ computed by Turing machine M_f and let g be the function in the reduction $B \leq_\ell C$ computed by M_g. Now let x be an input for A. The obvious way to compute $g(f(x))$ would be to first compute $f(x)$ using M_f and then compute $g(f(x))$ simulating M_g. But the problem is that we cannot store the output $f(x)$ since we only have logarithmic space available. But we can modify M_f so that on input of (x, i) it computes the bit $f(x)_i$ in $O(\log n)$ space. Now we simulate M_g on input $f(x)$ so that each time a bit $f(x)_i$ is being requested we run the simulation of M_f. □

Now we make the natural definition using the log space reduction:

Definition 4.15. A language $A \subseteq \{0,1\}^*$ is **NL**-*hard* if $B \leq_\ell A$ for all $B \in \mathbf{NL}$. Moreover A is **NL**-*complete* if $A \in \mathbf{NL}$ and A is **NL**-hard.

Next, it would be nice to have a natural problem that is **NL**-complete. And in fact, `PATH` has that property. Recall that `PATH` contains all the tuples (G, s, t) where G is a directed graph that contains an s-t path.

Theorem 4.16. `PATH` *is* **NL**-*complete.*

Proof. We have already argued that `PATH` $\in$ **NL**, so it remains to show **NL**-hardness. Let $L \in \mathbf{NL}$ and let M be a $O(\log n)$-space NTM that decides L. Fix an input $x \in \{0,1\}^*$ and consider the configuration graph $G_{M,x} = (V, E)$. Since M uses only

$O(\log n)$ space, the configurations are represented by $O(\log n)$ bits as well. Then the reduction is simply $f(x) = (G_{M,x}, C_{\text{start}}, C_{\text{accept}})$. Here the $G_{M,x}$ is written on the write-once output tape where it is crucial that given two configurations C and D we can check in $O(\log n)$-space whether $(C, D) \in E$. □

Revisiting the Cook-Levin Theorem (Theorem 2.10) we can see that the construction of the SAT formula can also be done in logarithmic space. Hence in fact, `SAT` is also **NP**-complete under log-space reduction.

4.5 NL = coNL

In the time complexity world, we conjecture that **NP** ≠ **coNP** as there does not seem to be a time-efficient way to certify that say a SAT formula is not satisfiable. Oddly, in the space-constrained world the classes that we consider behave very differently. First we want to give an alternative definition for **NL** that corresponds to the alternative **NP**-definition using the notion of a verifier.

Definition 4.17 (Alternative definition of **NL**)**.** A language $L \subseteq \{0,1\}^*$ is in **NL** if there is a deterministic Turing machine M and a polynomial p so that

$$L = \left\{x \in \{0,1\}^* \mid \exists u \in \{0,1\}^{p(|x|)} : M(x,u) = 1\right\}$$

Here M has

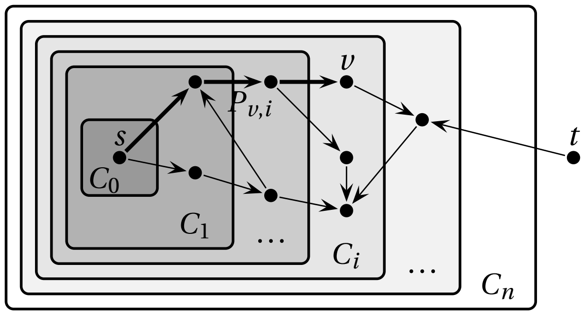
- a read-only input tape containing x
- $k = O(1)$ working tapes with $O(\log n)$ cells each
- a read-only <u>read-once</u> tape containing u

Here a read-once tape means that the head only has the options to either stay put or move to the right. It cannot move back to the left. Intuitively this models that a Turing machine corresponding to **NL** can make polynomially many non-deterministic choices but it doesn't have the space to remember the choices that it made in the past. We leave it to the reader to formally verify that definitions of **NL** in Def 4.3 and Def 4.17 are indeed equivalent. We also note that dropping the assumption of a read-once tape for the certificate we would actually recover the much more powerful class of **NP**.

We define **coNL** $:= \{\bar{L} \mid L \in$ **NL**$\}$. Then we prove the following result, which apparently was a surprise to researchers when it was first discovered:

Theorem 4.18 (Immerman-Szelepcsényi 1987)**. NL = coNL***.*

Proof. We know that PATH is **NL**-complete, hence it remains to prove that $\overline{\texttt{PATH}} \in$ **NL**. It suffices to design a deterministic $O(\log n)$-space Turing machine M so that for each directed graph $G = (V, E)$ and two vertices s, t there is a read-once certificate $u \in \{0,1\}^{p(n)}$ with $M((G, s, t), u) = 1$ if and only if t is *not* reachable from s. In other words, we need to design a certificate for the fact that there is no s-t path that we could check in $O(\log n)$-space. Note that there is actually a very simple certificate that certifies that s and t are disconnected: simply pick a set $S \subseteq V$ with $s \in S$, $t \notin S$ and $(u, v) \notin E$ for all $u \in S$, $v \notin S$. The problem is that this natural certificate is not verifiable in a read-once fashion. So we design a more complicated read-once certificate. The intuition is that the certificate will describe the sets $C_0, \ldots, C_n$ with $C_i = \{v \in V \mid v \text{ reachable from } s \text{ in } \leq i \text{ steps}\}$.



More formally, the certificate will look as follows (where the order is important as the verifer cannot remember more than $O(\log n)$ bits and can read the certificate only once). We assume that the vertices of $G = (V, E)$ are $V = \{1, \ldots, n\}$.

> **Certificate.** Define $c_0 := 1$. For $i = 1, \ldots, n$ the certificate contains a number $c_i \in \mathbb{N}$ and pairs $(u_k^{(i)}, P_k^{(i)})_{k=1,\ldots,c_i}$ of vertices with their certificate of inclusion in C_i. Moreover we include pairs $(v_k^{(i)}, Q_k^{(i)})_{k=1,\ldots,n-c_i}$ of nodes with their certificates of exclusion from C_i.

Here the truthful choice would be that $c_i = |C_i|$, $C_i = \{u_1^{(i)}, \ldots, u_{c_i}^{(i}\}$ and $[n] \setminus C_i = \{v_1^{(i)}, \ldots, v_{n-c_i}^{(i)}\}$ where the nodes are expected in increasing order. Each $P_k^{(i)}$ is expected to be an s-$u_k^{(i)}$ path and $Q_k^{(i)}$ is expected to be c_{i-1} many paths from s to all the nodes in C_{i-1} in increasing order of their end nodes. The verifier works as follows:

(1) FOR $i = 1$ TO n

  (2) Verify that $u_1^{(i)} < \ldots < u_{c_i}^{(i)}$ and that $P_k^{(i)}$ is an s-$u_k^{(i)}$ path in G of length at most i. If $i = n$ then verify that t is not in the list.

  (3) Verify that $v_1^{(i)} < \ldots < v_{n-c_i}^{(i)}$. Fo each k verify that $Q_k^{(i)}$ is a set of c_{i-1} many paths of length at most $i - 1$ from s to some nodes $w_1 < \ldots < w_{c_{i-1}}$. Also verify that $(w_\ell, v_k^{(i)}) \notin E$ for $\ell = 1, \ldots, c_{i-1}$.

(4) Accept if all verifications are successful.

It is important to keep in mind that in iteration i the algorithm only needs to remember the numbers c_i and c_{i-1} and a constant number of nodes at a time. If indeed there is no s-t path then as certificate one may simply pick the truthful choice described above. The tricky part is to prove the following:

Claim. *Let $G = (V, E)$ be a graph with $s, t \in V$ that contains an s-t path. Then the verifier will reject any certificate.*

Proof of claim. We prove the contrapositive, i.e. assuming the verifier accepts, we show that there is no s-t path. Consider an arbitrary certificate $(u_k^{(i)}, P_k^{(i)})_{k=1,\dots,c_i}$ and $(v_k^{(i)}, Q_k^{(i)})_{k=1,\dots,n-c_i}$ for $i = 1,\dots,n$ that the verifier accepts. We denote $C_i^* := \{v \in V \mid v \text{ reachable from } s \text{ in } \leq i \text{ steps}\}$ as the actual set of vertices reachable in i steps. We keep in mind that the verifier does not know C_i^*. We set $C_i := \{u_1^{(i)},\dots,u_{c_i}^{(i)}\}$ as the set that the certificate claims to be C_i^* and let $\bar{C}_i = \{v_1^{(i)},\dots,v_{n-c_i}^{(i)}\}$ be the set that the certificate claims to be the complement of C_i^*. Here we set $C_0 := \{s\}$ for convenience. By induction, for $i \in \{0,\dots,n+1\}$ suppose the certificate has been truthful before iteration i, i.e. $C_j = C_j^*$ for all $j < i$. If $i = n+1$ then the verifier only accepted because t does not appear in the final list and so there is no s-t path. So suppose that $0 \leq i \leq n$. Now consider the verifier in iteration i. Due to the order of the nodes, the verifier can check that the list defining C_i contains indeed c_i many different nodes and for each node $u_k^{(i)}$ it can verify whether $P_k^{(i)}$ indeed is a path of length at most i from s to $u_k^{(i)}$. From that we know that $|C_i| = c_i$ and $C_i \subseteq C_i^*$. Now consider what the verifier does in (3). Again due to the order of the nodes we know that indeed $|\bar{C}_i| = n - c_i$. For each k the verifier can also verify whether $Q_k^{(i)}$ indeed represents c_{i-1} paths of length at most $i-1$ to different nodes $w_1 < \dots < w_{c_{i-1}}$ that have no edge going to $v_k^{(i)}$. In particular $\{w_1,\dots,w_{c_{i-1}}\} \subseteq C_{i-1}^*$. By choice of the index i, the number c_{i-1} is indeed correct, i.e. $c_{i-1} = |C_{i-1}| = |C_{i-1}^*|$. From that we know that indeed $\{w_1,\dots,w_{c_{i-1}}\} = C_{i-1}^*$. That means we have verified that there is no edge going from C_{i-1}^* to $v_k^{(i)}$ which means that in fact $v_k^{(i)} \notin C_i^*$. Hence $\bar{C}_i \subseteq V \setminus C_i^*$. Since $|C_i| + |\bar{C}_i| = n$ we know that $C_i = C_i^*$ and $\bar{C}_i = V \setminus C_i^*$. That concludes the claim. $\square$

Chapter 5

The polynomial hierarchy

A relevant problem in logic is the one of minimizing the size of boolean formulas. This can be formalized as the language

$$\texttt{MINDNF} = \left\{(\varphi, k) \mid \varphi \text{ is a DNF that has an equivalent DNF of size } \leq k\right\}$$

This problem does not seem to obviously fall in either **NP** or **coNP**. But one can rewrite

$$\texttt{MINDNF} = \left\{(\varphi, k) \mid \exists\varphi'\forall x : \varphi, \varphi' \text{ are DNFs and } |\varphi'| \leq k \text{ and } \varphi(x) = \varphi'(x)\right\}$$

which means that one can express `MINDNF` efficiently with two quantifiers. That motivates the following the following definition:

Definition 5.1. A language $L \subseteq \{0,1\}^*$ is in Σ_k^P if there are polynomials $p_1, \ldots, p_k$ and a polynomial time Turing machine M so that

$$L = \left\{x \in \{0,1\}^* \mid \exists u_1 \in \{0,1\}^{p_1(|x|)} \forall u_2 \in \{0,1\}^{p_2(|x|)} \ldots Q_k u_k \in \{0,1\}^{p_k(|x|)} : M(x, u_1, \ldots, u_k) = 1\right\}$$

where $Q_k = \exists$ if k is odd and $Q_k = \forall$ if k is even.

Moreover L is in Π_k^P if there are $p_1, \ldots, p_k$ and M so that

$$L = \left\{x \in \{0,1\}^* \mid \forall u_1 \in \{0,1\}^{p_1(|x|)} \exists u_2 \in \{0,1\}^{p_2(|x|)} \ldots Q_k u_k \in \{0,1\}^{p_k(|x|)} : M(x, u_1, \ldots, u_k) = 1\right\}$$

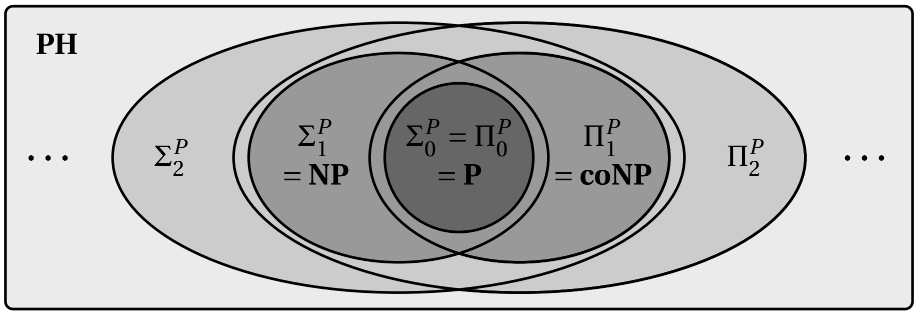
where $Q_k = \forall$ if k is odd and $Q_k = \exists$ if k is even.

Note that $\Pi_k^P = \{\bar{L} \mid L \in \Sigma_k^P\}$ by definition. We also observe that $\mathbf{P} = \Sigma_0^P = \Pi_0^P$ and $\mathbf{NP} = \Sigma_1^P$ while $\mathbf{coNP} = \Pi_1^P$. We can also see that `MINDNF` $\in \Sigma_2^P$. The following is also clear from the definition:

Lemma 5.2. *For all $k \geq 0$, $\Sigma_k^P \subseteq \Pi_{k+1}^P$ and $\Pi_k^P \subseteq \Sigma_{k+1}^P$.*

It will also be interesting to consider the class of problems that can be expressed with any (unspecified) constant number of alternating quantifiers:

Definition 5.3. We define the *polynomial hierarchy* as $\mathbf{PH} := \bigcup_{k\geq 0} \Sigma_k^P = \bigcup_{k\geq 0} \Pi_k^P$.


PH
...
Σ_2^P
Σ_1^P $= \mathbf{NP}$
$\Sigma_0^P = \Pi_0^P$ $= \mathbf{P}$
Π_1^P $= \mathbf{coNP}$
Π_2^P
...

As usually it will be interesting to understand complete problems for these classes.

Definition 5.4. A language $A \in \{0,1\}^*$ is Σ_k^P*-complete* if $A \in \Sigma_k^P$ and $B \leq_p A$ for all $B \in \Sigma_k^P$.

We define Π_K^P and **PH**-completeness analogously. It might not come totally surprising that we consider the natural restrictions of `TQBF` to k alternations of different quantifiers.

Definition 5.5. Define Σ_k`SAT` be the set of quantified boolean formulas of the form

$$\exists x_1 \forall x_2 \ldots Q_k x_k \varphi(x_1, \ldots, x_k)$$

that evaluate to true. Here each x_i is a vector of boolean variables. Similarly Π_k`SAT` is the set of quantified boolean formulas of the form

$$\forall x_1 \exists x_2 \ldots Q_k \varphi(x_1, \ldots, x_k)$$

Using insight from the Cook-Levin Theorem one can prove:

Theorem 5.6. *Σ_k`SAT` is Σ_k^P-complete and Π_k`SAT` is Π_k^P-complete.*

We skip the argument here. One can also show that `MINDNF` is Σ_2^P-complete though the argument is more involved. It is conjectured that $\Sigma_k^P \neq \Pi_k^P$ for all $k \geq 1$ and $\Sigma_k^P \neq \Sigma_{k+1}^P$ for all $k \geq 0$. Finally we can show that the class **PH** likely does not have complete problems.

Theorem 5.7. *The following holds:*

(a) *For all* $k \geq 1$ *one has:* $(\Sigma_k^P = \Pi_k^P) \Rightarrow (\mathbf{PH} = \Sigma_k^P)$

(b) $(\mathbf{P} = \mathbf{NP}) \Rightarrow (\mathbf{P} = \mathbf{PH})$

If (a) happens for a value of k, then we say that the polynomial hierarchy *collapses to level k*.

Proof. We only show (b), (a) works similarly. To be more precise we prove:
Claim. $\mathbf{P} = \mathbf{NP} \Rightarrow \forall k \geq 0 : \Sigma_k^P = \Sigma_{k+1}^P$.
Proof of Claim. Let $L \in \Sigma_{k+1}^P$. First consider the case that $k+1$ is even. Then

$$\begin{aligned} L &= \left\{x \in \{0,1\}^* \mid \exists u_1 \in \{0,1\}^{p_1(|x|)} \ldots \forall u_{k+1} \in \{0,1\}^{p_{k+1}(|x|)} : V(x, u_1, \ldots, u_{k+1}) = 1\right\} \\ &= \left\{x \in \{0,1\}^* \mid \exists u_1 \in \{0,1\}^{p_1(|x|)} \ldots \neg \underbrace{\exists u_{k+1} \in \{0,1\}^{p_{k+1}(|x|)} V(x, u_1, \ldots, u_{k+1}) = 0}_{(*)}\right\} \end{aligned}$$

where $p_1, \ldots, p_{k+1}$ are polynomials and V is a deterministic polynomial-time Turing machine. By assumption, the computation in $(*)$ is in $\mathbf{NP} = \mathbf{P}$ and hence can be replaced by a deterministic polynomial time Turing machine U so that

$$L = \left\{x \in \{0,1\}^* \mid \exists u_1 \in \{0,1\}^{p_1(|x|)} \ldots \exists u_k \in \{0,1\}^{p_k(|x|)} : U(x, u_1, \ldots, u_k) = 1\right\}.$$

That means $L \in \Sigma_k^P$. The claim with $k+1$ odd is similar (easier in fact as the negation is not needed). $\square$

The levels on the polynomial hierarchy can also be expressed using oracles. Recall that for a language $O \subseteq \{0,1\}^*$ we denote $\mathbf{NP}^O$ as the set of languages that can be computed using a non-deterministic polynomial time Turing machine that has access to an oracle for O.

Theorem 5.8. *The following holds*

(a) $\Sigma_2^P = \mathbf{NP}^{\mathtt{SAT}}$

(b) $\Pi_2^P = \mathbf{coNP}^{\mathtt{SAT}}$

(c) $\Sigma_{k+1}^P = \mathbf{NP}^{\Sigma_k \mathtt{SAT}}$

Proof. We prove (a); (b) and (c) are similar.
Claim I. $\Sigma_2^P \subseteq \mathbf{NP}^{\mathtt{SAT}}$.
Proof of Claim I. Let $L \in \Sigma_2^P$. Then we can write

$$\begin{aligned} L &= \left\{x \in \{0,1\}^* \mid \exists u_1 \in \{0,1\}^{p_1(|x|)} \forall u_2 \in \{0,1\}^{p_2(|x|)} : V(x, u_1, u_2) = 1\right\} \\ &= \left\{x \in \{0,1\}^* \mid \exists u_1 \in \{0,1\}^{p_1(|x|)} \neg \exists u_2 \in \{0,1\}^{p_2(|x|)} : V(x, u_1, u_2) = 0\right\} \end{aligned}$$

Then an $\mathbf{NP}^{\mathtt{SAT}}$ algorithm on input x works as follows: guess u_1, then use the Cook-Levin Theorem to compute a SAT formula ψ in polynomial time so that $\psi(x, u_1, u_2) = 1 \Leftrightarrow V(x, u_1, u_2) = 0$. Then we call the oracle to test whether $\psi \in \mathtt{SAT}$ and flip the outcome. □

Claim II. $\mathbf{NP}^{\mathtt{SAT}} \subseteq \Sigma_2^P$.

Proof of Claim II. Let $L \in \mathbf{NP}^{\mathtt{SAT}}$. Let M be the non-deterministic Turing machine with oracle access to `SAT` that decides L in polynomial running time $T(n)$. Consider an input $x \in \{0,1\}^*$. Then M guesses some computation path P of length at most $T(n)$ on which it queries a set $(\psi_i)_{i \in I}$ of SAT instances. Let $I_1 \subseteq I$ be the satisfiable ones and let $I_0 \subseteq I$ be the unsatisfiable ones. That means we can write

$$
\begin{aligned}
x \in L \quad \Leftrightarrow \quad & \exists \text{computation path } P \text{, SAT formulas } (\psi_i)_{i \in I}, \text{ partition } I = I_0 \dot\cup I_1, \\
& \text{assignments } x^{(i)} \text{ for } \psi_i \text{ with } i \in I_1: \\
& \forall \text{assignments } y^{(i)} \text{ for } \psi_i \text{ with } i \in I_0: \\
& P \text{ is indeed an accepting computation path quering the SAT formulas} \\
& (\psi_i)_{i \in I},\ x^{(i)} \text{ is satisfying for } \psi_i \text{ for all } i \in I_1, y^{(i)} \text{ is not satisfying for} \\
& \psi_i \text{ for all } i \in I_0
\end{aligned}
$$

This formulation is indeed in Σ_2^P.

□

5.1 Time space tradeoffs for SAT

At the time of this writing we cannot rule out that $\mathtt{SAT} \in \mathbf{DTIME}(n)$ and we cannot rule out that $\mathtt{SAT} \in \mathbf{DSPACE}(\log n)$. But curiously we can prove that there is no algorithm that satisfies both bounds.

Definition 5.9. For functions $S, T : \mathbb{N} \to \mathbb{N}$ we define $\mathbf{DTIME\text{-}SPACE}(T(n), S(n))$ as the set of languages $L \subseteq \{0,1\}^*$ that are decided by a deterministic Turing machine M that on any input x takes time $O(T(|x|))$ and space $O(S(|x|))$.

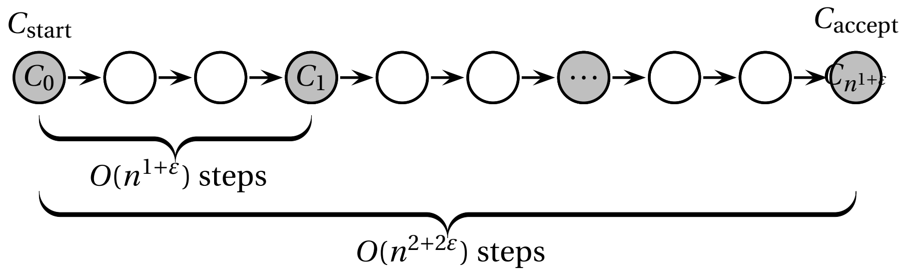
First we prove a more general result:

Theorem 5.10. *For* $0 < \varepsilon \leq \frac{1}{8}$ *one has* $\mathbf{NTIME}(n) \not\subseteq \mathbf{DTIME\text{-}SPACE}(n^{1+\varepsilon}, n^{\varepsilon})$.

Proof. For the sake of contradiction, we assume that $\mathbf{NTIME}(n) \subseteq \mathbf{DTIME\text{-}SPACE}(n^{1+\varepsilon}, n^{\varepsilon})$ for a small enough constant $\varepsilon > 0$ that we determine later. By the usual padding argument we then know that for any function $g(n)$ that is time and space-constructable one has

$$\mathbf{NTIME}(g(n)) \subseteq \mathbf{DTIME\text{-}SPACE}(g(n)^{1+\varepsilon}, g(n)^{\varepsilon}) \tag{5.1}$$

We will prove the following which then gives a contradiction to the Nondeterministic Time Hierarchy Theorem (Theorem 3.4) because $(1+3\varepsilon)\cdot(1+\varepsilon)^2 < 2$ for $\varepsilon \le \frac{1}{8}$.
Claim. *Assuming* (5.1) *one has* $\mathbf{NTIME}(n^2) \subseteq \mathbf{NTIME}(n^{(1+3\varepsilon)(1+\varepsilon)^2})$.
Proof of Claim. Let $L \in \mathbf{NTIME}(n^2)$. Applying (5.1) with $g(n) = n^2$ we know that $\mathbf{NTIME}(n^2) \subseteq \mathbf{DTIME\text{-}SPACE}(n^{2+2\varepsilon}, n^{2\varepsilon})$. Hence there is a deterministic Turing machine M deciding L in time $n^{2+2\varepsilon}$ on space $n^{2\varepsilon}$. Let $x \in \{0,1\}^*$ be an input with $n := |x|$ for which we have to decide whether $x \in L$. Consider the configuration graph $G_{M,x} = (V,E)$ (see Sec 4.2). We know that all configurations can be encoded with $O(n^{2\varepsilon})$ bits and if $x \in L$, then there is a path of length $n^{2+2\varepsilon}$ from C_{start} to C_{accept}. On the shortest computation path, pick a configuration every $O(n^{1+\varepsilon})$ time steps and denote them with $C_0, \dots, C_{n^{1+\varepsilon}}$.



There is a Turing machine U that computes the function

$$U(x,C,D) = \begin{cases} 1 & \text{if there is a path in } G_{M,x} \text{ from } C \text{ to } D \text{ in } O(n^{1+\varepsilon}) \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

for configurations $C, D \in V$. Since the number of steps is bounded by $O(n^{1+\varepsilon})$ and the encoding length of C and D is $O(n^{2\varepsilon})$ we can bound the non-deterministic running time of U by $O(n^{1+\varepsilon} \cdot n^{2\varepsilon})$. Applying (5.1) with $g(n) = n^{1+3\varepsilon}$ we know in particular that $\mathbf{NTIME}(n^{1+3\varepsilon}) \subseteq \mathbf{DTIME}(n^{(1+3\varepsilon)(1+\varepsilon)})$. That means we may assume that U is a deterministic Turing machine that takes time $O(n^{(1+3\varepsilon)(1+\varepsilon)})$. Next, we write

$$\begin{aligned} x \in L \quad &\Leftrightarrow \quad \exists C_1, \dots, C_{n^{1+\varepsilon}} \in V : \forall i \in [n^{1+\varepsilon}] : U(x, C_{i-1}, C_i) = 1 \\ &\Leftrightarrow \quad \exists C_1, \dots, C_{n^{1+\varepsilon}} \in V : \neg \underbrace{\exists i \in [n^{1+\varepsilon}] : U(x, C_{i-1}, C_i) = 0}_{(*)} \\ &\Leftrightarrow \quad \exists C_1, \dots, C_{n^{1+\varepsilon}} \in V : \neg W(x, C_1, \dots, C_{n^{1+\varepsilon}}) = 1 \end{aligned}$$

We can we phrase the computation in $(*)$ as a non-deterministic Turing machine W that takes input $x, C_1, \dots, C_{n^{1+\varepsilon}}$, guesses the index $i \in [n^{1+\varepsilon}]$ and runs the deterministic Turing machine U that takes time $O(n^{(1+3\varepsilon)(1+\varepsilon)})$. Overall the non-deterministic time is dominated by $O(n^{(1+3\varepsilon)(1+\varepsilon)})$. Applying (5.1) there is a deterministic version of W that we denote by W_{det} that runs in time $O(n^{(1+3\varepsilon)(1+\varepsilon)^2})$.

Then

$$x \in L \Leftrightarrow \underbrace{\exists C_1, \ldots, C_{n^{1+\varepsilon}} \in V : W_{\text{det}}(x, C_1, \ldots, C_{n^{1+\varepsilon}}) = 0}_{(**)}$$

Then $(**)$ is a non-deterministic computation that is in $\mathbf{NTIME}(n^{(1+3\varepsilon)(1+\varepsilon)^2})$. That finishes the claim. $\square$

Corollary 5.11. *For any* $0 < \delta < \frac{1}{8}$ *one has* $\mathtt{SAT} \notin \mathbf{DTIME}\text{-}\mathbf{SPACE}(n^{1+\delta}, n^{\delta})$*.*

Proof sketch. Suppose the claim is false. Let $L \in \mathbf{NTIME}(n)$ and fix an input $x \in \{0,1\}^*$ for L with $n := |x|$. Then by revisiting the Cook-Levin Theorem one can show that in time $n \log^{O(1)} n$ and space $\log^{O(1)} n$ one can construct a SAT fomula ψ of size $O(n \log n)$ with $x \in L \Leftrightarrow \psi \in \mathtt{SAT}$. Then one can decide whether $\psi \in \mathtt{SAT}$ in time $O((n \log n)^{1+\delta})$ and space $O((n \log n)^{\delta})$. That is a contradiction to Theorem 5.10 for $\delta < \frac{1}{8}$. $\square$

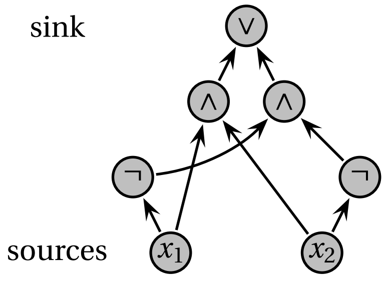
Chapter 6

Boolean circuits

6.1 Introduction to boolean circuits

In this chapter we want to discuss boolean circuits.

Definition 6.1. A *boolean circuit* C is a directed acyclic graph $G = (V, E)$ with n sources labelled by $x_1, \dots, x_n$ and a distinguished *sink*. All non-source nodes are labelled with one of the operations $\vee, \wedge, \neg$. Nodes labelled with $\vee$ or $\wedge$ have in-degree (also called *fan-in*) of 2 and nodes labeled $\neg$ have fan-in 1. The *size* $|C|$ of the boolean circuit is the number of vertices. On input of $x \in \{0,1\}^n$ the circuit computes a boolean value of $C(x) \in \{0,1\}$ at the sink node in the natural way.



Circuit C computing the function $C(x_1, x_2) = (x_1 = x_2)$

Naturally we are interested in the size of a circuit needed to compute certain boolean functions.

Theorem 6.2. *Any boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O(2^n/n)$.*

Proof. We prove a $O(2^n)$ bound and leave the slightly better bound for the homework. We can write a function f as

$$f(x) = (x_1 \wedge f(1, x_2, \dots, x_n)) \vee (\neg x_1 \wedge f(0, x_2, \dots, x_n))$$

Then recursively replace the function $f(1, x_2, \dots, x_n)$ and $f(0, x_2, \dots, x_n)$ that both have $n-1$ variables by circuits. If s_n is the circuit size that this recursive approach gives for n variables, then one can see that $s_n \leq 2s_{n-1} + O(1)$ and $s_1 = O(1)$. This can be resolved to $s_n \leq O(2^n)$. □

One can also prove a surprisingly tight lower bound.

Theorem 6.3. *For every n there is a function $f : \{0,1\}^n \to \{0,1\}$ for which all circuits have size at least $\Omega(2^n/n)$.*

Proof. We will use a counting argument. Fix a number $s \geq n$ so that every function $f : \{0,1\}^n \to \{0,1\}$ has indeed a circuit of size at most s. We discuss how to encode a circuit C where $|C| \leq s$. We number the nodes by $1, \dots$ (at most) s. For each gate we use $O(\log n)$ bits to encode the type ($\vee$, $\wedge$ or $\neg$ or $x_1, \dots, x_n$) and $O(\log(s))$ bits to encode the at most 2 predecessors. For the whole circuit we will need at most $Cs\log(s)$ bits for some constant $C > 0$ as $s \geq n$. Since there are many 2^{2^n} functions of the form $f : \{0,1\}^n \to \{0,1\}$ we have

$$2^{Cs\log(s)} > 2^{2^n} \quad \Rightarrow \quad Cs\log(s) > 2^n \quad \Rightarrow \quad s \geq \Omega(2^n/n)$$

□

One can rephrase the proof of Theorem 6.3 to also give that *most* functions $f : \{0,1\}^n \to \{0,1\}$ need circuits of size $\Omega(2^n/n)$. On the downside, Theorem 6.3 does not provide a lower bound for any natural function.

6.2 Circuits and complexity classes

We can also introduce complexity classes dealing with circuits:

Definition 6.4. For a function $T : \mathbb{N} \to \mathbb{N}$ we define

$$\textbf{SIZE}(T(n)) := \left\{ f : \{0,1\}^* \to \{0,1\} \mid \begin{array}{c} f \text{ can be computed by a family of} \\ \text{circuits } (C_n)_{n\in\mathbb{N}} \text{ of size } |C_n| \leq T(n) \end{array} \right\}$$

Note that one circuit only works for one input size n. So if we say that a function $f : \{0,1\}^* \to \{0,1\}$ is computed by a family $(C_n)_{n\in\mathbb{N}}$ of circuits then we mean that $f(x) = C_n(x)$ for all $n \geq 0$ and all $x \in \{0,1\}^n$. Then the class of functions with polynomial size circuits is the following:

Definition 6.5. $\mathbf{P/poly} := \bigcup_{c>0} \mathbf{SIZE}(cn^c)$.

We will later explain the reason for the name **P/poly**.

Theorem 6.6. *For any time constructable function $T(n)$ one has $\mathbf{DTIME}(T(n)) \subseteq \mathbf{SIZE}(O(T(n) \cdot \log(n)))$.*

The proof works by applying the argument from the Cook-Levin Theorem to a deterministic Turing machine. The logarithmic blowup comes from making the Turing machine oblivious. Then an oblivious Turing machine with running time $T'(n)$ can be turned into a circuit of size $O(T'(n))$. We will not give details here. Theorem 6.6 implies the following:

Corollary 6.7. $\mathbf{P} \subseteq \mathbf{P/poly}$.

There is also a rather simple hierarchy theorem for circuits:

Theorem 6.8. *There is a universal constant $C > 0$ so that for all $S(n) \leq o(2^n/n)$ one has $\mathbf{SIZE}(S(n)) \not\supseteq \mathbf{SIZE}(C \cdot S(n))$.*

Proof. Let us abbreviate size(f) as the minimum size of a circuit computing f. We know that there are constants $C_1 < C_2$ so that for all n there is a function $f : \{0,1\}^n \to \{0,1\}$ with size$(f) \geq C_1 2^n/n$ while all functions f have size$(f) \leq C_2 2^n/n$. Pick m minimal so that $S(n) < C_1 2^m/m$. Then there is a function f^* only defined on the first m coordinates so that size$(f^*) \geq C_1 2^m/m > S(n)$. But size$(f^*) \leq C_2 2^m/m \leq \frac{2C_2}{C_1} \cdot S(n)$. □

6.3 Uniform vs. non-uniform computation

Let $M_1, M_2, \ldots$ be the recurrent Turing machine sequence from Lemma 3.1 that contains every possible Turing machine infinitely often. Consider the unary variant of the Halting problem

$$\texttt{UHALT} := \{1^n \mid M_n \text{ halts on input } 1^n\}$$

Then we know that there is no Turing machine computing `UHALT`. But it is easy define a family of circuits $(C_n)_{n \in N}$ with size $|C_n| \leq O(n)$ computing `UHALT`. That means $\texttt{UHALT} \in \mathbf{P/poly}$. The issue is that Turing machines are a *uniform* model of computing where one has one Turing machine for all input lengths. In contrast circuits are a *non-uniform* model of computing where we have a different circuit for every input length n.

If one wants to compare the power of circuits with Turing machines, then the right model is the following:

Definition 6.9. Let $T, \alpha : \mathbb{N} \to \mathbb{N}$ be functions. A language $L \in \{0,1\}^*$ is in $\mathbf{DTIME}(T(n))/\alpha(n)$ if there is a sequence of strings $(a_n)_{n\in\mathbb{N}}$ with $a_n \in \{0,1\}^{\alpha(n)}$ so that

$$L = \left\{x \in \{0,1\}^* \mid M(x, a_{|x|}) = 1\right\}$$

for a deterministic $T(n)$-time Turing machine M.

If $L \in \mathbf{DTIME}(T(n))/\alpha(n)$, then we also say that L is decidable by a *time-$T(n)$ Turing machine with $\alpha(n)$ bits of advice*. And in fact, problems computable by polynomial size circuits correspond exactly to those computable by Turing machines taking poly(n) bits of advice:

Theorem 6.10. *One has* $\mathbf{P/poly} = \bigcup_{c>0} \mathbf{DTIME}(n^c)/n^c$

The proof is straightforward and we just sketch the argument.

Proof sketch. If $L \in \mathbf{P/poly}$ then for input n the advice is just the circuit C_n deciding the inputs of length n and on input of $x \in \{0,1\}^n$ the Turing machine M just evaluates $C_n(x)$.

Now let $L \in \mathbf{DTIME}(n^c)/n^c$ and fix an input length n with advice string $a_n \in \{0,1\}^{n^c}$. Then $M(x, \alpha_n)$ is a deterministic Turing machine machine that can be turned into a circuit following Theorem 6.6. □

Could it be true that `SAT` can be solved in polynomial time with polynomial advice? We cannot rule this out but we can give some evidence that this is unlikely as it would cause the polynomial hierarchy to collapse on the second level.

Theorem 6.11 (Karp, Lipton 1980)**.** $(\mathbf{NP} \subseteq \mathbf{P/poly}) \Rightarrow (\mathbf{PH} = \Sigma_2^P \cap \Pi_2^P)$.

Proof. First we prove an auxiliary result. Here we will make use of circuits that have more than one output bit which extends the original definition in a straightforward way.

Claim I. *Assume* $\mathbf{NP} \subseteq \mathbf{P/poly}$. *Let M be a polynomial time Turing machine. Then there is a family of circuits $(C_{n,m})_{n,m\in\mathbb{N}}$ with $C_{n,m} : \{0,1\}^n \to \{0,1\}^m$ of size poly$(n+m)$ so that for all $x \in \{0,1\}^n$ one has $M(x, C_{n,m}(x)) = 1 \Leftrightarrow \exists y \in \{0,1\}^m : M(x,y) = 1$.*

Proof of Claim I. Fix x and set $n := |x|$. By the Cook-Levin Theorem there is a SAT formula φ so that $\exists(y,z) : \varphi(y,z) = 1 \Leftrightarrow \exists y : M(x,y) = 1$ where z are some auxiliary variables. Assuming $\mathbf{NP} \subseteq \mathbf{P/poly}$ we know that there is a Turing machine M' with an advice string $a \in \{0,1\}^{\text{poly}(|\psi|)}$ that can decide whether $\varphi' \in$ `SAT` in polynomial time for all instances φ' with $|\varphi'| = |\varphi|$. We first run M' to test whether $\varphi \in$ `SAT`. If so, we go through all the variables in (y,z) and set them to 0 or 1, testing feasibility each time and keeping it satisfiable[1]. Then we obtain a Turing machine that using the advice string can find a satisfying assignment (y,z) in polynomial time if there is one. Then we turn that Turing machine with polynomial advice into a circuit, see Theorem 6.10. $\square$

Now we come to the main proof. We assume that **NP** $\subseteq$ **P/poly** and derive that then $\Pi_2^P \subseteq \Sigma_2^P$. By Theorem 5.7.(a), this then collapses the polynomial hierarchy on the 2nd level. Let $L \in \Pi_2^P$ which means that there is some polynomial p and some polynomial time Turing machine M so that

$$L = \left\{x \in \{0,1\}^* \mid \forall u \in \{0,1\}^{p(|x|)} \exists v \in \{0,1\}^{p(|x|)} : M(x,u,v) = 1\right\}$$

Fix an input $x \in \{0,1\}^*$ with $n := |x|$. We claim that for some polynomial q one has

$$\begin{aligned} & \forall u \in \{0,1\}^{p(n)} \exists v \in \{0,1\}^{p(n)} : M(x,u,v) = 1 \quad (*) \\ \Leftrightarrow \quad & \exists q(n)\text{-size circuit } C\ \forall u \in \{0,1\}^{p(n)} : M(x,u,C(x,u)) = 1 \quad (**) \end{aligned}$$

If $(**)$ is true then for any u one can pick $v := C(x,u)$ to make $(*)$ true. Now suppose $(*)$ is true. By Claim I, there is indeed a polynomial size circuit that for each u produces a v so that $M(x,u,v) = 1$. Hence also $(**)$ is true. The formulation $(**)$ shows that $L \in \Sigma_2^P$ which gives the claim. $\square$

One can fully unconditionally prove that there are problems in **PH** that require at least polynomially large circuits. Though we have to admit that the proof feels somewhat disappointing as it again relies on the counting argument from Theorem 6.3.

Theorem 6.12 (Kannan)**.** *For all $k \geq 1$, $\Sigma_2^P \cap \Pi_2^P \not\subseteq \mathbf{SIZE}(n^k)$.*

For the sake of space and time, we only sketch the argument.

Proof. If **NP** $\not\subseteq$ **P/poly** then the claim is trivially true, so suppose that **NP** $\subseteq$ **P/poly**. We use $\preceq$ to denote the (bit wise) lexicographic ordering on circuits. Among all circuits on n inputs, let C_n be the lexicographically minimal one so that $\text{size}(C_n) \geq n^{k+1}$ (we may assume that n is large enough so that this is possible). Define the language $L := \{x \in \{0,1\}^* : C_{|x|}(x) = 1\}$. By construction $L \notin \mathbf{SIZE}(n^k)$. We leave it as an exercise to show that L can be written using a few quantifiers; in fact $L \in \Sigma_4^P$. Now, by assumption **NP** $\subseteq$ **P/poly** and so Theorem 6.11 we have $\mathbf{PH} = \Sigma_2^P \cap \Pi_2^P$ which gives the claim as $L \in \Sigma_4^P \subseteq \mathbf{PH}$. $\square$

[1]We should agree that we imagine that a SAT formula has the same size whether it contains the variable y_i or a constant 0 or 1 instead.

Chapter 7

Randomized computation

In this chapter we want to extend the concept of Turing machines and allow them to use randomness.

7.1 Probabilistic Turing machines

Definition 7.1. A *probabilistic Turing machine* (PTM) M with k tapes is a Turing machine with <u>two</u> transition functions $\delta_0, \delta_1 : Q \times \Gamma^{k+1} \to Q \times \Gamma^k \times \{L, S, R\}^{k+1}$. In each time step independently, the algorithm choses a random bit $b \in \{0,1\}$ uniformly and moves using transition function δ_b. The running time is the maximum number of steps before M halts over all random choices.

Note that we have restricted the definition to the decision version of Turing machines, meaning that there is no output tape. On any input $x \in \{0,1\}^*$, the output $M(x) \in \{0,1\}$ is a *random variable* where $\Pr[M(x) = 1]$ is the probability to accept the input and $\Pr[M(x) = 0]$ is the probability to reject the input.

We will define several complexity classes depending on the "type" of the error. Similar to the case of **NP** we can rebuild a PTM so that it draws all the needed random bits r at the beginning and then runs a deterministic Turing machine based on the input x and the random bits r. So we define the complexity classes using this alternative view. By a slight abuse of notation, for a language $L \subseteq \{0,1\}^*$ and $x \in \{0,1\}^*$ we also write

$$L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

(as if L was a function).

Definition 7.2. Let $T : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. A language L is in $\mathbf{BPTIME}_{\varepsilon}(T(n))$ if there is a deterministic Turing machine M with running time $O(T(n))$ so that

$$\Pr_{r \in_R \{0,1\}^{O(T(|x|))}}[M(x,r) = L(x)] \geq 1 - \varepsilon(|x|) \quad \forall x \in \{0,1\}^*$$

We define $\mathbf{BPTIME}(T(n)) := \mathbf{BPTIME}_{1/3}(T(n))$ and $\mathbf{BPP} := \bigcup_{c>0} \mathbf{BPTIME}(n^c)$.

Here **BPP** stands for **b**ounded-error **p**robabilistic **p**olynomial time. When we write $r \in_R \{0,1\}^m$ then we mean that r is a uniform random sample from $\{0,1\}^m$. Phrased differently, the bits $r_1, \ldots, r_m$ are independent random bits so that $\Pr[r_i = 1] = \Pr[r_i = 0] = \frac{1}{2}$ for all i. Here the PTM will not need more than the running time $O(T(n))$ many random bits. Also note that the constant $1/3$ that we have chosen was arbitrary — any constant less than $1/2$ would have provided an equivalent definition.

In the definition of **BPP** we allowed *two-sided error*. It will also be useful to consider *one-sided error* where for example the decision of the PTM to accept an input is always correct.

Definition 7.3. Let $T : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. A language $L \subseteq \{0,1\}^*$ is in $\mathbf{RTIME}_{\varepsilon}(T(n))$ if there is a deterministic Turing machine M with running time $O(T(n))$ so that

$$\begin{aligned} x \in A \quad &\Rightarrow \quad \Pr_{r \in_R \{0,1\}^{O(T(|x|))}}[M(x,r) = 1] \geq 1 - \varepsilon(|x|) \\ x \notin A \quad &\Rightarrow \quad \Pr_{r \in_R \{0,1\}^{O(T(|x|))}}[M(x,r) = 0] = 1 \end{aligned}$$

We write $\mathbf{RTIME}(T(n)) := \mathbf{RTIME}_{1/3}(T(n))$ and $\mathbf{RP} := \bigcup_{c>0} \mathbf{RTIME}(n^c)$.

Here **RP** stands for **r**andomized **p**olynomial-time. Similarly one could have one-sided error going into the other direction. We define $\mathbf{coRTIME}(T(n)) := \{\bar{L} \mid L \in \mathbf{RTIME}(T(n))\}$ and $\mathbf{coRP} := \{\bar{L} \mid L \in \mathbf{RP}\}$.

We can also consider *zero-sided error* but in order for that to make sense we need to slightly adjust the definition of a PTM and allow that instead of terminating in time $T(n)$ on any input and any random choice, it terminates in *expected time* $T(n)$ on any input.

Definition 7.4. Let $T : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. A language $L \subseteq \{0,1\}^*$ is in $\mathbf{ZPTIME}(T(n))$ if there is a PTM M so that for any $x \in \{0,1\}^*$ one has (a) $M(x) = L(x)$ and (b) the expected running time on input x is $O(T(|x|))$. We set $\mathbf{ZPP} := \bigcup_{c>0} \mathbf{ZPTIME}(n^c)$.

Here **ZPP** stands for **z**ero-error **p**robabilistic **p**olynomial-time. Note that one could have defined **ZPP** in an equivalent way: the PTM M returns accept, reject or "?" in polynomial time where accept and reject are always correct and "?" is returned with probability at most $1/3$. In that case one may require that the running time is bounded by a polynomial for any input and any choice of random bits.

Theorem 7.5. ZPP = RP ∩ coRP.

Proof. By symmetry of **ZPP** it suffices to show the following:

Claim I. ZPP ⊆ RP.

Proof of Claim I. Let $L \in$ **ZPP** be a language that is decided by a PTM M with expected running time $T(n)$. On input $x \in \{0,1\}^*$ run M for $3T(|x|)$ time units. By Markov's inequality, with probability at least $2/3$, M has found a correct answer which we then output. If M has not terminated, output 0. Then if $x \notin L$ we never make a mistake. If $x \in L$ then we only make a mistake with probability at most $1/3$. □

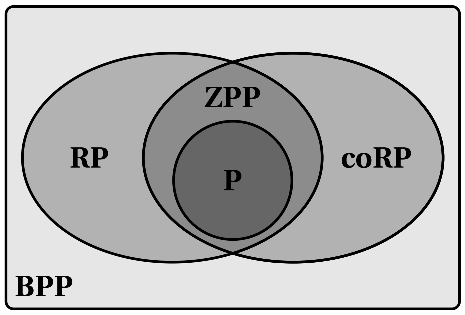
Claim II. RP ∩ coRP ⊆ ZPP.

Proof of Claim II. Let $L \in$ **RP** ∩ **coRP**. Let M^{RP} be the **RP** PTM for L and let M^{coRP} be the **coRP** PTM for L. Assume $T(n)$ is a common upper bound on the running time for both. Consider the following algorithm:

```
(1) REPEAT
        (2) Run $b_1 := M^{\mathrm{RP}}(x)$
        (3) Run $b_0 := M^{\mathrm{coRP}}(x)$
        (4) If $b_0 = b_1$ then return $b_0$
```

For symmetry reasons consider an input x with $x \in L$. Then in each iteration $b_1 \in \{0,1\}$ with $\Pr[b_1 = 1] \geq 2/3$. But $b_0 = 1$ always. That means the decision will always be correct and in each iteration we come to a decision with probability at least $2/3$. Then the expected number of iterations until termination is $\frac{3}{2}$. □

We visualize the complexity classes. Though it is entirely possible that **BPP** = **P** and all the classes collapse to one.



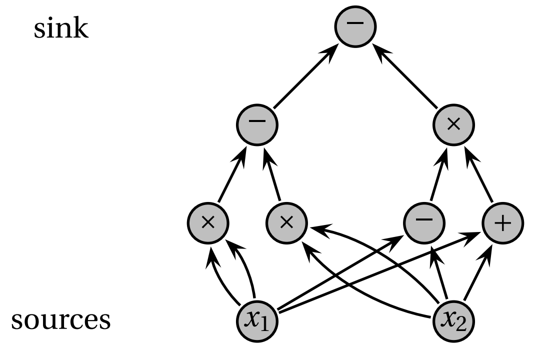
7.2 Arithmetic circuits

Randomization often helps to design easier or (polynomially) faster algorithms. Occasionally it happens that first a randomized polynomial time algorithm is known for a problem and then years later a (more complicated) deterministic algorithm is discovered. This was for example the case with deciding primality. In this section we want to give an example for a problem which is in **coRP** but is not known to be in **P**.

Definition 7.6. An *arithmetic circuit* C over $\mathbb{Z}$ is a directed acyclic graph where sources are labelled either with variable names $x_1, \ldots, x_n$ or constants 0 or 1. Non-source nodes are labelled with one of the operations $+$, $-$, $\times$ each having fan-in 2[1]. The graph contains a unique sink which given values $x_1, \ldots, x_n \in \mathbb{N}$ computes the output $C(x)$ in a natural way.

Consider the following example:

[1] The operation "$-$" does not commute. We agree that an order on the incoming edges is specified. In the pictures we will agree that the right input is to be subtracted from the left input.



arithmetic circuit C

The circuit computes the function $C(x_1, x_2) = (x_1^2 - x_2^2) - (x_1 - x_2)(x_1 + x_2) = 0$ for all $(x_1, x_2) \in \mathbb{Z}^2$. Hence the function f vanishes on the whole $\mathbb{Z}^2$ — but that was not actually obvious from just "looking" at the circuit. This motivates the following problem:

$$\texttt{ZEROP} := \left\{C \mid C \text{ is arithmetic circuit with } C(x_1, \dots, x_n) = 0 \;\; \forall x_1, \dots, x_n \in \mathbb{Z}\right\}$$

Our goal is to prove that ZEROP $\in$ **coRP**, but before that we need to introduce some auxiliary results. A *multivariate polynomial* f over $\mathbb{Z}$ with variables $x_1, \dots, x_n$ is of the form

$$f(x_1, \dots, x_n) = \sum_{a \in \mathbb{Z}^n_{\geq 0}} \beta_a \prod_{i=1}^n x_i^{a_i}$$

where the support $\{a \mid \beta_a \neq 0\}$ is finite and $\beta_a \in \mathbb{Z}$ for all $a \in \mathbb{Z}^n_{\geq 0}$. Each term $\beta_a \prod_{i=1}^n x_i^{a_i}$ is called a *monomial*, β_a is the *coefficient* of the monomial and $\sum_{i=1}^n a_i$ is the *total degree* of the monomial. The *total degree* of the polynomial f itself is defined as $\deg_{\text{tot}}(f) := \max_{a:\beta_a \neq 0} \|a\|_1$, which is the maximum total degree of any monomial. In contrast we use the term *degree* with symbol $\deg(f) := \max_{a:\beta_a \neq 0} \|a\|_\infty$ which is the maximum degree that any single variable has. For univariate polynomials, these quantities are the same while for arbitrary polynomials with n variables one has $\deg(f) \leq \deg_{\text{tot}}(f) \leq n \cdot \deg(f)$.

Lemma 7.7. *Let C be an arithmetic circuit over $\mathbb{Z}$ in variables $x_1, \dots, x_n$ that contains at most m multiplications. Then $C(x)$ is a polynomial in $x_1, \dots, x_n$ of total degree at most 2^m.*

One can easily prove this by induction because for any polynomials f, g one has $\deg_{\text{tot}}(f+g) \leq \max\{\deg_{\text{tot}}(f), \deg_{\text{tot}}(g)\}$ and $\deg_{\text{tot}}(f \cdot g) \leq \deg_{\text{tot}}(f) + \deg_{\text{tot}}(g)$.

That means each multiplication can at most double the total degree. As an arithmetic circuit computes a polynomial, the next question is: how many roots can a polynomial have?[2]. First, the answer is classical for univariate polynomials.

Theorem 7.8. *Let $f(x) = \sum_{i=0}^{d} \beta_i x^i$ be a univariate polynomial over a field $\mathbb{F}$ of degree $d > 0$, then f has at most d many roots (in $\mathbb{F}$).*

In particular applying Theorem 7.8 with any field that contains $\mathbb{Z}$ (for example $\mathbb{F} = \mathbb{R}$) this means that any polynomial with integer coefficients has at most d many roots in $\mathbb{Z}$. The *Fundamental Theorem of Algebra* by Gauss (1799) says that for $\mathbb{F} = \mathbb{C}$, there are exactly d roots if roots are counted with multiplicities. Next, we say that a polynomial f is the *zero polynomial* if all its coefficients are 0. From the next result it follows in particular that this is the same as $f(x) = 0\ \forall x \in \mathbb{Z}^n$. The question is in particular whether a multivariate polynomial that is not the zero polynomial could have arbitrarily many roots? Actually it can, see for example $f(x_1, x_2) = x_1$ which has all points $(0, \mathbb{Z})$ as roots. But restricting to a *combinatorial rectangle* we can bound the number of roots!

Lemma 7.9 (Schwarz-Zippel 1979)**.** *Let $f(x_1, \ldots, x_n)$ be a polynomial that is not the zero polynomial and let $d := \deg_{tot}(f)$ be the total degree. Let $S \subseteq \mathbb{Z}$ be a finite set. Then*

$$\Pr_{a_1,\ldots,a_n \in_R S}[f(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}$$

Note that here $a_1, \ldots, a_n$ are chosen independently at random from S.

Proof. We prove the claim by induction over $n \geq 1$. For $n = 1$ we know by Theorem 7.8 that f has at most d roots.

Now consider $f(x_1, \ldots, x_{n+1})$ with $n \geq 1$ with $d := \deg_{\text{tot}}(f)$. We can pull out powers x_{n+1}^i from each monomial and write

$$f(x_1, \ldots, x_{n+1}) = \sum_{i=0}^{\ell} f_i(x_1, \ldots, x_n) \cdot x_{n+1}^i$$

where we choose ℓ maximal so that f_ℓ is not the zero polynomial. Here each f_i is a polynomial in n variables with $\deg_{\text{tot}}(f_i) \leq d - i$.
Claim I. *Fix any $a_1, \ldots, a_n \in \mathbb{Z}$ so that $f_\ell(a_1, \ldots, a_n) \neq 0$. Then* $\Pr_{a_{n+1} \in_R S}[f(a_1, \ldots, a_{n+1}) = 0] \leq \frac{\ell}{|S|}$.
Proof of Claim I. Fix $a_1, \ldots, a_n$ with $f_\ell(a_1, \ldots, a_n) \neq 0$. Consider the univariate polynomial q defined by $q(x_{n+1}) := f(a_1, \ldots, a_n, x_{n+1})$. Then $\deg(q) = \ell$ and by

[2] Where a *root* of f is a point x with $f(x) = 0$.

assumption q is not the zero-polynomial. Then q has at most ℓ roots by Theorem 7.8. □

We continue with the main proof. We have

$$\begin{aligned}
& \Pr_{a_1,\dots,a_{n+1}\in_R S}[f(a_1,\dots,a_{n+1})=0] \\
\leq\; & \underbrace{\Pr_{a_1,\dots,a_n\in_R S}[f_\ell(a_1,\dots,a_n)=0]}_{\leq \frac{\deg_{\text{tot}}(f_\ell)}{|S|}\text{ by induction}} + \underbrace{\Pr_{a_1\dots,a_{n+1}\in_R S}[f(a_1,\dots,a_{n+1})=0 \mid f_\ell(a_1,\dots,a_n)\neq 0]}_{\leq \frac{\ell}{|S|}\text{ by Claim I}} \\
\leq\; & \frac{d-\ell}{|S|}+\frac{\ell}{|S|}=\frac{d}{|S|}
\end{aligned}$$

□

Now consider an arithmetic circuit C using variables $x_1,\dots,x_n$ that contains m multiplications. We know that $\deg_{\text{tot}}(C)\leq 2^m$ (considering C also as the polynomial computed at its sink) and hence setting $S:=\{1,\dots,2^{2m}\}$, for any non-zero polynomial C we have

$$\Pr_{x\in_R S^n}[C(x)=0] \overset{\text{Lem 7.9}}{\leq} \frac{\deg_{\text{tot}}(C)}{|S|} = \frac{2^m}{2^{2m}} = 2^{-m}$$

using the Schwarz-Zippel Lemma. There is only one issue: the numbers computed at the nodes in the circuit can double in every multiplication, i.e. even $C(0,\dots,0)$ can be as large as 2^{2^m} which would require 2^m bits to represent the numbers. There is however a trick to fix this and keep the numbers small:

Lemma 7.10. *Let C be an arithmetic circuit in n variables and let $a\in\mathbb{Z}^n$ and $k\in\mathbb{N}$. Then one can compute $C(a) \mod k$ in time polynomial in $|C|$ and the encoding length of both a and k.*

Proof. Starting at the sources compute all values computed by nodes in C modulo k. □

Note that it is possible that $C(a)\neq 0$ while $C(a) \mod k = 0$. But again randomization helps avoiding this.

Theorem 7.11. ZEROP $\in$ **coRP**.

Proof. Let C be an arithmetic circuit in n variables and let $m:=|C|$ be the number of nodes in the circuit[3]. We will assume that m is at least some large enough

[3]Differently from before, now m is not just the number of multiplications but certainly m is still an upper bound on the number of multiplications.

constant — if not, just add a few redundant nodes. We use the following algorithm:

```
(1) FOR $64m$ ITERATIONS DO
    (2) Sample $a_1, \dots, a_n \sim \{1, \dots, 2^{2m}\}$
    (3) Sample $k \sim \{1, \dots, 2^{4m}\}$
    (4) Compute $C(a_1, \dots, a_n) \mod k$. If outcome $\neq 0$ THEN reject
(5) Accept
```

First, the algorithm runs in polynomial time by Lemma 7.10. If $C \in$ ZEROP then in each iteration $C(a_1, \dots, a_n) \mod k = 0$ and hence the algorithm will always accept C. It remains to prove the following which then implies that the probability that an input $C \notin$ ZEROP is not rejected in $64m$ iterations is bounded by $(1 - \frac{1}{16m})^{64m} \leq e^{-4}$.

Claim I. *If $C \notin$ ZEROP, then in each iteration the probability that C is rejected in (4) is at least $\frac{1}{16m}$.*

Proof of Claim I. As argued earlier, with probability at least $1 - 2^{-m}$ one has $C(a) \neq 0$. We condition on this event to happen. As $0 \leq a_i \leq 2^{2m}$ and each gate in the circuit can at most double the value we certainly have $|C(a)| \leq (2^{2m})^{2^m} = 2^{2m \cdot 2^m} \leq 2^{2^{2m}}$. Then $|C(a)|$ has at most $\log_2 |C(a)| \leq 2^{2m}$ many different prime factors. We recall the following classical fact:

Prime Number Theorem (Hadamard, de la Vallée Poisson 1896). *For any $\varepsilon > 0$ there is a $N_\varepsilon \in \mathbb{N}$ so that for any $N \geq N_\varepsilon$ one has*

$$(1-\varepsilon) \cdot \frac{N}{\ln(N)} \leq |\{p \in \{1, \dots, N\} : p \text{ is prime}\}| \leq (1+\varepsilon) \cdot \frac{N}{\ln(N)}$$

So assuming m is big enough, we know that there are at least $\frac{1}{2} \cdot \frac{2^{4m}}{\ln(2^{4m})} \geq \frac{2^{4m}}{8m}$ many prime numbers in $\{1, \dots, 2^{4m}\}$. Hence

$$\Pr_{k \in_R \{1, \dots, 2^{4m}\}}[k \text{ is prime and } k \text{ not factor of } C(a)] \geq \frac{\frac{2^{4m}}{8m} - 2^{2m}}{2^{4m}} = \frac{1}{8m} - 2^{-2m} \quad (*)$$

If the event in $(*)$ happens, then $C(a) \mod k \neq 0$. Then overall the probability of rejection in (4) is at least $(1 - 2^{-m}) \cdot (\frac{1}{8m} - 2^{-2m}) \geq \frac{1}{16m}$ for m large enough. $\square$

The technique used in the algorithm is also called *finger printing*. More generally this refers to comparing two objects not bit-wise but rather by comparing their hash values.

7.3 Error reduction

A very useful property of randomized algorithms is that by repeating them one can make the error probability exponentially small. For this we will reply on the following fact:

Theorem 7.12 (Chernoff bound)**.** *Let $X_1, \dots, X_n \in \{0,1\}$ be independent random variables with sum $X := \sum_{i=1}^n X_i$ and mean $\mu := \mathbb{E}[X]$. Then for any $\varepsilon > 0$,*

$$\Pr\Big[\Big|\sum_{i=1}^{n} X_i - \mu\Big| \geq \varepsilon\mu\Big] \leq 2\exp\Big(-\min\Big\{\frac{\varepsilon^2}{4}, \frac{\varepsilon}{2}\Big\}\cdot\mu\Big)$$

We should mention that there are dozens of reformulations of similar results that bound the deviation of a sum of independent random variables from the mean. Somewhat inprecisely they are all called “the Chernoff bound”.

Theorem 7.13. *For any polynomial p one has*

(a) $\mathbf{BPP}_{\frac{1}{2}-\frac{1}{p(n)}} = \mathbf{BPP} = \mathbf{BPP}_{2^{-p(n)}}$

(b) $\mathbf{RP}_{1-\frac{1}{p(n)}} = \mathbf{RP} = \mathbf{RP}_{2^{-p(n)}}$

Proof. We only prove the technically more involved item (a). Let $L \in \mathbf{BPP}_{\frac{1}{2}-\frac{1}{p(n)}}$, meaning there is a randomized polynomial time algorithm M that makes a mistake with probability at most $\frac{1}{2}-\frac{1}{p(n)}$. For an input $x \in \{0,1\}^n$ we run the following algorithm:

```
(1) REPEAT $M(x)$ exactly $N := 4p(n)^3$ times.
(2) Let $b \in \{0,1\}$ be the value returned by $M(x)$ in the majority of cases.
    Return b.
```

For N_0 and N_1 be the number of times that in (1) we have $M(x) = 0$ and $M(x) = 1$, resp. Note that $N = N_0 + N_1$. For symmetry reasons we analyze the case that $x \in L$. Then $\Pr[M(x) = 1] \geq \frac{1}{2} + \frac{1}{p(n)}$. Then N_1 is the sum of N independent random variables and its mean is $\mu := \mathbb{E}[N_1] \geq N \cdot (\frac{1}{2} + \frac{1}{p(n)})$. We choose $\varepsilon > 0$ so that $\varepsilon\mu =$

$\frac{N}{p(n)}$. Then

$$\begin{aligned}
\Pr\left[N_1 \leq \frac{N}{2}\right] &\leq \Pr\Big[|N_1 - \mu| \geq \underbrace{\frac{N}{p(n)}}_{=\varepsilon\mu}\Big] \\
&\overset{\text{Thm 7.12}}{\leq} 2\exp\left(-\frac{\varepsilon^2}{4}\mu\right) \\
&= \exp\left(-\frac{1}{4}\frac{N^2}{\mu p(n)^2}\right) \\
&\overset{\mu\leq N}{\leq} \exp\left(-\frac{N}{4p(n)^2}\right) \leq 2^{-p(n)}
\end{aligned}$$

□

7.4 The relationship of BPP to other classes

We want to discuss some results on how **BPP** relates to other problems. Again, one might conjecture that **BPP** = **P**, but here we list what can actually be proven right now:

Theorem 7.14. BPP ⊆ P/poly.

Proof. Let $L \in$ **BPP**. By Theorem 7.13 we know that there is a polynomial time deterministic TM M so that $\Pr_{r\in_R\{0,1\}^{p(|x|)}}[M(x,r) = L(x)] \geq 1-2^{-(|x|+1)}$. Now fix an input length n. We call a random string r *bad* for $x \in \{0,1\}^n$ if $M(x,r) \neq L(x)$ and *good* otherwise. There are 2^n different inputs and only a $2^{-(n+1)}$-fraction of strings is bad for any given input. Hence there must be a string $r_n^* \in \{0,1\}^{p(n)}$ that is good for all inputs of length n. Then $M(x,r_n^*)$ is a polynomial time Turing machine with advice r_n^* that solves L. □

It should most certainly be true that **BPP** ⊆ **NP**. But even that is not known! Hence we prove a weaker statement. Here, for two strings $a,b \in \{0,1\}^m$ we write $a \oplus b \in \{0,1\}^m$ as their bitwise addition modulo 2. Moreover, for a set $S \subseteq \{0,1\}^m$ and $b \in \{0,1\}^m$ we define $S \oplus b := \{a \oplus b \mid a \in S\}$ as the shift of S by b modulo 2. For the next complexity result, we need a combinatorial lemma:

Lemma 7.15. *Let $S \subseteq \{0,1\}^m$ and $0 < \delta < 1$.*

(a) *Assume $|S| \leq \delta 2^m$ and $k < \frac{1}{\delta}$. Then for all $u_1,\ldots,u_k \in \{0,1\}^m$ one has $\bigcup_{i=1}^k (S \oplus u_i) \neq \{0,1\}^m$.*

(b) *Assume* $|S| \geq (1-\delta)2^m$ *and* $k > \frac{m}{\log_2(1/\delta)}$. *Then there are* $u_1, \ldots, u_k \in \{0,1\}^m$ *so that* $\bigcup_{i=1}^k (S \oplus u_i) = \{0,1\}^m$.

Proof. (a) is simple as $|\sum_{i=1}^k (S \oplus u_i)| \leq k \cdot |S| < \frac{1}{\delta} \cdot \delta 2^m = 2^m$. For (b), we use the probabilistic method to argue that $u_1, \ldots, u_k$ exist. Choose $u_1, \ldots, u_k \in_R \{0,1\}^m$ uniformly at random.

Claim I. *For a fixed* $x \in \{0,1\}^m$ *one has* $\Pr_{u_1,\ldots,u_k}[x \notin \bigcup_{i=1}^k (S \oplus u_i)] < 2^{-m}$.

Proof of Claim I. We have

$$\Pr\left[x \notin \bigcup_{i=1}^k (S \oplus u_i)\right] \overset{(*)}{=} \Pr_{r \in_R \{0,1\}^m} [x \notin (S \oplus r)]^k \overset{(**)}{\leq} \delta^k < 2^{-m}$$

where we use independence in $(*)$ and in $(**)$ we use that $x \in (S \oplus r) \Leftrightarrow (x \oplus r) \in S$ and $x \oplus r$ is still a uniform random choice from $\{0,1\}^m$. □

Now by the union bound and Claim I, the probability that any of the 2^m many $x \in \{0,1\}^m$ is not covered is strictly less than $2^m \cdot 2^{-m} = 1$. Hence there exists a choice of $u_1, \ldots, u_k$ that make that event true. □

Now to the actual complexity result:

Theorem 7.16 (Sipser, Gács 1983)**.** $\mathbf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$

Proof. By symmetry of **BPP** it suffices to prove that $\mathbf{BPP} \subseteq \Sigma_2^P$. Let $L \in \mathbf{BPP}$. Again, by Theorem 7.13 there is a deterministic Turing machine M so that $\Pr_{r \in_R \{0,1\}^{p(|x|)}}[M(x) = L(x)] \geq 1 - \delta(|x|)$ for all $x \in \{0,1\}^*$ where we will make the choice for the error δ later. Our goal is to design a $\exists\forall$ type of predicate for L. Intuitively one would use the $\exists$ quantifier to guess one correct random string. That does not quite work, but one can use the help of the second quantifier to make sure that M produces the same output for exponentially many strings. We fix an input x with $n := |x|$ and denote the number of random bits by $m := p(n)$. Let $S := \{r \in \{0,1\}^m \mid M(x,r) = 1\}$ be the random bits that let the Turing machine accept. We know that

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

We want to use Lemma 7.15 and we need to find a value of k so that

$$\frac{m}{\log_2(1/\delta)} < k < \frac{1}{\delta}$$

For example we can pick $\delta := \frac{1}{4m}$ and $k := 2m$. We claim that

$$x \in L \Leftrightarrow \exists u_1, \ldots, u_k \in \{0,1\}^m \underbrace{\forall r \in \{0,1\}^m \bigvee_{i=1,\ldots,k} (M(x, r \oplus u_i) = 1)}_{(***)}$$

In fact, if $x \in L$, then by Lemma 7.15.(b) there are $u_1, \dots, u_k$ so that $\bigcup_{i=1}^{n}(S \oplus u_i) = \{0,1\}^m$ which makes $(***)$ true. If $x \notin L$, then by Lemma 7.15.(a), for any $u_1, \dots, u_k$ there is an $r^* \notin \bigcup_{i=1}^{k}(S \oplus u_i)$ that makes $(***)$ false. Here note that for all $u_1, \dots, u_k$ and r one has

$$\bigvee_{i=1,\dots,k}(M(x, r \oplus u_i) = 1) \quad \Leftrightarrow \quad \exists i \in [k] : (r \oplus u_i) \in S \quad \Leftrightarrow \quad r \in \bigcup_{i=1}^{k}(S \oplus u_i)$$

□

Chapter 8

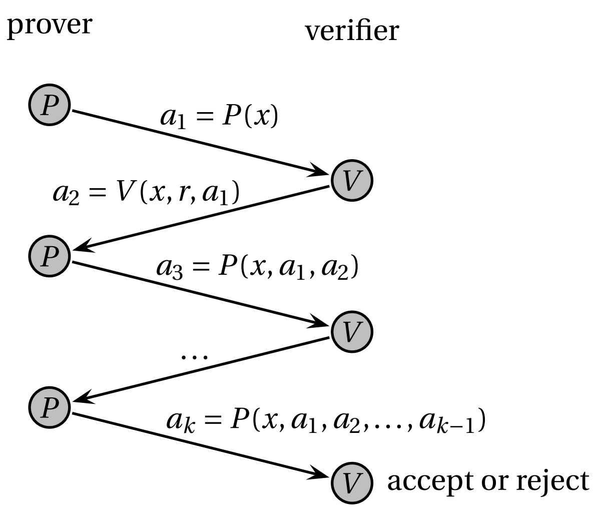
Interactive proofs

In this chapter, we want to discuss *interactive proofs* which have applications to cryptography and hardness of approximation. Suppose we have a language L and instead of constructing a Turing machine deciding whether a given input x is in L we have two parties that send each other messages. The parties are:

- *The verifier:* This is a probabilistic polynomial time Turing machine whose goal is to correctly decide whether $x \in L$ or not.

- *The prover:* The prover has unlimited computational power and the goal of the prover is to make the verifier accept whether $x \in L$ or not.

More formally, we can model the behaviour of the verifier as a polynomial time computable function $V : \{0,1\}^* \to \{0,1\}^*$ where $a_i := V(x, r, a_1 \dots, a_{i-1})$ is the message that the verifier sends in round i, depending on the input x, random bits $r \in \{0,1\}^{p(|x|)}$ and messages $a_1, \dots, a_{i-1}$ exchanged so far with the prover. The prover can be modeled as an arbitrary function $P : \{0,1\}^* \to \{0,1\}^*$ where $a_i := P(x, a_1, \dots, a_{i-1})$ is the next message that the prover sends to the verifier, depending on the input x and the messages exchanged so far. After some number k of rounds the prover decides to either accept or reject.



We note that the prover does not see the random bits r — and this is important. We write $\text{out}_V(P, V, x) \in \{0,1\}$ as the decision of the verifier at the end of the protocol. Note that $\text{out}_V(P, V, x)$ is a random variable depending on the random bits r. The way we phrased it here, the prover sends the first message but one could also let the verifier to send the first message. As we allow an arbitrary polynomial number of rounds, it does not actually matter here.

Definition 8.1. A language L is in **IP** if there exists a probabilistic polynomial time Turing machine V so that

$$\begin{aligned} x \in L &\quad\Rightarrow\quad \exists P : \Pr[\text{out}_V(V, P, x) = 1] \geq \frac{2}{3} \\ x \notin L &\quad\Rightarrow\quad \forall P : \Pr[\text{out}_V(V, P, x) = 1] \leq \frac{1}{3} \end{aligned}$$

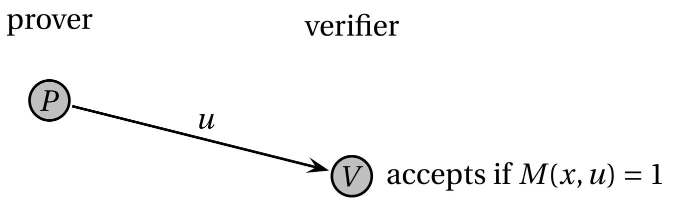
Here the number of rounds and the length of the messages has to be at most polynomial in $|x|$.

8.1 Some facts on IP

To understand the class **IP** better, we give two examples.

Lemma 8.2. NP $\subseteq$ **IP**.

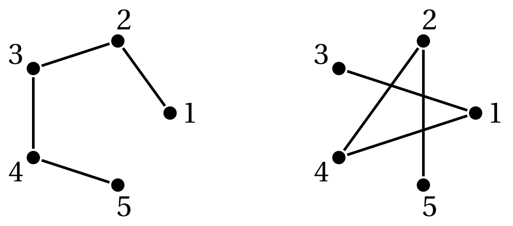
Proof. Let $L \in$ **NP**. Then we can write $L = \{x \in \{0,1\}^* : \exists u \in \{0,1\}^{p(|x|)} : M(x, u) = 1\}$ where M is a polynomial time Turing machine. On input x, the prover sends the string u to the verifier. The verifier accepts if $M(x, u) = 1$.



□

Note that the protocol is correct in both cases with probability 1. Also the verifier did not need any randomness and is deterministic.

Next, we want to discuss a problem that is not known to be in **NP** but which has a clever non-trivial protocol. We say that two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a permutation $\pi : V_1 \to V_2$ on the vertices so that $\{u, v\} \in E_1 \Leftrightarrow \{\pi(u), \pi(v)\} \in E_2$. Note that the answer is trivially no if the graphs have a different number of vertices, so it is often just assumed that $|V_1| = |V_2|$.



two isomorphic graphs

Consider the language

$$\texttt{GI} := \big\{(G_1, G_2) \mid G_1 \text{ and } G_2 \text{ are isomorphic}\big\}$$

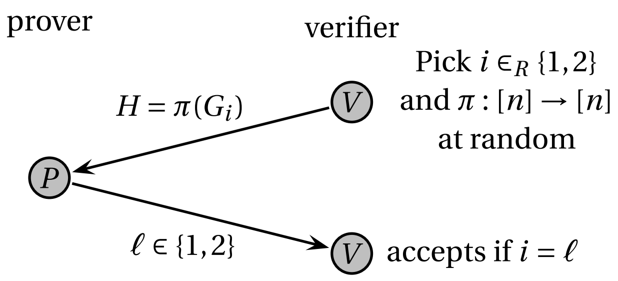
where GI stands for *graph isomorphism*. Then trivially $\texttt{GI} \in \textbf{NP}$ while it is not known whether $\texttt{GI} \in \textbf{P}$ or at least $\texttt{GI} \in \textbf{coNP}$. Although we should mention that there is a $n^{\text{polylog}(n)}$-time algorithm due to Babai from 2015, which comes close to settling this question. Either way, while Babai's paper is 100+ pages long, there is a very simple interactive protocol that can prove that two graphs are *not* isomorphic.

Theorem 8.3. $\overline{\texttt{GI}} \in \textbf{IP}$.

Proof. Consider graphs[1] G_1, G_2 and suppose that $G_1 = ([n], E_1)$, $G_2 = ([n], E_2)$ and we have to design a protocol that decides whether the graphs are isomorphic. The verifier picks a random index $i \in_R \{1,2\}$ and a random permutation $\pi : [n] \to$

[1]If one wanted to be picky, then the complement of $\overline{\texttt{GI}}$ also contains all bit strings that do not represent two graphs on the same number of vertices. This can be easily checked in polynomial time and we ignore this here.

$[n]$ and sends the permutated graph $H = \pi(G_i) = ([n], \pi(E_i))$. Then the prover sends an index $\ell \in \{1,2\}$ and the prover accepts if $i = \ell$.



We analyze the protocol:

- *Case* $(G_1, G_2) \in \overline{\texttt{GI}}$. If both graps are not isomorphic, then there is exactly one index i so that $\pi(G_i) = H$ which the prover can compute and send to make the verifier accept with probability 1.

- *Case* $(G_1, G_2) \notin \overline{\texttt{GI}}$. Assume G_1 and G_2 are isomorphic. When the prover receives a graph H, it is equally likely that H is a permutation of G_1 as it is of G_2[2]. Hence $\Pr[i = \ell] = 1/2$ no matter the choice of ℓ. One can reduce the probability to below $1/3$ by repeating the protocol.

□

There is also an upper bound on the power of **IP** that one can show:

Lemma 8.4. IP ⊆ PSPACE.

We do not want to give a formal proof here, but the idea is as follows: Fix a language $L \in$ **IP** and let V be the corresponding verifier (which is a polynomial time PTM). For any input x, the number of rounds k, the number of random bits and the maximum length of messages is upper bounded by some polynomial $p(|x|)$. Then one can solve the problem of finding the prover P that maximizes the probability of V to accept x in polynomial space.

[2]More formally one can argue that in the isomorphic case, the distribution $\pi(G_1)$ and the distribution of $\pi(G_2)$ is the same when π is a uniform random permutation.

8.2 The sumcheck protocol

So far we only discussed that **NP** ⊆ **IP** ⊆ **PSPACE**. Our goal for the remainder of this chapter is to prove that actually **IP** = **PSPACE**. This fact was a quite surprising result at the time it was discovered.

In order to prepare for this result, we consider a problem that seems somewhat artificial but it will be enourmously helpful:

$$\texttt{SUMCHECK} = \left\{ (g, K, p) \mid \begin{array}{c} g(x_1, \dots, x_n) \text{ is a polynomial with } \sum_{b \in \{0,1\}^n} g(b) \equiv_p K \\ \text{and } p \text{ is prime with } p \geq 2n \deg(g) \end{array} \right\}$$

We claim that there is an efficient interactive protocol for `SUMCHECK`. Note that we have the somewhat artificial condition $p \geq 2nd$ with $d := \deg(g)$ for a technical reason and it is needed for our protocol to work. In the later application we can pick the number p arbitrarily large so this is not a restriction for us.

We first give an overview over the protocol that decides `SUMCHECK`. Consider a polynomial $g(x_1, \dots, x_n)$ with degree by d. Then define a univariate polynomial

$$h(x_1) := \sum_{b' \in \{0,1\}^{n-1}} g(x_1, b')$$

that also has a degree of at most d. Intuitively, h arises from g by summing over the last $n-1$ variables but leaving x_1 as an indeterminant. Ideally we would like the prover to send the polynomial $h(x_1)$ (in form of its $d+1$ coefficients which are in $\mathbb{Z}_p$) to the verifier. If the prover was truthful, the verifier could just check whether $h(0) + h(1) \equiv_p K$. But of course the verifier might try to cheat and rather send a different degree-d polynomial $s(x_1)$. Now we will use that the verifier knows the whole polynomial $s(x_1)$ and not just the two values $s(0)$ and $s(1)$. The verifier wants to check whether indeed s and h are the same polynomials, where it is useful that in the non-affirmative case one has that $\Pr_{a \in_R \mathbb{Z}_p}[s(a) \neq_p h(a)] \geq 1 - \frac{d}{p}$. Hence the verifier picks a random $a \in_R \mathbb{Z}_p$ and asks the prover to prove that

$$s(a) \equiv_p \sum_{b' \in \{0,1\}^{n-1}} g(a, b')$$

Then this is again a `SUMCHECK` instance on $n-1$ variables and we can recurse. More formally, we prove the following:

Theorem 8.5. `SUMCHECK` $\in$ **IP** *whenether the encoding of the polynomial is chosen so that (i) for any assignment $a \in setZ_p^n$ one can evaluate $g(a)$ in time poly$(|g|)$ and (ii)* $\deg(g) \leq |g|$.

Proof. We formally state the protocol that we already described earlier: For $n = 1$, the verifier simply checks if $g(0) + g(1) \equiv_p K$. For $n \geq 2$, the protocol is as follows:

SUMCHECK PROTOCOL

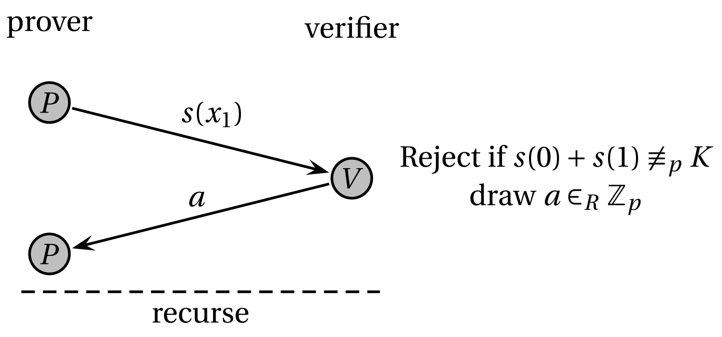
Input: Polynomial g of degree at most d, prime $p \in \mathbb{N}$, $K \in \mathbb{Z}_p$

(1) **Prover:** Sends a univariate polynomial $s(x_1)$ of degree at most d

(2) **Verifier:** Reject if $s(0) + s(1) \not\equiv_p K$. Otherwise draw $a \in_R \mathbb{Z}_p$ and send it.

(3) **Both:** Recursively run the protocol on the `SUMCHECK` problem

$$s(a) \equiv \sum_{b' \in \{0,1\}^{n-1}} g(a, b')$$


prover
verifier
P
$s(x_1)$
V
Reject if $s(0)+s(1) \not\equiv_p K$
draw $a \in_R \mathbb{Z}_p$
a
P
recurse

For the analysis it will be helpful to consider the following univariate polynomial:

$$h(x_1) := \sum_{b' \in \{0,1\}^{n-1}} g(x_1, b')$$

Claim I. $(g, K, p) \in \texttt{SUMCHECK} \Rightarrow \exists P : \Pr[\mathrm{out}_V(P, v, x) = 1] = 1$.

Proof of Claim I. The prover sends $s := h$. Then $s(0) + s(1) \equiv_p K$. The protocol recurses on $s(a) \equiv_p h(a)$ which is true and hence succeeds by induction with probability 1 as well.

Claim II. $(g, K, p) \notin \texttt{SUMCHECK} \Rightarrow \forall P : \Pr[\mathrm{out}_V(P, v, x) = 0] \geq (1 - \frac{d}{p})^n$.

Proof of Claim II. We prove the claim by induction over n. For $n = 1$, the verifier rejects with probability 1 and the statement is true. Now suppose $n \geq 2$. If the prover sends $s = h$, then $s(0) + s(1) \equiv_p h(0) + h(1) \not\equiv_p K$ and the verifier rejects right away. So assume that the prover sends a polynomial s with $s \neq h$. Then

$$\begin{aligned}\Pr[V \text{ rejects}] \;&\geq\; \underbrace{\Pr_{a \in_R \mathbb{Z}_p}[s(a) \not\equiv_p h(a)]}_{\geq 1 - \frac{d}{p}} \cdot \underbrace{\Pr\left[\begin{matrix} V \text{ rejects proof of} \\ s(a) \equiv_p \sum_{b' \in \{0,1\}^{n-1}} g(a, b') \end{matrix} \;\middle|\; s(a) \not\equiv_p h(a)\right]}_{\geq (1 - \frac{d}{p})^{n-1} \text{ by induction}} \\ &\geq\; \left(1 - \frac{d}{p}\right)^n\end{aligned}$$

Here we use on the left that by Theorem 7.8 the non-zero polynomial $s-h$ has at most d roots. This is where we use that p is a prime so that $\mathbb{Z}_p$ is a field. On the right we use induction. $\square$

Finally we note that $(1-\frac{d}{p})^n \geq \exp(-\frac{2dn}{p}) \geq \frac{1}{e}$ using the inequality that $1-z \geq e^{-2z}$ for $0 \leq z \leq \frac{1}{2}$. $\square$

8.3 #SAT$_D \in$ IP

The next step is to demonstrate how to use SUMCHECK on generalized satisfiability problems. Consider the language

$$\texttt{\#SAT}_D = \big\{(\varphi, K) \mid \varphi \text{ is a CNF with exactly } K \text{ satisfying assignments}\big\}$$

Theorem 8.6. #SAT$_D \in$ **IP**.

Proof. Consider a CNF φ with variables $x_1,\dots,x_n$ and clauses $C_1,\dots,C_m$. For $j \in [m]$, define the polynomial[3]

$$p_j(X_1,\dots,X_n) := 1 - \prod_{j:x_j \text{ appears in } C_j} (1-X_j) \cdot \prod_{j:\bar{x}_j \text{ appears in } C_j} X_j$$

One may check that for any $a \in \{0,1\}^n$ one has

$$p_j(a) = \begin{cases} 1 & \text{if } a \text{ satisfies clause } C_j \\ 0 & \text{otherwise} \end{cases}$$

Hence

$$P_\varphi(X_1,\dots,X_n) := \prod_{j=1}^{m} p_j(X_1,\dots,X_n) \tag{8.1}$$

is a polynomial of degree at most m (and total degree at most nm) so that $P_\varphi(a) = 1$ if and only if a satisfies φ. That means $\sum_{a\in\{0,1\}^n} P_\varphi(a)$ denotes the number of satisfying assignments for φ and so

$$(\varphi, K) \in \texttt{\#SAT}_D \iff \underbrace{\sum_{a\in\{0,1\}^n} P_\varphi(a) = K}_{(*)}$$

[3]For example if C_j is $x_1 \vee \bar{x}_2 \vee x_3$ then the corresponding polynomial is $p_j(X_1,\dots,X_n) = 1-(1-x_1)x_2(1-x_3)$.

Here $(*)$ is almost a SUMCHECK instance, just that we need to phrase it modulo p. But the left hand side of $(*)$ is in $\{0,\dots,2^n\}$ and hence we compute[4] a prime number p with $2^n < p \leq 2^{2n}$ and run the SUMCHECK protocol on $\sum_{a\in\{0,1\}^n} g(a) \equiv_p K$. Finally note that the expression in (8.1) allows to evaluate $P_\varphi(a) \mod p$ for any $a \in \mathbb{Z}_p^n$ in polynomial time. □

8.4 TQBF ∈ IP

Finally we want to prove that **IP** = **PSPACE**. By Lemma 8.4 we already know that **IP** ⊆ **PSPACE**, so it suffices to prove that some **PSPACE**-complete problem is in **IP**; naturally we choose TQBF[5]. We recall that a *quantified boolean formula (QBF)* is of the form

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \, \varphi(x_1,\dots,x_n)$$

where $Q_i \in \{\exists,\forall\}$ and φ is an arbitrary boolean formula with variables $x_1,\dots,x_n \in \{0,1\}$. For the time being, to simplify notation, let us assume that the formula is indeed of the form

$$\psi := \exists x_1 \forall x_2 \exists x_3 \dots \forall x_n \, \varphi(x_1,\dots,x_n)$$

Then one can observe that $\psi \in$ TQBF if and only if

$$\sum_{x_1\in\{0,1\}} \prod_{x_2\in\{0,1\}} \sum_{x_3\in\{0,1\}} \dots \prod_{x_n\in\{0,1\}} P_\varphi(x_1,\dots,x_n) \neq 0 \tag{8.2}$$

where P_φ is the polynomial defined in the proof of Theorem 8.6. So one might be tempted to just run a straightforward generalization of the SUMCHECK protocol on (8.2). But there is the problem that (8.2) may contain up to n multiplications and each one may double the degree, possibly leading to a polynomial of degree 2^n. But in (8.2), only function values $P_\varphi(x_1,\dots,x_n)$ with $x \in \{0,1\}^n$ matter. So we could manipulate the polynomial P_φ (and any of the intermediate polynomials $Q_{x_i\in\{0,1\}} \dots \prod_{x_n\in\{0,1\}} P_\varphi(x_1,\dots,x_n)$ where $Q \in \{\exists,\forall\}$) as long as we do not change the values on those binary points.

[4]By the prime number theorem, we know that a $\Theta(\frac{1}{n})$ fraction of integers between 2^n and 2^{2n} are prime numbers. For poly(n) rounds we can pick a random integer between 2^n and 2^{2n} and then test in polynomial time if p is prime.

[5]This statement requires a short argument: consider any language $L \in$ **PSPACE** and suppose we can prove that TQBF ∈ **IP**. As $L \leq_p$ TQBF, there is a polynomial time computable map f so that $x \in L \Leftrightarrow f(x) \in$ TQBF. The verifier can compute $f(x)$ and then run the TQBF-protocol that proves whether $f(x)$ is in TQBF and use the answer to determine whether $x \in L$.

Let $\mathcal{P} := \mathbb{Z}[x_1,\dots,x_n]$ be the vector space of all polynomials in variables $x_1,\dots,x_n$ with coefficients over $\mathbb{Z}$. For each variable index $i \in [n]$, we define $L_i : \mathcal{P} \to \mathcal{P}$ as the unique linear map so that

$$L_i\Big(\prod_{j=1}^{n} x_j^{\alpha_j}\Big) = x_i^{\min\{\alpha_i,1\}} \cdot \prod_{j\neq i} x_j^{\alpha_j}$$

for all $\alpha \in \mathbb{Z}^n_{\geq 0}$. For example $L_1(4x_1^3x_2 + 2x_1x_2 + 5x_1 + 3x_2^4) = 6x_1x_2 + 5x_1 + 3x_2^4$. Intuitively speaking, the L_i operator[6] reduces the exponents that appear with variable x_i down to 1. Hence L_i is also called a *linearization* operator. We note that also

$$L_i(p)(x_1,\dots,x_n) = x_i \cdot p(x_1,\dots,x_{i-1},1,x_{i+1},\dots,x_n) + (1-x_i)\cdot p(x_1,\dots,x_{i-1},0,x_{i+1},\dots,x_n)$$

which we also could have used to define L_i. Either way, since $x_i^a = x_i$ for all $x_i \in \{0,1\}$ and all $a \in \mathbb{N}$ we know the following:

Observation 8.7. For any polynomial $p(x_1,\dots,x_n)$ and any variable $i \in [n]$ one has

$$p(x_1,\dots,x_n) = (L_i p)(x_1,\dots,x_n) \quad \forall x_1,\dots,x_n \in \{0,1\}$$

It will also be useful to introduce two other types of functions $\forall_{x_i}, \exists_{x_i} : \mathcal{P} \to \mathcal{P}$ defined by

$$\begin{aligned} (\forall_{x_i} p)(x_1,\dots,x_n) &:= \prod_{b\in\{0,1\}} p(x_1,\dots,x_{i-1},b,x_{i+1},\dots,x_n) \\ (\exists_{x_i} p)(x_1,\dots,x_n) &:= \sum_{b\in\{0,1\}} p(x_1\dots,x_{i-1},b,x_{i+1},\dots,x_n) \end{aligned}$$

Then we can rewrite (8.2) into the equivalent condition

$$\exists_{x_1} L_1 \forall_{x_2} L_1 L_2 \exists_{x_3} L_1 L_2 L_3 \dots \forall x_n L_1 L_2 \dots L_n P_\varphi(x_1,\dots,x_n) \neq 0 \tag{8.3}$$

Note that the description length of (8.3) blew up at most quadratically compared to (8.2) while all intermediate polynomials will have degree at most polynomial in $|\varphi|$. Now we are read for the final result of this chapter.

Theorem 8.8 (Lund, Fortnow, Karloff, Nisan 1990)**. IP** = **PSPACE**.

[6]There does not seem to be a generally accepted definition of what an "operator" is but the term is typically used for any linear map that maps elements in a vector space (such as $\mathcal{P}$) into that same vector space.

Proof. Consider a `TQBF` instance ψ with inner CNF φ consisting of m clauses. Then as discussed above, there is a polynomial time computable sequence $O_1, \dots, O_T$ so that $\psi \in$ `TQBF` if and only if

$$O_1 O_2 \dots O_T P_\varphi(x_1, \dots, x_n) \neq 0 \tag{8.4}$$

where (i) $T \leq O(n^2)$, (ii) each O_t is of the form L_i, $\exists x_{x_i}$ or $\forall_{x_i}$ for some i, (iii) each of the polynomials $O_t O_{t+1} \dots O_T P_\varphi(x_1, \dots, x_n)$ as a degree upper bounded by $d \leq O(m)$. One can also prove that the number that the left hand side of (8.4) represents is upper bounded by $2^{2^{O(n)}}$ which corresponds to at most $O(n)$ squaring operations. If the left hand side of (8.4) is not 0, then analogous to the argument in Theorem 7.11, there is a prime number $p \leq 2^{O(n)}$ and a $K \in \{1, \dots, p-1\}$ so that

$$O_1 O_2 \dots O_T P_\varphi(x_1, \dots, x_n) \equiv_p K \tag{8.5}$$

The prover then tries to convince the verifier that (8.5) holds. Before the start of the main protocol, the prover sends (p, K) and the verifier checks that p is indeed a prime.

The main protocol itself will again be defined in a recursive way. As often with recursive arguments, the intermediate problem is different and somewhat more complicated than the global problem that it aims to solve. So the actual intermediate task for the prover is the following:

> Given a polynomial $P_\varphi(x_1, \dots, x_n)$, a partition $[n] = S \dot\cup F$ of the variables into *set* variables S and *free* variables F, a sequence of operations $O_1, \dots, O_{T'}$ of the form $O_t \in \{\exists_{x_i}, \forall_{x_i}, L_i\}$ that eliminate each free variable exactly once, a prime p, $K' \in \mathbb{Z}_p$ and values $a_i \in \mathbb{Z}_p$ for all set variables $i \in S$. Prove that
>
> $$O_1 \underbrace{O_2 \dots, O_{T'} P_\varphi((x_i)_{i \in F}, (a_i)_{i \in S})}_{=:h} \equiv_p K'$$

Then the protocol splits off the first operation O_1 and proceeds as follows:

- Case $O_1 = \exists_{x_i}$. In this case, $h(x_i)$ is a polynomial of degree at most d depending on x_i.

  1. Prover: send polynomial $s(x_i)$ (which truthfully would be $h(x_i)$)
  2. Verifier: If $s(0) + s(1) \not\equiv_p K'$ then reject. Otherwise pick $a_i \in_R \mathbb{Z}_p$, and ask prover to prove $s(a) \equiv_p h(a)$

- Case $O_1 = \forall_{x_i}$. Again, $h(x_i)$ is a polynomial of degree at most d depending on x_i.

1. Prover: send polynomial $s(x_i)$ (which truthfully would be $h(x_i)$)
2. Verifier: If $s(0) \cdot s(1) \not\equiv_p K'$ then reject. Otherwise pick $a_i \in_R \mathbb{Z}_p$, and ask prover to prove $s(a) \equiv_p h(a)$

- Case $O_1 = L_i$. In this case we have $i \in S$, i.e. the variable x_i in h has already been set to a_i. Consider the polynomial

$$h'(x_i) := O_2 \ldots, O_{T'} P_\varphi((x_j)_{j \in F \cup \{i\}}, (a_j)_{j \in S \setminus \{i\}})$$

obtained by "freeing" the ith variable.

1. Prover: send polynomial $s(x_i)$ (which truthfully would be $h'(x_i)$)
2. Verifier: If $a_i \cdot s(1) + (1 - a_i) \cdot s(0) \not\equiv_p K'$ then reject. Otherwise pick $a'_i \in_R \mathbb{Z}_p$, and ask prover to prove $s(a'_i) \equiv_p h'(a'_i)$

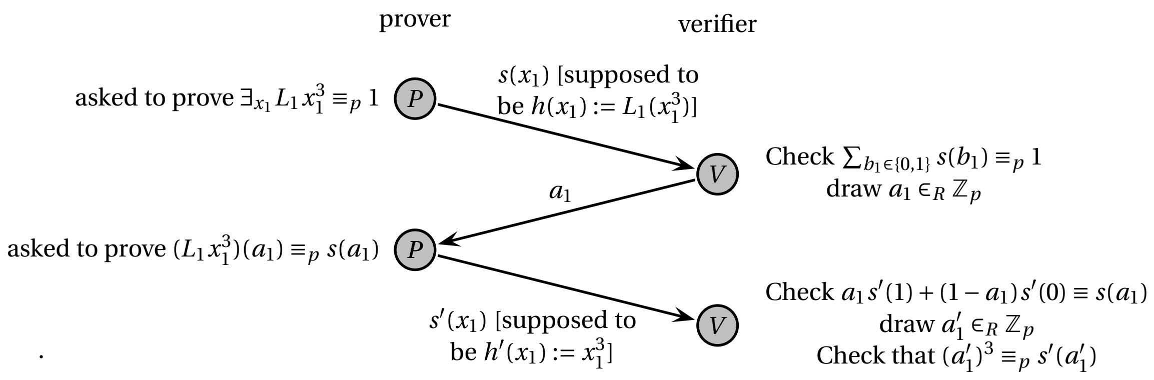
If $\psi \in$ TQBF, then we can check that the truthful choice will make the verifier accept with probability 1. Now assume $\psi \notin$ TQBF. Then one can prove that $\Pr[\text{verifier accepts}] \le (1 - \frac{d}{p})^{T'}$ by induction over the number of operations T'. The analysis for the first 2 cases is similar to Theorem 8.5, so we only discuss the last case where $O_1 = L_i$. If the prover indeed sends $s = h'$ then

$$a_i \cdot s(1) + (1 - a_i) \cdot s(0) \equiv_p a_i \cdot h'(1) + (1 - a_i) \cdot h'(0) \equiv_p (L_i h')(a_i) \not\equiv_p K'$$

and the verifier rejects[7] Otherwise, $s - h'$ is a non-zero polynomial with at most d roots and $\Pr[s(a'_i) \not\equiv_p h(a'_i)] \ge 1 - \frac{d}{p}$. We condition on this event and recurse; running the protocol on the instance with $T' - 1$ operations, the verifier will reject with probability at least $(1 - \frac{d}{p})^{T'-1}$. That concludes the claim. ☐

We note that the protocol in Theorem 8.8 has one-sided error. That implies that we could have strengthened the definition of **IP** to require that $x \in L \Rightarrow \exists P : \Pr[\text{out}_V(V, P, x) = 1] = 1$ without changing the class **IP**.

[7]We provide an example. Consider the protocol to prove that $\exists_{x_1} L_1 x_1^3 \equiv_p 1$ (which indeed is true).


prover
verifier
asked to prove $\exists_{x_1} L_1 x_1^3 \equiv_p 1$
P
$s(x_1)$ [supposed to be $h(x_1) := L_1(x_1^3)$]
V
Check $\sum_{b_1 \in \{0,1\}} s(b_1) \equiv_p 1$
draw $a_1 \in_R \mathbb{Z}_p$
a_1
asked to prove $(L_1 x_1^3)(a_1) \equiv_p s(a_1)$
P
$s'(x_1)$ [supposed to be $h'(x_1) := x_1^3$]
V
Check $a_1 s'(1) + (1 - a_1) s'(0) \equiv s(a_1)$
draw $a'_1 \in_R \mathbb{Z}_p$
Check that $(a'_1)^3 \equiv_p s'(a'_1)$

.

Chapter 9

The PCP Theorem

9.1 Introduction

The interactive proof class **IP** that we defined in Chapter 8 characterizes precisely **PSPACE**. It would be nice if we had an interactive proof-type characterization of the much weaker class **NP** that is more relevant for us.

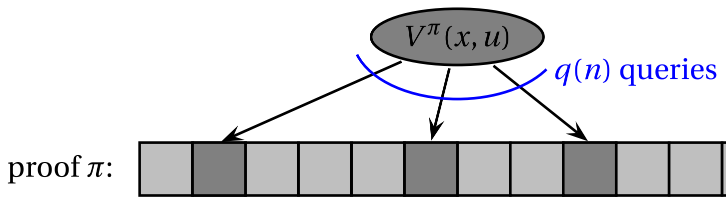
Definition 9.1. A $(r(n), q(n))$-**PCP** *verifier* is a deterministic polynomial-time Turing machine $V^\pi(x, u)$ that receives random bits $u \in_R \{0,1\}^{r(|x|)}$ and non-adaptive access to $q(|x|)$ bits of a proof string $\pi \in \{0,1\}^*$. More precisely, the Turing machine can write indices $i_1, \dots, i_{q(n)}$ $(n := |x|)$ on a special tape and then receive the bits $\pi_{i_1}, \dots, \pi_{i_{q(n)}}$ but it can make such a query only once. We say that a $(r(n), q(n))$-**PCP** verifier V^π *decides* a language $L \subseteq \{0,1\}^*$ if

$$x \in L \quad \Rightarrow \quad \exists \pi : \Pr_{u \in_R \{0,1\}^{r(n)}} [V^\pi(x, u) \text{ accepts}] = 1$$

$$x \notin L \quad \Rightarrow \quad \forall \pi : \Pr_{u \in_R \{0,1\}^{r(n)}} [V^\pi(x, u) \text{ accepts}] \leq \frac{1}{2}$$

Then **PCP**$(r(n), q(n))$ is the set of languages L that can be decided by a $(O(r(n)), O(q(n)))$-**PCP** verifier.

Here **PCP** stands for **p**robabilistically **c**heckable **p**roof.



Since randomness is limited to $r(n)$ bits, we can limit the length of the proof string to $2^{r(n)}q(n)$ as this is the maximum number of bits that may have a positive probability of being read. But as we are using a random access model based on indices, the proof may indeed have such a length. We can check a few simple inclusions:

Theorem 9.2. *One has*

(a) $\mathbf{PCP}(0,0) = \mathbf{P}$

(b) $\mathbf{PCP}(0, poly(n)) = \mathbf{NP}$

(c) $\mathbf{PCP}(\log n, 1) \subseteq \mathbf{NP}$

Proof. **For (a).** Without access to a proof and without randomness, V^π is just a deterministic poly-time TM. **For (b).** In this case, there is no randomness and a proof of polynomial length; this equals **NP**. **For (c).** Let $V^\pi(x,u)$ be a $(r(n),q(n)) = (O(\log n), O(1))$-**PCP** verifier for L. A non-deterministic TM can guess the proof $\pi \in \{0,1\}^*$ where $|\pi| \leq 2^{r(n)} \cdot q(n) \leq \text{poly}(n)$. Then one can deterministically compute the value $\Pr_{u \in_R \{0,1\}^{r(n)}}[V^\pi(x,u)]$ by enumerating all $2^{r(n)} = \text{poly}(n)$ choices for the random bits u. □

One of the deepest results in complexity theory is as follows:

Theorem 9.3 (PCP Theorem — Arora, Feige, Goldwasser, Lund, Lovász, Motwani, Safra, Sudan, Szegedy 1992[1]). $\mathbf{PCP}(\log n, 1) = \mathbf{NP}$.

The reader should appreciate at this point that it is mindblowing how just checking a constant number of bits could suffice for **NP**-hard problems. This has dramatic consequences for the approximatility of **NP**-hard problems as we discuss later in Section 9.8. The proof of Theorem 9.3 is beyond the scope of this lecture[2]. In order to demonstrate the techniques prove the following weaker result:

Theorem 9.4 (Weak PCP Theorem). $\mathbf{PCP}(poly(n), 1) \supseteq \mathbf{NP}$.

[1]Really this is a combination of several works and we cite the set of authors that received the 2001 Gödel prize.

[2]Chapter 22 in the textbook gives a complete proof of the PCP Theorem following the more recent work of Dinur that uses a gap-amplification argument. However the iterative nature of that argument makes it also harder to understand how exactly a proof looks like that is checkable by querying $O(1)$ bits.

While the proof length in this result is exponential, the number of bits that are read is still $O(1)$. We will now work towards a proof of Theorem 9.4 and develop a few tools on the way.

9.2 Quadratic equations

It suffices to prove that for some **NP**-hard language L one has $L \in \mathbf{PCP}(\text{poly}(n), 1)$. One could work with 3SAT as usually, but it will actually be more convenient to use a different **NP**-hard problem. We consider the problem of quadratic equations modulo 2.

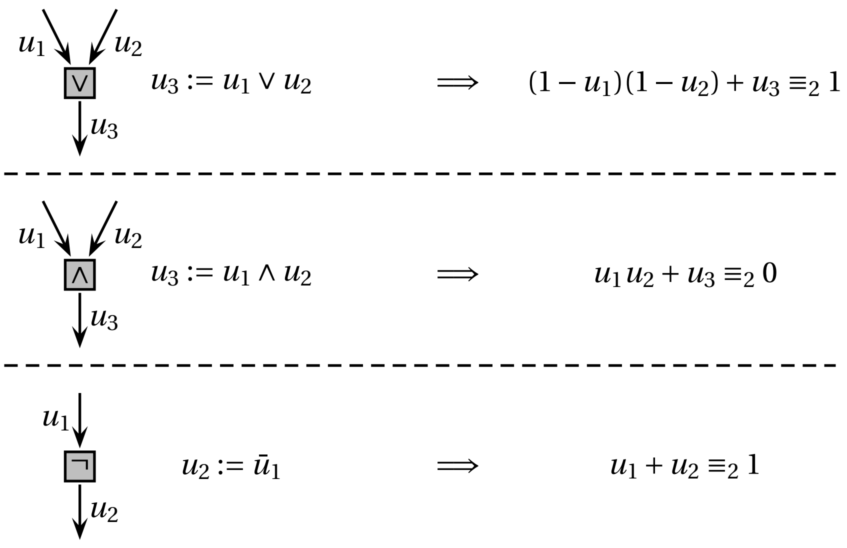
$$\texttt{QUADEQ} := \left\{(A_1, \dots, A_m, b_1, \dots, b_m) \mid \exists u \in \{0,1\}^n : \langle A_i, uu^T \rangle \equiv_2 b_i \forall i = 1, \dots, m\right\}$$

An example instance is as follows

$$\begin{aligned} u_1^2 + u_2 u_3 + u_1 u_4 &\equiv_2 1 \\ u_1 u_3 + u_2 u_4 &\equiv_2 0 \\ u_1 u_4 + u_2^2 + u_2 u_4 &\equiv_2 1 \end{aligned}$$

Theorem 9.5. QUADEQ *is* **NP***-complete.*

Proof. It is clear that QUADEQ $\in$ **NP**, so we only show **NP**-hardness. Recall that given a circuit $C : \{0,1\}^n \to \{0,1\}$ with gates $\vee, \wedge, \neg$ it is **NP**-hard to decide whether there is a $x \in \{0,1\}^n$ so that $C(x) = 1$. Now introduce a variable u_i for each wire (including the input wires). Then any variable that is not on an input wire, is defined in terms of two other variables. And indeed we can transform each such definition into a quadratic equation:



Linear terms such as u_i can be replaced by u_i^2 which is equivalent in $\mathbb{Z}_2$. Then there is an $x \in \{0,1\}^n$ with $C(x) = 1$ if and only if the system constructed above plus the equation $u_m^2 \equiv_2 1$ has a feasible solution where u_m is the variable for the output wire. $\square$

We can now give an otherview of the **PCP**(poly(n), 1) verifier for `QUADEQ`. The expected proof that certifies that u is satisfiable will be the function tables for the two linear functions $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^{n\times n} \to \{0,1\}$ with

$$f(x) = \langle u, x\rangle \mod 2 \quad \text{and} \quad g(X) = \langle uu^T, X\rangle \mod 2$$

Note that the total length of the function tables is $2^n + 2^{n^2}$. Given an instance for `QUADEQ`, the PCP verifier V^π will have 3 components:

- **Linearity test.** By reading $O(1)$-bits, test whether the functions f and g are at least nearly linear.
- **Consistency test.** By reading $O(1)$-bits, test whether the functions f and g are of the form $f(x) \equiv_2 \langle u, x\rangle$ and $g(X) \equiv_2 \langle uu^T, X\rangle$ for a common vector u.
- **Satisfiability test.** Check whether the encoded assignment u satisfies the `QUADEQ` instance.

We will fill in the details in the upcoming sections.

9.3 Fourier analysis

In this section we want to make an detour to study functions of the form $f : \{\pm 1\}^n \to \mathbb{R}$. For two functions $f, g : \{\pm 1\}^n \to \mathbb{R}$ we define an inner product

$$\langle f, g\rangle := \mathop{\mathbb{E}}_{x \in_R \{\pm 1\}}[f(x) \cdot g(x)] = \frac{1}{2^n} \sum_{x \in \{\pm 1\}^n} f(x) \cdot g(x)$$

that is sometimes called the *expectation inner product.* For a set $S \subseteq [n]$, consider the special function

$$\chi_S : \{\pm 1\}^n \to \{\pm 1\} \quad \text{with} \quad \chi_S(x) := \prod_{i \in S} x_i$$

We will call χ_S a *character.* We denote $S \Delta T := (S \setminus T) \cup (T \setminus S)$ as the *symmetric difference* of sets $S, T \subseteq [n]$.

Lemma 9.6. *For $S, T \subseteq [n]$ one has*

$$\langle \chi_S, \chi_T \rangle = \begin{cases} 1 & \text{if } S = T \\ 0 & \text{otherwise} \end{cases}$$

Proof. We write

$$\langle \chi_S, \chi_T \rangle = \mathop{\mathbb{E}}_{x \in_R \{\pm 1\}^n} [\chi_S(x) \cdot \chi_T(x)] = \mathop{\mathbb{E}}_{x \in_R \{\pm 1\}^n} [\chi_{S\Delta T}(x)] = \prod_{i \in S\Delta T} \underbrace{\mathop{\mathbb{E}}_{x_i \in_R \{\pm 1\}} [x_i]}_{=0} = \begin{cases} 0 & \text{if } |S\Delta T| > 0 \\ 1 & \text{if } |S\Delta T| = 0 \end{cases}$$

Here we use that $\chi_S(x) \cdot \chi_T(x) = \chi_{S\Delta T}(x)$. We also use that for independent random variables X and Y one has $\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y]$. ☐

We note that the set $\{f \mid f : \{\pm 1\}^n \to \mathbb{R}\}$ is a vector space of dimension 2^n and Lemma 9.6 says that the family of 2^n many functions $\{\chi_S\}_{S \subseteq [n]}$ is pairwise orthogonal and even orthonormal. Hence $\{\chi_S\}_{S \subseteq [n]}$ must be an orthonormal basis for that vector space. It then makes sense to consider the *coordinates* that an element $f : \{\pm 1\}^n \to \mathbb{R}$ has with respect to that basis:

Definition 9.7. For $f : \{\pm 1\}^n \to \mathbb{R}$ we denote the *S-th Fourier coefficient* as $\hat{f}_S := \langle f, \chi_S \rangle = \mathbb{E}_{x \in_R \{\pm 1\}^n}[f(x)\chi_S(x)]$.

By orthonormality we know the following:

Lemma 9.8. *For every function $f : \{\pm 1\}^n \to \mathbb{R}$ one has $f(x) = \sum_{S \subseteq [n]} \hat{f}_S \chi_S(x)$ for all $x \in \{\pm 1\}^n$.*

The following can be obtained by applying Lemma 9.8 and using the orthonormality of the characters.

Lemma 9.9. *For any $f, g : \{\pm 1\}^n \to \mathbb{R}$ one has*

1. *$\langle f, g \rangle = \sum_{S \subseteq [n]} \hat{f}_S \cdot \hat{g}_S$*
2. *Parsival's identity: $\langle f, f \rangle = \sum_{S \subseteq [n]} \hat{f}_S^2$*

For two vectors $x, y \in \{\pm 1\}^n$ we write $x \odot y \in \{\pm 1\}^n$ as the vector with entries $(x \odot y)_i := x_i \cdot y_i$. Then $\chi_S(x \odot y) = \chi_S(x) \cdot \chi_S(y)$ for all $x, y \in \{\pm 1\}^n$. One may think of this property as an analogue to linearity (we will get back to that later). We can prove that the only functions f that satisfy $f(x \odot y) = f(x) \cdot f(y)$ for all x, y are the character functions and moreover, if $f(x \odot y) = f(x) \cdot f(y)$ holds for most pairs (x, y), then f must be close to one particular such character function. Here the Fourier view will be invaluable.

Theorem 9.10. *Let $f : \{\pm 1\}^n \to \{\pm 1\}$ be a function so that*

$$\Pr_{x,y \in_R \{\pm 1\}^n}[f(x \odot y) = f(x)f(y)] \geq \frac{1}{2} + \varepsilon \tag{9.1}$$

with $0 \leq \varepsilon \leq \frac{1}{2}$. Then there is a set $S \subseteq [n]$ so that $\hat{f}_S \geq 2\varepsilon$.

Proof. We write

$$\begin{aligned}
2\varepsilon &= \Big(\frac{1}{2} + \varepsilon\Big) - \Big(\frac{1}{2} - \varepsilon\Big) \\
&\overset{(9.1)}{\leq} \mathop{\mathbb{E}}_{x,y \in_R \{\pm 1\}^n}[f(x \odot y) \cdot f(x) \cdot f(y)] \\
&\overset{\text{Lem 9.8}}{=} \mathop{\mathbb{E}}_{x,y \in_R \{\pm 1\}^n}\Big[\Big(\sum_{S \subseteq [n]} \hat{f}_S \chi_S(x \odot y)\Big)\Big(\sum_{T \subseteq [n]} \hat{f}_T \chi_T(x)\Big)\Big(\sum_{R \subseteq [n]} \hat{f}_R \chi_R(y)\Big)\Big] \\
&\overset{\chi_S(x \odot y) = \chi_S(x)\chi_S(y)}{=} \sum_{S,T,R \subseteq [n]} \hat{f}_S \hat{f}_R \hat{f}_T \mathop{\mathbb{E}}_{x,y \in_R \{\pm 1\}^n}[\chi_S(x) \cdot \chi_S(y) \cdot \chi_T(x) \cdot \chi_R(y)] \\
&\overset{\text{indep.}}{=} \sum_{S,T,R \subseteq [n]} \hat{f}_S \hat{f}_T \hat{f}_R \underbrace{\mathop{\mathbb{E}}_{x \in_R \{\pm 1\}^n}[\chi_S(x)\chi_T(x)]}_{=1 \text{ if } S=T, =0 \text{ o.w.}} \underbrace{\mathop{\mathbb{E}}_{y \in_R \{\pm 1\}^n}[\chi_S(y)\chi_R(y)]}_{=1 \text{ if } S=R, =0 \text{ o.w.}} \\
&= \sum_{S \subseteq [n]} \hat{f}_S^3 \\
&\leq \max_{S \subseteq [n]}\{\hat{f}_S\} \cdot \underbrace{\sum_{S \subseteq [n]} \hat{f}_S^2}_{=\langle f,f \rangle = 1} \\
&\leq \max_{S \subseteq [n]}\{\hat{f}_S\}
\end{aligned}$$

□

9.4 Linearity testing

Recall that a function $f : \{0,1\}^n \to \{0,1\}$ is *linear* over $\mathbb{Z}_2$ if $f(x \oplus y) \equiv_2 f(x) + f(y)$ where $(x \oplus y) \in \{0,1\}^n$ is the vector with $(x \oplus y)_i \equiv_2 x_i + y_i$. Note that we cannot actually expect to test whether f is linear with only $O(1)$ many queries because linearity of f may fail just due to a single entry out of 2^n many entries. So we define a relaxed notion:

Definition 9.11. Let $0 \leq \delta < \frac{1}{2}$. We say that $f : \{0,1\}^n \to \{0,1\}$ is *δ-close to a linear function* if there exists a linear function $g : \{0,1\}^n \to \{0,1\}$ with

$$\Pr_{x \in_R \{\pm 1\}^n}[f(x) = g(x)] \geq 1 - \delta$$

Theorem 9.12 (Linearity test)**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a function with*

$$\Pr_{x,y \in_R \{0,1\}^n}[f(x \oplus y) \equiv_2 f(x) + f(y)] \geq 1 - \delta$$

for $0 \leq \delta < \frac{1}{2}$. Then f is δ-close to a linear function.

Proof. We have already proven this! Note that $f(x \oplus y) \equiv_2 f(x) + f(y)$ is equivalent to $(-1)^{f(x\oplus y)} = (-1)^{f(x)} \cdot (-1)^{f(y)}$. Apply Theorem 9.1 choosing ε so that $\frac{1}{2} + \varepsilon = 1 - \delta$. Then for some $S \subseteq [n]$ there is a function $(-1)^{\sum_{i\in S} x_i}$ so that $\mathbb{E}_{x\in_R\{0,1\}^n}[(-1)^{f(x)}(-1)^{\sum_{i\in S} x_i}] \geq 2\varepsilon$ which means that $\Pr_{x\in_R\{0,1\}^n}[f(x) \equiv_2 \sum_{i\in S} x_i] \geq \frac{1}{2} + \varepsilon = 1 - \delta$. Then $g(x) := \sum_{i\in S} x_i \mod 2$ is the linear function that is δ-close to f. □

Theorem 9.13 (Local decodability)**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a function that is δ-close to the linear function $g : \{0,1\}^n \to \{0,1\}$. Then for each $x \in \{0,1\}^n$ one has*

$$\Pr_{y\in_R\{0,1\}^n}[g(x) \equiv_2 f(y) + f(x \oplus y)] \geq 1 - 2\delta$$

Proof. Fix x. For $y \in_R \{0,1\}^n$, both of the points y and $x \oplus y$ are uniform samples from $\{0,1\}^n$. Hence by assumption and the union bound

$$\Pr[\underbrace{f(y) \equiv_2 g(y) \text{ and } f(y \oplus x) \equiv_2 g(x \oplus y)}_{(*)}] \geq 1 - 2\delta$$

If the event in $(*)$ happens then indeed

$$g(x) \overset{g \text{ linear}}{\equiv_2} g(y \oplus x) + g(y) \equiv_2 f(y \oplus x) + f(y)$$

□

We want to comment on a connection to *coding theory* at this point. Consider the map

$$\mathtt{WH} : \{0,1\}^n \to \{0,1\}^{2^n} \quad \text{with} \quad \mathtt{WH}(u) := (\langle u, x\rangle \mod 2)_{x\in\{0,1\}^n}$$

which is also called the *Walsh Hadamard code*. That means every vector $u \in \{0,1\}^n$ is mapped to the function table of the linear map $x \mapsto \langle u, x\rangle \mod 2$. This code has a few nice properties. First, it is an *error correcting code*, which means that the code words $\mathtt{WH}(u)$ are far from each other:

Lemma 9.14 (Random subset principle)**.** *For $u, v \in \{0,1\}^n$ with $u \neq v$, the codewords $\mathtt{WH}(u)$ and $\mathtt{WH}(v)$ differ in half the coordinates.*

Proof. Fix an index i with $u_i \neq v_i$. Now pick $x \in_R \{0,1\}^n$ at random by first drawing all the bits $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \in_R \{0,1\}$ and only in the second phase drawing x_i. Then conditioning on any outcome from the first phase we have

$$\Pr_{x_i \in_R \{0,1\}}[\langle u,x\rangle \equiv_2 \langle v,x\rangle] = \Pr_{x_i \in_R \{0,1\}}\Big[x_i \equiv_2 \sum_{j \neq i}(v_j - u_j)x_j\Big] = \frac{1}{2}$$

as exactly one of the outcomes $x_i \in \{0,1\}$ makes the event true. □

Note that the lemma means one could corrupt up to a quarter of the bits in $\mathtt{WH}(u)$ adversarially and one could still reconstruct u itself. Along the lines of Theorem 9.13 we know that for every i and every x one has $\mathtt{WH}(u)_x + \mathtt{WH}(u)_{x \oplus e_i} \equiv_2 \langle u,x\rangle + \langle u, x \oplus e_i\rangle \equiv_2 u_i$. This means the code is *locally decodable* as we can reconstruct every bit in u by inspecting $O(1)$ positions in the codeword and this is still possible if a $< \frac{1}{4}$ fraction of the codeword is corrupted.

9.5 Consistency test

Now we discuss the second building block of the PCP verifier which is the consistency test.

Definition 9.15. We call a pair of functions $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^{n^2} \to \{0,1\}$ *consistent* if there exists a vector $u \in \{0,1\}^n$ so that $f(x) \equiv_2 \langle u,x\rangle$ and $g(X) \equiv_2 \langle uu^T, X\rangle$.

The next step will be to show that one can test consistency by evaluating $O(1)$ many random positions in f and g.

```
CONSISTENCY TEST
Input: Access to functions $f : \{0,1\}^n \to \{0,1\}$, $g : \{0,1\}^{n \times n} \to \{0,1\}$.
  (1) Pick independent random $x, y, a, b \in_R \{0,1\}^n$ and $B \in_R \{0,1\}^{n \times n}$.
  (2) Accept if
        $(f(a) + f(a+x)) \cdot (f(b) + f(y+b)) \equiv_2 g(B) + g(B + xy^T)$    (9.2)
```

Note that whenever f and g are linear, the test can be simplified to pick $x, y \in \{0,1\}^n$ at random and accepting if

$$f(x) \cdot f(y) \equiv_2 g(xy^T)$$

But the more complicated version will also work if f and g are merely δ-close to linear functions. To be more precise, if f is close to a linear function $\tilde{f}$ and g is δ-close to a linear function $\tilde{g}$, then by Theorem 9.13 we know that with probability $1 - O(\delta)$, (9.2) is equivalent to $\tilde{f}(x) \cdot \tilde{f}(y) \equiv_2 \tilde{g}(xy^T)$.

Theorem 9.16 (Consistency test)**.** *The following holds:*

(a) If f and g are consistent, then they pass the consistency test with probability 1.

(b) If f and g are δ-close to linear functions and those linear functions are not consistent, then they pass the consistency test with probability at most $3/4 + O(\delta)$.

Proof. For (a). If indeed f and g are consistent, then $f(x) \equiv_2 \langle u, x\rangle$ and $g(X) \equiv_2 \langle uu^T, X\rangle$. Then for any $x, y \in \{0,1\}^n$ one has $g(xy^T) \equiv_2 \langle uu^T, xy^T\rangle \equiv_2 \langle u, x\rangle \cdot \langle u, y\rangle \equiv_2 f(x) \cdot f(y)$.
For (b). To keep the notation simple, we prove the statement for $\delta = 0$, that means we assume that f and g are actually linear. Then $f(x) \equiv_2 \langle u, x\rangle$ and $g(X) \equiv_2 \langle W, X\rangle$ for some $u \in \{0,1\}^n$ and $W \in \{0,1\}^{n\times n}$. Assuming that f and g are not consistent we have $W \neq uu^T$. It suffices to show the following:
Claim. *If $W \neq uu^T$, then* $\Pr_{x,y\in_R\{0,1\}^n}[\langle W, xy^T\rangle \not\equiv_2 \langle u, x\rangle\langle u, y\rangle] \geq \frac{1}{4}$.
Proof of Claim. We imagine to first pick x at random and then in a second phase y. In the first phase we have $\Pr_{x\in_R\{0,1\}^n}[x^T W \not\equiv_2 x^T(uu^T)] \geq \frac{1}{2}$ by the random subset principle from Lemma 9.14 (in fact, even applied to a single column where W and uu^T are different). Now condition on this event and fix x. Then in the second phase $\Pr_{y\in_R\{0,1\}^n}[x^T W y \not\equiv_2 x^T(uu^T)y] \geq \frac{1}{2}$ (since $x^T W \not\equiv_2 x^T(uu^T)$). If this event also happens, then

$$\langle W, xy^T\rangle \equiv_2 x^T W y \not\equiv_2 x^T(uu^T)y \equiv_2 \langle uu^T, yy^T\rangle$$

□

9.6 The satisfiability test

So far we can test whether the proofs f and g are (nearly) linear functions that are consistent and encode an assignment u. Now we come to the actual test whether the encoded assignment u satisfies the QUADEQ instance.

```
SATISFIABILITY TEST
Input: A QUADEQ instance $A_1, \dots, A_m \in \{0,1\}^{n\times n}$, $b_1, \dots, b_m \in \mathbb{Z}_2$. Access to function $g : \{0,1\}^{n\times n} \to \{0,1\}$.
  (1) Pick a random subset $I \subseteq \{1, \dots, m\}$ and a random $B \in \{0,1\}^{n\times n}$.
  (2) Accept if $g(B + \sum_{i\in I} A_i) + g(B) \equiv_2 \sum_{i\in I} b_i$
```

As before, if g happens to be linear then the test simplifies to $g(\sum_{i\in I} A_i) \equiv_2 \sum_{i\in I} b_i$.

Theorem 9.17 (Satisfiability test)**.** *The following holds:*

(a) If $g(X) \equiv_2 \langle X, uu^T \rangle$ and u is satisfying, then the satisfiability test accepts with probability 1.

(b) If g is δ-close to a linear function $\tilde{g}$ with $\tilde{g}(X) = \langle uu^T, X \rangle$ and u is not satisfying, then the satisfiability test accepts with probability at most $\frac{1}{2} + O(\delta)$.

Proof. (a) is simple, so consider (b). Again, to keep the notation clean we prove the case with $\delta = 0$, that means $g(X) \equiv_2 \langle uu^T, X \rangle$ for some $u \in \{0,1\}^n$ that does not satisfy the system of quadratic equations. Then there is at least one index $i \in [m]$ so that $\langle A_i, uu^T \rangle \not\equiv_2 b_i$. Then by the random subset principle, for a uniform random subset $I \subseteq [m]$ we have $\Pr[\sum_{i\in I} \langle A_i, uu^T \rangle \not\equiv_2 \sum_{i \in I} b_i] = \frac{1}{2}$. □

9.7 Proof of the weak PCP Theorem

Now we can put everything together.

Theorem (Weak PCP Theorem — restated)**.** $\mathbf{PCP}(poly(n), 1) \supseteq \mathbf{NP}$.

Proof. We use the following PCP verifier where $O(1)$ refers to a large enough constant.

```
PCP VERIFIER
Input: A QUADEQ instance $A_1, \ldots, A_m \in \{0,1\}^{n\times n}$, $b_1, \ldots, b_m \in \mathbb{Z}_2$. Access to
functions $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^{n\times n} \to \{0,1\}$.
  (1) Run the Linearity Test O(1) times on both $f$ and $g$
  (2) Run the Consistency Test O(1) times
  (3) Run the Satisfiability Test $O(1)$ times
  (4) If any test fails, reject. Otherwise accept.
```

Note that these tests are non-adaptive. If there is a satisfying assignment u, then all tests will pass with probability 1. Now suppose otherwise. If for some small constant δ, the functions f and g are not δ-close to linear functions then we reject in (1) with high probability by Theorem 9.12. So suppose indeed f is δ-close to $\tilde{f}$ and g is δ-close to $\tilde{g}$. If $\tilde{f}$ and $\tilde{g}$ are not consistent then we will reject with high probability in (2) by Theorem 9.16. So suppose that there is a $u \in \{0,1\}^n$ with $f(x) \equiv_2 \langle u, x \rangle$ and $g(X) \equiv_2 \langle uu^T, X \rangle$ while u does not satisfy the quadratic equations. Then we reject in (4) with high probability by Theorem 9.17. □

9.8 Hardness of approximation

We also want to explain why the PCP Theorem is so important. Let us go back to the 3SAT problem and consider a CNF φ with m clauses and n variables. Instead of just considering satisfiability, let us define the *value* $\text{val}(\varphi) \in \{0, \dots, m\}$ as the maximum number of clauses satisfied by any assignment. In other words $\varphi \in \texttt{3SAT} \Leftrightarrow \text{val}(\varphi) = m$. We are interested in how well one can approximate $\text{val}(\varphi)$ in polynomial time. Let $0 < \alpha \leq 1$. We say that an an algorithm is an α-*approximation algorithm* for 3SAT, if on any input φ, it finds an assignment a in polynomial time so that a satisfies at least $\alpha \cdot \text{val}(\varphi)$ many clauses. Similarly we can define approximation algorithms for any other maximization problem.

Actually one can always find a satisfying assignment that satisfies $\frac{7}{8}m$ many clauses (just take a random assignment). On the other hand, the Cook-Levin Theorem says that for any language $L \in \mathbf{NP}$ there is a polynomial time computable function f that produces a 3-CNF formula so that

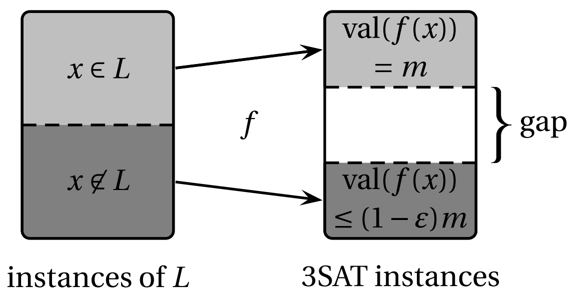
$$\begin{aligned} x \in L \quad &\Rightarrow \quad \text{val}(f(x)) = m \\ x \notin L \quad &\Rightarrow \quad \text{val}(f(x)) \leq m - 1 \end{aligned}$$

But the Cook-Levin Theorem really does not give a stronger guaranteee — maybe in the produced 3SAT formula one can really satisfy all but one clause. So this does not give any useful lower bound on hardness of approximation for 3SAT.

In particular, until the discovery of the PCP theorem it was absolutely plausible that for any $\varepsilon > 0$ there is an algorithm that finds an assignment satisfying $(1-\varepsilon)\text{val}(\varphi)$ many clauses in time $n^{g(\varepsilon)}$ where $g(\varepsilon)$ is some function depending only on the accuracy ε. Such an algorithm is also called a *PTAS (polynomial time approximation scheme)*. PTAS-type algorithms were known for a range of problems such as Knapsack (with one or constantly many constraints) — so why not for 3SAT. But the PCP Theorem provides a stronger reduction than Cook-Levin where a *gap* is created:

Theorem 9.18. *There is a universal constant $\varepsilon > 0$ so that for any $L \in \mathbf{NP}$ there is a polynomial time computable function f that produces a 3-CNF with $m = \text{poly}(|x|)$ clauses so that*

$$\begin{aligned} x \in L \quad &\Rightarrow \quad \textit{val}(f(x)) = m \\ x \notin L \quad &\Rightarrow \quad \textit{val}(f(x)) \leq (1-\varepsilon)m \end{aligned}$$



Proof. By the PCP Theorem (Theorem 9.3) there is a PCP verifier using $r(n) \leq C_L \log(|x|)$ random bits that reads q_L bits from the proof where the definition allows that C_L and q_L depend on the language L. But for any **NP**-complete language L' we know that there is a polynomial time computable function g with $x \in L \Leftrightarrow g(x) \in L'$. So we could have instead used the PCP verifier for the other language L' applied to instance $g(x)$. This tells us that $q_L \leq q$ for some *universal* constant q.

Now back to the main argument. We fix an input x with $n := |x|$ for language L. Let $V^\pi(x,u)$ be the PCP verifier for L. For each fixed u, we can think of $V^\pi(x,u)$ as a function depending on q many bits of the proof π. As we have shown in Claim I of Theorem 2.10 (combined with the reduction in Lemma 2.12), there is a 3SAT formula φ_u with exactly $D \leq q2^q$ clauses so that $V^\pi(x,u) = \varphi_u(\pi)$ for all π. Let $\varphi := \bigwedge_{u \in \{0,1\}^{r(n)}} \varphi_u$ be the AND of all those constant size CNFs. By construction, φ has $m := D2^{r(n)}$ many clauses. We claim the following:

$$\begin{aligned} x \in L \quad &\Rightarrow \quad \mathrm{val}(\varphi) = m \\ x \notin L \quad &\Rightarrow \quad \mathrm{val}(\varphi) \leq m\Big(1 - \frac{1}{2D}\Big) \end{aligned}$$

The first case is clear since for $x \in L$, there is a proof π so that $V^\pi(x,u) = 1 = \varphi_u(\pi)$ for all u. So consider the case $x \notin L$. Fix any proof π. Then for at least half of the choices of $u \in \{0,1\}^{r(n)}$ one has $V^\pi(x,u) = 0$. For each of those u's, at least one of the clauses in φ_u are not satisfied by π. That means at most $m - \frac{1}{2}2^{r(n)} = m - \frac{m}{2D}$ clauses in φ are satisfied. □

One can rephrases this as follows:

Corollary 9.19. *Assuming* **P** $\neq$ **NP**, *there is a universal constant* $\varepsilon > 0$ *so that there is no polynomial time* $(1-\varepsilon)$*-approximation for* 3SAT.

Results as above in Cor 9.19 are called *hardness of approximation*. Using more advanced arguments one can prove that if **P** $\neq$ **NP**, then there is no polynomial time $(\frac{7}{8} + \varepsilon)$ for 3SAT for any constant $\varepsilon > 0$. In other words, there is no polynomial time algorithm that (in the worst case) beats the trivial approximation algorithm. Hardness of approximation is a vast field and the obtained inapproximability ratios are highly problem dependent. For example, assuming **P** $\neq$ **NP**, one can prove that there is no $n^{1-\varepsilon}$-approximation for `INDSET`. That means even though the decision versions of `3SAT` and `INDSET` are equally hard, in terms of approximability, the problems behave very differently.

Chapter 10

Circuit lower bounds

We want to continue in the spirit of Chapter 6 and discuss lower bounds on the size of circuits. Certainly this topic has been a failure of complexity theory. We cannot even prove that any function $f : \{0,1\}^n \to \{0,1\}$ belonging to an **NP**-problem requires circuits of size $\omega(n)$ and in order to prove that $\mathbf{P} \neq \mathbf{NP}$ one would need lower bounds of the form $n^{\omega(1)}$. In this chapter we want prove lower bounds for monotone circuits and then discuss why lower bounds on general circuits seem to be so difficult to obtain.

10.1 Lower bounds for monotone circuits

Consider the function $\texttt{CLIQUE}_{k,n} := \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ where $\texttt{CLIQUE}_{k,n}(x) = 1$ if the graph $G := ([n], \{\{i,j\} \mid x_{ij} = 1\})$ contains a clique of size k. By a slight abuse of notation we will also write $\texttt{CLIQUE}_{n,k}(G)$. We recall that a *clique* in an undirected graph $G = (V,E)$ is a subset $S \subseteq [n]$ so that for all distinct $i,j \in S$ one has $\{i,j\} \in E$. By Theorem 2.13 we know that the uniform variant of $\texttt{CLIQUE}_{k,n}$ is **NP**-hard and certainly we expect that any circuit for $\texttt{CLIQUE}_{k,n}$ must have super polynomial size – just that we cannot prove that at this moment. Recall that for a general circuit we allow gates $\vee, \wedge, \neg$ with fan-in 1 and 2, resp. We can also see that the function $\texttt{CLIQUE}_{k,n}$ is monotone as adding edges to a graph cannot destroy an existing k-clique. In general we define:

Definition 10.1. A function $f : \{0,1\}^n \to \{0,1\}$ is called *monotone* if for all $x,y \in \{0,1\}^n$ one has $x \leq y \Rightarrow f(x) \leq f(y)$.

Here we write $x \leq y$ for vectors $x,y \in \{0,1\}^n$ if coordinate-wise $x_i \leq y_i$ for all $i = 1,\dots,n$. Any monotone function f can be written in the form

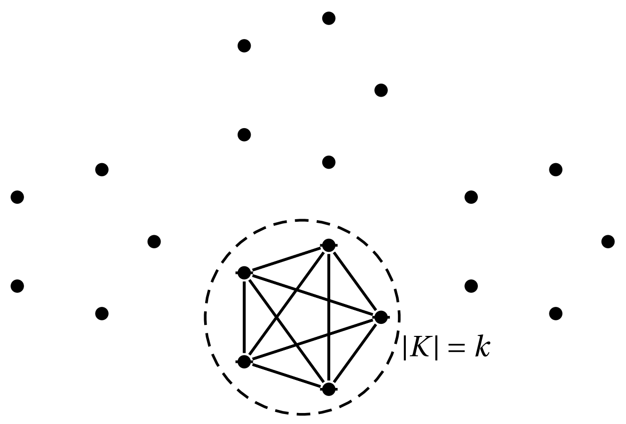
$$f(x) = \bigvee_{y \in \{0,1\}^n : f(y)=1} \Big(\bigwedge_{i \in [n] : y_i = 1} x_i \Big)$$

In other words, f can be computed with a *monotone circuit* which is a circuit that only contains $\vee$ and $\wedge$ gates. Note that in monotone circuit, all the functions computed at non-output gates are monotone as well. But is it true that for a monotone function there is also a *minimum* size circuit that happens to be monotone? Or maybe a polynomial blowup suffices to make any circuit for a monotone function also monotone? The answer is "no" in both cases, but these questions were open before the work that we present here. For the remainder of this chapter we will prove that (for a suitable choice of $k := k(n)$), any monotone circuit for $\texttt{CLIQUE}_{k,n}$ must have exponential size.

10.1.1 Two distributions over inputs

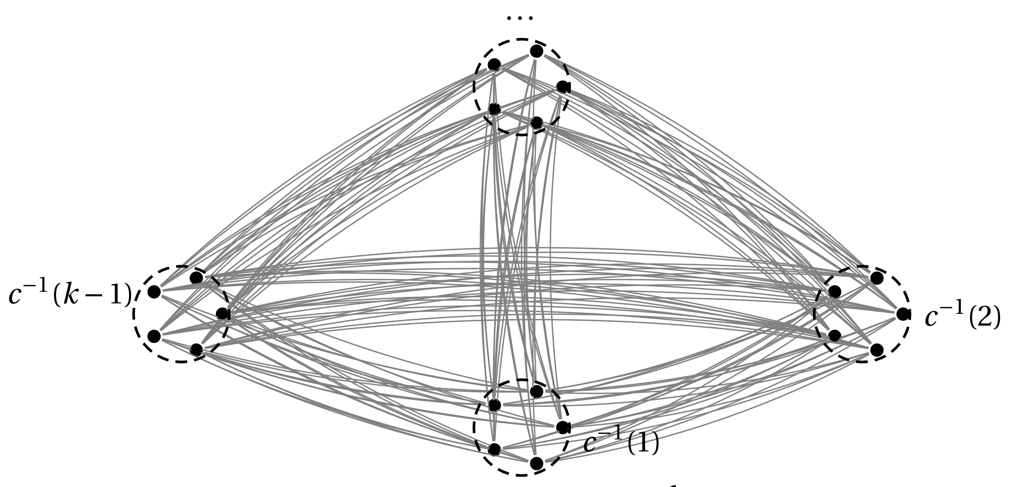
Depending on parameters $1 \le k \le n$, we will define two distributions $\mathcal{Y}$ and $\mathcal{N}$ over graphs and then prove that any small monotone circuit will not be able to correctly decide $\texttt{CLIQUE}_{k,n}(\mathcal{Y})$ and $\texttt{CLIQUE}_{k,n}(\mathcal{N})$.

Definition 10.2. The distribution $\mathcal{Y}$ over n-vertex graphs is as follows: Pick a uniform set $K \subseteq [n]$ with $|K| = k$ and output the graph G that has a clique on K but no other edges.



graph sampled from $\mathcal{Y}$

Definition 10.3. The distribution $\mathcal{N}$ over n-vertex graphs is as follows: Pick a function $c : [n] \to [k-1]$ at random and insert an edge $\{u, v\}$ if $c(u) \neq c(v)$.



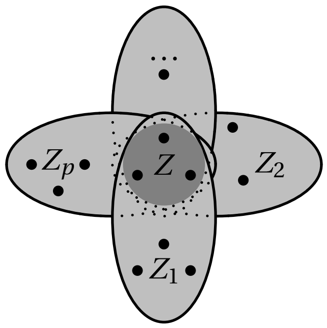
graph sampled from $\mathcal{N}$

Intuitively, the distribution $\mathcal{Y}$ gives a graph with a minimal number of edges so that always $\texttt{CLIQUE}_{k,n}(\mathcal{Y}) = 1$ and the distribution $\mathcal{N}$ gives a graph with an approximately maximal number of edges so that that $\texttt{CLIQUE}_{k,n}(\mathcal{N}) = 0$. We note that with high probability, graphs drawn according to $\mathcal{N}$ have a lot of cliques of size $k-1$.

10.1.2 Two combinatorial lemmas

We need a combinatorial lemma that says that in a large enough set system one can find many sets that have the same pairwise intersection.

Lemma 10.4 (Sunflower Lemma — Erdős, Rado 1960)**.** *Let $\mathcal{F} \subseteq 2^X$ be family of sets over a groundset X so that $|S| \le \ell$ for all $S \in \mathcal{F}$ and $|\mathcal{F}| > (p-1)^\ell \ell!$ for some $p \in \mathbb{N}$. Then there are sets $Z_1, \dots, Z_p \in \mathcal{F}$ and $Z \subseteq X$ so that $Z_i \cap Z_j = Z$ for all $i \neq j$.*



Proof. We prove the claim by induction over ℓ. For $\ell = 1$, all sets are disjoint and the condition $|\mathcal{F}| > p-1$ clearly suffices. Now suppose $\ell > 1$. Let $\mathcal{M} \subseteq \mathcal{F}$ be an inclusion-wise maximal set of disjoint sets. If $|\mathcal{M}| \geq p$ then we are done (with $Z := \emptyset$). So suppose $|\mathcal{M}| < p$. The union $U := \bigcup_{S \in \mathcal{M}} S$ of elements in $\mathcal{M}$ has size $|U| \leq (p-1)\ell$ and by maximality, each set in $\mathcal{F}$ contains at least one element in U. Hence by averaging, there is an element $u \in U$ that is in at least $\frac{|\mathcal{F}|}{|U|} \geq \frac{(p-1)^{\ell}\ell!}{(p-1)\ell} = (p-1)^{\ell-1}(\ell-1)!$ many sets. Let $\mathcal{F}' := \{S \setminus \{u\} : S \in \mathcal{F} \text{ and } u \in S\}$ be those sets after deleting the element u. Note that sets in $\mathcal{F}'$ have cardinality at most $\ell - 1$. By induction we can find a sunflower $Z_1, \ldots, Z_p \in \mathcal{F}'$ and $Z_1 \cup \{u\}, \ldots, Z_p \cup \{u\}$ is a sunflower for the original set system $\mathcal{F}$. □

The *birthday paradox* usually refers to the fact that if one has significantly more than $\sqrt{365}$ random people in room, then likely some of them will have the same birthday. We can also get a reverse bound. Recall that a function $c : A \to B$ is *injective* if for all distinct $i, j \in A$ one has $c(i) \neq c(j)$. For $I \subseteq A$ we also denote $c_{|I} : I \to B$ as the *restriction* to I.

Lemma 10.5 (Birthday bound). *Let $1 \leq \ell \leq k$. Then a random function $c : [\ell] \to [k]$ is injective with probability at least $1 - \frac{\ell^2}{2k}$. Moreover, for any subset $I \subseteq [\ell]$ one also has $\Pr[c \text{ injective} \mid c_{|I} \text{ injective}] \geq 1 - \frac{\ell^2}{2k}$.*

Proof. Let X be the random variable that gives the number of *collisions* (i.e. the number of pairs $\{i, j\}$ with $c(i) = c(j)$). Then using linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{\ell} \sum_{j=i+1}^{\ell} \underbrace{\Pr[c(i) = c(j)]}_{=1/k} \leq \frac{\ell^2}{2k}$$

Note that $\Pr[X \geq 1] \leq \mathbb{E}[X]$ by Markov inequality which gives the first claim. The moreover part follows along the same lines just that for $i, j \in I$ one has $\Pr[c(i) = c(j)] = 0$ since we condition on not having a collision in I. □

10.1.3 Approximating CLIQUE by Clique Indicators

For a graph $G = ([n], E)$ and set set $S \subseteq [n]$ we define define the *clique indicator*

$$C_S(G) = \begin{cases} 1 & \text{if } S \text{ is a clique in } G \\ 0 & \text{otherwise} \end{cases}$$

Note that $\texttt{CLIQUE}_{k,n}(G) = \sum_{S \subseteq [n] : |S| = k} C_S$ which is a natural way to write $\texttt{CLIQUE}_{k,n}$ as a monotone circuit. Assuming for the sake of contradiction that there is a monotone circuit for $\texttt{CLIQUE}_{k,n}$ of (not too large) size s, we can approximate all the intermediate functions computed by the circuit with clique indicators — at least on instances drawn from $\mathcal{Y}$ and $\mathcal{N}$.

Theorem 10.6 (Razborov, Andreev, Alon, Boppana 1985)**.** *For $1 \leq k \leq n^{1/4}$ with k large enough, there is no monotone circuit of size $2^{\sqrt{k}}$ for $\texttt{CLIQUE}_{k,n}$.*

Proof. We fix large enough parameters k and n with $k \leq n^{1/4}$ and consider a monotone circuit H of size $s < 2^{\sqrt{k}}$. We will actually prove the stronger statement that H will not be able to correctly distinguish the two distributions $\mathcal{Y}$ and $\mathcal{N}$, i.e. we will prove that

$$\Pr_{G \in_R \mathcal{Y}}[H(G) \neq \texttt{CLIQUE}_{k,n}(G)] + \Pr_{G \in_R \mathcal{N}}[H(G) \neq \texttt{CLIQUE}_{k,n}(G)] \geq 0.8$$

For a large constant D, we define $\ell := \frac{\sqrt{k}}{D}$, $p := D\sqrt{k}\log(n)$ and $m := (p-1)^{\ell}\ell!$. Then $\ell \cdot p \leq n^{1/7}$ for k large enough and so $m \leq n^{\ell/7} \leq n^{\sqrt{k}}$. We say that a function $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ is an (m,ℓ)*-function* if there are sets $S_1, \ldots, S_m \subseteq [n]$ with $|S_i| \leq \ell$ so that[1]

$$f(G) = \bigvee_{i=1}^{m} C_{S_i}(G)$$

Our goal is prove that all functions computed at the s many gates of the circuit H can be well approximated by (m,ℓ)-functions which then in particular applies to the output H itself. When combining two (m,ℓ)-functions we will have the issue that the resulting function has either too many sets or their size exceeds ℓ. We first develop tools to approximate such functions again with (m,ℓ)-functions.

Claim I. *Given an (m^2,ℓ)-function $h = \bigvee_{i=1}^{m^2} C_{Z_i}$, there is an (m,ℓ) function $\tilde{h}$ so that (i) for every graph G one has $\tilde{h}(G) \geq h(G)$ and (ii) $\Pr_{G \in_R \mathcal{N}}[\tilde{h}(G) > h(g)] \leq \frac{1}{10s}$.*

Proof of Claim I. Suppose h is defined by more than m different sets, otherwise there is nothing to prove. By the Sunflower Lemma (Lemma 10.4) there are indices $i_1, \ldots, i_p$ so that $Z_{i_1}, \ldots, Z_{i_p}$ have the common intersection of Z. Replace $Z_{i_1}, \ldots, Z_{i_p}$ in the definition with Z and denote the outcome h'. We will first analyze h' — the desired function $\tilde{h}$ follows by repeating the argument at most m^2 times. For any graph G, if any of the sets Z_{i_j} are a clique, then also Z is a clique and so $h'(G) \geq h(G)$ holds. For (ii) it suffices to prove that $\Pr_{G \in_R \mathcal{N}}[\forall j \in [p] : Z_{i_j}$ not is clique $\mid Z$ is clique$] \leq \frac{1}{10m^2 s}$. Recall that a sample from $\mathcal{N}$ is generated using a uniform random function $c : [n] \to [k-1]$. We condition on the even that

[1]We allow dublicate sets so that we can get exactly m sets instead of at most m many.

Z is a clique, i.e. $c_{|Z}$ is injective. Then

$$\begin{aligned} & \Pr\left[\forall j \in [p] : c_{|Z_{i_j}} \text{ not injective} \mid c_{|Z} \text{ injective}\right] \\ \overset{(*)}{=} \quad & \prod_{j=1}^{p} \Pr\left[c_{|Z_{i_j}} \text{ not injective} \mid c_{|Z} \text{ injective}\right] \\ \overset{\text{Lem 10.5}}{\leq} \quad & \left(\frac{\ell^2}{k-1}\right)^p \leq 2^{-p} = n^{-D\sqrt{k}} \leq \frac{1}{10m^2 s} \end{aligned}$$

here we use in $(*)$ that $Z_{i_1} \setminus Z, \dots, Z_{i_p} \setminus Z$ are disjoint so that the events are independent. □

Claim II. *Let $m' \leq m^2$. Given a (m',∞)-function $h = \bigvee_{i=1}^{m'} C_{Z_i}$, there is an (m',ℓ)-function $\tilde{h}$ so that (i) for every graph G one has $\tilde{h}(G) \leq h(G)$ and (ii) $\Pr_{G\in\mathcal{Y}}[\tilde{h}(G) < h(G)] \leq \frac{1}{10s}$.*

Proof of Claim II. Let $\tilde{h}$ be the function obtained by dropping all sets Z with $|Z| > \ell$. Then trivially $\tilde{h}(G) \leq h(G)$ for every graph which gives (i). For (ii), recall that a graph $G \in_R \mathcal{Y}$ contains a single randomly placed k-clique. So it suffices to prove that for any fixed set Z with $|Z| > \ell$, it is extremely unlikely that Z is a clique. And indeed,

$$\begin{aligned} \Pr_{G\in_R\mathcal{Y}}[C_Z(G) = 1] &= \Pr_{|K|=k}[Z \subseteq K] \leq \frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \ldots \cdot \frac{k-(\ell-1)}{n-(\ell-1)} \\ &\leq \left(\frac{k}{n}\right)^{\ell} \overset{k\leq n^{1/4}}{\leq} n^{-0.75\ell} \leq \frac{1}{10m^2 s} \end{aligned}$$

Next, we will prove that if f and g are two (m,ℓ)-functions then we can approximate $f \vee g$ and $f \wedge g$ again with (m,ℓ)-functions. For this purpose we will introduce two operations $f \sqcup g$ and $f \sqcap g$.

> *Operation $f \sqcup g$.* Consider two (m,ℓ)-functions $f = \bigcup_{i=1}^m C_{S_i}$ and $g = \bigcup_{j=1}^m C_{T_j}$. Consider $h := (\bigvee_{i=1}^m C_{S_i}) \vee (\bigvee_{j=1}^m C_{T_j})$ which is a $(2m,\ell)$-function. Then apply Claim I to h and return the obtained (m,ℓ)-function.

Now we define the second operation:

> *Operation $f \sqcap g$.* Consider two (m,ℓ)-functions $f = \bigcup_{i=1}^m C_{S_i}$ and $g = \bigcup_{j=1}^m C_{T_j}$. Define the $(m^2, 2\ell)$-function $h := \bigvee_{i,j\in[m]} C_{S_i\cup T_j}$. First apply Claim II to approximate h with a (m^2,ℓ)-function h'. Then apply Claim I to approximate h' by a (m,ℓ)-function h'' which we return.

We summarize the properties of these two functions.
Claim III. *Let f and g be two (m,ℓ)-functions. Then for both combinations $(h,\tilde{h}) = (f \vee g, f \sqcup g)$ and $(h,\tilde{h}) = (f \wedge g, f \sqcap g)$ one has*

$$\Pr_{G \in_R \mathcal{Y}}[\tilde{h}(G) \neq h(G)] \leq \frac{1}{10s} \quad \text{and} \quad \Pr_{G \in_R \mathcal{N}}[\tilde{h}(G) \neq h(G)] \leq \frac{1}{10s}$$

Now we can prove the main statement. Consider the functions $f_1,\ldots,f_s$ computed at the gates of the circuit H in topological order, i.e. each f_i depends only on $f_1,\ldots,f_{i-1}$ and $f_s = H$. We construct a sequence $\tilde{f}_1,\ldots,\tilde{f}_s$ of (m,ℓ)-functions where each $\tilde{f}_i$ is an approximation to $\tilde{f}_i$. For an input gate i that corresponds to variable $x_{\{u,v\}}$ we set $\tilde{f}_i := f_i := C_{\{u,v\}}$ (which trivially is an (m,ℓ)-function). Now consider a non-input gate i and suppose it is a $\vee$ gate depending on gates i_1, i_2, i.e. $f_i = f_{i_1} \vee f_{i_2}$. Then we set $\tilde{f}_i := \tilde{f}_{i_1} \sqcup \tilde{f}_{i_2}$; similar if $f_i = f_{i_1} \wedge f_{i_2}$ then we set $\tilde{f}_i := \tilde{f}_{i_1} \sqcap \tilde{f}_{i_2}$. Then by Claim III, the error that we make in each step is at most $\frac{1}{10s}$ for each of the distributions $\mathcal{Y}$ and $\mathcal{N}$ and so $\Pr[\tilde{f}_s(G) \neq f_s(G)] \leq \frac{1}{10}$ if either $G \in_R \mathcal{Y}$ or $G \in_R \mathcal{N}$. Then the following shows that it is impossible that $H = \texttt{CLIQUE}_{k,n}$.
Claim IV. *For any (m,ℓ)-function f one has either* $\Pr_{G \in_R \mathcal{Y}}[f(G) = 0] \geq 0.99$ *or* $\Pr_{G \in_R \mathcal{N}}[f(G) = 1] \geq 0.99$.
Proof of Claim IV. Write $f = \bigcup_{i=1}^{m'} C_{Z_i}$ with $m' \leq m$. If $m' = 0$ then $f(G) = 0$ for all graphs G. Otherwise, we know that $|Z_1| \leq \ell$ and then $\Pr_{G \in \mathcal{N}}[Z_1$ is clique in $G] \geq 1 - \frac{\ell^2}{k-1} \geq 0.99$ if D is large enough. □

We leave it as an exercise to the reader to prove that there is indeed a *non-monotone circuit* H of size poly(n) so that $H(G) = 1$ for all G drawn from $\mathcal{Y}$ while $H(G) = 0$ for all $G \in_R \mathcal{N}$.

10.2 Natural proofs

We also want to provide some evidence as to why it might be difficult to prove general circuit lower bounds.

10.2.1 Introduction

As in Chapter 6, we use size(f) to denote the minimum number of gates of a circuit that computes a function $f : \{0,1\}^n \to \{0,1\}$. Now consider an explicit function $f : \{0,1\}^n \to \{0,1\}$ — for example f could be the function `CLIQUE` — and let us speculate how a circuit lower bound proof for f could work. A natural approach would be to come up with a simple proxy μ that approximately measures circuit complexity, i.e. for any function $g : \{0,1\}^n \to \{0,1\}$ it assigns a value $\mu(g)$ where a large value of $\mu(g)$ means that g has a high circuit complexity. Such a proxy is also called a *formal complexity measure*. The issue is that one can prove that such a formal complexity measure (with some additional natural properties) cannot exist, assuming a well established cryptographic conjecture!

Theorem 10.7. *Consider a formal complexity measure μ that for all large enough n satisfies:*

(a) Small for input functions: $\mu(x_i) \leq 1$ and $\mu(\bar{x}_i) \leq 1$

(b) Small growth: $\mu(f \wedge g) \leq \mu(f) + \mu(g)$, $\mu(f \vee g) \leq \mu(f) + \mu(g)$, $\mu(\bar{f}) \leq \mu(f) + 1$.

(c) Largeness: There is a function $f : \{0,1\}^n \to \{0,1\}$ with $\mu(f) \geq n^{\omega(1)}$

(d) Constructiveness: Given the function table of a function $g : \{0,1\}^n \to \{0,1\}$ one can compute $\mu(g)$ in time $2^{O(n)}$.

Assuming Conjecture 5, such a complexity measure μ does not exist.

We have not yet introduced Conjecture 5, but we will get to that. Note that any μ satisfying $(a) + (b)$ gives a lower bound of $\text{size}(f) \geq \mu(f)$ for all functions f. Moreover, setting $\mu(f) := \text{size}(f)$ satisfies $(a) + (b) + (c)$[2], just that (d) would violate Conjecture 5.

10.2.2 Pseudo-random functions

Our result is going to be conditional on the existence of *pseudorandom functions* which we will formalize now. As in Section 3.2 we write A^g for an oracle Turing machine that has oracle access to a function $g : \{0,1\}^* \to \{0,1\}$. We abbreviate $\mathcal{F}_n := \{f \mid f : \{0,1\}^n \to \{0,1\}\}$ as all boolean functions on n variables.

Definition 10.8. A function $\delta : \mathbb{N} \to [0,1]$ is called *neglibile* if for all $c > 0$ there is an $n_0 \in \mathbb{N}$ so that $\delta(n) < n^{-c}$ for all $n \geq n_0$.

In other words, a function is negligible if it is super-polynomially small, i.e. $\delta(n) \leq n^{-\omega(1)}$. Then we will rely on the following conjecture:

Conjecture 5 (Existence of subexponentially-secure pseudorandom functions)**.** *There is a universal constant $\varepsilon > 0$ and a family of functions $\{f_s\}_{s \in \{0,1\}^*}$ where*[3] *$f_s : \{0,1\}^{|s|} \to \{0,1\}$ so that*

[2]Note that provably a random function f satisfies (c).

[3]The textbook defines pseudo random functions to be in the form $f_s : \{0,1\}^{|s|} \to \{0,1\}^{|s|}$, i.e. the output length equals the input length. In our application we only need a single bit and so we can use a simpler (and weaker) definition.

1. *There is a polynomial time Turing machine that on input of (s,x) with strings $s \in \{0,1\}^*$ and $x \in \{0,1\}^{|s|}$, computes $f_s(x)$ in polynomial time.*

2. *For any probabilistic 2^{n^ε}-time oracle Turing machine A, the quantity*

$$\delta(n) := \left| \Pr_{s \in_R \{0,1\}^n}[A^{f_s}(1^n) = 1] - \Pr_{g \in_R \mathcal{F}_n}[A^g(1^n) = 1] \right|$$

is negligible.

In other words, if we consider a Turing machine A that has only subexponential running time, then A cannot distinguish between a true random function g generated with 2^n random bits and a function f_s that is generated with only n random bits (in the form of $s \in_R \{0,1\}^n$). It is known that one could construct such a family $\{f_s\}_{s\in\{0,1\}^*}$ of pseudorandom functions from a *one way function*, which is a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ for which collisions are computationally difficult to find.

We also want to emphasize that there is no information-theoretic problem to design a distinguisher breaking Conj 5. The algorithm A may depend on the family $\{f_s\}_{s\in\{0,1\}^*}$ and for input length n, there are only 2^n pseudorandom functions while the number of actual random functions is $|\mathcal{F}_n| = 2^{2^n}$. If running time does not matter, then after checking $O(n)$ random inputs we know with high probability whether the function represented by the oracle is of the form f_s or purely random.

We have not covered pseudo random generators in this course and we want to give the reader some intuition why they are useful. Also we want to put Conj 5 into context with concepts that we already know. The next theorem is not needed in the context of natural proofs and can be skipped if desired.

Theorem 10.9. *Conj 5* $\Rightarrow$ **BPP** $\subseteq$ **DTIME**$(2^{(\log n)^{O(1)}})$.

Proof. Let $L \in$ **BPP**. Consider a probabilistic Turing machine M with running time at most n^c for some constant $c \geq 1$ and fix the constant $0 < \varepsilon < 1$ from Conj 5. Let x be an input and set $n := |x|$.

We choose a parameter $k \in \mathbb{N}$ so that $n^c \ll 2^{k^\varepsilon}$, for example $k := (2c\log(n))^{1/\varepsilon}$ will work. Then we draw $s \in_R \{0,1\}^k$ uniformly at random. Note that one can interpret f_s as vector with 2^k many pseudo random bits. In particular $2^k > n^c$ so this is more than enough random bits for the algorithm M. Since M has running time $n^c < 2^{k^\varepsilon}$ we may conclude from Conj 5 that

$$\left| \Pr_{s \in_R \{0,1\}^k}[M^{f_s}(x) = 1] - \Pr_{r \in_R \{0,1\}^{n^c}}[M^r(x) = 1] \right| \leq \frac{1}{n^{\omega(1)}}$$

That means M^{f_s} is again a **BPP**-type algorithm (with marginally worse success probability) but M^{f_s} uses only $k = (2c\log(n))^{1/\varepsilon}$ many random bits. We can make that algorithm deterministic by trying out all 2^k possibilities. □

10.2.3 Non-existence of natural proofs

We will now describe the argument by Razborov and Rudich that *natural proofs* do not exist. For this, we will not directly prove Theorem 10.7 using formal complexity measures but we use a "binary" variant. Recall that a *predicate* $\mathcal{P}$ *on boolean functions* assigns a value $\mathcal{P}(f) \in \{0,1\}$ to all boolean functions $f : \{0,1\}^n \to \{0,1\}$.

Definition 10.10. A predicate $\mathcal{P}$ on boolean functions is called a *natural proof predicate with parameter* $T(n)$ if for all n large enough the following holds:

- *Usefulness.* $\mathcal{P}(g) = 0$ for all $g : \{0,1\}^n \to \{0,1\}$ with $\text{size}(g) \leq T(n)$.
- *Largeness.* One has $\Pr[\mathcal{P}(g) = 1] \geq \frac{1}{n}$ for a uniform random function $g : \{0,1\}^n \to \{0,1\}$.
- *Constructiveness.* With access to the function table of g one can compute $\mathcal{P}(g)$ in time $2^{O(n)}$.

Theorem 10.11 (Razborov, Rudich 1994)**.** *Assuming Conjecture 5, there is a constant $c > 0$ so that there is no natural proof predicate $\mathcal{P}$ with parameter $T(n) = n^c$.*

Proof. Consider a family of pseudo-random functions $\{f_s\}_{s\in\{0,1\}^*}$ according to Conj 5 that is safe against a 2^{n^ε}-time adversary. Draw $s \in_R \{0,1\}^n$ at random and draw a uniform random function $f : \{0,1\}^n \to \{0,1\}$. Fix a choice of $h \in \{f_s, f\}$. We will build an algorithm that given access to h decides whether $h = f_s$ or $h = f$.

Set $m := n^{\varepsilon/2}$ and consider the function $g : \{0,1\}^m \to \{0,1\}$ with $g(x) := h(x0^{n-m})$. Assume for the sake of contradiction that there is a natural proof predicate with $T(n) = n^c$. Then the value $\mathcal{P}(g)$ can be computed in time $2^{O(m)} = 2^{O(n^{\varepsilon/2})}$. We will prove that $\mathcal{P}$ is the algorithm that contradicts Conj 5.

Claim I (Pseudo random case). *If $g = f_s$, then $\mathcal{P}(h) = 0$.*

Proof of Claim I. By assumption, the function f_s can be computed in time $\text{poly}(n)$ and so the restriction h can be computed by a circuit of size $\text{poly}(n)$ which is of the form m^c for some (large) constant $c > 0$. Hence $\mathcal{P}(h) = 0$ by the usefulness assumption. □

Claim II. (Uniform random case). *If $g = f$, then* $\Pr[\mathcal{P}(h) = 1] \geq \frac{1}{m}$.

Proof of Claim II. In this case also the restriction h is a uniform random function and so the claim follows by the Largeness part of Def 10.10. □

Note that this argument rules out natural proofs with parameter n^c where c is some large constant depending on the constant ε in the pseudorandom assumption and on the exponent in the running time needed to evaluate the functions f_s. If one wanted to make the stronger assumption that for any $\delta > 0$, there are pseudorandom families that are safe against $2^{n^{1-\delta}}$-time adversaries and that can be computed in time $O(n^{1+\delta})$, then presumably one can rule out natural proofs with parameter $n^{1+\gamma}$, for any constant $\gamma > 0$.

10.2.4 Non-existence of formal complexity measures

Finally, we will prove that also formal complexity measure, satisfying the properties listed in Theorem 10.7 do not exist, assuming Conj 5. Here the main technical difference is that in the setting Theorem 10.7 we are only guaranteed the existence of a single hard function while Theorem 10.11 requires that a substantial fraction of random functions are hard.

Theorem 10.7. Assume n is large enough and fix the exponent $c > 0$ as in Theorem 10.11. Fix the formal complexity measure μ and let $f : \{0,1\}^n \to \{0,1\}$ be a function so that $\mu(f) \geq 8n^c$. Let $g : \{0,1\}^n \to \{0,1\}$ be a uniform random function. We use $\oplus$ as the *exclusive or*. Then

$$f = (f \oplus g) \oplus g = (\overline{(f \oplus g)} \wedge g) \vee ((f \oplus g) \wedge \bar{g})$$

and so

$$\mu(f) \leq \mu(\overline{f \oplus g}) + \mu(g) + \mu(f \oplus g) + \mu(\bar{g})$$

But each of those function $\overline{f \oplus g}$, g, $f \oplus g$, $\bar{g}$ is again a uniform random function and at least one of them needs to have a μ-value of at least $\frac{\mu(f)}{4} \geq 2n^c$. Overall we can conclude that $\Pr_g[\mu(g) \geq 2n^c] \geq \frac{1}{4}$ where g is a uniform random function. So we define the predicate $\mathcal{P}$ by letting

$$\mathcal{P}(h) := \begin{cases} 1 & \text{if } \mu(h) \geq 2n^c \\ 0 & \text{otherwise} \end{cases}$$

As for any boolean function we have $\mu(h) \leq \text{size}(h)$, the claim then follows by Theorem 10.11. □