SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" University of Rome
Faculty of Information Engineering, Informatics and Statistics
Department of Computer Science

Computational Complexity

Lecture notes integrated with the book "Computational Complexity: a modern approach", S. Arora, B. Barak

Author
Simone Bianco

November 10, 2024

Contents

Information and Contacts

Personal notes and summaries collected as part of the *Computational Complexity* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Sufficient knowledge of computability theory and algorithm complexity

Licence:

1

Introduction on computation

1.1 Turing Machines

Throughout history, humans have been solving problems through a wide variety of models capable of computing valid results, ranging from their intellect to mechanical devices capable of solving problems. In particular, a computation made by a model can be described as a list of sequential operations and initial conditions that will always yield the same result each time the computation is executed.

In modern mathematics, this idea is formalized through the concept of **algorithm**, a finite list of unambiguous instructions that, given some set of initial conditions, can be performed to compute the answer to a given problem. Even though this is a straightforward definition, it isn't as "mathematically stable" as it seems: each computational model could have access to a different set of possible operations, meaning that the same problem could be solved by different computational models in various ways. This innate nature of computational models makes life difficult for mathematicians, who want to prove results that are as general as possible. In 1950, Alan Turing defined the now-called **Turing machine**, an abstract machine capable of capturing the concept of computation itself through simple — but sufficient — operations.

A Turing machine is made of:

- A finite number of *tapes*, each divided into cells. Each cell contains a symbol from a finite set called *alphabet*, usually assumed to contain only 0 and 1, or a special symbol $\sqcup$, namely the *blank character*. The tape is finite on the left side but infinite on the right side. We assume that there always are a read-only *input tape* and a read-write *output tape*. The other tapes are called *work tapes*.
- A finite number of independent *read-write heads*, one for each tape, capable of reading and writing symbols on the tapes. The heads are always positioned on a single cell of their tape and can shift left and right only one cell per shift.

- A finite set of *states* that can be assumed by the machine. At all times the machine only knows its current state. The set contains at least one state that is capable of immediately halting the machine when reached (such states could be unreachable, making the machine go in an infinite loop).
- A finite set of *instructions* which, given the current state and the current cells read by the read-write heads, dictate how the machine behaves. Each instruction tells the machine to do three things: replace the symbols of the current cells (which can be replaced with themselves), move each head one cell to the left, one cell to the right or stay in place, while also moving from the current state to a new one (which can be the current state itself).

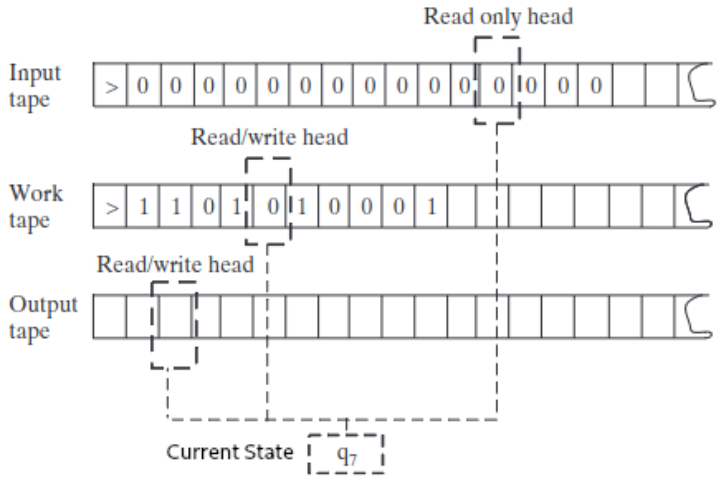


Figure 1.1: Representation of a Turing machine computation step

Definition 1

A k-tape Turing machine is a 7-uple $M = (Q, F, \Gamma, \Sigma, q_{\text{start}}, \delta)$ where:

- Q is a finite set of states, $F \subseteq Q$ is a finite set of halting states and $q_{\text{start}} \in Q$ is the initial state taken by the machine.
- Γ is a finite set of symbols, usually called the tape alphabet. The tape alphabet always contains the symbols $>$, i.e the start of tape symbol, and $\sqcup$, i.e. the blank tape cell symbol.
- Σ is a finite set of symbols, usually called the input alphabet, where $\Sigma \subseteq \Gamma - \{<, \sqcup\}$. The input string can be formed only of these characters.
- $\delta : (Q - F) \times \Gamma^k \to Q \times \Gamma^k \times \{\mathsf{L}, \mathsf{R}\}^k$ is a partial function, usually called the transition function, where L and R represent a left or right shift of the read-write head. Intuitively, if $\delta(q, a_1, \ldots, a_k) = (p, b_1, \ldots, b_k, X_1, \ldots, X_k)$, where $X_i \in \{\mathsf{L}, \mathsf{R}\}$, then, when the machine is in state q and it reads the symbols $a_1, \ldots, a_k$ on the current cells of the tapes, it transitions to the state p, replaces the symbols with $b_1, \ldots, b_k$ and moves the heads in the directions $X_1, \ldots, X_k$.

Given an input $x \in \Sigma^*$, we denote the output of the computation as $M(x)$. By definition, nothing prevents the computation made by a Turing machine to go into infinite loops, thus never halting. The *Church-Turing thesis* defines computation in terms of non-looping Turing machines: any problem that is computable can be solved by **a Turing machine that always halts**, i.e. returns a solution for every possible input.

Definition 2: Computable function

Given a function $f : \Sigma^* \to \Sigma^*$ and a Turing machine M that always halts, we say that M **computes** f if $\forall x \in \Sigma^*$ it holds that $M(x) = f(x)$.

Due to this definition, any Turing machine can be viewed as a function $M : \Sigma^* \to \Sigma^*$. For example, consider the following function:

$$\text{PALINDROME}(x) = \begin{cases} 1 & \text{if } x \text{ is a palindrome string} \\ 0 & \text{if } x \text{ is not a palindrome string} \end{cases}$$

To compute this function, we define the following 3-taped Turing machine M, made of an input tape, a work tape and an output tape:

M = "Given the input string x:

1. Copy the string x from the input tape to the work tape.
2. Move the head of the input tape to the first cell and the head of the work tape to the last written cell.
3. While moving the input tape forward and the work tape backwards, check if the current cell of both heads contains the same symbol. If one pair of different cells is found, write 0 on the output tape and halt the computation.
4. If the input tape head reached the last written cell and the work tape head reached the first cell, write 1 on the output tape and halt the computation"

Clearly, a lot of computational models, such as the modern computer, are more powerful than a Turing machine. First, we have to give a proper definition of *"power"*. In the context of computability theory, we are interested in studying the maximum amount of resources needed by a computational model to compute an answer to the problem they are designed for. In the case of Turing machines, we are interested in **running time** and **required space**.

Definition 3: Running Time and Required Space

Given a TM M that halts on all inputs, we define the **running time** and the **required space** respectively as the functions $T, S : \mathbb{N} \to \mathbb{R}^+$ such that $T(n)$ as the number of transitions of δ executed by M for an input of length n, while $S(n)$ is the number of tape cells written by the heads during the computation for an input of length n (except the input tape's cells).

For example, consider the previous TM that computes the function PALINDROME. Let $|x| = n$. The first operation requires $2n$ steps of computation to copy the n symbols to the work tape. Then, n more steps are required to adjust the position of the two heads, followed by $2n$ steps to check all pairs of cells. Finally, the last operation requires a single computation step to write the single cell. We conclude that $T(n) = 5n$, while $S(n) = n + 1$.

1.2 Simulation of Turing machines

The definition of Turing machine given in the previous chapter is quite general. There are many variants of our Turing machine model, some which grant more freedom and some which restrict the model. We will show that any computational model can be reduced to a **1-tape minimal alphabet TM**, with a slight loss of power. First, we have to restrict our attention to **time-constructable functions**, a particular subset of functions from that can be computed in an amount of steps that is at most equal to themselves.

Definition 4: Time-constructable function

A function $T : \mathbb{N} \to \mathbb{R}^+$ is said to be **time-constructable** if there is a TM M that computes it in at most $T(n)$ steps for all $n \in \mathbb{N}$.

For example, all the common functions such as $n^2, \log_2 n$ and 2^n are time constructable. We'll start by reducing the tape alphabet. The idea here is pretty simple: for each symbol $a \in \Gamma$, we can choose a binary encoding that represents it. This is similar to how modern computers use standards such as ASCII and UTF to represent characters as strings of 0s and 1s. Given the number of symbols in Γ, we require $c \cdot \log_2 |\Gamma|$ bits to encode each symbol, where $c \geq 1$. When M' has to write a symbol $a \in \Gamma$ on a tape, M' writes the binary encoding $\langle a \rangle$ of such symbol. This means that each step of the computation has to be multiplied by a factor of $c \cdot \log_2 |\Gamma|$.

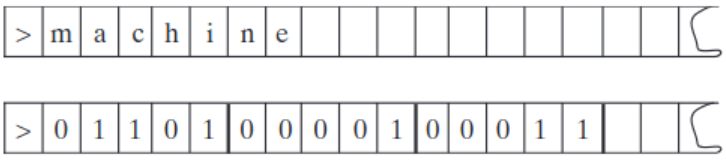


Figure 1.2: Simulation a general alphabet TM through a minimal alphabet TM

Proposition 1: Minimal alphabet Turing machine

Given a TM M with tape alphabet Γ and running time $T(n)$, where T is time-constructable, there is a TM M' with tape alphabet $\{>, \sqcup, 0, 1\}$ and running time $c \cdot \log_2 |\Gamma| \cdot T(n)$, for some constant $c \in \mathbb{R}$, such that $M(x) = M'(\langle x \rangle)$ for all $x \in \Sigma^*$, where $\langle x \rangle$ is the binary encoding of x.

Another change that could be made to the computational model involves the tapes of the machine: in our model, each tape is infinite on the right and finite on the left. This model is usually called **Bidirectional Turing Machine**. Again, we can easily reduce this variant to our model: we can simply double the number of tapes and treat each pair of tapes as two conjoined tapes. This idea is similar to how the number set $\mathbb{Z}$ can be viewed as the union of $\mathbb{N}$ and $-\mathbb{N}$. The running time of the new machine is clearly equal to the original one's.

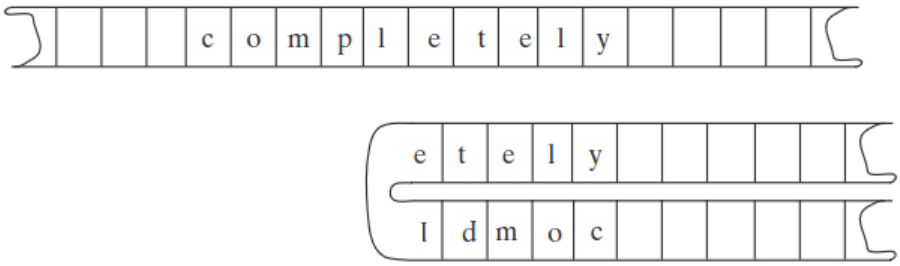


Figure 1.3: Simulation of a bidirectional TM through a monodirectional TM

Proposition 2: Monodirectional Turing machine

Given a bidirectional TM M with tape alphabet Γ and running time $T(n)$, where T is time-constructable, there is a monodirectional TM M' with tape alphabet Γ and running time $T(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma$.

Last but not least, we can reduce a k-tape TM to a single tape TM. Differently from the other two solutions, this idea is not so trivial: we can define the new tape alphabet as a tuple made of k symbols from the original alphabet. For example, given $a_1, \ldots, a_k$ such that a_i is the first cell of the i-th tape, the first cell of the single taped machine will contain the symbol $(a_1, \ldots, a_k)$. This means that the new alphabet requires $|\Gamma|^k$ symbols to represent all possible combinations of cells in the k tapes.

However, this is not sufficient: in the original machine, the k heads were independent from one another. We have to add a way to track where each independent head is in the simulation. To achieve this, we add new special symbols to the tape alphabet: if during a computation step in M the i-th head is on cell j, the i-th symbol in the j-th cell of M' will have a small dot above it to represent the position of the head. This means that the new final alphabet requires $2\,|\Gamma|^k$ symbols to represent all possible combinations of cells and head positions in the k tapes.

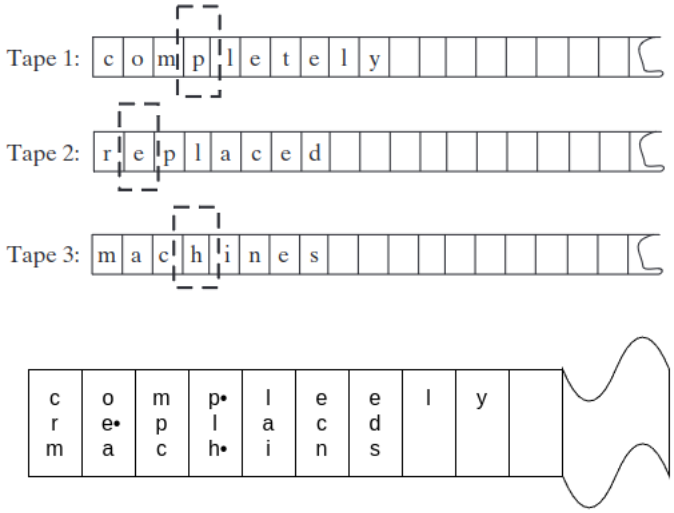


Figure 1.4: Simulation of a k-tape TM through a single tape TM

Before each step of the simulation, the new TM has to first find all the positions of the heads and only then move accordingly. This requires the machine to possibly go back and forth from the first written cell to the latest one.

Proposition 3: Single tape Turing machine

Given a k-tape TM M with running time $T(n)$ and required space $S(n)$, where T is time-constructable, there is a single tape TM M' with running time $n^2 + T(n) \cdot S(n)$ and required space $S(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma^*$.

We can even define an unconventional type of Turing machine, i.e. the **Oblivious Turing machine**. In an Oblivious TM, the movements of the heads are depend only on the length of the input string: for every input $x \in \Sigma^*$ and $i \in \mathbb{N}$, the location of each of the machine's heads at the i-th step of $M(x)$ is only a function of $|x|$ and i. In other words, instead of being defined through the transition function, in an Oblivious TM the movements done by the heads are "implicit". In fact, in this case the transition function is defined as $\delta : (Q - F) \times \Gamma^k \to Q \times \Gamma^k$.

Any standard TM can be transformed into an oblivious one. The idea behind this conversion is similar to how we transformed a k-tape TM to a single tape one. Let $T(n)$ be the running time of the initial TM M. After constructing the single tape, we mark the $T(n)$-th cell with a special symbol, marking a boundary on the right side of the tape. On each step of the computation, the new machine swipes from the leftmost cell to the rightmost one, reading the k head-marks and changing them as M would, to then return to the start of the tape. This small modification makes the new machine's movements oblivious.

The conversion process that we just explained is pretty simple, but quite inefficient since we get a runtime of at most $T(n)^2$. In a more clever way, in which we won't delve into, it's possible to reduce this overhead to a small logarithmic factor.

Proposition 4: Oblivious Turing machine

Given a TM M with running time $T(n)$, where T is time-constructable, there is an single taped Oblivious TM M' with running time $T(n) \cdot \log_2 T(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma^*$.

This capability of reciprocal simulation achievable by different types of Turing machines inspired Turing to show that there is a way to define a singular TM capable of simulating all the others, i.e. an **Universal Turing machine**. To prove this, Turing extended the concept of encoding to machines themselves: if anything can be encoded then TMs can also be encoded! In particular, every encoding $\alpha \in \{0,1\}^*$ corresponds to a single Turing machine M_α, while each TM M' can be obtained by infinitely many encodings. In fact, since we decide how the encoding works, we can encode a TM in multiple ways.

Again the idea here is extraordinarily simple. We assume that U has 5 tapes: an input tape, an output tape, a simulation tape, a tape containing the description of the machine's transition function and one containing the current state of the simulation.

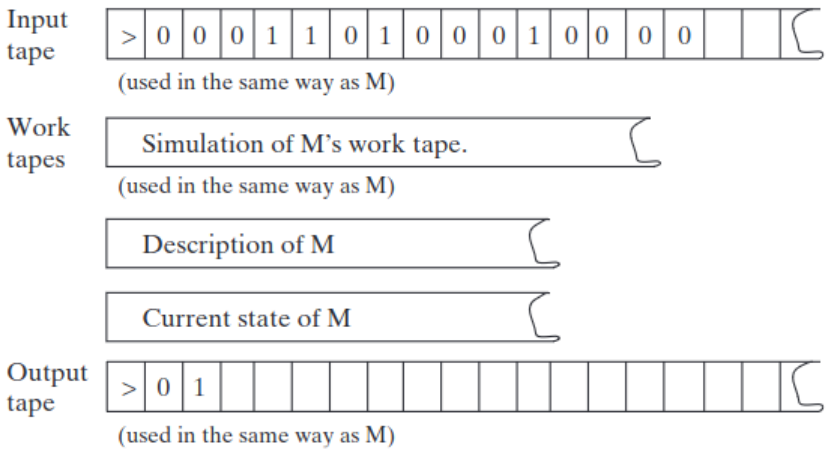


Figure 1.5: Tapes of the Universal Turing machine

Theorem 1: Universal Turing machine

There is a TM U called Universal TM such that for every $x, \alpha \in \{0,1\}^*$ it holds that $U(\langle x, \alpha \rangle) = M_\alpha(x)$. If M_α has running time $T(n)$, the simulation done by U has running time $c \cdot T(n) \cdot \log_2 T(n)$

The existence of such Universal TM shouldn't be a surprise: modern computers are nothing more than a UTMs that can execute any given algorithm, producing an output for a given input. The concept of Universal Turing machine also allows us to easily prove that many other computational models are capable of characterizing computation: if a model is capable of simulating an UTM then it is capable of making any possible computation. This idea is known as **Turing completeness**.

1.3 Uncomputability and Gödel's theorems

After achieving a mathematically stable definition of computation through Turing machines, Turing's focus shifted to understanding which problems are computable and which aren't. Consider the set $\{0,1\}^*$ and the set $\{M_{\alpha_1}, M_{\alpha_2}, \ldots\}$ containing all the possible Turing machines. Through a **diagonal argument**, at some point, each machine M_{α_i} will take its own binary encoding as input:

	M_{α_1}	M_{α_1}	M_{α_1}	$\ldots$	M_{α_i}	$\ldots$
α_1	0	1	0	1	1	$\cdots$
α_2	1	0	0	1	0	$\cdots$
α_3	1	1	1	0	1	$\cdots$
$\vdots$	0	1	0	0	0	$\cdots$
α_i	1	0	0	1	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

From this idea, we can considered the following function $\mathrm{UC} : \{0,1\}^* \to \{0,1\}$:

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

This functions takes in input an encoding α and returns 0 if the computation $M_\alpha(\alpha)$ returns 1, while it returns 1 if $M_\alpha(\alpha)$ returns 0 or goes into an infinite loop. By way of contradiction, suppose that there is a TM M that computes UC.

Let $\beta \in \{0,1\}^*$ such that $M = M_\beta$. Then, we get that:

$$\mathrm{UC}(\beta) = 0 \iff M_\beta(\beta) = 1 \iff \mathrm{UC}(\beta) = 1$$

which raises a contradiction. Thus, the only possibility is that this function is actually **uncomputable**. Through this function, we can show that many other functions are also uncomputable. For example, consider the function $\mathrm{HALT} : \{0,1\}^* \to \{0,1\}$:

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

This functions takes in input an input x and an encoding α, runs $M_\alpha(x)$ and returns 1 if and only if the computation halts. By way of contradiction, suppose that HALT is computable. Then, we define can the following TM M:

$M =$ "Given the encoding α:

1. Compute $h = \mathrm{HALT}(\langle \alpha, \alpha \rangle)$.
2. If $h = 0$, return 1.
3. Otherwise, return $1 - M_\alpha(x)$."

This machine actually computes UC, which we proved to be uncomputable, raising a contradiction. Thus, it must hold that HALT is also uncomputable.

Theorem 2

The functions UC and HALT are uncomputable.

The existence of uncomputable functions — and thus uncomputable problems — gives a negative answer to the *Entscheidungsproblem* (german for *decision problem*), a question posed by David Hilbert in 1928 which asks if there is an algorithm that for each input statement answers "yes" or "no" according to whether the statement is universally true. In addition to this question, Hilbert also posed the question «*is there a strong enough logical system based on recursive axioms and rules that is complete and consistent?*», where:

- A *strong enough* logical system is any proof system that captures basic arithmetic, i.e. any proof system that can describe the natural numbers.
- In a *complete* logical system, if a statement ϕ is true then it is provable
- In a *consistent* logical system, there is no statement ϕ such that both ϕ and $\neg\phi$ are true.

Between completeness and consistency, the latter is clearly more important: if both ϕ and $\neg\phi$ are true, any statement can actually be proved to be both true and false at the same time! In his two acclaimed theorems, Kurt Gödel proved the the answer to Hilbert's additional question is also negative:

- If a strong enough logical system cannot be both complete and consistent
- A consistent logical system cannot prove his own consistency

These two theorems imply that we can do nothing but hope that mathematics is incomplete but at least consistent. Interestingly, Turing's answer can actually be used to give an alternative proof of Gödel's first theorem. First, we have to prove a weaker version of the theorem, where we substitute the concept of consistency with the concept of *soundness*: a logical system is said to be sound if there is no false statement that can be proved. Given $\alpha, x \in \{0,1\}^*$, let $\phi_{\langle\alpha,x\rangle} =$ "$M_\alpha(x)$ halts". Consider the following Turing machine:

$M =$ "Given $\langle\alpha, x\rangle$ in input:

1. Build the formula $\phi_{\langle\alpha,x\rangle}$ and $\neg\phi_{\langle\alpha,x\rangle}$
2. Repeat for all $n \in \mathbb{N}$:
    3. Repeat for all $\pi \in \{0,1\}^n$:
        4. If π is a proof of $\phi_{\langle\alpha,x\rangle}$ then return 1
        5. If π is a proof of $\neg\phi_{\langle\alpha,x\rangle}$ then return 0"

By way of contradiction, suppose that our logical system is both complete and sound. Then, it holds that any statement is true if and only if it can be proven. This makes the machine M capable of computing HALT, which we know to be impossibile. Thus, we get that such logical system cannot be both complete and sound. From this result, is easy to replace soundness with consistency.

2

Basics of complexity theory

2.1 Decision problems and the class P

Until now, we have discussed about problems described by functions of the general form $f : \Sigma^* \to \Sigma^*$. Furthermore, we have seen how the language Σ^* used by a Turing machine can be restricted to the minimal language comprised of 0 and 1.

Now, we will restrict our attention to **decision problems**, i.e. functions of the form $f : \{0,1\}^* \to \{0,1\}$. These problems can be described as simple questions with a «*yes*» or «*no*» answer, such as asking if some input object has some property or not. A «*yes*» answer is represented by a 1, while a «*no*» answer is represented by a 0. For example, given the language $\mathbb{N}$, the question «*is n a prime number?*» is modeled by the decision problem PRIMES $= \{\langle n \rangle \in \{0,1\}^* \mid n \text{ is prime}\}$.

Definition 5: Language of a Decision problem

The language of a decision problem $f : \{0,1\}^* \to \{0,1\}$ is a subset $L \subseteq \{0,1\}^*$ such that $L = \{x \in \{0,1\}^* \mid f(x) = 1\}$.

A decision problem is said to be *decidable* if there is a Turing machine answers the question posed by the problem with 0 or 1 for any input $x \in \{0,1\}^*$. This also implies that the machine has to halt for every input. The language of a problem decided by a Turing machine M is denoted as $L(M)$. Decidability theory plays a core role in math and computer science since most problems can be modeled through it. These problems can be grouped into different classes based on the minimal running time required for any known TM that solves them.

Definition 6: The DTIME class

Given a function $T : \mathbb{N} \to \mathbb{R}^+$, we define DTIME$(T(n))$ as the class of the languages of decision problems computable by a TM for which any input $x \in \{0,1\}^*$ of length $|x| \leq n$ is accepted or refused in at most $O(T(n))$.

The most important subclass corresponds to the set of problems that can be **efficiently solved**. This class is referred to as P, i.e. the class of problems solvable by a Turing machine in polynomial time. In this context, we define an algorithm as "efficient" if it doesn't require an exponential amount of time.

Definition 7: The class P

We define P as the class of the languages decidable in polynomial time:

$$\mathsf{P} = \bigcup_{k \geq 0} \text{DTIME}(n^k)$$

For example, consider the *graph st-connectivity problem*:

$$\text{PATH}(x) = \begin{cases} 1 & \text{if } x = \langle G, s, t \rangle \text{ and } G \text{ is a graph with a path } s \to t \\ 0 & \text{otherwise} \end{cases}$$

This problem can easily be solved in polynomial time through a Depth-first Search algorithm, thus PATH $\in$ P.

But why are we interested in efficiently solving only decision problems? Isn't this type of problem too weak? Surprisingly, it turns out that **decision is all you need**. In general, any real world problem can be modeled in one of three ways:

- **Decision problem**: the problem asks to answer «*yes*» or «*no*» for any given input
- **Search problem**: the problem asks to find a solution for any given input
- **Optimization problem**: the problem asks to find the best solution for any given input

For instance, consider again the graph connectivity problem. We can define this problem as a decision problem by asking «*is there a path from s to t?*», as a search problem by asking «*which is a path from s to t?*» or as an optimization problem by asking «*which is the longest length of a path from s to t?*». These three problems clearly have different types of complexness. However, we can solve the search problem and the optimization problem through two auxiliary decision problems:

- To solve the search problem for an input x, we can use the problem that asks the question «*is this string the prefix of the solution for x?*». Starting from the empty string ε, we ask to machine of the auxiliary problem if the bit 0 is the prefix of the solution. If the answer is «*yes*», we proceed by ask if 00 is the prefix of the solution, otherwise we ask if 10 is part of the solution. We repeat this process until we get the full solution.

- To solve the optimization problem, we can use the problem that asks the question «*is there a path from s to t of at least k nodes?*». Starting from $k = 0$, we ask to machine of the auxiliary problem if there is a path of length k. If there is, we increment k by one and ask again, repeating the process until the machine answers «*no*».

2.2 Verifiability, non-determinism and the class NP

Consider the following *formula satisfiability problem*, defined as:

$$\text{SAT}(x) = \begin{cases} 1 & \text{if } x = \langle\phi\rangle \text{ and } \phi \text{ is a satisfiable formula} \\ 0 & \text{otherwise} \end{cases}$$

Currently, we do not know if this problem is in P or not: the best known algorithms have a worst case running time complexity of 2^{cn}, for some $c \in \mathbb{R}$. This result seems strange: given an assignment $\alpha(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the variables of a formula ϕ, we can easily check in linear time if the assignment is valid or not. Shouldn't this problem be easy? The problem here lies in the amount of possible assignments for a given formula ϕ. If ϕ has n variables, then there are 2^n possible assignments to be checked for satisfiability. In other words, it's easy to verify if a given a assignment can satisfy ϕ, but it's very hard to find this assignment. This idea allows us to introduce the concept of **verification**.

Definition 8: Verifier

Given a computable function f, we say that a TM M is a **verifier** for f if $\forall x \in \{0,1\}^*$ it holds that $f(x) = 1$ if and only if there at least one additional string $w \in \{0,1\}^*$, called **witness** (or *certificate*), such that $M(x, w) = 1$

This definition can be rewritten in terms of *completeness* and *soundness* of the verifier, where the witness w acts as a proof for the statement $f(x) = 1$:

- Completeness: $f(x) = 1 \implies \exists w \in \{0,1\}^*$ such that $M(x, w) = 1$
- Soundness: $f(x) = 0 \implies \forall w \in \{0,1\}^*$ it holds that $M(x, w) = 0$

Mimicking the class P of efficiently solvable decision problems, we define the class of efficiently verifiable decision problems.

Definition 9: The class NP

We define NP as the class of the languages decidable in polynomial time:

$$\text{NP} = \{L \subseteq \{0,1\}^* \mid L \text{ is verifiable in polynomial time}\}$$

By definition, it clearly holds that $\text{P} \subseteq \text{NP}$: if f is solvable in polynomial time then we can use any string as a witness and proceed to solve the problem. Moreover, in order for a verifier to have polynomial time complexity, the length of the witness w must always

be at most poly(n), since otherwise the machine wouldn't even be able to read all of it. Thus, we can always assume that $|w| \leq \text{poly}(|x|)$.

Lots of decision problems are known to be in the class NP, such as the 3-COL problem, which asks the question «*is this graph G 3-colorable?*», the CNF-SAT problem, which asks the question «*is this CNF formula ϕ satisfiable?*», the 3-SAT problem, which asks the question «*is this 3-CNF formula ϕ satisfiable?*» and the GRAPH-ISO problem, which asks the question «*are these two graphs G_1, G_2 isomorphic to each other?*». We can easily notice that we can use efficient verifiers to solve decision problems. However, this process still requires exponential time.

Proposition 5: Decidability through verifiability

Given a language L, if L is verifiable in at most $T(n)$ time then it is decidable in at most $T(n)^2 \cdot 2^{T(n)}$ time

Proof. Let M be a verifier for L that runs in at most $T(n)$ time. We define M' as follows:

M' = "Given the input string x:

1. Let $n = |x|$.
2. Repeat for $i = 0, \ldots, T(n)$:
    3. Repeat for each $w \in \{0,1\}^i$:
        4. Compute $b = M(x, w)$.
        5. If $b = 1$, return 1.
6. Return 0"

M' clearly decides the language L, but requires to generate all the possible witnesses of length at most $T(n)$ and then run $M(x, w)$, requiring at most $T(N)^2 \cdot 2^{T(N)}$ steps. □

Currently, it is not known whether P = NP or not. Researchers believe that the answer to this conjecture is false. For instance, the *Exponential time hypothesis* states that the 3-CNF version of the SAT problem, i.e. 3-SAT, cannot be solved in subexponential time, meaning that it requires at least 2^{cn} steps of computation, for some $c \in \mathbb{R}$, and thus that the currently known algorithms are the best we can achieve.

The answer to this question is considered to be one of the most important questions in mathematics. If P = NP were to be true, a lot of key problems in mathematics that are currently only efficiently verifiable could be solved in a reasonable amount of time by a modern computer. On the other hand, a large number of current technologies are based on the assumption that P ≠ NP. For example, cryptography assumes that it's easy to check that each encrypted string is the result of the encryption scheme being applied to the original message, which works as the certificate, and very hard to find this message only through the encrypted string. If P ≠ NP were proven false, we would have to reconsider a large portion of the modern world, even digital currencies themselves.

We saw how a Turing machine computes a solution following a precise, step-by-step procedure dictated by a set of rules. This type of computation is said to be **deterministic**. In a deterministic computation, the TM has only one possible action to take for each state it can assume. A **non-deterministic** computation, instead, can be thought of as having the ability to explore many different potential computation paths at once.

Definition 10: Non-deterministic Turing machine

A k-tape Non-deterministic Turing machine (NDTM) is a TM provided with two distinct transition functions δ_1, δ_2. On each step, the computation made by the machine forks: one branch follows δ_1 and the other follows δ_2. The machine accepts an input if and only if at least one of the branches of the computation tree accepts. The running time of an NDTM is the length of the longest path in the computation tree for the input x.

Intuitively, non-deterministic Turing machines are more powerful than deterministic ones due to the possibility of exploring multiple choices without increasing the running time. The whole computation can be viewed as a binary tree. We also notice that the number of transition functions doesn't matter: if we have a NDTM with h transition functions, we can simulate it with a NDTM with 2 transitions functions.

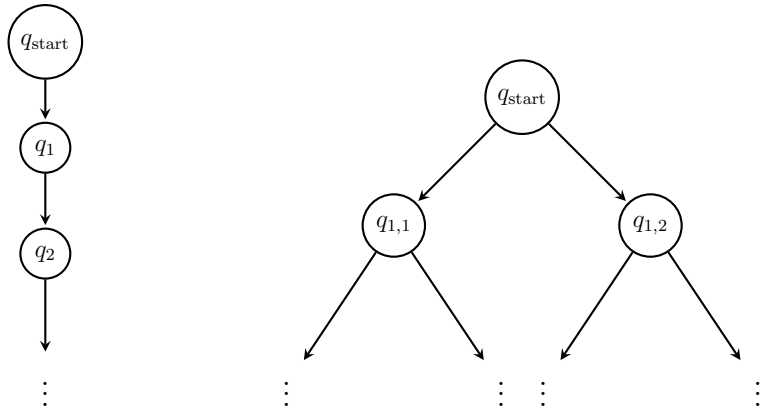


Figure 2.1: A deterministic computation (left) and a non-deterministic one (right)

Definition 11: The NTIME class

Given a function $T : \mathbb{N} \to \mathbb{R}^+$, we define NTIME($T(n)$) as the class of the languages of decision problems computable by a NDTM for which any input $x \in \{0,1\}^*$ of length $|x| \leq n$ is accepted or refused in at most $O(T(n))$.

Surprisingly, the concept of efficient non-deterministic Turing machine is equivalent to the concept of efficient verification, meaning that NP is also the class of all the languages decidable in polynomial time by a NDTM. In fact, this was the original definition of this class of decision problems, hence the name NP standing for Non-deterministic P. However, since non-deterministic Turing machines are only a theoretical tool and not a

real, concrete and realizable model of computation, it was quickly changed to the one based on verification.

Theorem 3: The class NP (2nd Definition)

$$\mathsf{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k)$$

Proof. Suppose L has a polynomial time verifier V in time at most n^k, for some $k \in \mathbb{N}$. We know that for each input $x \in L$ there is a witness $w \in \{0,1\}^{n^k}$. We define a NDTM N that takes in input the string x, non-deterministically generates all the possible strings $w \in \{0,1\}^{n^k}$ and then runs $V(x,w)$, accepting if and only if V accepts. Each branch takes at most $2n^k$ steps.

$$x \in L \iff \exists w \in \{0,1\}^* \;\; V(x,w) = 1 \iff \text{A branch of } N \text{ accepts}$$

Now suppose that L is decided by a NDTM N' in time at most n^k, for some $k \in \mathbb{N}$. Then, we know that $\forall x \in L$ there is at least a path P of length at most n^k that accepts x. We define a verifier V' for which the encoding of the accepting path of each input acts as the witness, concluding that:

$$x \in L \iff \text{A path } P \text{ of } N'(x) \text{ accepts} \iff \exists \langle P \rangle \in \{0,1\}^* \;\; V'(x,P) = 1$$

concluding that $L \in \mathsf{NP}$. □

2.3 Reductions and NP-Completeness

One of the most interesting aspects of computable (and uncomputable) problems is the ability to be transformed into another problem in order to achieve a solution. Suppose that we have an instance a of problem A and that we know an algorithm that transforms a into an instance b of a problem B such that a is a «*yes*» answer if and only if b is a «*yes*» answer. Then, by solving b we would get an answer to a.

In computer science, this concept is known as **reduction**: a problem A is said to be reducible into a problem B, written as $A \leq B$, if any instance a of A can be mapped into an instance b of B whose solution gives a solution to the former. For instance, the proof of the uncomputability of HALT given for Theorem 2. In that case, we *reduced* UC to HALT.

Many-to-one reductions are the simplest type of reduction possible, where a function f maps strings from a language A to strings of a language B.

Definition 12: Many-to-one reduction

Given two languages A, B, we say that A is **many-to-one reducible** to B, written as $A \leq_m B$, if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ such that:

$$x \in A \iff f(x) \in B$$

For example, consider the problems 3-SAT and SAT. The former is clearly reducible to the latter through the identity function $\mathrm{f}(x) = x$: a 3-CNF formula F is satisfiable if and only if F itself is satisfiable!.

For example, consider the problems CNF-SAT and 3-SAT. We can construct a many-one-reduction from the former problem to the latter. Consider a CNF formula $F = \bigwedge_{i=1}^{m} C_i$ where C_i is a logical clause, i.e. $C_i = \bigvee_{j=0}^{t} \ell_j$. We can construct a new 3-CNF formula $F' = \bigwedge_{i=1}^{m} D_i$ by substituting each clause C_i with a new subformula D_i:

1. If $C_i = \ell_1$, we define D_i as:

$$D_i = (\ell_1 \vee y_1 \vee y_2)(\ell_1 \vee \overline{y_1} \vee y_2)(\ell_1 \vee y_1 \vee \overline{y_2})(\ell_1 \vee \overline{y_1} \vee \overline{y_2})$$

   where y_1, y_2 are two new variables.

2. If $C_i = \ell_1 \vee \ell_2$, we define D_i as:

$$D_i = (\ell_1 \vee \ell_2 \vee y_1)(\ell_1 \vee \ell_2 \vee \overline{y_1})$$

   where y_1 is a new variable.

3. If $C_i = \ell_1 \vee \ell_2 \vee \ell_3$, we define $D_i = C_i$.

4. If $C_i = \ell_1 \vee \ldots \vee \ell_t$ with $t > 3$, we define D_i as:

$$D_i = (\ell_1 \vee \ell_2 \vee y_1)(\overline{y_1} \vee \ell_3 \vee y_2) \ldots (\overline{y_{t-3}} \vee \ell_{t-2} \vee y_{t-2})(\overline{y_{t-2}} \vee \ell_{t-1} \vee \ell_t)$$

For the first three cases, it's easy to see that given an assignment α it holds that $C_i(\alpha) = 1 \iff D_i(\alpha)$. The fourth case, instead, requires more attention. Suppose that there is an assignment α that satisfies C_i. This implies that at least one ℓ_j must be set to true in α. Then, in every j-th subclause of D_i, we can set one between y_j or $\overline{y_{j-1}}$ to true in order to satisfy D_i. Vice versa, suppose that there is an assignment β that satisfies D_i. By way of contradiction, suppose that $\ell_1, \ldots, \ell_t = 0$ in β. Without loss of generality, we can assume that $y_1, \ldots, y_{t-3} = 1$ in β. Then, for any assignment of y_{t-2}, one between the clause $(\overline{y_{t-3}} \vee \ell_{t-2} \vee y_{t-2})$ or $(\overline{y_{t-2}} \vee \ell_{t-1} \vee \ell_t)$ cannot be satisfiable, raising a contradiction. Thus, there must be at least one ℓ_j set to true in β, implying that β also satisfies C_i.This concludes that F is satisfiable if and only if F' is satisfiable. Moreover, F' is a 3-CNF, hence CNF-SAT $\leq_m$ 3-SAT.

Many-to-one reductions between problems are transitive: starting from a problem A, we can reduce it to a problem B through a function f and then reduce it to a problem C through a function g. This implies that $h = g \circ f$ is a reduction from A to C. Hence, we have that CNF-SAT $\leq_m$ 3-SAT $\leq_m$ SAT.

Proposition 6: Transitive reduction

Given three languages A, B, C, if $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.

In particular, we notice that both of the previous reductions are easily computable in polynomial time by a Turing machine. When this happens, the reduction is said to be a **Karp reduction**.

Definition 13: Karp reduction

Given two languages A, B, we say that A is **Karp reducible** to B, written as $A \leq_P B$, if there is a many-to-one reduction from A to B computable in polynomial time

For instance, the previous reduction from CNF-SAT to 3-SAT can actually be computed in polynomial time, hence CNF-SAT $\leq_P$ 3-SAT. Clearly, transitivity between reductions also holds in this case: if two reductions f and g are both computable in polynomial time then $h = g \circ f$ also is.

Proposition 7: Transitive Karp reduction

Given three languages A, B, C, if $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.

When a problem A is Karp reducible to a problem B solvable (or verifiable) in polynomial time by a machine M, i.e. a problem that is in P (or NP), we can build a new Turing machine M' that first computes the reduction from A to B, then runs M and finally accepts if and only if M accepts the reduced instance. This new machine M' clearly also requires polynomial time, meaning that A is also solvable (or verifiable) in polynomial time.

Proposition 8

Let A, B be two languages such that $A \leq_P B$. Then:

- If $B \in \mathsf{P}$ then $A \in \mathsf{P}$.
- If $B \in \mathsf{NP}$ then $A \in \mathsf{NP}$.

Reductions between problems enable us to *classify* problems based on their "hardness": if a problem A is efficiently reducible to a problem B, then solving A cannot be harder than solving B, forming some sort of hierarchy between problems. In other words, if $A \leq_P B$ then A is at most hard as B and B is at least as easy as A.

Building on this idea of hardness, we can identify some privileged problems that are harder than a whole class of problems! For example, every problem from the class NP is efficiently reducible to a single problem B, then each NP problem is as hard as B. When this happens, we say that B is **NP-Hard**. When an NP-Hard problem also lies in NP, we say that such problem is **NP-Complete**.

Definition 14: NP-Hardness and NP-Completeness

A language B is said to be NP-Hard if $\forall A \in \mathsf{NP}$ it holds that $A \leq_P B$. If B is also in NP, we say that it is NP-Complete.

The simplest NP-Complete problem is the *TM satisfiability problem*, defined as:

$$\text{TM-SAT}(x) = \begin{cases} 1 & \text{if } x = \langle \alpha, y, 1^n, 1^t \rangle \text{ and } \exists w \in \{0,1\}^n \text{ s.t. } M_\alpha(y, w) = 1 \text{ within } t \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

First we show that this problem is in NP. Given an input $x = \langle \alpha, y, 1^n, 1^t \rangle$ and a witness $w \in \{0,1\}^*$, we start by asserting that $|w| = n$. Then, we simulate $M_\alpha(y, w)$ with a counter that immediately stops if the simulation executes more than t steps. Since the input has size $|x| \geq t$ and the simulation takes at most t steps, the verifier requires polynomial time, concluding that TM-SAT $\in$ NP. Finally, we show that this problem is also NP-Hard. Given a language $A \in$ NP, we know that it can be verified in at most $|x|^k$ steps for some $k \in \mathbb{N}$. Suppose that such verifier is encoded by β. The reduction is easily given by mapping any input $x \in A$ to $f(x) = \left\langle \beta, x, 1^{|w|}, 1^{|x|^k} \right\rangle$, where w is x's witness for A, concluding that $A \leq_P$ TM-SAT.

We notice that, by definition, NP-Hard problems can even be uncomputable! For instance, consider the *input acceptance problem*:

$$\text{ACCEPT}(x) = \begin{cases} 1 & \text{if } x = \langle \alpha, y \rangle \text{ and } M_\alpha(y) = 1 \\ 0 & \text{otherwise} \end{cases}$$

It's pretty easy to see that any NP problem (and more generally any computable problem) is Karp reducible to ACCEPT, meaning that ACCEPT is NP-Hard. However, this problem is clearly uncomputable since HALT $\leq_m$ ACCEPT.

If an NP-Hard problem could be solved in polynomial time, i.e. be inside P, then every single problem NP-Hard problem would also be solvable in polynomial time, meaning that P = NP. However, when an NP-Hard problem also lies in the class NP, this relation also becomes valid in the opposite direction: if P = NP then any NP-Hard problem that is in NP is also in P.

Proposition 9

Given a language B, it holds that:

- If B is NP-Hard then $B \in$ P implies that P = NP
- If B is NP-Complete then $B \in$ P if and only if P = NP

In other words, if an NP-Complete problem is proven to be solvable or unsolvable in polynomial time, we would get a proof for the P $\stackrel{?}{=}$ NP question. Moreover, by definition, each NP-Complete is Karp reducible to one another, meaning that they all of them share the same hardness. These are the reasons why these problems are referred to as *complete*.

Even though it is indeed NP-Complete, the problem TM-SAT(x) is quite artificial since, by definition, in order to efficiently solve it we would have to be capable of efficiently solving any efficiently verifiable problem, which is literally the same as asking to solve the

$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question. A more interesting NP-Complete problem is the *satisfiability problem* discussed in the previous sections, which actually is the first ever problem to be shown to be NP-Complete. This result is known as the Cook-Levin theorem.

Theorem 4: The Cook-Levin theorem

SAT and 3-SAT are NP-Complete

The standard proof of this theorem is quite long and tedious, requiring a very careful notation in order to make things clearer. Instead, we will achieve this theorem by proving that the *circuit satisfiability problem* is NP-Complete and then reducing it to both SAT and 3-SAT. For the moment being, we will assume to have proved this theorem.

2.4 The Web of Reductions

The easiest way to show that a problem is NP-Complete is to show that it lies inside NP and then find a Karp-reduction from another NP-Complete problem to it. Through this idea, we can form a *web of reductions* between NP-Complete problems. In fact, this was the original idea used by Karp to prove his famous 21 NP-Complete problems starting from the SAT problem (hence the name Karp reductions).

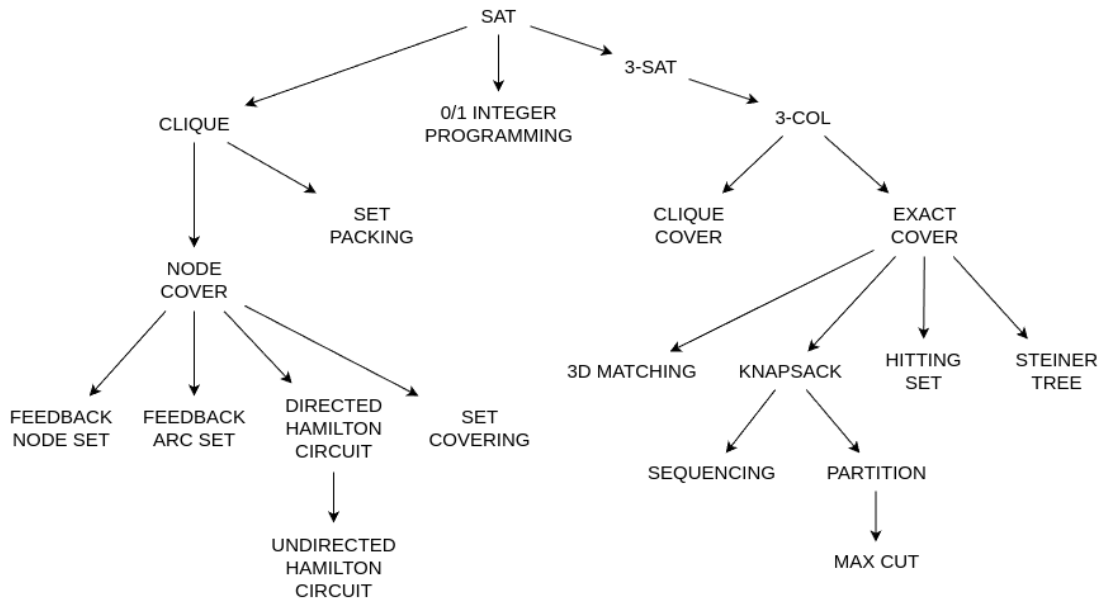


Figure 2.2: Karp's original 21 NP-Complete problems reduction tree

Some problems can be shown to be NP-Complete simply by being an extension of an already known NP-Complete problem. For instance, we have shown that CNF-SAT $\leq_P$ 3-SAT, but the reverse also holds since the former is just an extension of the latter (the reduction is the identity function). Moreover, CNF-SAT is trivially in NP: the witness is the satisfying assignment. This concludes that CNF-SAT is also NP-Complete.

Theorem 5

CNF-SAT is NP-Complete.

Consider now the *0/1 integer programming problem*:

$$\text{0/1-PROG} = \begin{cases} 1 & \text{if } x = \langle P \rangle \text{ and } P \text{ is a feasible 0/1 program} \\ 0 & \text{otherwise} \end{cases}$$

An 0/1 integer program is a linear program defined on n variables, m inequalities, called constraints, and an objective function, where $x_1, \ldots, x_n \in \{0, 1\}$. In a linear program, we want to find the optimal combination of the values of $x_1, \ldots, x_n$ that maximize the value of the objective function. A linear program is said to be feasible if there is at least one possible assignment for $x_1, \ldots, x_n$ that satisfies all the constraints.

Theorem 6

0/1-PROG is NP-Complete.

Proof. Here, the witness is nothing more than a feasible assignment for $x_1, \ldots, x_n$, which can be checked in polynomial time. Moreover, we can easily reduce 3-SAT by expressing every clause inside a CNF formula as a sum between the three literals whose value must be at least 1. For example, the clause $(x_1 \vee \overline{x_2} \vee \overline{x_3})$ can be expressed as the constraint $x_1 + (1 - x_2) + (1 - x_3) \geq 1$. Clearly, each formula is satisfiable if and only if its corresponding linear program has a feasible solution. □

Now, we shift our focus to a more complex reduction. Consider the *independent set problem*:

$$\text{IND-SET} = \begin{cases} 1 & \text{if } x = \langle G, k \rangle \text{ and } G \text{ is a graph with an ind. set. } S \text{ such that } |S| \geq k \\ 0 & \text{otherwise} \end{cases}$$

An independent set of a graph G is a subset of vertices that share no edge between each other. Formally, $S \subseteq V(G)$ is an independent set of G when $\forall u, v \in S$ it holds that neither (u, v) nor (v, u) is inside $E(G)$.

Theorem 7

IND-SET is NP-Complete.

Proof. For IND-SET, the certifying witness is trivially an independent set of size k inside G, which can be checked in linear time.

Again, we can reduce 3-*SAT* to this problem. However, this time we require a less immediate conversion. Consider a CNF formula $F = \bigwedge_{i=1}^{m} C_i$ where C_i is a logical clause with 3 literals. We construct a graph G_F from F as follows:

1. For each clause $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$, add three nodes ℓ_1, ℓ_2, ℓ_3 (repeat the nodes if the same literal appears in more than a clause) and connect these three nodes by adding the edges $(\ell_1, \ell_2), (\ell_2, \ell_3)(\ell_3, \ell_1)$, forming a triangle
2. For each node ℓ added in the previous step, add an edge between ℓ and every other node labeled with its negation, i.e. $\overline{\ell}$.

Let α be an assignment of F. Without loss of generality, let ℓ_1^i be the literal that satisfies each clause C_i (if there are more than one satisfying literal, just pick one of them). Let S be a subset of vertices composed of each ℓ_1^i from each clause C_i. By construction, we know that ℓ_1^i is adjacent to ℓ_2^i, ℓ_3^i. Moreover, it is also adjacent to every other node labeled with its negation, that being the nodes $\overline{\ell_1^j}, \ldots, \overline{\ell_1^t}$.

Suppose that α is a satisfying assignment for F. Since $\ell_1^i = 1$ inside α, we know that $\overline{\ell_1^j}, \ldots, \overline{\ell_1^t} = 0$ inside α. Since S contains only $\ell_1^1, \ldots, \ell_1^m$, this is an independent set of size at least m inside G_F.

Vice versa, suppose that α doesn't satisfy F. This implies that at least one clause C_j evaluates to 0, hence every literal inside it must be set to 0. This implies that S has size less than m, hence its not an independent set of size at least m.

We get that F can be satisfiable if and only if there is an independent set of size at least m inside G_F. Moreover, since each clause has a constant number of literals, this conversion can be built in polynomial time by a TM, concluding that 3-SAT $\leq_P$ IND-SET. □

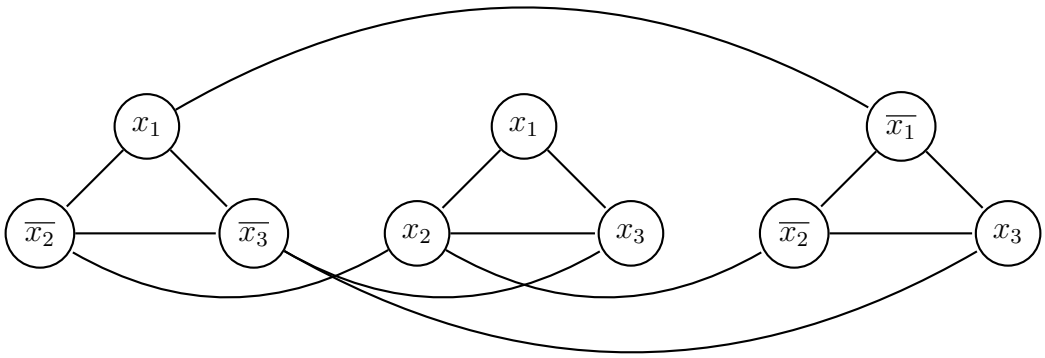


Figure 2.3: Conversion of the formula $(x_1 \vee \overline{x_2} \vee x_3)(x_1 \vee x_2 \vee x_3)(\overline{x_1} \vee \overline{x_2} \vee x_3)$

Once we have proven that the independent set problem is NP-Complete, we have a link between graph theory and complexity theory. This allows us to use the many known results in graph theory to easily extend our web of complete problems. For instance, consider the *clique problem* and the *vertex cover problem*:

$$\text{CLIQUE} = \begin{cases} 1 & \text{if } x = \langle G, k \rangle \text{ and } G \text{ is a graph with a clique } C \text{ of size } |C| \leq k \\ 0 & \text{otherwise} \end{cases}$$

$$\text{VERT-COVER} = \begin{cases} 1 & \text{if } x = \langle G, k \rangle \text{ and } G \text{ is a graph with a v. cov. } C \text{ of size } |C| \leq k \\ 0 & \text{otherwise} \end{cases}$$

A clique of a graph G is a subset of vertices all adjacent to each other. Formally, $C \subseteq V(G)$ is a clique of G when $\forall u, v \in C$ it holds that (u, v) or (v, u) is inside $E(G)$. A vertex cover of a graph G, instead, is a subset of vertices capable of covering every edge of G. Formally, $V \subseteq V(G)$ is a vertex cover of G when $\forall (u, v) \in E(G)$ it holds that either u or v is inside V.

Like in IND-SET, the certifying witness for CLIQUE and VERT-COVER are a clique and a vertex cover of size k inside G. Two very standard results in graph theory state the following property, which descend directly from the definitions of independent set, clique and vertex cover.

Proposition 10

Let $S \subseteq V(G)$ a subset of vertices of a graph G. Then, it holds that:

1. S is an independent set of G if and only if S is a clique of $\overline{G}$
2. S is an independent set of G if and only if $V(G) - S$ is a vertex cover of G

Note: $\overline{G}$ is the complementary graph of G, the graph with the same nodes of G and all the opposite edges of G. Formally, $V(\overline{G}) = V(G)$ and $E(\overline{G}) = V(G) \times V(G) - E(G)$.

This properties can be used to make two trivial reductions:

1. $\langle G, k \rangle \in$ IND-SET if and only if $\langle \overline{G}, k \rangle \in$ CLIQUE
2. $\langle G, k \rangle \in$ IND-SET if and only if $\langle G, |V(G)| - k \rangle \in$ VERT-COVER

which can both be built in polynomial time.

Theorem 8

CLIQUE and VERT-COVER are NP-Complete.

Sometimes, graph properties aren't enough to establish a reduction from a graph problem to another. Case in point is the *Hamiltonian path problem*. A Hamiltonian path on a graph G is a path that traverses all the nodes of the graph. This problem is quite important in graph theory and it's part of the most studied ones.

$$\text{d-HAMPATH(x)} = \begin{cases} 1 & \text{if } x = \langle G \rangle \text{ and } G \text{ is a directed graph with a Hamiltonian path} \\ 0 & \text{otherwise} \end{cases}$$

This problem is also NP-Complete, however its proof is not so easy. First, we prove this for the directed version to then reduce it to the undirected case. It's easy to see that d-HAMPATH(x) is in NP, since a Hamiltonian path can act as a witness. To show its completeness, we define a reduction from 3-SAT. Consider a CNF formula $F = \bigwedge_{i=0}^{m-1} C_i$ where C_i is a logical clause with 3 literals with a total of n variables. We notice that the clauses of F are numbered starting from 0 instead of 1. This will make the notation of the proof easier to read.

For each variable x_i, we construct an horizontal "chain" of $6m$ vertices $v_0^i, \ldots, v_{6m-1}^i$. Each node inside the chain is bidirectionally adjacent to its near nodes. In other words, for each $j \in [0, 6m-2]$ we have that (v_j^i, v_{j+1}^i) and (v_{j+1}^i, v_j^i). The idea behind this chain of $6m$ vertices will be explained later in the proof.

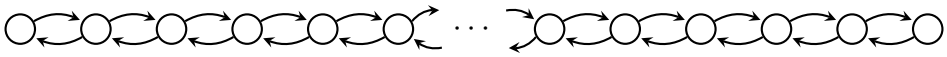


Figure 2.4: Horizontal chain of $6m$ vertices for a variable x_i

Then, we add $n+1$ more vertices $s_0, \ldots, s_n$ that connect these n chains. The nodes $s_0, s_1, \ldots, s_{n-1}$ have two edges out-going to the extremes of their chain, i.e. (s^i, v_1^i) and (s^i, v_{6m-1}^i), and the nodes $s_1, \ldots, s_{n-1}, s_n$ have two in-going edges from the extremes of the previous chain, i.e. (v_0^{i-1}, t^i) and (v_{6m-1}^{i-1}, t^i). The final result is a sequence of diamond-shaped subgraphs.

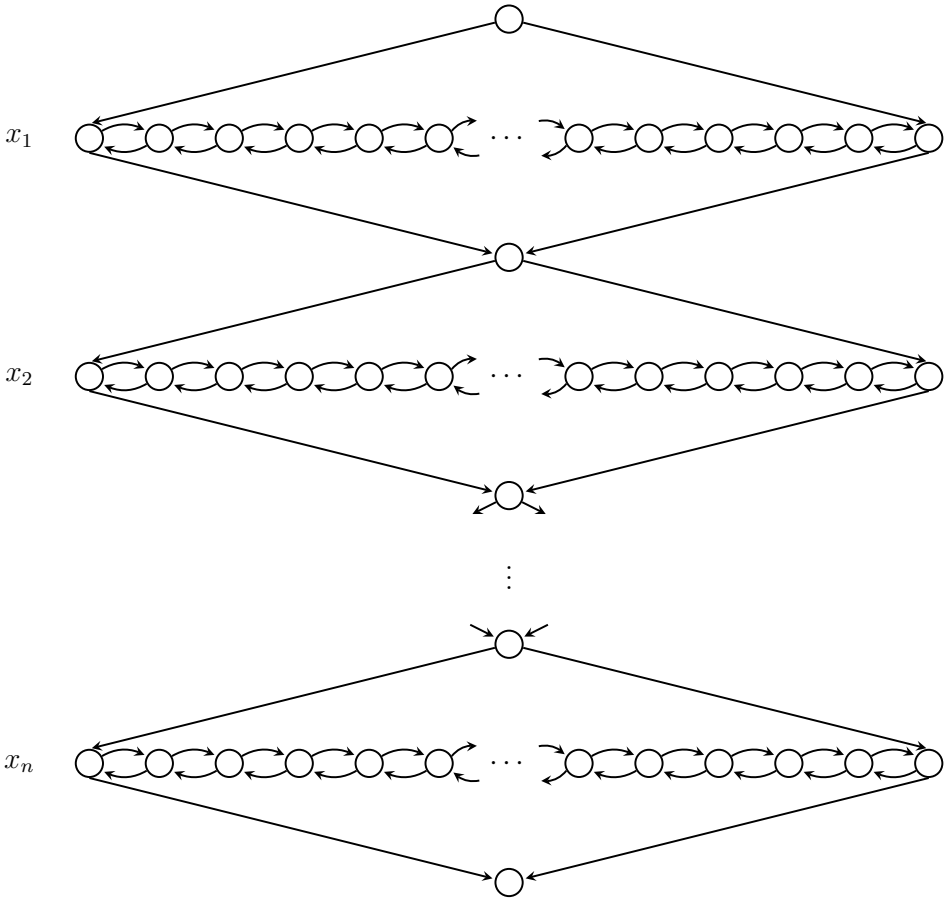


Figure 2.5: Sequence of diamond-shaped subgraphs for the formula F

Let's discuss what we have constructed until now. Consider a Hamiltonian path that starts from s_0 and ends on s_n. On the first edge, this path has to take either the left or the right edge out-going from s_0. Without loss of generality, suppose that it takes the edge on the left. Then, this path has to traverse the whole chain of x_0 from left to right

and then go to the node s_1. The same argument holds for all the other chain-connecting nodes.

Hence, this graph contains exactly 2^n Hamiltonian paths. Now, we will assume that when the chain i is traversed by a path *from left to right* then the variable x_i will be considered as set to *true*, while when the path traverses it *from right to left* we will assume that x_i is *false*. With this idea in mind, we get a perfect bijection from each Hamiltonian path to each possible assignment of F.

However, this means that each formula with n variables and $6m$ clauses will produce the same graph. To distinguish between each formula, we have to introduce some "gadgets" that model the clauses of F.

First, we introduce m nodes $c_0, \ldots, c_{m-1}$, each corresponding to a clause of F. The idea here is that the first 6 nodes of the chain i will be reserved for possible connections to the clause C_0, while the next 6 nodes will be reserved for connections to the clause C_1 and so on. Thus, each chain is partitioned into 6 subchains, one for each clause. Moreover, each of these subchains is again partitioned into 3 pairs of adjacent nodes used to describe if the x_i variable or its negation $\overline{x_i}$ is inside C_i and in what position.

Formally, consider a clause $C_j = \ell_0 \vee \ell_1 \vee \ell_2$. When the literal ℓ_t of such clause C_j corresponds to the variable x_i, we add the edges $(v^i_{6j+2t}, c_j), (c_j, v^i_{6j+2t+1})$. Instead, when the literal ℓ_t of such clause C_j corresponds to the negation $\overline{x_i}$ of a variable, we add the edges $(v^i_{6j+2t+1}, c_j), (c_j, v^i_{6j+2t})$. In other words, if $\ell = x_i$ then we force the direction $v^i_{6j+2t} \to c_j \to v^i_{6j+2t+1}$ on the path, while if $\ell = \overline{x_i}$ then we force the reverse direction $v^i_{6j+2t+1} \to c_j \to v^i_{6j+2t}$ on the path.

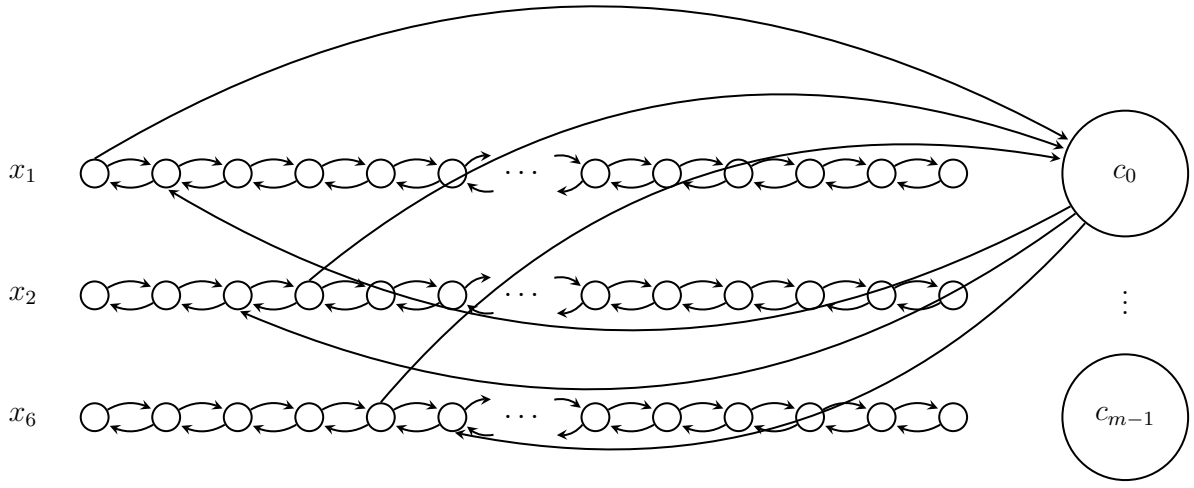


Figure 2.6: Example of path restriction given by $C_0 = (x_1 \vee \overline{x_2} \vee x_6)$

Consider now an assignment α that satisfies F. We construct a path P that traverses each chain i from left to right if $\alpha(x_i) = 1$ and from right to left if $\alpha(x_i) = 0$. While traversing a chain, if the literal ℓ_t is the one that satisfies the clause C_i, the path also traverses the node c_i through the two vertices reserved for that literal (choose only one if

there is more than one satisfying literal). This path traverses all the nodes of the graph, concluding that P is an Hamiltonian path of G_F.

Vice versa, suppose that G_F has a Hamiltonian path P'. We construct an assignment α' where the truthfulness of the variable x_i depends on the direction which P' traverses for the i-th chain. This assignment clearly satisfies F since each clause C_i will have to be traversed by at least one literal in order for P' to be Hamiltonian.

We conclude that F is satisfiable if and only if G_F has a Hamiltonian path. Since G_F is a huge graph by construction, we have to check that we can build it in polynomial time. Each chain contains $6m$ nodes and $2(6m-1)$ edges, for a total of $6mn$ vertices and $2n(6m-1)$ edges. The we also have n chain-connecting nodes, each with at most 2 out-going and 2 in-going edges, for a total of n vertices and at most $4n$ edges. Finally, we have m clause nodes, each with 6 edges, for a total of m more vertices and $6m$ more edges.

Summing all of these values, we conclude that G_F has a total of $6mn+n+m$ nodes and at most $2(6m-1)+4n+6m$ edges. This means that the reduction can indeed be built in polynomial time, concluding that 3-SAT $\leq_P$ d-HAMPATH.

Theorem 9

d-HAMPATH is NP-Complete.

Once we have proved this result, we can extend it to other kinds of graph problems related to Hamiltonian paths, such as the undirected version of the *Hamiltonian path problem* and the *Hamiltonian cycle problem*, both directed and undirected. Here, a cycle is said to be Hamiltonian if it traverses each node only once, exception made for the first and last node of the cycle. These problems can be easily shown to be NP-Complete by reducing d-HAMPATH to them.

Theorem 10

u-HAMPATH, d-HAMCYCLE and u-HAMCYCLE are NP-Complete.

To summarize, we have shown that, under the assumption that The Cook-Levin theorem is true, which states that SAT and 3-SAT are NP-Complete, the problems CNF-SAT, 0/1-PROG, IND-SET, CLIQUE, VERT-COVER, u-HAMPATH, d-HAMCYCLE and u-HAMCYCLE are all NP-Complete.

Thousands of NP problems have been proven to be NP-Complete, but none of them has been proven to be inside or outside of P. Proving one of the two results would answer the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ answer. The fact that we are incapable of efficiently solving any of these problems gives enough insight for modern researchers to believe that the answer to the conjecture is negative, i.e. $\mathsf{P} \neq \mathsf{NP}$. Moreover, some of these problems have also been proven to be **unapproximable** under a certain ratio. In the following sections and chapters, we will provide more results that make us believe that $\mathsf{P} \neq \mathsf{NP}$ must hold, since otherwise most "*obvious*" things in complexity theory would actually collapse.

2.5 The classes EXP, NEXP

We have seen how for some problems we still don't know if they lie only inside NP or if they also lie in P. However, we do know that some decision problem are strictly outside of P. For instance, consider the following variant of the halting problem:

$$\text{k-HALT}(x) = \begin{cases} 1 & \text{if } x = \langle \alpha, y, k \rangle \text{ and } M_\alpha(y) \text{ halts in at most } k \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

This problem strictly requires exponential time to be solved: the input value k is encoded by $c \log k$ bits but the problem has to simulate $M_\alpha(y)$ for at least k steps, which is exponential in terms of the input size. This means that k-HALT $\notin$ P.

We define the *exponential versions* of the classes P and NP by relaxing the time constraint to being exponential with respect to the input.

Definition 15: The classes EXP and NEXP

We define EXP as the class of the languages decidable in exponential time by a deterministic Turing machine as follows:

$$\text{EXP} = \bigcup_{k \geq 0} \text{DTIME}(2^{n^k})$$

Likewise, we define NEXP as the class of the languages decidable in exponential time by a non-deterministic Turing machine as follows:

$$\text{NEXP} = \bigcup_{k \geq 0} \text{NTIME}(2^{n^k})$$

Note: we can also define NEXP through exponential time verifiers

From Proposition 5, we easily get that any problem that is verifiable in polynomial time can be decided in exponential time by a deterministic TM. Moreover, like in the polynomial case, any deterministic exponential time Turing machine can be simulated by a non-deterministic Turing machine that has two exactly equal transition functions, requiring the same amount of time.

Proposition 11

$\text{P} \subseteq \text{NP} \subseteq \text{EXP} \subseteq \text{NEXP}$

Moreover, we know that k-HALT $\notin$ P but it clearly holds that k-HALT $\in$ EXP. In the same fashion, we can define a very artificial problem that cannot be in NP but that definitely is inside NEXP.

Proposition 12

$\mathsf{P} \subsetneq \mathsf{EXP}$ and $\mathsf{NP} \subsetneq \mathsf{NEXP}$.

These results give us some insight on the P vs. NP question. In particular, since $\mathsf{P} \subsetneq \mathsf{EXP}$, we have only three possible alternatives for the decision hierarchy:

1. Each level is separated, meaning that $\mathsf{P} \subsetneq \mathsf{NP} \subsetneq \mathsf{EXP}$
2. The separation lies only between P and NP, meaning that $\mathsf{P} \subsetneq \mathsf{NP} = \mathsf{EXP}$
3. The separation lies only between NP and EXP, meaning that $\mathsf{P} = \mathsf{NP} \subsetneq \mathsf{EXP}$

Furthermore, the P vs. NP question also translates to the exponential time world: we do not know whether $\mathsf{EXP} = \mathsf{NEXP}$ or not. However, we have an interesting correlation between the two questions:

Theorem 11

If $\mathsf{EXP} \neq \mathsf{NEXP}$ then $\mathsf{P} \neq \mathsf{NP}$

Proof. We prove the contrapositive statement. We already know that $\mathsf{EXP} \subseteq \mathsf{NEXP}$ holds by definition, so we have to only show that if $\mathsf{P} = \mathsf{NP}$ then $\mathsf{NEXP} \subseteq \mathsf{EXP}$ also holds.

Given a language $L \in \mathsf{NEXP}$, let L_{pad} be the language defined as $L_{\text{pad}} = \{x1^{2^{|x|^k}} \mid x \in L\}$, where k is some pre-fixed sufficiently large constant. The idea behind this theorem is to basically cheat by abusing definitions: since the running time of a machine is defined with respect to its input size, if we make the input size extremely large then the execution time will be "efficient" with respect to the input size.

Let M be the machine that verifies L in exponential time. Then, we define the machine M' as follows:

M' = "Given the input string x:

1. Check that $x = \left\langle y1^{2^{|y|^k}}, w \right\rangle$. If false, reject.
2. Return $M(y, w)$."

This machine clearly verifies L_{pad}. Moreover, since the input size already is exponential, the execution of M' will always run in polynomial time. Hence, we get that $L_{\text{pad}} \in \mathsf{NP}$. Then, since $\mathsf{P} = \mathsf{NP}$ by assumption, we get that $L_{\text{pad}} \in \mathsf{P}$. Through a machine M'' that decides P in polynomial time, we can define another machine that decides L in exponential time by bloating the input with an exponential number of ones and then running M'' on that input, concluding that $L \in \mathsf{EXP}$. □

The argument used in this theorem is usually called **time padding argument**. Even though this result is indeed interesting, finding a separation between EXP and NEXP is clearly more difficult than finding one for P and NP: when we allow the time to be exponential, we can solve virtually any "non-artificial" problem.

2.6 Disqualification, duality and the class coNP

We saw how SAT lies inside NP and how it actually is NP-Complete. What can we say about his complementary language, i.e. $\overline{\text{SAT}} = \{x \in \{0,1\}^* \mid x \notin \text{SAT}\}$? We seem to not be able to show that this language is inside NP: in order to *verify* that a string lies in $\overline{\text{SAT}}$, we would have to find a way to certify that this string is *not* inside SAT, which means that every possible witness cannot certify the membership of this string in SAT. We define the class of all such languages as coNP.

Definition 16: The class coNP

We define coNP as the class of languages whose complement is in NP:

$$\text{coNP} = \{L \subseteq \{0,1\}^* \mid \overline{L} \in \text{NP}\}$$

We observe that, by definition, the class coNP **is not equal to** $\overline{\text{NP}}$, as the latter is the class of languages that aren't in NP. Here, the prefix "co" stands for *dual* and **not for *complement***.

Moreover, the concept of complementary language is different from the concept of **opposite** language: the opposite of a problem is the problem that asks the opposite question of the former (this doesn't imply that the answer to the latter is always the opposite of the former). For instance, consider the following problems:

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a CNF formula and } \exists\alpha \;\; \phi(\alpha) = 1\}$$

$$\text{UNSAT} = \{\langle\phi\rangle \mid \phi \text{ is a CNF formula and } \forall\alpha \;\; \phi(\alpha) = 0\}$$

These two problems are opposites. However, by definition we have that $\text{UNSAT} \subsetneq \overline{SAT}$: the language $\overline{\text{SAT}}$ doesn't only contain CNF formulas that are unsatisfiable, but also gibberish strings in $\{0,1\}^*$ that aren't even encodings of CNF formulas! Nonetheless, the language UNSAT does still lie inside coNP since the language $\overline{\text{UNSAT}}$ — which is not equal to SAT — can be easily verified in polynomial time. First, we check if the input string is a valid formula or not. If it is, the witness for its membership in $\overline{\text{UNSAT}}$ is a satisfying assignment. Otherwise, if the string is gibberish, the witness can be any string.

Take a closer look at what we just did: in order to verify $\overline{\text{UNSAT}}$, we have found a way to polynomially **disqualify** strings that are *not* inside UNSAT. In fact, coNP can also be defined as the class of languages that can be disqualified in polynomial time.

Theorem 12: The class coNP (2nd Definition)

Given a language L, it holds that $L \in \text{coNP}$ if and only if there is a polynomial time TM M such that:

$$x \in L \iff \forall w \in \{0,1\}^* \; M(x,w) = 1$$

Proof. Given $L \in \mathsf{coNP}$, by definition we have that $\overline{L} \in \mathsf{NP}$. Hence, there is a polynomial time verifier M such that:

$$x \in \overline{L} \iff \exists w \in \{0,1\}^* \; M(x,w) = 1$$

Let $\neg M$ be the machine that always returns the opposite of M. We notice that:

$$x \in L \iff x \notin \overline{L} \iff \forall w \in \{0,1\}^* \; M(x,w) = 0 \iff \forall w \in \{0,1\}^* \; \neg M(x,w) = 1$$

hence $\neg M$ is a polynomial time disqualifier for L. All the steps that we did can be reversed: the opposite of any disqualifier for L is a verifier for $\overline{L}$. □

This trivially implies that $\mathsf{P} \subseteq \mathsf{coNP} \subseteq \mathsf{EXP}$: if a problem is in P then we ignore the witnesses, while if a problem is in coNP then we can just enumerate all the possible witnesses and check them deterministically in exponential time.

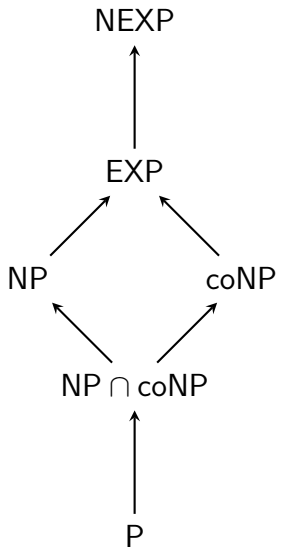


Figure 2.7: Recap of the class inclusions discussed in this chapter

Moreover, this new definition of coNP opens the doors to the concept of **duality**: to the property of two objects of being "two faces of the same medal". For example, consider the two following languages

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a CNF formula and } \exists\alpha \;\; \phi(\alpha) = 1\}$$

$$\text{TAUT} = \{\langle\phi\rangle \mid \phi \text{ is a DNF formula and } \forall\alpha \;\; \phi(\alpha) = 1\}$$

These two languages are dual to each other: the concepts of CNF and DNF formulas are two faces of the same medal and the same goes for the quantifiers $\exists$ and $\forall$. Clearly, the language TAUT lies inside coNP, since the same disqualification technique that we did for UNSAT can be used in this case by replacing the satisfying assignment with an unsatisfying one.

Duality and complementariness are two distinct concepts. For instance, TAUT is the dual problem of SAT, but TAUT and $\overline{\text{SAT}}$ are two totally different problems. Likewise, the concepts of duality and opposites are also distinct from each other. The dual of the UNSAT problem can be defined as:

$$\text{coUNSAT} = \{\langle\phi\rangle \mid \phi \text{ is a DNF formula and } \exists\alpha \;\; \phi(\alpha) = 0\}$$

which, curiously, turns out to be the opposite of the TAUT problem.

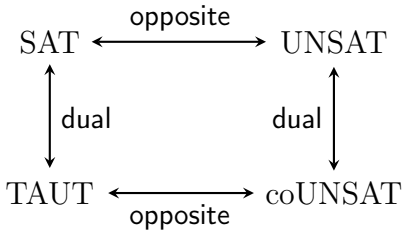


Figure 2.8: Summary of the relations between types of satisfiability problems.

Now that we understand how disqualification and duality are two equivalent concepts for NP and coNP, Let's try to extend this concept to the class P. By definition, we have that $\text{coP} = \{L \subseteq \{0,1\}^* \mid \overline{L} \in \text{P}\}$. However, we can easily see that these two classes are actually the same: if a language L is decided by M in polynomial time, then $\neg M$ clearly decides $\overline{L}$ in polynomial time. Hence, we get that $\text{P} = \text{coP}$. This property is usually called **closure under complement**.

However, the same argument cannot be used to show that $\text{NP} = \text{coNP}$. For instance, the opposite of a verifier is (usually) not a verifier for the complementary language. Similarly, the opposite of a NDTM is not a valid NDTM for the complementary language: the original machine could have an accepting and a rejecting branch, hence the opposite machine would also have a rejecting and an accepting branch, but the latter may accept even strings that aren't in the original language.

This property is an intrinsic difference between determinism and non-determinism. In fact, the opposite-machine-argument can be used to show that $\text{EXP} = \text{coEXP}$, but it doesn't work for the question $\text{NEXP} \stackrel{?}{=} \text{coNEXP}$.

Moreover, this closure under complement property of the class P can be used show that if $\text{P} = \text{NP}$ then trivially $\text{NP} = \text{P} = \text{coP} = \text{coNP}$. By contrapositive, we get the following theorem.

Theorem 13

If $\text{coNP} \neq \text{NP}$ then $\text{P} \neq \text{NP}$

This result gives us our first way to potentially show that $\text{P} \neq \text{NP}$. Nowadays, researchers believe that $\text{coNP} \neq \text{NP}$ is true, but none has yet proven it. The hardness of this question follows from the fact that a language can be NP-Complete if and only if its complement

is coNP-Complete. Here, the concept of coNP-Completeness follows the same definition of NP-Completeness, where NP is replaced by coNP.

Theorem 14

L is NP-Complete if and only if $\overline{L}$ is coNP-Complete.

Proof. Suppose that B is NP-Complete. Since $B \in \mathsf{NP}$, we trivially get that $\overline{B} \in \text{coNP}$. Moreover, since B is also NP-Hard, we know that for each problem $A \in \mathsf{NP}$ there is a function f such that:

$$x \in A \iff f(x) \in B$$

We notice that this very same functions is also a reduction from the complement of A to the complement of B:

$$x \in \overline{A} \iff x \notin A \iff f(x) \notin B \iff f(x) \in \overline{B}$$

Hence, for each problem $\overline{A} \in \mathsf{coNP}$, we have that $\overline{A} \leq_m \overline{B}$ □

Hence, if we are able to show that an NP-Complete problem (or a coNP-Complete one) also lies inside coNP (or NP) then NP = coNP. However, this is equal to proving that NP-Completeness and coNP-Completeness are the same thing, which is indeed an extremely hard question.

Furthermore, the previous theorem automatically implies that $\overline{\text{SAT}}$ is coNP-Complete. Despise opposite and complementary problems being two very distinct things, we can actually always reduce the complementary problem to the opposite problem. For instance, consider the following machine:

M = "Given the input string x:

1. Check if $x = \langle \phi \rangle$ for some formula ϕ.
2. If true, return x. Otherwise, return $\langle y \wedge \neg y \rangle$"

The trick here is to map every string that isn't a valid encoding of an unsatisfiable CNF formula to a pre-fixed unsatisfiable formula, i.e. the formula $y \wedge \neg y$, while every string that is a valid gets mapped to itself. This reduction implies that $\overline{\text{SAT}} \leq_P \text{UNSAT}$, meaning that UNSAT is coNP-Hard. Moreover, this problem is also clearly in coNP: the disqualifying witness is an assignment that satisfies the formula. Hence, we get that UNSAT is coNP-Complete.

Moreover, when a problem is the dual of the opposite (or the opposite of the dual) of a problem, we can often reduce the former to the latter and vice versa. For instance, the problem UNSAT can easily be reduced to the problem TAUT: a CNF formula ϕ is unsatisfiable if and only if the DNF formula $\neg\phi$ is a tautology, hence $\text{UNSAT} \leq_P \text{TAUT}$. This implies that TAUT is also coNP-Complete.

3

Diagonalization and Hierarchies

3.1 Hierarchy theorems

In Section 1.3 we discussed how the *diagonal argument* can be used to prove that uncomputable functions exist. This argument is quite useful in order to prove many other theorems in the field of computational complexity. In particular, we're interest in showing the **hierarchy theorems**. These theorems highlight a fundamental property of computation: as we permit more resources, the computational power of the model strictly increases. This result may seem trivial, but it actually provides insights into the structural distinctions between different classes, underscoring the trade-offs between resource constraints and computational capability.

First, we'll discuss the **Time Hierarchy Theorem** which shows that allowing Turing machines more computation time strictly increases the set of decidable languages. For now, we focus on deterministic Turing machines.

Theorem 15: Determinisitc Time Hierarchy Theorem

For all functions $f, g : \mathbb{N} \to \mathbb{N}$ with g time-constructible and $f(n) = o(g(n))$

$$\text{DTIME}(f(n)) \subsetneq \text{DTIME}(g(n) \log g(n))$$

Proof. It trivially holds that $\text{DTIME}(f(n)) \subseteq \text{DTIME}(g(n)) \subseteq \text{DTIME}(g(n) \log g(n))$. We will show that the inclusion between $f(n)$ and $g(n) \log g(n)$ is actually strict.

Consider the following TM M defined as:

M = "Given the input string x:

1. Compute $\langle x \rangle = n$
2. Compute $g(n)$ in a time-constructible way

3. Store the value $\lceil g(n) \rceil$ in a counter. If such counter ever reaches the value 0 during the following instructions, M immediately *rejects*.
4. Check if $x = \alpha 10^*$, where M_α is a TM. If the interpretation fails, M *rejects*.
5. Simulate M_α with input w. After each step of the simulation, decrement the counter by one.
6. If M_α accepts, M *rejects*. Otherwise, M *accepts*"

We notice that each step of the simulation run by M has to decrement the counter, which has size $\log \lceil g(n) \rceil$, requiring $d \cdot \log g(n)$ steps for some $d \in \mathbb{R}$. We also notice that even if the simulated machine would go into infinite loops, the simulation done by M will still halt when the counter reaches zero, meaning that M always halts after at most $d \cdot g(n) \log g(n)$ steps (the logarithmic factor is due to the counter). Hence, we know that the language $L(M)$ can be decided in time $O(g(n) \log g(n))$.

Through the diagonal argument, we know that there is an encoding $\beta \in \{0,1\}^*$ such that M_β that $L(M)$ is also decided by a TM M_β. By way of contradiction, suppose that M_β runs in time $f(n)$. By definition of little-oh notation, we have that:

$$\forall c \in \mathbb{R}_{>0}\ \exists n_0 \in \mathbb{N}_{>0} \mid \forall n \geq n_0 \quad f(n) < c \cdot g(n)$$

Hence, given that $\frac{1}{d} \in \mathbb{R}^+$, we know that $\exists n_0 \in \mathbb{N}$ such that $\forall n \geq n_0$ it holds that $f(n) < \frac{1}{d} \cdot g(n)$, and thus that $d \cdot f(n) < g(n)$.

Consider the input string $x = \beta 10^{n_0}$. Since $|x| \geq n_0$, the simulation of M_β done by M will run in $f(n) < d \cdot f(n) < g(n)$ steps, thus the counter will never reach the value zero. This means that $M(x)$ will simulate all the computation of $M_\beta(x)$ and return the opposite result, hence we get that $x \in L(M)$ if and only if $x \notin L(M_\beta) = L(M)$, which is a contradiction. This concludes that M_β cannot run in time $f(n)$, hence $\text{DTIME}(f(n)) \subsetneq \text{DTIME}(g(n) \log g(n))$. □

This hierarchy theorem can be used to give another proof of the fact that P $\subsetneq$ EXP: for all $k \in \mathbb{N}$ we have that $n^k = o\left(\frac{2^n}{\log 2^n}\right)$, hence $\text{DTIME}(n^k) \subsetneq \text{DTIME}(2^n) \subseteq \text{DTIME}(2^{n^h})$ for all $h > 1$. Since each $\text{DTIME}(n^k)$ is strictly contained inside each $\text{DTIME}(2^{n^h})$, this concludes that each $\text{DTIME}(n^k)$ is strictly contained in EXP, thus P is also strictly contained. For Non-deterministic Turing machines, instead, we can define an even stronger hierarchy theorem, where the logarithmic factor is not needed. The idea is similar to the previous proof (with some little tweaking), so we will omit it.

Theorem 16: Non-deterministic Time Hierarchy Theorem

For all functions $f, g : \mathbb{N} \to \mathbb{N}$ with g time-constructible and $f(n+1) = o(g(n))$

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

Proof. Omitted. □

The same idea also holds for the **Space Hierarchy Theorem**. However, in this case the deterministic space hierarchy theorem is actually stronger than the deterministic time hierarchy theorem. This is due to the possibility of re-using the same cells, preventing the logarithmic blow-up. Moreover, the same result holds for both the deterministic and non-deterministic versions of the theorem.

In this context, the concept of space-constructible function follows the same idea of time-constructible functions: a function $S : \mathbb{N} \to \mathbb{R}^+$ is said to be **space-constructible** if there is a TM M that computes it using at most $S(n)$ cells for all $n \in \mathbb{N}$. Likewise, the classes DSPACE and NSPACE follow the same definition of DTIME and NTIME, where time is replaced by space.

Theorem 17: Space Hierarchy Theorem

For all functions $f, g : \mathbb{N} \to \mathbb{N}$ with g space-constructible and $f(n) = o(g(n))$

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

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

Proof. Omitted. □

Interestingly, there is an hierarchy theorem even for the size of non-uniform circuits. Even though this result can indeed be proved through diagonalization, previously shown results for upper and lower bounds on circuits are sufficient.

Theorem 18: Non-uniform Hierarchy Theorem

For all functions $f, g : \mathbb{N} \to \mathbb{N}$ with $n < 10f(n) < g(n)\frac{2^n}{n}$, it holds that

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

Proof. From Theorem 26, we know that for each $\ell > 1$ there is a Boolean function $h : \{0,1\}^\ell \to \{0,1\}$ such that $h \notin \text{SIZE}\left(\frac{2^\ell}{10\ell}\right)$. Moreover, from Theorem 25 we know that for such function h it holds that $h \in \text{SIZE}\left(\frac{2^\ell}{\ell}\right)$ gates.

Choose a value of ℓ such that $f(n) = \Theta\left(2^\ell\right) 10\ell$, trivially concluding that $h \notin \text{SIZE}(f(n))$. Since $10f(n) < g(n)$,we conclude that:

$$h \in \text{SIZE}\left(\frac{2^\ell}{\ell}\right) = \text{SIZE}\left(10f(n)\right) \subseteq \text{SIZE}\left(g(n)\right)$$

□

3.2 Ladner's theorem

One of the striking aspects of NP-Completeness is that a surprisingly large number of NP problems turned out to be be NP-Complete. This phenomenon suggests a bold conjecture: every problem in NP is either in P or NP-Complete. In other words, this conjecture states that there is no NP-Intermediate language, i.e. a language in NP that is not in P and not NP-Complete.

Definition 17: NP-Intermediateness

A language $L \in \mathsf{NP} - \mathsf{P}$ is said to be **NP-Intermediate** if it also isn't NP-Complete.

If $\mathsf{P} = \mathsf{NP}$, this conjecture is trivially true since every problem in P is Karp reducible to each other. However, we believe that $\mathsf{P} \neq \mathsf{NP}$. In this case, the conjecture turns out to be false. Ladner was able to prove that, under this assumption, there is an NP-Intermediate language.

Theorem 19: Ladner's theorem

If $\mathsf{P} = \mathsf{NP}$ then there is an NP-Intermediate language.

Proof. For every function $f : \mathbb{N} \to \mathbb{N}$, we define SAT_f as the language of all the satisfiable formulas of length n that are padded with $1^{n^{f(n)}}$.

$$\mathsf{SAT}_f = \left\{ \left\langle \phi 01^{n^{f(n)}} \right\rangle \mid \phi \in \mathsf{SAT}, |\phi| = n \right\}$$

Consider now the function $H : \mathbb{N} \to \mathbb{N}$ such that $H(n)$ is the smallest number $i < \log\log n$ such that for every $x \in \{0,1\}^*$ with $|x| \leq \log n$, the machine M_i (where i's binary expansion acts as the encoding) computes $\mathrm{SAT}_H(x)$ within $i\,|x|^i$ steps. If there is no such number i then $H(n) = \log\log n$.

We notice that the function H is well defined: $H(n)$ determines membership in SAT_H of strings whose length is greater than n and the definition of $H(n)$ only relies upon checking the status of strings of length at most $\log n$. In fact, we can easily define a recursive algorithm that computes $H(n)$ in time $O(n^3)$.

Claim: $\mathrm{SAT}_H \in \mathsf{P}$ if and only if $H(n) = O(1)$. Moreover, if $\mathrm{SAT}_H \notin \mathsf{P}$ then $H(n) \to +\infty$.

Proof of the claim. Suppose that $\mathrm{SAT}_H \in \mathsf{P}$ and let M the TM that computes SAT_H in polynomial time, say kn^k steps for some $k \in \mathbb{N}$. By definition of H, there must be a number i satisfying the constraints. Moreover, since M can encoded by infinitely many strings, we can assume that i is large enough to imply that $k < i$ and $M = M_i$. Then, for $2^{2^i} < n$ we have that $H(n) \leq i$, hence $H(n) = O(1)$.

Suppose now that $H(n) \leq c$ for some constant $c \in \mathbb{R}$. Then, H's image is finite, hence there must be an i such that $H(n) = i$ for infinitely many n. By way of contradiction, suppose that M_i doesn't compute SAT_H within in^i steps. Then, there must be an input

x such that for every $2^{|x|} < n$ it holds that $H(x) \neq i$, raising a contradiction. Thus, we get that

$$\exists c \in \mathbb{N} \mid H(n) \leq c \implies \text{SAT}_H \in \mathsf{P}$$

The previous implication can be used to get two results:

- If $H(n) = O(1)$ then clearly there is a constant $c \in \mathbb{N}$ such that $H(n) \leq c$, concluding that $H(n) = O(1)$ implies that $\text{SAT}_H \in \mathsf{P}$
- The contrapositive implication states that if $\text{SAT}_H \notin \mathsf{P}$ then $H(n) \to +\infty$.

□

The previous claim holds even without the assumption that $\mathsf{P} \neq \mathsf{NP}$. However, if we assume the latter, then the language SAT_H must be NP-Intermediate.

Suppose that $\text{SAT}_H \in \mathsf{P}$. Then, by the previous claim, we know that $\exists c \in \mathbb{N}$ such that $H(n) \leq c$. Hence, we can build a machine M that takes a formula ϕ as input, pads it with 01^{n^c} and runs the polynomial algorithm for SAT_H in order to solve SAT in polynomial time, which would conclude that $\text{SAT} \in \mathsf{P}$ and thus that $\mathsf{P} = \mathsf{NP}$, contradicting the assumption. Hence, $\text{SAT}_H \notin \mathsf{P}$.

Suppose now that SAT_H is NP-Complete. This implies that there is a Karp reduction from SAT to SAT_H that runs in time $O(n^i)$ for some $i \in \mathbb{N}$. This means that, for large enough input lengths, the reduction must map instances of length n in SAT to instances of SAT_H with length smaller than $n^{H(n)}$, since otherwise the reduction wouldn't be computed in $O(n^i)$ steps. Thus, for large enough formulae, the reduction must map ϕ to $\psi 01^{H(|\psi|)}$ instead of $\psi 01^{n^{H(|\psi|)}}$.

However, we already showed that $\text{SAT}_H \notin \mathsf{P}$, thus the claim implies that $H(n) \to +\infty$. This means that the reduction can be used to — in some sense — recursively reduce the size of the formula, giving us a polynomial time algorithm for SAT, again contradicting the assumption. Hence, SAT_H cannot be NP-Complete.

□

Ladner's theorem is a significant result in computational complexity theory. However, the problem used by the theorem is clearly artificial and "weird". Currently, no "natural" problems are known to be NP-Intermediate under the assumption that $\mathsf{P} \neq \mathsf{NP}$. Finding such problem would provide valuable insight on what characteristics make a problem computable in polynomial time or NP-Complete.

A good candidate for being a natural NP-Intermediate is the *graph isomorphism problem*, which asks to determine if two graphs are isomorphic to each other. This problem is known to be solvable in **quasi-polynomial time**, that is in $O(2^{\log^c n})$ or equivalently $O(n^{\log^{c-1} n})$, even without the assumption that $\mathsf{P} \neq \mathsf{NP}$.

3.3 Oracles and the limits of diagonalization

Quantifying the limits of diagonalization is not easy. For concreteness, let us say that a *diagonal argument* is any technique that relies solely on the existence of an effective representation of Turing machines by strings and the ability of an Universal TM to simulate any another without much overhead in running time or space. Any argument based only in these facts is treating machines as **black boxes**: the machine's internal workings do not matter. This implies that any diagonal argument is also valid for **oracle Turing machines**.

Definition 18: Oracle Turing machine

An **oracle** is a black-box device that can magically solve in $O(1)$ the decision problem for some language $O \subseteq \{0,1\}^*$. An **oracle Turing machine** M^O can query an oracle for O through a *special oracle tape* on which it can write a string $q \in \{0,1\}^*$, go into a *special oracle state* to activate the oracle. After the query, the oracle tape will contain 0 if $q \notin O$ and 1 if $q \in O$.

Clearly. if O is a difficult language then this oracle gives a huge amount of power to the TM M^O. For every language $O \subseteq \{0,1\}^*$, we denote with P^O the set containing every language that can be decided by a polynomial-time deterministic TM with access to an oracle for O. Likewise, NP^O is the set of every language that can be decided by a polynomial-time nondeterministic TM with access to an oracle for O. It trivially holds that $\mathsf{P} \subseteq \mathsf{P}^O$ and $\mathsf{NP} \subseteq \mathsf{NP}^O$. These notations can be extended in an intuitive way. For instance, we have that:

$$\mathsf{P}^{\mathsf{P}} = \bigcup_{O \in \mathsf{P}} \mathrm{P}^O$$

Oracles make reasoning about computation very easy. For instance, consider an oracle for a language $O \in \mathsf{P}$. Then, we have that $\mathsf{P}^O = \mathsf{P}$: each query to the oracle can be replaced by an algorithm that solves O in polynomial time. This clearly concludes that $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$. Likewise, it's easy to see that for any NP-Complete language L it holds that $\mathsf{P}^L = \mathsf{P}^{\mathsf{NP}}$.

The key fact about oracle TMs is that regardless of what the oracle O is , the set of all TMs with access to O satisfy the two properties of any diagonal argument: just like a standard machine, we can represent each oracle machine M^O as strings and simulate such TM using a Universal TM U^O that itself also has access to an oracle for O. Thus any result about Turing machine that are based on diagonalization can also be extended for oracle machines. Such concept is called **relativization**. Many results in complexity theory are based on such concept.

However, relativization actually *limits* the power of diagonalization: we can define two languages A and B for which $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B = \mathsf{NP}^B$. We wont dive into this proof as it requires the languages A an B are both convoluted — think of what we did for Ladner's theorem.

Theorem 20: The Baker-Gill-Solovay Theorem

There are two languages A and B for which $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B = \mathsf{NP}^B$.

Proof. Omitted □

This theorem implies that any diagonalization or simulation proof that solves the question $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ must have some *non-relativizable constraint* that impose that the proof would cannot be translated into a result for oracle machines. Even though many results in complexity relativize, there are some notable exceptions that will be discussed in future chapters.

3.4 The polynomial hierarchy

We have already explored several methods for "capturing" the core characteristics of families of computational problems by demonstrating their completeness within certain natural complexity classes. This section advances this exploration by examining another family of natural problems whose complexity cannot be fully represented by nondeterminism alone. To capture these problems, we define the **Polynomial Hierarchy** (PH), which generalizes the classes P, NP and coNP.

To motivate the study of PH, we focus on some computational problems that seem to not be captured by NP-Completeness. First, recall that we proved how *independent set problem* is NP-Complete.

$$\text{IND-SET} = \begin{cases} 1 & \text{if } x = \langle G, k \rangle \text{ and } G \text{ is a graph with an ind. set } S \text{ of size } |S| \geq k \\ 0 & \text{otherwise} \end{cases}$$

In order to certify that $\langle G, k \rangle \in$ IND-SET, any independent set of size k can act as a polynomial sized witness. Now, consider the following variation of the independent set problem, i.e. the *exact independent set problem*:

$$\text{EXACT-IND-SET} = \begin{cases} 1 & \text{if } x = \langle G, k \rangle \text{ and } G \text{ is a graph whose largest ind. set is of size } k \\ 0 & \text{otherwise} \end{cases}$$

By definition, $\langle G, k \rangle \in$ EXACT-IND-SET if and only if there is an independent set of size k in G and every other independent set has size at most k. In this case, the only possible certificate seems to be the set of all the independent sets of G, which requires exponential length in the worst case. This means that it probably holds that EXACT-IND-SET $\in \mathsf{NP}$.

To better underline the difference between these two problems, Let's rewrite their languages through *quantifiers*:

$$\text{IND-SET} = \{\langle G, k \rangle \mid \exists S \text{ ind. set such that } |S| \geq k\}$$

$$\text{EXACT-IND-SET} = \{\langle G, k \rangle \mid \exists S \text{ ind. set such that } \forall S' \text{ind. set } |S'| \leq |S| = k\}$$

The difference between the two problems seems to rely on the presence of an **additional universal quantifier** after the existence quantifier. This universal quantifier is what makes the witness too large. To bypass this issue, we can extend the concept of verification by *splitting* the process of verification in two parts: one for the existential quantifier and one for the universal quantifier. We define the class Σ_2^P as the set of languages for which there is a polynomial time TM M and a polynomial q such that:

$$L \in \Sigma_2^P \iff \forall x \in L \; \exists u \{0,1\}^{q(|x|)} \; \forall v \{0,1\}^{q(|x|)} \; M(x,u,v) = 1$$

We observe that in this case the exponent P is just a notation and thus not related to oracles. This class is an extension of the class NP: we can just ignore the universal quantifier. Hence, it holds that $\mathsf{NP} \subseteq \Sigma_2^P$. We can easily show that EXACT-IND-SET $\in \Sigma_2^P$: just use the existential verification for the maximum cardinality independent set and the universal verification for every other independent set.

Now, consider the opposite problems opIND-SET and opEXACT-IND-SET:

$$\text{opIND-SET} = \{\langle G, k\rangle \mid \forall S \text{ ind. set } |S| \geq k\}$$

$$\text{opEXACT-IND-SET} = \{\langle G, k\rangle \mid \forall S \text{ ind. set } \exists S' \text{ind. set } |S'| \leq |S| = k\}$$

We easily notice that opIND-SET $\in \mathsf{coNP}$. However, for opEXACT-IND-SET we have the same issue as before: we need two quantifiers. To capture this language, we can extend the class coNP by defining the class Π_2^P as the set of languages for which there is a polynomial time TM M and a polynomial q such that:

$$L \in \Pi_2^P \iff \forall x \in L \; \forall u \{0,1\}^{q(|x|)} \; \exists v \{0,1\}^{q(|x|)} \; M(x,u,v) = 1$$

Interestingly, we notice that this class is still an extension of the class NP: we can just ignore the universal quantifier. Hence, it holds that $\mathsf{NP} \subseteq \Pi_2^P$. In the same way, we also get that $\mathsf{coNP} \subseteq \Sigma_2^P$ and $\mathsf{coNP} \subseteq \Pi_2^P$. To creare the polynomial hierarchy, we can generalize these two classes in order to allow for an alternating amount of existence and universal quantifiers (or viceversa).

Definition 19: The classes Σ_i^P and Π_i^P

For $i \geq 0$, we define the class Σ_i^P as the set of languages for which there is a polynomial time TM M and a polynomial q such that $L \in \Sigma_i^P$ if and only if

$$\forall x \in L \; Q_1 w_1 \{0,1\}^{q(|x|)} \; Q_2 w_2 \{0,1\}^{q(|x|)} \; \ldots \; Q_i w_i \{0,1\}^{q(|x|)} \; M(x, w_1, \ldots, w_i) = 1$$

where Q_j is equal to $\exists$ if j is odd and equal to $\forall$ if j is even. Likewise, we define the class Π_i^P as the set of languages for which there is a polynomial time TM M and a polynomial q such that $L \in \Pi_i^P$ if and only if

$$\forall x \in L \; Q_1 w_1 \{0,1\}^{q(|x|)} \; Q_2 w_2 \{0,1\}^{q(|x|)} \; \ldots \; Q_i w_i \{0,1\}^{q(|x|)} \; M(x, w_1, \ldots, w_i) = 1$$

where Q_j is equal to $\forall$ if j is odd and equal to $\exists$ if j is even.

We notice that, by definition, we have that $\mathsf{P} = \Sigma_0^P = \Pi_0^P$, while $\mathsf{NP} = \Sigma_1^P$ and $\mathsf{coNP} = \Pi_1^P$. Moreover, we also notice that $\Sigma_i^P = \mathsf{co}\Pi_i^P$ and thus $\Pi_i^P = \mathsf{co}\Sigma_i^P$. Generalizing the idea showed before, we notice that $\Sigma_i^P \subseteq \Pi_{i+1}^P$ and that $\Pi_i^P \subseteq \Sigma_{i+1}^P$. The **Polynomial Hierarchy** is formed of all the quantification levels:

$$\mathsf{PH} = \bigcup_{i \geq 0} \Sigma_i^P = \bigcup_{i \geq 0} \Pi_i^P$$

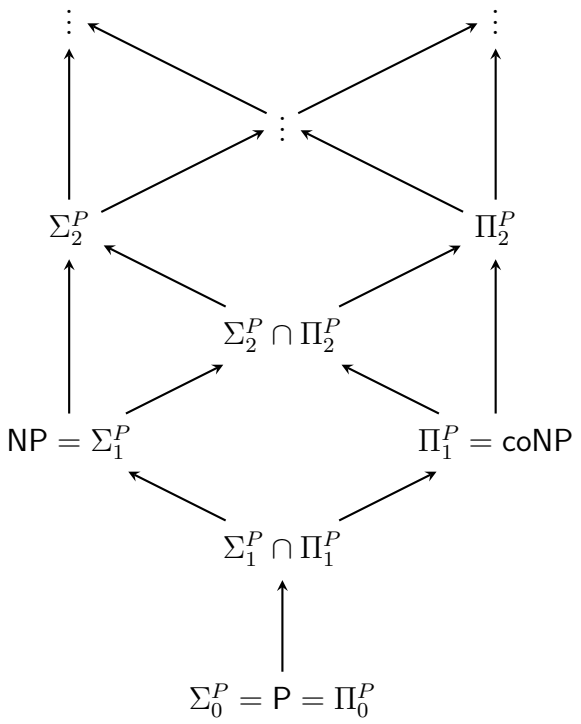


Figure 3.1: The polynomial hierarchy

Interestingly, for each i-th level of the hierarchy we can define a Σ_i^P**-Complete** problem, i.e. the Σ-alternated quantified version of the satisfiability problem:

$$\Sigma_i^P\text{SAT} = \{\langle\phi\rangle \mid Q_1 x_1\ Q_2 x_2\ \ldots\ Q_i x_i\ \phi(x_1, \ldots, x_i) = 1\}$$

where Q_j is equal to $\exists$ if j is odd and equal to $\forall$ if j is even. Likewise, we can define a Π_i^P**-Complete** problem, i.e. the Π-alternated quantified version of the satisfiability problem:

$$\Pi_i^P\text{SAT} = \{\langle\phi\rangle \mid Q_1 x_1\ Q_2 x_2\ \ldots\ Q_i x_i\ \phi(x_1, \ldots, x_i) = 1\}$$

where Q_j is equal to $\forall$ if j is odd and equal to $\exists$ if j is even.

We believe that $\mathsf{P} \neq \mathsf{NP}$ and $\mathsf{NP} \neq \mathsf{coNP}$. An appealing generalization of these conjectures is that for every $i \geq 0$, it holds that $\Sigma_i^P \subsetneq \Sigma_{i+1}^P$ and that $\Pi_i^P \subsetneq \Pi_{i+1}^P$. This conjecture is used often in complexity theory and is usually stated as "the polynomial hierarchy does not collapse", where the polynomial hierarchy is said to collapse if for some j it holds that $\Sigma_j^P = \Sigma_{j+1}^P$. For instance, one can easily prove the following result by induction.

Theorem 21: Collapses of the Polynomial Hierarchy

For all $i \geq 1$, it holds that:

1. If $\Sigma_{i-1}^P = \Sigma_i^P$ then $\mathsf{PH} = \Sigma_{i-1}^P$
2. If $\Pi_{i-1}^P = \Pi_i^P$ then $\mathsf{PH} = \Pi_{i-1}^P$
3. If $\Sigma_i^P = \Pi_i^P$ then $\mathsf{PH} = \Sigma_i^P$

Proof. Omitted. □

As a corollary of this theorem, we get that if $\mathsf{P} = \mathsf{NP}$ then $\mathsf{P} = \mathsf{PH}$ and that if $\mathsf{NP} = \mathsf{coNP}$ then $\mathsf{NP} = \mathsf{PH}$. These two results should be enough evidence to believe that $\mathsf{P} \neq \mathsf{NP}$ and $\mathsf{NP} \neq \mathsf{coNP}$ are almost certainly true — how could anyone believe such collapses?

The polynomial hierarchy can also be recursively described through **oracles**: for all $i \geq 1$, it holds that $\Sigma_i^P = \mathsf{NP}^{\Sigma_i^P}$ and that $\Pi_i^P = \mathsf{coNP}^{\Sigma_i^P}$. Following this idea, we can also define a new type of hierarchy level: for all $i \geq 0$, we define $\Delta_i^P = \mathsf{P}^{\Sigma_i^P}$. In particular, we have that $\Delta_i^P \subseteq \Sigma_i^P \cap \Pi_i^P$ and that $\mathsf{P} = \Delta_0^P = \Delta_1^P$.

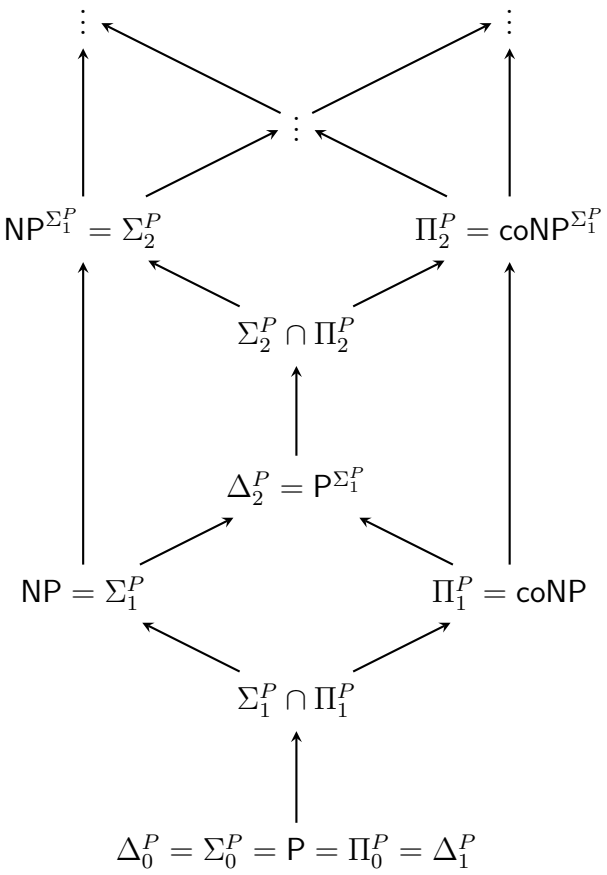


Figure 3.2: Oracle definition of the polynomial hierarchy

4

Boolean circuits

4.1 Boolean circuits and the class P/poly

Boolean circuits are defined as sets of logical AND, logical OR and logical NOT gates connected by cables. Boolean circuits have been proven to be Turing complete due to Turing machines and circuits being capable of simulating each other up to a polynomial factor. Again, none should be dumbfounded by this result: any modern computer is just a large amount of Boolean gates wired together.

Definition 20: Boolean circuit

A **Boolean circuit** is a directed acyclic graph whose nodes, called gates, each associated with either an input variable or a Boolean operator. Each input gate has in-degree 0 and unlimited out-degree. Each Boolean gate has an out-degree gate equal to 1 (except for the output gate which has out-degree 0) and in-degree equal to either 1 or 2. All the 1 in-degree gates compute the logical NOT and all 2 in-degree gates compute the logical AND or the logical OR of their given input variables or Boolean function.

Each gate v is associated with the Boolean function f_v computed by it. An assignment $x = x_1, \ldots, x_n$ defines the result of the computation for a gate. A function $f : \{0,1\}^n \to \{0,1\}$ is said to be computed by a circuit with output gate u if for all inputs $x \in \{0,1\}^n$ it holds that $f(x) = C(x)$, where $C(x)$ is the function computed by the output gate.

The complexity of Boolean circuits is measured in terms of their *size* and *depth*, i.e. the number of gates of the circuit and the length of the longest directed path from an input gate to the output gate. The **circuit complexity** of a function f is defined as the size of the smallest Boolean circuit that computes it.

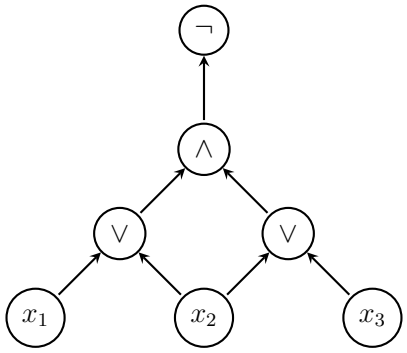


Figure 4.1: A Boolean circuit of size 7 and depth 3 computing $\overline{(x_1 \vee x_2)(x_2 \vee x_3)}$.

Boolean circuits can also be defined in a different way. Instead of using logical NOT gates, we can assume that the the negations $\overline{x_1}, \ldots, \overline{x_n}$ are also input variables. Any standard circuit of size S and depth D can be easily transformed into this different type of circuit by repeatedly applying the De Morgan rule on all the logical NOT gates, producing a circuit of size at most $2S$ and depth at most D due to the number of input gates being doubled and the logical NOT gates being removed. These circuits are usually called **De Morgan circuits**.

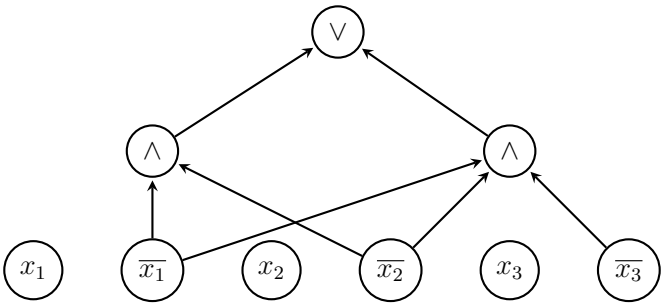


Figure 4.2: A De Morgan circuit of size 9 and depth 2 computing $\overline{(x_1 \vee x_2)(x_2 \vee x_3)}$.

An important thing to notice is that Boolean circuits do **not allow computations to be reused**. For example, consider the *n bit XOR function*:

$$\mathrm{XOR}(x) = \bigoplus_{i=1}^{n} x_i$$

Intuitively, we can compute this function through recursion:

$$(\neg\mathrm{XOR}(x_1, \ldots, x_{\frac{n}{2}}) \vee \mathrm{XOR}(x_{\frac{n}{2}+1}, \ldots, x_n)) \wedge (\mathrm{XOR}(x_1, \ldots, x_{\frac{n}{2}}) \vee \neg\mathrm{XOR}(x_{\frac{n}{2}+1}, \ldots, x_n))$$

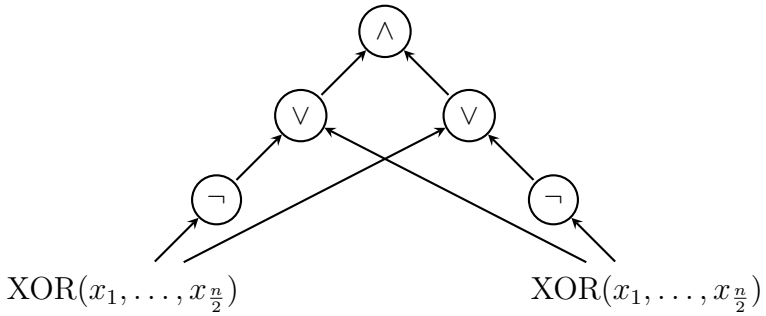


Figure 4.3: Recursive computation of XOR(x)

If we could model a circuit in this way, the size would be given by $S(n) = 5 + 2S\left(\frac{n}{2}\right)$, which is approximately $O(n)$. Without reusing computations, instead, we have to make two copies of each recursion.

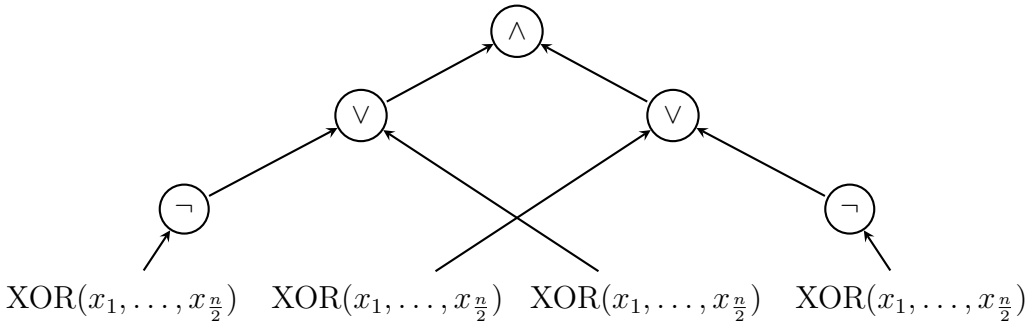


Figure 4.4: Recursive Boolean circuit that computes XOR(x)

Hence, the size of the circuit is actually given by $S(n) = 5 + 4S\left(\frac{n}{2}\right)$, that is $O(n^2)$. Differently from Turing machines, circuits are capable of computing only functions with a fixed amount of input bits. To compute a variable input size, we need a *family of circuits* $\{C_n\}_{n\in\mathbb{N}}$, where C_n computes all the inputs of length i.

Definition 21: Circuit family

A $T(n)$-size **circuit family** is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs, a single output and size at most $T(n)$. We say that a language L is decidable by $\{C_n\}_{n\in\mathbb{N}}$ if $\forall x \in \{0,1\}^n$ it holds that $x \in L$ if and only if $C_n(x) = 1$.

Through circuit decidability, we can define an equivalent of the class P, i.e. the class $\mathsf{P}_{/\mathsf{poly}}$. The notation "/poly" comes from the equivalence between polynomial sized circuit families and Turing machines that take a polynomial amount of "advice bits", a topic that will be discussed in later sections. Circuit families are even capable of simulating non-advice-taking Oblivious Turing machines. This result will enable us to get many strong results, including the proof of The Cook-Levin theorem that we omitted before.

Definition 22: The class $\mathsf{P}_{/\mathsf{poly}}$

We define the class of the languages decidable by a poly-sized circuit family:

$$\mathsf{P}_{/\mathsf{poly}} = \bigcup_{k \geq 0} \mathrm{SIZE}(n^k)$$

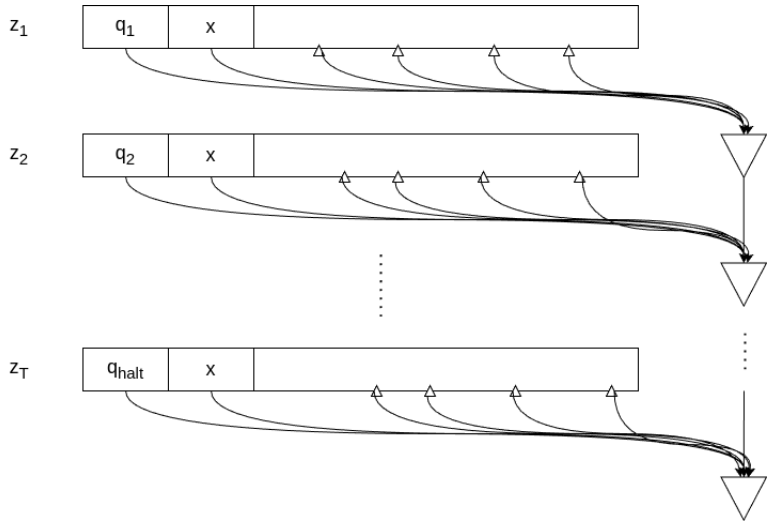
where $\mathrm{SIZE}(f(n))$ is the set of all languages decided by a $f(n)$-sized circuit family

Theorem 22: Simulating TMs through circuit families

Let L be a language decided by a TM in time $T(n)$, where T is time-constructable, and let $L_n = L \cap \{0,1\}^n$ for all $n \in \mathbb{N}$. There is a circuit C_n of size $cT(n)\log_2 T(n)$ such that $x \in L_n$ if and only if $C_n(x) = 1$,

Proof. Let M be the k-taped TM that decides L in time $T(n)$. We know that M can be converted into an Oblivious TM M' that decides L in time $T(n)\log_2 T(n)$. Let $T = T(n)\log_2 T(n)$ and le $z_1, \ldots, z_T$ be the sequence of *snapshots* of the computation of M', that is the sequence of the machine's state and symbols read by all the k-heads. We know that M' can use at most T cells, meaning that we can restrict our focus on the first T cells of every tape. Moreover, since M' is oblivious, we know that each i-th step of computation only depends on n and i, thus we can ignore the movements of the heads.

Hence, each snapshot z_i is encoded by a constant-sized binary string, meaning that we can compute the string z_i based on the input length n, the previous snapshot z_{i-1} and the snapshots $z_{i_1}, \ldots, z_{i_k}$, where z_{i_j} denotes the last step that the j-th head of M' was in the same position as it is in the i-th step. The composition of all the T constant-sized circuits gives rise to a circuit C_n of size $O(T)$ that computes from the input x the snapshot z_T of the last step of $M'(x)$, meaning that $C_n(x) = M'(x)$. Applying this process for all $n \in \mathbb{N}$, we get a circuit family that decides L. □



Encoding of an Oblivious TM as a circuit

The previous result concludes that from each TM that runs in time $T(n)$ we can build a circuit family $\{C_n\}_{n\in\mathbb{N}}$ that can decide the same language with size $cT(n)\log_2 T(n)$, giving us the following additional result.

Corollary 1

$\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$

4.2 Proof of the Cook-Levin theorem

We're finally ready to give a simple proof for The Cook-Levin theorem. Consider the following *circuit satisfiability problem* as follows:

$$\text{CIRCUIT-SAT}(x) = \begin{cases} 1 & \text{if } x = \langle C \rangle \text{ and } C \text{ is a satisfiable circuit} \\ 0 & \text{otherwise} \end{cases}$$

A circuit is said to be satisfiable if there is at least one assignment of its variables such that $C(x) = 1$. Each circuit C_n built in the conversion from a TM to a circuit family can also be constructed in time $cT(n)\log_2 T(n)$. This allows us to prove that CIRCUIT-SAT is actually NP-Complete.

Lemma 1

CIRCUIT-SAT is NP-Complete

Proof. CIRCUIT-SAT is clearly in NP, since a satisfying assignment can act as a witness verifiable in polynomial time. We show that CIRCUIT-SAT is also NP-Hard. Let L be any language in NP and let V be verifier of L that runs in time at most n^k for some $k \in \mathbb{N}$. We define a TM M as follows:

M = "Given the input string $\langle x \rangle$:

1. Let $w \in \{0,1\}^m$ be a witness such that $V(x, w) = 1$
2. Compute $n = |x|$ and $m = |w|$.
3. Build the circuit C_V such that $C_V(x, w) = V(x_w)$ as described in Theorem 22
4. Build the circuit $C_x : \{0,1\}^m \to \{0,1\}$ such that $C_x(w) = C_V(x, w)$"

This machine computes a many-to-one reduction from L to CIRCUIT-SAT:

$$x \in L \iff \exists w \in \{0,1\}^* \;\; V(x, w) = 1 \iff$$

$$\exists w \in \{0,1\}^* \;\; C_x(w) = 1 \iff C_x \in \text{CIRCUIT-SAT}$$

Since both C_V and C_x can be built in $cn^k \log_2(n^k)$, we conclude that this reduction is actually a Karp reduction. □

Lemma 2

CIRCUIT-SAT $\leq_P$ CNF-SAT

Proof. Given a circuit C, let S be its size. We encode each gate g inside the circuit as a CNF ϕ_g:

- If g is a logical NOT gate with the input gate g_1 then we define ϕ_g as:

$$\phi_g = (g \vee g_1)(\overline{g} \vee g_1)$$

  This subformula evaluates as true if and only if $g = \neg g_1$.

- If g is a logical AND gate with input gates g_1, g_2, we define ϕ_g as:

$$\phi_g = (g \vee \overline{g_1} \vee \overline{g_2})(\overline{g} \vee g_1)(\overline{g} \vee g_2)$$

  This subformula evaluates as true if and only if $g = g_1 \wedge g_2$.

- If g is a logical OR gate with input gates g_1, g_2, we define ϕ_g as:

$$\phi_g = (\overline{g} \vee g_1 \vee g_2)(g \vee \overline{g_1})(g \vee \overline{g_2})$$

  This subformula evaluates as true if and only if $g = g_1 \vee g_2$.

Let $\phi_C = \bigwedge_{i=1}^{S} \phi_{g_i}$ be the formula that encodes every gate in C. This encoding perfectly describes the computation made by C. In order to force this computation, we consider the formula $\phi = g_S \wedge \phi_C$, where g_S is the output gate. We get that C is satisfiable if and only if the ϕ is satisfiable. Moreover, since each subformula ϕ_g has at most 7 literals inside it, the final formula ϕ has at most $1 + 7S$ literals, implying that it can be constructed in polynomial time. □

With this lemma, we have finally completed the following Karp reduction chain:

$$\text{CIRCUIT-SAT} \leq_P \text{CNF-SAT} \leq_P \text{3-SAT} \leq_P \text{SAT}$$

Since CIRCUIT-SAT is NP-Complete, through transitivity of Karp reductions we get that all these problems are NP-Complete, finally concluding the proof of the Cook-Levin theorem!

4.3 Machines that take advice and non-uniformity

As briefly mentioned before, the name of the class $\mathsf{P}_{/\mathsf{poly}}$ comes from it's alternative definition through Turing machines that take bits of advice. As per the NP case, this was the original definition, which quickly got replaced once the equivalence with circuits was proven. But what does "bits of advice" mean? The idea here is that if our machine has some tips on how the solution is formed, the computation can become stronger.

Definition 23: Advice-taking TM

Given a computable function f and a function $a \to \mathbb{N} \to \mathbb{N}$, we say that f is decided by a **TM** M **that takes** $a(n)$ **bits of advice** if there is a sequence $(\alpha_n)_{n\in\mathbb{N}}$, with $\alpha_n \in \{0,1\}^{a(n)}$, such that $\forall x \in \{0,1\}^n$ it holds that $f(x) = 1$ iff $M(x, a_{|x|}) = 1$

We observe that this definition is different from the concept of witness. In a verifier, we have to check that the witness w is actually valid, while in an advice-taking TM we assume that the bits are always correct. The other major difference is how the additional strings are pre-fixed: given two inputs of the same size $x, x' \in \{0,1\}^n$, they can have two different witnesses $w, w' \in \{0,1\}^*$, but they must use the same n bits of advice α_n.

Theorem 23: The class $\mathsf{P}_{/\mathsf{poly}}$ (2nd Definition)

We define the class of the languages decidable by a polynomial time TM that takes a polynomial amount of bits of advice:

$$\mathsf{P}_{/\mathsf{poly}} = \bigcup_{k,h\geq 0} \mathrm{DTIME}(n^k)_{/n^h}$$

where $\mathrm{DTIME}(f(n))_{/a(n)}$ is the set of all languages decided by a TM that runs in time $f(n)$ and takes $a(n)$ bits of advice

Proof. First, we notice that the same construction given in Theorem 22 can be used to also simulate an advice-taking TM by just adding gates for the advice. Vice versa, given $L \in \mathsf{P}_{/\mathsf{poly}}$ we know that there is a sequence of poly-sized circuits $(C_n)_{n\in\mathbb{N}}$ such decides L. Let $(a_n)_{n\in\mathbb{N}}$ be the sequence such that $a_n = \langle C_n \rangle$ and let $a(n) = |a_n|$. We can construct a machine M that for each $x \in \{0,1\}^n$ takes the additional input $a_n \in \{0,1\}^{a(n)}$ and simulates C_n, meaning that $M(x, a_n) = C_n(x)$. □

Advice-taking machines can decide any unary language with a single bit of advice. A language L is said to be **unary** if $L \subseteq 1^*$. In other words, every string of the language is in the form 1^n for some $n \in \mathbb{N}$. Given a language L, we define a machine M that first checks if the input x is of the form 1^n for some $n \in \mathbb{N}$ and then reads a single bit of advice a_n such that $a_n = 1$ if and only if $x \in L$. This result can also be proved through circuits: for each $n \in \mathbb{N}$, if $1^n \in L$ then we consider the circuit made of n AND gates, otherwise we consider the circuit that always outputs 0.

Proposition 13

Any unary language $L \subseteq 1^*$ is in $\text{DTIME}(n)_{/1}$ and $\text{SIZE}(n)$

We already proved that $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$. In fact, advice-taking Turing machines are clearly at least as strong as standard ones. However, we can easily prove that this type of machine is indeed way more stronger than the standard model. For example, consider the *unary halting problem*:

$$\text{UHALT}(x) = \begin{cases} 1 & \text{if } x = 1^n \text{ and } \langle n \rangle = \langle M, y \rangle \text{ for some pair } (M, y) \text{ s.t. } M(y) \text{ halts} \\ 0 & \text{otherwise} \end{cases}$$

UHALT is clearly uncomputable by a TM since otherwise we would be able to solve the halting problem. However, since this a unary language, we know that it is inside $\mathsf{P}_{/\mathsf{poly}}$. This result concludes that $\mathsf{P}_{/\mathsf{poly}}$ is different from every single class that we previously encountered and in particular that the inclusion $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$ is strict.

Corollary 2

$\mathsf{P} \subsetneq \mathsf{P}_{/\mathsf{poly}}$

Since circuits can simulate Turing machines and decide even uncomputable problems, this clearly concludes that circuits are actually stronger than machines. But where does this weird difference in power come from? The answer is simple: in the definition we require the circuit family to simply "exist", even if we have no idea of how the circuits are constructed. In a Turing machine, the computation is **uniform**, meaning that the instructions that define the computation are the same for all input lengths. For a circuit family, instead, each input length could be computed by a circuit completely different from the others, giving us a **non-uniform** computation. In order for a TM to be capable of simulating a circuit family, we need this family to also be uniform, i.e. capable of being generated through an algorithm.

Definition 24: Uniform circuit family

A circuit family $(C_n)_{n \in \mathbb{N}}$ is said to be **uniform** if there is a TM M such that $\forall n \in \mathbb{N}$ it holds that $M(1^n) = \langle C_n \rangle$

We notice that the circuit family obtained through the conversion process given by Theorem 22 is actually uniform since this very process can indeed be computed by a TM. When an uniform circuit family is also poly-sized, we say that it is P-uniform. It's easy to see that the class of languages decidable by a P-uniform circuit family is equivalent to the class P, hence a language L is in P if and only if it is decidable by a P-uniform circuit family.

4.4 The Karp-Lipton theorem

We discussed how $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$ through circuits being capable of simulating a Turing machine computation. However, the same technique used in the proof of Theorem 22 cannot be used for verifiers since each input have a witness of different length, hence we cannot make the computation oblivious. Since $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$, if we were able to show that $\mathsf{NP} \not\subseteq \mathsf{P}_{/\mathsf{poly}}$ then we would automatically conclude that $\mathsf{P} \neq \mathsf{NP}$. The **Karp-Lipton theorem** gives us some insight to believe that $\mathsf{NP} \not\subseteq \mathsf{P}_{/\mathsf{poly}}$ should be true.

Theorem 24: The Karp-Lipton theorem

If $\mathsf{NP} \subseteq \mathsf{P}_{/\mathsf{poly}}$ then $\mathsf{PH} = \Sigma_2^P$

Proof. We want to show that $\Pi_2^P\text{SAT} \leq_P \Sigma_2^P\text{SAT}$ in order to prove that $\Pi_2^P \subseteq \Sigma_2^P$ holds if $\mathsf{NP} \subseteq \mathsf{P}_{/\mathsf{poly}}$. The theorem will immediately follow from Theorem 21.

Claim: If $\mathsf{NP} \subseteq \mathsf{P}_{/\mathsf{poly}}$ then $\Pi_2^P\text{SAT} \leq_P \Sigma_2^P\text{SAT}$

Proof of the claim. We recall that $\Pi_2^P\text{SAT}$ is Π_2^P-Complete and that it is defined as:

$$\langle\phi\rangle \in \Pi_2^P\text{SAT} \iff \forall u \in \{0,1\}^n \; \exists v \in \{0,1\}^n \; \phi(u,v) = 1$$

Let $\phi_u(v) = \phi(u,v)$. If $\mathsf{NP} \subseteq \mathsf{P}_{/\mathsf{poly}}$ then we know that $\text{SAT} \in \mathsf{P}_{/\mathsf{poly}}$. Hence, there is a poly-sized circuit family $\{C_n\}_{n\in\mathbb{N}}$ that solves SAT, implying that:

$$\exists v \in \{0,1\}^n \; \phi_u(v) = 1 \iff \langle\phi_u\rangle \in \text{SAT} \iff C_n(\langle\phi\rangle, u) = 1$$

Now, consider the idea we discussed in Section 2.1 on how if $\mathsf{NP} \subseteq \mathsf{P}$ then search problems can be computed through decision problems. Here, we have a similar situation: if $\mathsf{NP} \subseteq \mathsf{P}_{/\mathsf{poly}}$ then we can use circuits to solve search problems. In particular, we're interested in solving the search problem that finds a string $v \in \{0,1\}^n$ for any given a formula ϕ and $u \in \{0,1\}^n$ such that $\phi_u(v) = 1$. We know that there is a new poly-sized circuit family $\{C'_n\}_{n\in\mathbb{N}}$ such that for every formula ϕ and $u \in \{0,1\}^n$ if there is a string $v \in \{0,1\}^n$ for which $\phi_u(v) = 1$ then $C'_n = (\langle\phi\rangle, u) = v$ (we didn't formally define circuits with multiple bit outputs, but their definition follows as a natural extension).

Due to non-uniformity, we only know that this circuit family exists, but we do not know how it is made. The idea here is that the circuits of this family can be "guessed" using an additional existence quantification. Let q be the the polynomial that defines the size of $\{C'_n\}_{n\in\mathbb{N}}$. We notice that each circuit in this family can be encoded with $O(q(n^2))$ bits. Hence, each circuit can be guessed:

$$\exists \langle C'_n\rangle \in \{0,1\}^{O(q(n^2))} \; \forall u \in \{0,1\}^n \; \phi(C'(\langle\phi\rangle, u), u) = 1$$

This implies that $\langle\phi\rangle \in \Pi_2^P\text{SAT}$ if and only if $\langle\phi\rangle \in \Sigma_2^P\text{SAT}$. □

Since $\Pi_2^P \subseteq \Sigma_2^P$ holds, thanks to duality we also get that $\Sigma_2^P = \mathsf{co}\Pi_2^P \subseteq \mathsf{co}\Sigma_2^P = \Pi_2^P$, hence we conclude that $\Pi_2^P \subseteq \Sigma_2^P$. By Theorem 21, we get that $\mathsf{PH} = \Sigma_2^P$.

□

4.5 Upper and lower bounds for circuits

Consider a generic function $f : \{0,1\}^n \to \{0,1\}$. Consider an assignment α such that $f(\alpha) = 1$. We define the logical formula ϕ_α as the conjunction of all the literals described by α. For instance, given a function $f : \{0,1\}^3 \to \{0,1\}$, if $\alpha = (1,0,1)$ is such that $f(\alpha) = 1$ then we have that $\phi_\alpha = (x_1 \wedge \overline{x_2} \wedge x_3)$.

Consider now the formula $\phi = \bigvee_{\alpha : f(\alpha)=1} \phi_\alpha$. This formula is a DNF for which $\phi(\alpha) = 1 \iff f(\alpha) = 1$. Each clause ϕ_α can be computed by a sequence of n AND gates. Let C_α be such sub-circuit that computes ϕ_α. The circuit that computes ϕ (and f) is given by a sequence of k OR gates, where k is the number of assignments such that f accepts. Since the function f has 2^n assignments, in the worst case all of them will be accepting, concluding that $k \leq 2^n$. Hence, we get the upper bound of $n2^n$.

This upper bound is clearly too high, but it is still sufficient to conclude that we only need circuits of at most exponential size to compute any Boolean function on n inputs. Shannon was able to give a better upper bound, proving that we actually need only subexponential circuits.

Theorem 25: Shannon's size upper bound

Any function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O\left(\frac{2^n}{n}\right)$.

Proof. First, we notice that for any Boolean funcion $f\{0,1\}^n \to \{0,1\}$ can be decomposed as:

$$f(x_1, \ldots, x_{n-1}, x_n) = (x_n \wedge f(x_1, \ldots, x_{n-1}, 1)) \vee (\overline{x_n} \wedge f(x_1, \ldots, x_{n-1}, 0))$$

Claim: Any boolean function $f\{0,1\}^n \to \{0,1\}$ can be computed with a circuit of size $O(2^n)$ gates

Proof of the claim. The decomposition process described above can be recursively applied, giving the following recursive equation:

$$S(n) = \begin{cases} 2S(n-1) + 3 & \text{if } n > 1 \\ 0 & \text{if } 0 \leq n \leq 1 \end{cases}$$

where $S(n)$ is the size of the resulting circuit. Solving the equation, we get that $S(n) = \frac{3}{2}(2^{n-1} - 1)$ □

Consider now the set Y_k of all functions from $\{0,1\}^k$ to $\{0,1\}$. This set contains exactly 2^{2^k} functions. Let $\text{All}_k : \{0,1\}^k \to \{0,1\}^{2^{2^k}}$ be the function that computes all the functions in Y_k at the same time — in other words, $\text{All}_k(x) = (f_1(x), f_2(x), \ldots, f_{2^{2^k}}(x))$. Hence, we can construct a trivial circuit with $2^{2^k}(k2^{2^k})$ gates that computes every possible function in Y_k. Now, we can use the recursive circuit of the claim only for $n - k$ levels of the recursion, using up to $O(2^{n-k})$ gates. Setting k to $\log n$ gives the result.

□

Let's now focus on lower bounds. Existing lower bounds for circuits are usually proved through the so-called **gate elimination argument**. The proofs themselves consist of a rather involved case analysis and we will not present them here. Instead, we will show the idea behind this type of proof by proving weaker lower bounds.

The gate-elimination argument does the following: given a circuit for the function in question, we first argue that some variable x_1 (or a set of variables $x_1, \ldots, x_k$) must fan out to several gates. If we replace this variable with a constant value, also called **1-bit restriction**, several gates will be eliminated. For instance, given a gate $x_1 \vee x_2$, if we set x_1 then the whole gate can be eliminated. By repeatedly applying this process, we can conclude the number of gates in the original circuit. For a Boolean function f, we denote with $f_{|x_i=b}$ the 1-bit restriction of f where the i-th variable gets replaced with the constant b:

$$f_{|x_i=b}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

Consider the Boolean function $\mathrm{XOR}(x) = \bigoplus_{i=1}^{n} x_i$. We already showed in Section 4.1 that this function can be computed by a circuit of size $O(n^2)$. We will now prove a lower bound on the number of AND and OR gates of a circuit that computes such function.

Proposition 14: Circuit lower bounds on parity functions

Any circuit that computes XOR or ¬XOR on n bits requires at least $3(n-1)$ AND gates and OR gates.

Proof. Let XOR_k denote the XOR function applied to the first k bits.

Claim: In any circuit C that computes XOR_k or $\neg\mathrm{XOR}_k$ over $k \geq 2$ bits, there is a 1-bit restriction that eliminates at least 3 gates of C.

Proof of the claim. We prove the claim for the function XOR_k. The same argument can be applied to $\neg\mathrm{XOR}_k$. Let g be the first AND or OR gate (starting from the bottom) of an optimal circuit that computes XOR_k on k bits. Let x_i, x_j be its two inputs.

Suppose now that x_i has out-degree 1. This would imply that g is the only gate with x_i acting as an input. Then, we could replace x_j with a constant b such that $g_{|x_j=b_1}$ is equal to a constant bit d_1 (e.g. if $g = x_i \wedge x_j$ we can replace x_j with 0 in order to also replace g with 0). This would imply that for any assignment where $x_j = b_1$ the output of the circuit is independent of the value of x_i, which cannot happen by definition of the XOR_k function itself. Hence, x_i must have at least 2 outgoing edges.

Let h be another gate for which x_i acts as an input. We notice that h cannot be the output gate of the circuit: if it wes, we could replace x_i with a constant b_1 such that $h_{|x_i=b_2}$ is equal to a constant bit d_2, making the XOR_k function dependent only on the value of x_i, which is impossible. Hence, h must have a successor gate p.

Moreover, since we chose g as the first AND or OR gate of the circuit, we know that g's inputs can only be variables. Hence, we get that $g \neq p$. To recap, we have that g, h, p are three different gates, where x_i is the input of g and h and h is the input of p.

We notice that for any value $b_3 \in \{0,1\}$ we get that both $g_{x_i=b_3}$ and $h_{x_i=b_3}$ can be replaced by two constants. However, one of the two values is such that $p_{|x_i=b_3}$ can also be replaced by a constant. Hence, we conclude that there is a 1-bit restriction such that g, h, p can be replaced. □

We now proceed by induction. Clearly, when $n = 1$ we need no gates at all to compute both XOR_n and $\neg\text{XOR}_n$. making the result trivially true. Assume now that for XOR_{n-1} and $\neg\text{XOR}_{n-1}$ we need at least $3(n-2)$ gates. We notice that $\text{XOR}_{n|x_n=b}$ is equal to either XOR_{n-1} or $\neg\text{XOR}_{n-1}$ for any value $b \in \{0,1\}$. Thus, since the restriction $\text{XOR}_{n|x_n=b}$ eliminates 3 gates and both XOR_{n-1} and $\neg\text{XOR}_{n-1}$ require at least $3(n-2)$ gates, we get that any circuit computing XOR_n must have at least $3(n-1)$ gates. □

We saw how circuit lower bounds for specific functions can be achieved through techniques such as gate elimination. However, we can also define general lower bounds for most functions. In fact, Shannon also proved that there are indeed some Boolean functions that require exponential circuits, even though we do not know how they are made.

Theorem 26: Shannon's size lower bound

For every $n > 1$, with probability at least $1 - o(1)$, a random function $f : \{0,1\}^n \to \{0,1\}$ requires a circuit of size greater than $\frac{2^n}{10n}$.

Proof. We prove the theorem through a simple counting argument. Let C be a circuit of size at most S. We notice that C can be encoded with $9S \log S$ bits by representing it as an adjacency list. Let X_S be the set of all the circuits of size at most S. We get that $|X_S| = 2^{9S \log S}$. Consider now the set Y of all functions from $\{0,1\}^n$ to $\{0,1\}$. This set contains exactly 2^{2^n} functions. We notice that:

$$\frac{|X_S|}{|Y|} = \frac{2^{9\frac{2^n}{10n}\log\left(\frac{2^n}{10n}\right)}}{2^{2^n}} = \frac{1}{2^{2^n\left(1-\frac{9}{10n}\log\left(\frac{2^n}{10n}\right)\right)}} < \frac{1}{2^{\frac{2^n}{10}}} = o(1)$$

Hence, the probability of a random function to be computable by a circuit of size at most $\frac{2^n}{10n}$ is $o(1)$, which concludes that with probability $1 - o(1)$ it cannot be computed by such circuit. □

Together with his upper bound proof, Shannon's results show that **almost every Boolean function** strictly requires exponential size. Even though we know that some functions are hard even for non-uniform circuits, we cannot use this fact to show that $\mathsf{NP} \not\subseteq \mathsf{P}_{/\mathsf{poly}}$ holds due to the structure of these functions being unknown. This gives hope that we should be able to find one such function that also happens to lie in NP, separating the two classes thanks to The Karp-Lipton theorem. At the moment, the best circuit size lower bound for an NP language is only $5n - o(n)$.

5 Randomized computation

5.1 Randomization and the class BPP

Up until now, we've used the deterministic Turing machine as our standard model of computation. However, this model doesn't capture one important aspect of reality: the ability to make random choices during computation. For instance, most programming languages offer a built-in pseudo-random number generator for this purpose. While scientists and philosophers continue to debate the existence of true randomness, it's clear that outcomes like coin tosses—or other physical measurements—appear sufficiently random and unpredictable for practical use. Therefore, it's reasonable to consider Turing machines capable of simulating coin tosses through a source of random bits.

Definition 25: Probabilistic Turing machine

A **probabilistic Turing machine (PTM)** is a TM provided with two distinct transition functions δ_0, δ_1 . On each step, the computation made by the machine "flips a coin" and choses to apply δ_0 or δ_1 both with probability $\frac{1}{2}$. The running time of an PTM is the maximum number of steps that the computation may take regardless of the random choices that it makes.

We notice that this definition is similar to the one we gave for Non-deterministic Turing machines. In fact, PTMs may be seen as a "restriction" of NDTMs. In the latter, the machine explores all the $2^{T(n)}$ paths in the **binary computation tree** at the same time through non-determinism. In PTMs, instead, the machine explores only one such path, which is chosen **randomly** in each computation with probability $\frac{1}{2^{T(n)}}$. However, even though they look similar, PTMs and NDTMs are very different: like DTMs, and unlike NDTMs, probabilistic machines are intended to model realistic computational devices.

Now that we have formally defined how Turing machines can "toss coins" to compute, we have to formally defined what it means for a language to be computed by such machines.

We say that a language is decided by a PTM if such machine accepts strings in the language and rejects other strings with a **good bounded-error**. In particular, we require that this bounded-error at least $\frac{2}{3}$, both for accepted and rejected strings. We can define a new class BPP, standing for **bounded-error probabilistic polynomial time**.

Definition 26: The class BPP

We define BPP as the class of languages for which there is a polynomial time PTM M such that:

$$x \in L \iff \Pr[M(x) = L(x)] \geq \frac{2}{3}$$

where $L(x) = 1$ if $x \in L$ and $L(x) = 0$ if $x \notin L$. The machine M is often referred to as a **Monte Carlo machine**.

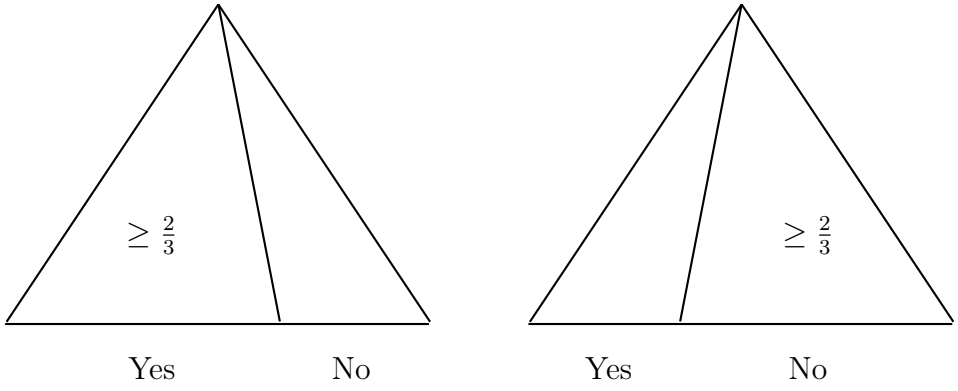


Figure 5.1: Distribution of branches for BPP when $x \in L$ (left) and when $x \notin L$ (right)

The idea behind this definition is that by running the machine k times on the same input we will have the following probability table:

	$L(x) = 1$	$L(x) = 0$
$M(x) = 1$	$> 1 - \frac{1}{2^{ck}}$	$< \frac{1}{2^{ck}}$
$M(x) = 0$	$< \frac{1}{2^{ck}}$	$> 1 - \frac{1}{2^{ck}}$

For a sufficiently large k, we are pretty sure that the machine decides the language.

Since any deterministic TM is a special case of a PTM where both transition functions are equal, it trivially holds that P ⊆ BPP. A central open question of complexity theory is whether or not P = BPP. Surprisingly, many complexity theorists actually believe that answer to the question P $\stackrel{?}{=}$ BPP is actually true. This derives from important results shown in the theory of *de-randomization* and *pseudo-randomization*. Moreover, it's also easy to see that BPP ⊆ EXP: we can build a deterministic TM that simulates every possible sequence of coin tosses in exponential time.

Moreover, we do not know if BPP ⊆ NP or if BPP ⊆ coNP: any PTM computing a language in BPP may have an accepting and a rejecting branch both when $x \in L$ and $x \notin L$, thus it cannot be easily converted into an NDTM.

Surprisingly, probabilistic algorithms can sometimes be even more efficient that non-random ones. For instance, consider the search problem MEDIAN which asks to find the median of n input numbers. This search problem is actually in P as it can be solved efficiently in $O(n \log n)$ by first sorting the numbers and then returning the middle element. However, we can solve this search problem in time $O(n)$ through randomization. In fact, we can solve an ever harder problem, i.e. finding the k-th smallest element in the set.

`kSmallest`$(k, a_1, \ldots, a_n)$:

1. Pick a random $i \in [n]$ and count the number m of elements a_j such that $a_j \leq a_i$
2. If $m = k$, output a_i
3. If $m > k$, output kSmallest$(k, b_1, \ldots, b_m)$, where $b_1, \ldots, b_m$ are the m elements counted before
4. If $m < k$, output kSmallest$(k, c_1, \ldots, c_{n-m})$, where $c_1, \ldots, c_{n-m}$ are the $n - m$ elements not counted before"

In the worst case, the runtime is given by $T(n) = T\left(\frac{9}{10}n\right) + O(n)$, which is indeed $O(n)$. Moreover, this algorithm is actually guaranteed to always return the correct median value. There are indeed some non-randomized alternatives that can also run in $O(n)$, such as the *median of medians* algorithm [BFP+73], but they are based on way more complex techniques. The randomized algorithm that we provided is one of the most used in practice, showing that randomness can often be a very good tool.

Without much surprise, the class BPP can also be defined through the concept of random verification: at least $\frac{2}{3}$ of the random paths will act as the witnesses through which a standard TM can verify the membership.

Theorem 27: The class BPP (2nd Definition)

Given a language L, it holds that $L \in$ BPP if and only if there is a polynomial time deterministic TM M such that:

$$x \in L \iff \Pr_{r \in_R \{0,1\}^*}[M(x, r) = L(x)] \geq \frac{2}{3}$$

The choice of the constant $\frac{2}{3}$ that we used up until now may seem pretty arbitrary. However, this choice is actually sufficient to form a robust definition. In fact, we now show that we can replace $\frac{2}{3}$ with any constant larger than $\frac{1}{2}$ or even with $\frac{1}{2} + \frac{1}{n^c}$ for any constant $c > 0$. We denote with BPP_c the class of languages L for which there is a polynomial time PTM M such that:

$$\Pr[M(x) = L(x)] \geq \frac{1}{2} + \frac{1}{|x|^c}$$

for any $x \in \{0, 1\}^*$. Trivially, we have that $\mathsf{BPP} \subseteq \mathsf{BPP}_c$ for any $c > 0$. We show that $\mathsf{BPP}_c \subseteq \mathsf{BPP}$ also holds by proving an actually stronger result.

Theorem 28: Error reduction for BPP

Given a language in BPP_c, for every constant $d > 0$ there exists a polynomial time PTM M such that for any $x \in \{0,1\}^*$ it holds that:

$$\Pr[M(x) = L(x)] \geq 1 - \frac{1}{2^{|x|^d}}$$

Proof. The idea behind the proof is to simply run multiple times the original machine. Let M be the machine for which $L \in \mathsf{BPP}_c$. The machine M' simply does the following: for every input $x \in \{0,1\}^*$, run $M(x)$ for $k = 8\,|x|^{2c+d}$ times, obtaining k outputs $y_1, \ldots, y_k \in \{0,1\}$. If the majority of these outputs is 1, then output 1; otherwise, output 0.

For every $i \in [k]$, we define the Bernoulli random variable X_i such that $X_i = 1$ if and only if $y_i = L(x)$. Notice that $X_1, \ldots, X_k$ are independent variables with $\mathbb{E}[X_i] = \Pr[X_i = 1] \geq p$, where $p = \frac{1}{2} + \frac{1}{|x|^c}$ is the probability that defines M. By Chernoff's bound — which we won't prove — we get that:

$$\Pr\left[\left|\sum_{i=1}^{k} X_i - pk\right| > \delta pk\right] < e^{-\frac{\delta^2}{4}pk}$$

where the left side corresponds to the probability of the output of M' being wrong Since in our case we have that $p = \frac{1}{2} + \frac{1}{|x|^c}$, by setting $\delta = \frac{1}{2|x|^c}$ we get that:

$$\Pr\left[\left|\sum_{i=1}^{k} X_i - pk\right| > \delta pk\right] < e^{-\frac{\delta^2}{4}pk} \leq \frac{1}{2^{|x|^d}}$$

Hence, the probability of the output of M' being correct is at least $1 - \frac{1}{2^{|x|^d}}$. □

5.2 One-sided error and the classes RP, coRP

The class BPP captures probabilistic algorithms with "two-sided error". These algorithms are allowed to give a wrong answer with a small probability, both when $x \in L$ and $x \notin L$. However, many probabilistic algorithms are based on "one-sided error", i.e. they may return a wrong answer only for one of the two possibilities. For example, if $x \notin L$, they will never output 1, although they may output 0 when $x \in L$. This type of behavior is captured by the class RP, standing for **randomized polynomial time**.

Definition 27: The class RP

We define RP as the class of languages for which there is a polynomial time PTM M such that:

$$x \in L \implies \Pr[M(x) = 1] \geq \frac{2}{3}$$
$$x \notin L \implies \Pr[M(x) = 1] = 0$$

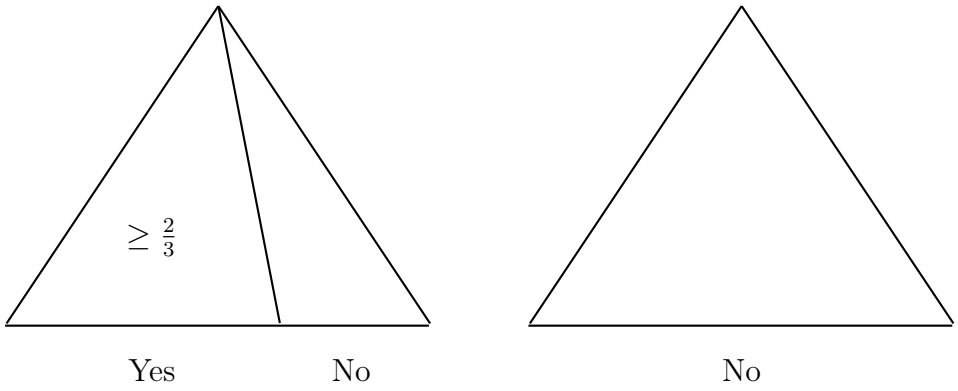


Figure 5.2: Distribution of branches for RP when $x \in L$ (left) and when $x \notin L$ (right)

Historically, this class was defined earlier than BPP. The class $\mathsf{coRP} = \{L \subseteq \{0,1\}^* \mid L \in \mathsf{RP}\}$ is the dual of the class RP, hence it can also be defined in the following more intuitive way, where the concept of "one-sided error" lies on the other side.

Definition 28: The class coRP

We define coRP as the class of languages for which there is a polynomial time PTM M such that:

$$x \in L \implies \Pr[M(x) = 1] = 1$$

$$x \notin L \implies \Pr[M(x) = 1] < \frac{1}{3}$$

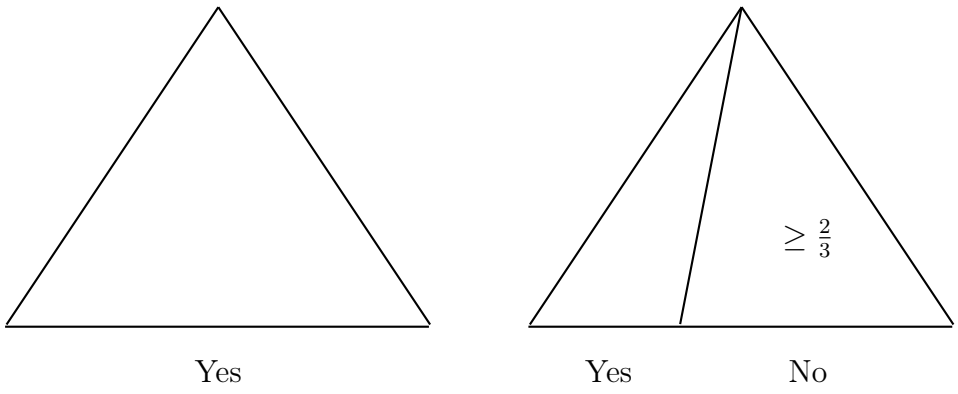


Figure 5.3: Distribution of branches for coRP when $x \in L$ (left) and when $x \notin L$ (right)

Like BPP, the classes RP and coRP can also be defined through random verification. For example, we have that $L \in \mathsf{RP}$ if and only if there is a polynomial time deterministic TM M such that:

$$x \in L \implies \Pr_{r \in_R \{0,1\}^*}[M(x,r) = 1] \geq \frac{2}{3}$$

$$x \notin L \implies \Pr_{r \in_R \{0,1\}^*}[M(x,r) = 1] = 0$$

However, differently from BPP, these two classes can indeed by compared to NDTMs since in one of the two cases we will have that the computing PTM has all rejecting branches or all accepting branches. Hence, we have that RP $\subseteq$ NP and coRP $\subseteq$ coNP. In a certain sense, we can view RP and coRP as two restrictions of the classes NP and coNP, where at least $\frac{2}{3}$ of the branches must satisfy the process of verification or disqualification.

5.3 The power of randomness

To show the power of randomized computation, we will consider two problems: the *primality test* and the *polynomial identity test*. In the **primality testing problem**, we are given an integer p and wish to determine whether or not it is prime. In other words, we want to decide the language PRIMES $= \{\langle N \rangle \mid N \in \mathbb{P}\}$.

For obvious reasons, this problem has been of main interest for mathematicians since the ancient times. In the 70', a very efficient probabilistic algorithm was found. In particular, this algorithm shows that PRIMES $\in$ coRP. Researchers believed that this algorithm could also be *de-randomized* through some number theory property, while preserving a polynomial running time. In 2004, Agrawal, Kayal, and Saxena [AKS04] found a deterministic polynomial time algorithm to solve this problem, proving that it lies in P. However, the randomized version is still widely used due to it being way more efficient and simpler.

Theorem 29

PRIMES $\in$ coRP

Proof. For every number $N \in \mathbb{N}$ and $A \in [N-1]$, we define $\mathrm{QR_N}$ as follows

$$QR_N(A) = \begin{cases} 0 & \text{if } \gcd(A,N) \neq 1 \\ +1 & \text{if } \exists B \in \mathbb{N} \ \ \gcd(B,N) = 1 \text{ and } A \equiv B^2 \pmod{N} \\ -1 & \text{otherwise} \end{cases}$$

When $QR_N(A) = +1$, we say that A is a *quadratic residue modulo* N. We consider the following number theory facts, which won't be proven:

- For every odd prime N and $A \in [N-1]$, it holds that $QR_N(A) = A^{\frac{N-1}{2}} \pmod{N}$
- For every odd numbers N and A, the *Jacobi symbol*, written as $\left(\frac{N}{A}\right)$, is defined as $\left(\frac{N}{A}\right) = \Pi_{i=1}^{k} QR_{P_i}(A)$, where $P_1, \ldots, P_k$ are the prime factors of N.
- For every odd composite N, among all $A \in [N-1]$ such that $\gcd(N,A) = 1$, at most half of the A values satisfy $\left(\frac{N}{A}\right) = A^{\frac{N-1}{2}} \pmod{N}$.

First, we notice that the Jacobi symbol is computable in time $O(\log A \cdot \log N)$. We define a PTM M as follows:

M = "Given the string x in input:

1. Check if $x = \langle N \rangle$. If false, reject

2. If $N = 2$, accept.
3. Pick a random $A \in [N-1]$.
4. If $\gcd N, A > 1$ or $\left(\frac{N}{A}\right) \neq A^{\frac{N-1}{2}} \pmod N$, reject
5. Otherwise, accept."

When the input value N is prime, the machine will always accept. When N is not prime, instead, the machine will wrongly accept with probability at most $\frac{1}{2}$. By running M a constant number of times, this probability can be amplified to the required value.

□

Curiously, the search problem corresponding to primality testing, i.e. finding the factorization of a given composite number, seems very different and much more difficult. The conjectured hardness of this problem underlies many current cryptosystems. In recent years, it has been shown that *quantum computers* are able to solve this search problem efficiently, even though it requires some currently infeasible architectures.

We now switch to another coRP decision problem, which, however, has no known efficient deterministic algorithm. This problem is known as the **polynomial identity testing**, which asks to determine if two polynomials $p_1, p_2 \in \mathbb{Z}[x_1, \ldots, x_m]$ are equivalent. A trivial question arises: many modern softwares are capable of deciding if two polynomials are equivalent almost instantly, so how can this problem not be in P? The answer to this question is simple: the input polynomials are given in an *implicit form*. In particular, we assume that the two input polynomials are represented by an *algebraic circuit*, i.e. a circuit defined on the basis $\{+, -, \cdot\}$ (we can also allow the circuit to contain constants).

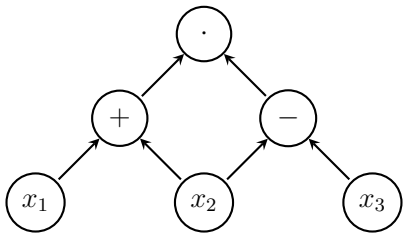


Figure 5.4: An algebraic circuit encoding the polynomial $(x_1 + x_2)(x_2 - x_3)$.

Formally, the problem is defined as $\text{PIT} = \{\langle P_1, P_2\rangle \mid \forall x \in \{0,1\}^m \quad P_1(x) = P_2(x)\}$. This representation allows us to express polynomials in a very compact way. Many other problems in complexity theory are represented through circuits, such as the *polynomial pigeonhole principle* or the *polynomial parity argument*. The idea behind these problems is that we are "forced" to avoid known simple algorithms since we're working with objects of exponential size with respect to the real input.

For instance, given two polynomials p and q, we could easily check if they are equal by first expanding until they reach the standard form $a_0 + a_1 x_1 + \ldots + a_n x_n$ and then check if $p - q = 0$. This procedure clearly requires polynomial time with respect to the size of the polynomial. However, by expanding the polynomial represented by an algebraic circuit

we get a polynomial of exponential size with respect to the input length (remember that the circuits are the input in PIT). Hence, if we apply the previous procedure to PIT, we get an exponential running time. Moreover, the expansion procedure alone clearly requires exponential time. These constraints make PIT not so trivial.

However, this problem can be solved through randomization by reducing it to another coRP problem, i.e. the *zero polynomial* problem. This new problem takes an algebraic circuit as input and asks to determine if the circuit represents the zero polynomial. Formally, we have that ZEROP $= \{\langle P\rangle \mid \forall x \in \{0,1\}^m \;\; P(x) = 0\}$. The reduction from PIT to ZEROP is immediate: given the two input circuits P_1, P_2, we construct a new circuit $P = P_1 - P_2$ by joining them with a single gate. Vice versa, we can also reduce ZEROP to PIT: given the input circuit P, we test if it is equal to a circuit that represents the zero polynomial. This makes the two problems completely equivalent. The proof of ZEROP lying inside coRP is based on a special case of the Schwartz-Zippel lemma, which we won't prove.

Lemma 3: Schwartz-Zippel lemma

Let $p \in \mathbb{Z}[x_1, \ldots, x_m]$ be a non-zero polynomial with total degree at most d and let S be a finite set of integers. Given $a_1, \ldots, a_m$ randomly chosen from S (with replacement), it holds that:

$$\Pr[p(a_1, \ldots, a_m) \neq 0] \geq 1 - \frac{d}{|S|}$$

Here, The total degree of a monomial $x^{d_1} \cdot \ldots \cdot x^{d_m}$ is equal to $d_1 + \ldots + d_m$. The total degree of a polynomial is the largest total degree of its monomials.

Theorem 30

ZEROP and PIT are in coRP

Proof. Let P be a circuit representing a polynomial over $\mathbb{Z}[x_1, \ldots, x_m]$. We notice that a depth d circuit contains at most d multiplications, so it defines a polynomial of degree at most 2^S — think of the circuit made of d multiplication gates where the inputs of each gate are both the output of the previous gate.

We can easily define the following probabilistic algorithm: choose m numbers $a_1, \ldots, a_m$ from $S = \{1, \ldots, 10 \cdot 2^d\}$, evaluate $P(a_1, \ldots, a_m) = y$, accept if $y = 0$ and reject otherwise. We notice that the m random numbers require $O(m \log 2^d) = O(md)$ random bits, thus they can be generated in polynomial time. If the input circuit P represents the zero polynomial, the output will clearly always be correct. Otherwise, the Schwartz-Zippel lemma ensures that the probability of a wrong answer is low enough.

However, the previous algorithm has a catch: the values that arise during the evaluation of $P(a_1, \ldots, a_m) = y$ may be require even $(10 \cdot 2^d)^{2^d}$ bits, making it impossible to compute in polynomial time. To fix this problem, we use a technique called *fingerprinting*: we choose a random value $k \in [2^{2d}]$ and evaluate $P(a_1, \ldots, a_m) \equiv y \pmod{k}$. Clearly, if

$y = 0$ then $y \equiv 0 \pmod{k}$, hence we still always accept when P represents the zero polynomial. When $y \neq 0$, instead, we claim the following.

Claim: If $y \neq 0$, with probability at least $\frac{1}{4d}$ a random $k \in [2^{2d}]$ doesn't divide y.

Proof of the claim. Let $B = \{p_1, \ldots, p_t\}$ be the set of distinct prime factors of y. By the Prime Number Theorem, for sufficiently large d we have that the number of primes in $[2^{2d}]$ is at least $\frac{2^{2d}}{\log 2^{2d}} = \frac{2^{2d}}{2d}$.

Since y can have at most $\log y \leq 5d2^d = o\left(\frac{2^{2d}}{2d}\right)$ prime factors, for sufficiently large values of d, the number of primes in $[2^{2d}] - B$ is at least $\frac{2^{2d}}{4d}$, thus a random $k \in [2^{2d}]$ will have this property with probability at least $\frac{1}{4d}$. □

Through this claim, we can choose a constant number of values $k_1, \ldots, k_h$ to boost the probability: we compute $P(a_1, \ldots, a_m) \equiv y_1 \pmod{k_i}$ for each k_i and accept when the majority of the outputs $y_1, \ldots, y_h$ is different from zero and reject otherwise. Hence, we get that ZEROP $\in$ coRP and by consequence that PIT $\in$ coRP through the previous reduction.

□

5.4 Zero-sided error and the class ZPP

Last but not least, we define a class based on "zero-sided error". The idea here is that when a machine cannot be 100% sure if a string is in the language or not, it is allowed to output "I don't know", i.e. the symbol " ? ". This allows us to remove error from the computation for the price of *uncertainty*. Of course, we have to keep the number of " ? " outputs low, otherwise every existing language would trivially be inside this class: we could just output " ? " for all the strings of the language. To define the **zero-error probabilistic polynomial time** class, we will directly use a definition based on random verifiability due to it being simpler.

Definition 29: The class ZPP

We define ZPP as the class of languages for which there is a polynomial time PTM M such that:

- If $\forall r \in \{0,1\}^*$ we have that $M(x, r) = 1$ then $x \in L$
- If $\forall r \in \{0,1\}^*$ we have that $M(x, r) = 0$ then $x \notin L$
- The general probability of a " ? " output is low:

$$\Pr_{r \in_R \{0,1\}^*}[M(x, r) = ?] \leq \frac{1}{2}$$

First of all, it's easy to see that $\mathsf{P} \subseteq \mathsf{ZPP}$ since any deterministic TM here would output " ? " with zero probability. Moreover, a good eye may notice that $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$, a fact that fully expresses the idea of "zero-sided error".

Theorem 31: The class ZPP (2nd Definition)

$\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$

Proof. If a language is in ZPP then every time the output of the machine is " ? " we can convert it into a 0, making the machine valid for RP. Since $\mathsf{ZPP} \subseteq \mathsf{RP}$, we get that $\mathsf{coZPP} \subseteq \mathsf{coRP}$, but the class ZPP is actually closed under complement (it follows from the definition), thus $\mathsf{ZPP} \subseteq \mathsf{coRP}$ is also true.

Vice versa, if a language is in $\mathsf{RP} \cap \mathsf{coRP}$, we can simultaneously run the two machines M and M' which prove that the language is in RP and coRP. If we get 0 as output from both machines, we run them again. Eventually, the machine M (or M') will output 1, while the other machine will output 0, giving us the final output. □

The presence of a " ? " output is mostly "unwanted" since it makes reasoning about computation harder. Thankfully, we can remove this type of output through another way more convenient — and actually very surprising — definition.

Theorem 32: The class ZPP (3rd Definition)

Given a language L, it holds that $L \in \mathsf{ZPP}$ if and only if there is a polynomial p and a PTM M such that $\forall x \in \{0,1\}^*$ it holds that $M(x) = L(x)$ and:

$$\forall x \in \{0,1\}^* \quad \mathbb{E}[T_{M,x}] \leq p(|x|)$$

where $T_{M,x}$ is the random variable describing the running time of $M(x)$. In other words, the expected runtime of M is at most polynomial. The machine M is often referred to as a **Las Vegas machine**.

Proof. Let M be the PTM machine that proves that $L \in \mathsf{ZPP}$. Let q be the polynomial describing the running time of M. We define a new machine M' that repeatedly runs M until it outputs 1. $X_{M,x}$ be the variable denoting the number of times $M(x)$ gets run inside M', implying that $\Pr[X_{M,x} = i] \leq \frac{1}{2^{i-1}}$.Since every run requires $q(|x|)$ steps, we get that:

$$\mathbb{E}[T_{M,x}] \leq \mathbb{E}[q(|x|) \cdot X_{M,x}] = q(|x|) \sum_{i=0}^{+\infty} 1 \cdot \Pr[X_{M,x} = i] = q(|x|) \sum_{i=1}^{+\infty} \frac{1}{2^{i-1}} = 2q(|x|)$$

hence the expected runtime of M' is polynomially bounded.

Vice versa, suppose that there is a Las Vegas machine M for a language L, where p is the polynomial that bounds the expected runtime. We define a new PTM M' as follows. Run M for at least k times its expected running time. If it gives an answer under $k \cdot \mathbb{E}[T_{M,x}]$ steps, return that answer. Otherwise, output 0.

Since $T_{M,x}$ is a non-negative random variable with a finite expected value, by Markov's inequality — which we won't prove — we have that:

$$\Pr[T_{M,x} \geq k \cdot \mathbb{E}[T_{M,x}]] \leq \frac{1}{k}$$

Hence, by choosing a nice value for k — say 100 — we get that the probability of M halting under $100 \cdot \mathbb{E}[T_{M,x}]$ inside M' is greater than $1 - \frac{1}{100}$. Thus, M' is an RP machine. Similarly, we can define a machine M'' constructed in the same way as M', with the difference that it returns 1 if it times out, making it a coRP machine. Hence, we get that $L \in \text{RP} \cap \text{coRP} = \text{ZPP}$. □

While reasoning about this new definition, we can with most certainty conclude that if a problem is inside ZPP then we're pretty much fine: we're now allowed to use randomness in order to perfectly decide a language, with the cost of getting a bad runtime in some cases. In fact, Las Vegas algorithms are often better than normal decision algorithms — consider what we showed for the MEDIUM search problem.

5.5 Randomness, circuits and PH

In the previous sections we discussed the relationships between probabilistic classes, P, NP and coNP. We also discussed how researchers believe that $\text{P} \neq \text{NP}$ and $\text{P} = \text{BPP}$ — and consequently that all the probabilistic classes collapse into P. We'll now prove two results [Adl78; Sip83; Lau83] that show us how randomness is not a sufficient alternative to non-determinism.

Theorem 33: Adleman's theorem

$\text{BPP} \subseteq \text{P}_{/\text{poly}}$

Proof. Given a language $L \in \text{BPP}$. We saw how this probability can be boosted by running M multiple times. Let M be the polynomial time PTM with boosted probability such that:

$$x \in L \iff \Pr_{r \in_R \{0,1\}^*}[M'(x,r) = L(x)] > 1 - \frac{1}{2^{n+1}}$$

We say that a random string $r \in_R \{0,1\}^m$ is *bad* for an input $x \in \{0,1\}^n$ if $M(x,r) \neq L(x)$. Otherwise, we say that r is *good* for the input x. Since M' makes an error with probability at most $\frac{1}{2^{n+1}}$, for every input x we have at most $\frac{2^m}{2^{n+1}}$ bad random strings. Adding up this result for all the possible 2^n inputs, we get that there are at most $2^n \cdot \frac{2^m}{2^{n+1}} = 2^{m-1}$ bad strings for some inputs. However, there will always be at least one random string $r_n \in \{0,1\}^m$ that is good for all the possible 2^n inputs. We call this string *perfect* for n.

In order for M' (and M) to be able to verify each input in polynomial time, the length of r_n must be at most polynomial with respect to n. Thus, for each $n \in \mathbb{N}$, we can use the perfect string r_n as advice bits that can be given as additional inputs to an advice-taking TM, concluding that $L \in \text{P}_{/\text{poly}}$. □

Together with The Karp-Lipton theorem, the previous result implies that we cannot hope to find a randomized polynomial time algorithm that solves NP-Complete problems. For instance, if we could solve 3SAT in randomized polynomial time, then we would get that $\mathsf{NP} \subseteq \mathsf{BPP}$. Hence, we would get that $\mathsf{NP} \subseteq \mathsf{P}_{/\mathsf{poly}}$, which would also imply that $\mathsf{PH} = \Sigma_2^P$, which is conjectured to be false with high probability. This leaves us with only one option: maybe we have that $\mathsf{BPP} \subseteq \mathsf{NP}$ and $\mathsf{BPP} \subseteq \mathsf{P}_{/\mathsf{poly}}$ but $\mathsf{NP} \not\subseteq \mathsf{P}_{/\mathsf{poly}}$, meaning that randomness is too weak. There are no current conjectures for the inclusion between BPP and NP.

Theorem 34: The Sipser-Gács-Lautemann theorem

$\mathsf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$

Proof. Since BPP is closed under complement and $\Pi_2^P = \mathsf{co}\Sigma_2^P$, it is sufficient to prove that $\mathsf{BPP} \subseteq \Sigma_2^P$ to also get $\mathsf{BPP} \subseteq \Pi_2^P$.

As in the previous theorem, given a language $L \in \mathsf{BPP}$, let M be the polynomial time PTM with boosted probability such that:

$$x \in L \iff \Pr_{r \in_R \{0,1\}^*}[M'(x,r) = L(x)] > 1 - \frac{1}{2^n}$$

For any $x \in \{0,1\}^n$, let S_x be the set of random strings $r \in \{0,1\}^m$ such that $M(x,r) = L(x)$ — using the previous theorem's notation, this is the set of all the good random strings for x. We notice that if $x \in L$ then $|S_x| \geq \left(1 - \frac{1}{2^n}\right) 2^m$, while if $x \notin L$ then $|S_x| \leq \frac{2^m}{2^n}$, meaning that for each input this set is either very large or very small.

Given a subset $S \subseteq \{0,1\}^m$ and a string $u \in \{0,1\}^m$, we denote with $S + u$ the set given by the "bitwise XOR" of S with u, that is $S \oplus u = \{x \oplus u \mid x \in S\}$. Let $k = \left\lceil \frac{m}{n} \right\rceil + 1$. We claim the following two statements.

Claim 1: For sufficiently large n, for every subset $S \subseteq \{0,1\}^m$ with $|S| \leq \frac{2^m}{2^n}$ and every k strings $u_1, \ldots, u_k \in \{0,1\}^m$, it holds that $\bigcup_{i=1}^k (S \oplus u_i) \neq \{0,1\}^m$.

Proof of the first claim. Since $|S \oplus u_i| = |S|$ by definition, we have that

$$\left| \bigcup_{i=1}^{k} (S \oplus u_i) \right| \leq k\,|S| \leq k \frac{2^m}{2^n}$$

For a sufficiently large n, we get that $k\frac{2^m}{2^n} < 2^m$, concluding that the union cannot cover the whole set □

Claim 2: For sufficiently large n and for every subset $S \subseteq \{0,1\}^m$ with $|S| \geq \left(1 - \frac{1}{2^n}\right) 2^m$, there exist k strings $u_1, \ldots, u_k \in \{0,1\}^m$ such that $\bigcup_{i=1}^k (S \oplus u_i) = \{0,1\}^m$.

Proof of the second claim. We want to prove that the probability of the existence of such k strings is greater than 0. Suppose that $u_1, \ldots, u_k$ are chosen independently at random

from $\{0,1\}^m$. Let B_r denote the even $r \notin \bigcup_{i=1}^{k}(S \oplus u_i)$. By extension, for each $i \in [k]$ let B_r^i denote the event $r \notin S \oplus u_i$. First, we notice that, by properties of the bitwise XOR, it holds that $r \notin S \oplus u_i$ if and only if $r \oplus u_i \notin S$. However, since $r \oplus u_i \in \{0,1\}^m$ and $|S| \geq \left(1 - \frac{1}{2^n}\right) 2^m$, it holds that $Pr[r \oplus u_i \notin S] < \frac{1}{2^n}$.

Since $u_1, \ldots, u_k$ are independent from each other, we know that $\Pr[B_r^i] = \Pr[B_r^j]$ for all $i, j \in [k]$. Moreover, since $B_r = \bigwedge_{i=1}^{k} B_r^i$, we get that $\Pr[B_r] = (\Pr[B_r^i])^k < \frac{1}{2^{nk}}$

For a sufficiently large n, we get that $\frac{1}{2^{nk}} < \frac{1}{2^m}$. Since for every $r \in \{0,1\}^m$ we have that $\Pr[B_r] < \frac{1}{2^m}$, the probability of the existence of a random string $r' \in \{0,1\}^m$ such that B_r is true is less than 1, i.e. $\Pr[\exists r' \in \{0,1\} \;\; B_{r'}] < 1$

This concludes that $\Pr[\exists u_1, \ldots, u_k \in \{0,1\}^m \;\; \bigcup_{i=1}^{k}(S \oplus u_i) = \{0,1\}^m] > 0$ □

Thanks to the two claims, we get that:

$$x \in L \iff \exists \langle u_1, \ldots, u_k \rangle \in \{0,1\}^{O(mk)} \; \forall r \in \{0,1\}^m \;\; \bigvee_{i=1}^{k} M(x, r \oplus u_i) = 1$$

concluding that $L \in \Sigma_2^P$. □

Together with the Adleman's theorem, this result suggests that BBP may indeed lie inside NP, giving us more insight on how BPP = P might be true. In fact, every single property that is true for P seems to also be true for BPP.

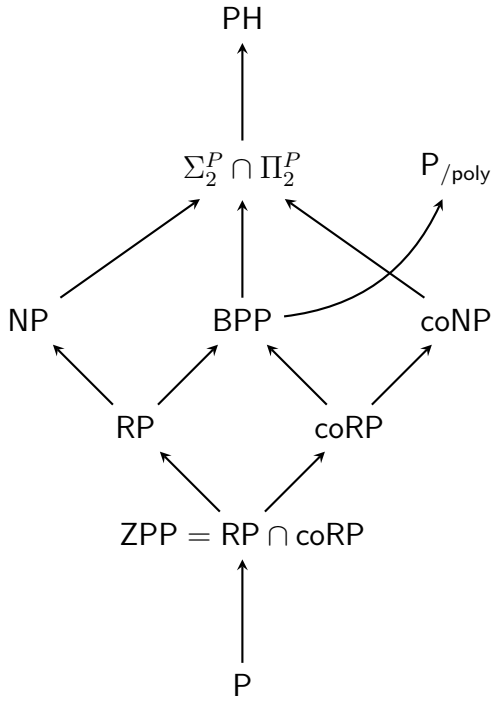


Figure 5.5: Recap of the class inclusions discussed in this chapter

6 Space complexity

6.1 Relations between time and space

In the previous chapters we have focused on the running time of Turing machines, while the **space requirements** have been neglected. Generally, time complexity is considered more important than space complexity, even though the latter is also a fundamental topic in computation. In particular, we will show that this type of complexity is highly related with *games*.

We recall that the *required space* $S(n)$ of a Turing machine M the number of tape cells written by the heads during the computation for an input of length n, except the input tape's cells. While working with decision problems, we can also omit the space complexity of the output tape, since only 1 bit is required for the output. Thus, for decision problems we consider only the cells written on the **work tape**. In Section 3.1, we defined the classes DSPACE and NSPACE through the same definition of the classes DTIME and NTIME, where time is simply replaced by space.

Just like the trivial inclusion $\text{DTIME}(f(n)) \subseteq \text{NTIME}(f(n))$, it also trivially holds that $\text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$. We already mentioned how time easily limits space: if a machine M runs in time $T(n)$ then it can write at most $T(n)$ cells, meaning that $S(n) \leq T(n)$. Thus, it clearly holds that $\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$ and $\text{NTIME}(f(n)) \subseteq \text{NSPACE}(f(n))$.

These last inclusions can be improved through a fundamental property that differentiates space complexity from time complexity: the possibility of **re-using space**, i.e. using cells that were previously written without writing new ones. This property makes space complexity way more powerful than time complexity: since we don't care about time, we could make an exponential number of steps even in a very low amount of space. In fact, deterministic space machines are even stronger even than non-deterministic time ones.

Proposition 15: Time-bounded space

For any time-constructible function $T : \mathbb{N} \to \mathbb{N}$, it holds that:

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

Proof. Let V be a verifier for a language L with running time $R(n)$. Since V runs in $T(n)$ steps, we know that the length of each witness will be at most $T(n)$. We define a new machine M as follows: given the input string x, cycle between all the possible strings $w \in \{0,1\}^{T(n)}$ and check if $V(x,w) = 1$. If V accepts then M also accepts. If none of the possible strings are a valid witness, M rejects.

The running time of M is clearly $O(2^{T(n)})$, but we only care about space. Each witness has length at most $T(n)$, while V's simulation requires at most $T(n)$ cells. On each iteration, we can reuse the cells of the previous iteration, concluding that we require at most $O(T(n))$ space and thus that $L \in \text{DSPACE}(T(n))$. □

Space can also limit time, even though this bound is not as powerful as the one imposed by time on space. To achieve this limitation, we use the notion of **configuration graph**. Let M be a deterministic (or non-deterministic) TM. A *configuration* of M is a string containing of all non-blank entries of M's work tape, along with its state and head positions, at a particular point in its execution. For instance, the encoding $\texttt{WORKTAPE};j;q_i$ represents a configuration such that:

- The work tape contains the string `WORKTAPE`
- The work tape's head is positioned on the cell A
- The input tape's head is positioned on the j-th cell
- The current state is q_i

Definition 30: Configuration graph of a computation

Let M be a TM (or NDTM) and let x an input. The **configuration graph** of the computation $M(x)$ is the directed graph $G_{M,x}$ whose nodes are all the possible configurations of M, where two nodes C_i, C_j are connected by a directed edge if M can transition from C_i to C_j when the input is x, according to its transition function. The node corresponding to the initial configuration is C_{start}. Since $G_{M,x}$ may have more than one accepting configuration, to ensure that we have only one final node, we add an additional node C_{accept} whose in-going edges come only from all the accepting configurations.

By definition, for each pair of inputs x, x' the graphs $G_{M,x}$ and $G_{M,x'}$ will have the same set of nodes. However, their edges may be different. The configuration graph of a computation allows us to convert the computation $M(x)$ to a simple graph problem: we have that $M(x) = 1$ if and only if $G_{M,x}$ has a path from C_{start} to C_{accept}.

Proposition 16: Space-bounded time

For any space-constructible function $f : \mathbb{N} \to \mathbb{N}$ with $f(n) \geq \log n$, it holds that:

$$\text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n)) \subseteq \text{DTIME}(2^{f(n)})$$

Proof. First, we notice that if M is deterministic, then the graph $G_{M,x}$ has out-degree one, while if M is non-deterministic then the graph has out-degree at most two. Thus, non-determinism doesn't really change the graph size, only the number of edges, which can still be at most the square of the number of vertices.

If the space complexity of M is $S(n)$, then each configuration requires $O(S(n))$ bits to be encoded. This implies that there are at most $2^{O(S(n))}$ configurations, meaning that the graph has exponential size. We can define a machine M' that, given the input x, first constructs the graph $G_{M,x}$ and then runs a DFS on such graph to find a path from C_{start} to C_{accept}. Since a DFS requires linear time with respect to the graph size, this computation requires at most $2^{O(S(n))}$ steps. □

In the previous proof, we reasoned on how to use time by reasoning on space. However, a very similar idea can be used to work with space while reasoning on space itself. Through this intuition, Savitch [Sav70] showed that deterministic space is even capable of simulating non-deterministic space with an small blow-up in required space.

Theorem 35: Savitch's theorem

For any space-constructible function $f : \mathbb{N} \to \mathbb{N}$ with $f(n) \geq \log n$, it holds that:

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

Proof. Let N be a NDTM with required space $S(n)$. For any input x, we know that the configuration graph $G_{N,x}$ has $2^{O(S(n))}$ nodes. We notice that that, for any vertices u, v, there is a path $u \to v$ of length at most 2^k if and only if there's a node z with two paths $u \to z$ and $z \to y$ both of at most 2^{i-1} nodes. Based on this idea, we define the following deterministic recursive procedure that returns 1 if and only if there is a path $u \to v$ of length at most 2^k.

`Reach`?(G, x, y, k):

1. If $k = 0$, check if $x = y$. If true, return 1. Otherwise, return 0.
2. If $k = 1$, check if $(x, y) \in E(G)$. If true, return 1. Otherwise, return 0.
3. If $k > 0$, repeat the following for each node $z \in V(G)$:
    4. Recursively run Reach?$(x, z, k-1)$ and Reach?$(z, y, k-1)$. If they both return 1, then return 1. Otherwise, return 0.

By defining a TM M that computes Reach?$(G_{N,x}, C_{\text{start}}, C_{\text{accept}}, S(n))$ on input x, we get that $M(x) = 1$ if and only if there is a path of at most $2^{S(n)}$ nodes, which covers the whole graph.

Now we focus on the space complexity of the procedure. Let $R(m, k)$ be the space complexity of the procedure while being run on a graph G with m nodes and a value k. We notice that on each recursive step, the procedure requires $\Theta(\log m)$ cells to enumerate all the vertices since — assuming the nodes are also labeled by a number and not only by a configuration — we only need to store the index of the current node. Moreover, while doing the second recursive call, the procedure can re-use the space of the first recursive call. We get that $R(m, k) = R(m, k-1) + \Theta(\log m)$, where $R(m, 0) = R(m, 1) = \Theta(1)$. By solving this recursive equation, we conclude that $R(m, k) = k \log m$.

Thus, the computation of Reach?$(G_{N,x}, C_{\text{start}}, C_{\text{accept}}, S(n))$ requires $O(S(n)^2)$ space. However, since the graph $G_{N,x}$ has exponential size, the space required by M to compute the whole graph $G_{N,x}$ before running the procedure would be too large. To fix this issue, we can simply compute the nodes of the graph during the recursion: the next node to be iterated on is simply the next generate configuration. □

6.2 Games and the class PSPACE

After defining the relationships between time and space, we are ready to define the space complexity equivalents of P and NP, i.e. the classes PSPACE and NPSPACE

Definition 31: The classes PSPACE and NPSPACE

We define PSPACE as the class of the languages decidable in polynomial space:

$$\mathsf{PSPACE} = \bigcup_{k \geq 0} \mathrm{DSPACE}(n^k)$$

Likewise, we define NPSPACE as the class of the languages decidable in non-deterministic polynomial space:

$$\mathsf{NPSPACE} = \bigcup_{k \geq 0} \mathrm{NSPACE}(n^k)$$

Savitch's theorem directly concludes that PSPACE = NPSPACE, proving that space is indeed very different from time. This quite surprising since our intuition is that the corresponding classes for time complexity are separated. From Proposition 15 and Proposition 16, instead, we easily get that:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$$

As for the class NP, we're interested in defining a concept of PSPACE-Hard. However, we have to make some clever observations. Since we defined NP-Hardness through Karp reductions, i.e. polynomial time reductions, the simplest idea would be to define PSPACE-Hardness through polynomial space reductions. However, we notice that, in this case, any single problem in PSPACE would also be PSPACE-Complete: given two problems $A, B \in \mathsf{PSPACE}$, two inputs $x_{\text{yes}}, x_{\text{no}}$ such that $x_{\text{yes}} \in B$ and $x_{\text{no}} \notin B$ and a poly-space machine M that solves A, we can define a function f such that $f(x) = x_{\text{yes}}$ if $M(x) = 1$

and $f(x) = x_{\text{no}}$ if $M(x) = 0$. Since the simulation of M requires polynomial space, f is a polynomial space reduction for which $x \in A$ iff $f(x) \in B$ — notice that this "trick" doesn't work for NP-Completeness since the computation of f must be deterministic.

In order to give a better definition of PSPACE-Hardness, we have to look to the *implications* of NP-Complete: if any NP-Complete problem is proven to be in P, then the whole class NP collapses into P. With this idea in mind, a more interesting definition of PSPACE-Hardness derives from the preservation of Karp reductions.

Definition 32: PSPACE-Hardness and PSPACE-Completeness

A language B is said to be PSPACE-Hard if $\forall A \in \text{NP}$ it holds that $A \leq_P B$. If B is also in PSPACE, we say that it is PSPACE-Complete.

This definition clearly preserves the idea of "collapse under reductions": if any PSPACE-Complete problem is proven to be in NP (or P), then the whole class PSPACE collapses into NP (or P). The most interesting PSPACE-Complete problem that satisfies this definition is the *true quantified Boolean formula problem*, defined as follows:

$$\text{TQBF} = \{\langle\psi\rangle \mid \psi = [Q_1 x_1 \ \ldots \ Q_k x_k \ \phi(x_1, \ldots, x_k)] \text{ is true}\}$$

where ϕ is an unquantified Boolean formula and $Q_1, \ldots, Q_k$ are either $\exists$ or $\forall$, without restrictions. We notice that since all the variables of a QBF are bound by some quantifier, the QBF is always either true or false. Moreover, we notice that this problem differs from Σ_n^PSAT and Π_n^PSAT for two reasons: the quantifiers don't have to be alternating and the whole quantified formula must be an always-true statement.

We won't dive into the proof of PSPACE-Completeness for this problem, but the idea is similar to what can be done for SAT: just encode the whole computation as a QBF that is always true.

Theorem 36

TBQF is PSPACE-Complete

(proof omitted)

We recall that the central feature of NP problems is that a yes answer has a short certificate. The analogous concept for PSPACE problems seems to be that of a **winning strategy for a two-player game** with perfect information. A good example of such a game is Chess: Two players alternately make moves, and the moves are made on a board visible to both, hence the term *perfect information*. Here, a *winning strategy* refers to a predefined set of moves that guarantees a player victory in a game, regardless of how the opponent plays. Clearly, any winning strategy can be represented as a QBF. There are $2k$ variables $x_1, \ldots, x_{2k}$ such that odd numbered variables correspond to the move of Player 1 and even numbered variables correspond to the move of Player 2.

In order for Player 1 to have a winning strategy, they must have a way to win for all possible sequences of moves by Player 2. In other words, we have that:

$$\exists x_1 \, \forall x_2 \, \exists x_3 \, \ldots \, \exists x_{2k-1} \, \forall x_{2k} \, \phi(x_1, \ldots, x_{2k}) \text{ is true}$$

The same idea also holds for a winning strategy for Player 2: we just have to swap each $\exists$ with a $\forall$ and vice versa. Deciding whether or not a player has a winning strategy seems to require searching the tree of all possible moves. This looks like something we would do in NP. However, just like in the case of the polynomial hierarchy, the crucial difference is the lack of a short witness for the statement "Player 1 has a winning strategy," since the only certificate we can think of is the winning strategy itself, which requires exponentially many bits to even describe.

In 1976, Tarjan and Even suggested that this difference between NP and PSPACE is the same as for why *puzzles* are easier than *games*, that is the fact that the initiative can shift back and forth between the players.

Proving PSPACE-Completeness of games may seem like a frivolous pursuit, but similar ideas lead to PSPACE-Completeness of some practical problems. Usually, these problems involve repeated moves by an agent who faces an adversary with unlimited computational power. For instance, many computational problems of robotics involve a robot navigating in a changing environment.

6.3 Logarithmic space and the classes L, NL

In the context of time complexity, the running time of a machine clearly cannot be lower than the input length, since otherwise the machine wouldn't even be able to read the whole input. Thus $T(n) \geq n$ is always true. In space complexity, instead, the number of cells written on the work tape can be way lower than the length of the input string. However, we still require that the space is at least logarithmic, since otherwise we wouldn't be able to achieve interesting computations. Thus, we assume that $S(n) \geq \log n$. We define the additional two classes based on **logarithmic space**.

Definition 33: The classes L and NL

We define PSPACE as the class of the languages decidable in logarithmic space:

$$\mathsf{L} = \mathrm{DSPACE}(\log n)$$

Likewise, we define NPSPACE as the class of the languages decidable in non-deterministic logarithmic space:

$$\mathsf{NL} = \mathrm{NSPACE}(\log n)$$

Clearly, we have that $\mathsf{L} \subseteq \mathsf{NL}$. Since $S(n) \geq \log n$ is assumed to always be true, any language that is in L (or NL) can be computed in $\Theta(\log n)$. This makes low space a nice topic of study: many algorithms may have to be executed on devices with very limited

memory. From Proposition 15 and Proposition 16, we get that $\mathsf{L} \subseteq \mathsf{NL} \subseteq \text{DTIME}(n) \subseteq \mathsf{P}$. This means that any low space computation requires at most linear time. This makes low space computation not only very efficient in space but also very efficient in time. Savitch's theorem, instead, shows that $\mathsf{NL} \subseteq \mathsf{L}^2$, where the latter is the class $\mathsf{L}^2 = \text{DSPACE}(\log^2 n)$. Even thought space is more flexible than time, the question $\mathsf{L} \stackrel{?}{=} \mathsf{NL}$ is as-hard-as the question $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

Following the same argument that we made for PSPACE-Completeness, in order to define NL-Completeness we have to slightly tweak our definitions. In this particular case, we're trying to make the class NL collapse onto the class L. Hence, we have to preserve logarithmic space. When A reduces to B in logarithmic space, we write $A \leq_L B$.

Definition 34: NL-Hardness and NL-Completeness

A language B is said to be NL-Hard if $\forall A \in \mathsf{NP}$ it holds that $A \leq_L B$. If B is also in NL, we say that it is NL-Complete.

In Karp reductions, one of the main properties is the transitivity between reductions, that is the fact that if $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$. For logarithmic space reductions, this transitivity property still holds, but it isn't as trivial as the polynomial time case: a log-space machine might not even have the memory to write down its output. To solve this issue, we can compute the two reduction bit-by-bit: after computing the next bit of the first reduction, we feed this bit to the second reduction, producing the next final bit. This is similar to how in Theorem 35 the graph gets generated during the recursive calls. This idea preserves the transitive property for log-space reductions, meaning that if $A \leq_L B$ and $B \leq_L C$ then $A \leq_L C$. With the same idea, we get that if $A \leq_L B$ and $B \in \mathsf{L}$ then $A \in \mathsf{L}$: we compute the reduction bit-by-bit and feed the result to B's log-space machine.

The main NL-Complete problem is the *st-connectivity problem*, i.e. the PATH problem that we already discussed. This shouldn't come as a surprise: Savitch's theorem essentially reduces any non-deterministic computation to a PATH instance. Again, the idea is to generate the graph $G_{N,x}$ bit-by-bit while feeding it to a machine that solves PATH. This easily makes the problem NL-Hard. To show that it also lies in NL, we consider a non-deterministic BFS: each branch has to store only the previous node, meaning that logarithmic space suffices.

Theorem 37

PATH is NL-Complete

As for the class NP, in NL class we can also derive the alternative definition via verifier. However, in this case we need needs a small change in order to respect the strict space constraints:

- By constraining the witness to a logarithmic length we obtain a too restricted subset of problems, due to the witness being too small, therefore the subset of considered

problems would not cover NL

- By keeping the witness of polynomial length the witness can be used as *additional memory*: we could read the same information multiple times without having to store it in the work tape, saving a considerable amount of space

In other words, in the first case we obtain verifiers that are too weak, while in the second they are too powerful. To get a middle ground between the two, we can impose that the witness is **read-once**: the verifier has an additional tape containing the witness whose cells can be read only once — in other words, the head cannot move to the left.

Theorem 38: The class NL (2nd Definition)

Given the NL class, we have that:

$$\mathsf{NL} = \{L \subseteq \{0,1\}^* \mid L \text{ is verifiable by read-once witnesses in log space}\}$$

This alternative definition of NL also gives us a nice alternative definition for the class coNL. In particular, since PATH is NL-Complete, we know that $\overline{\text{PATH}}$ is coNL-Complete. Unlike in the case of PATH, there is no natural witness for the non-existence of a path between two nodes, thus it seemed "obvious" to researchers that $\overline{\text{PATH}} \notin \mathsf{NL}$. In two surprising results, Immerman [Imm88] and Szelepcsényi [Sze88] showed that this intuition was wrong. As a corollary, this implies that the class NL is actually closed under complementation, i.e. that $\mathsf{NL} = \mathsf{coNL}$.

Theorem 39: The Immerman-Szelepcsényi theorem

$\overline{\text{PATH}} \in \mathsf{NL}$

Proof. The idea behind the following proof is that the polynomial length of the certificate can be abused: we cannot read the same cells multiple times, but we can insert an adequate number of copies of the previous cells into the certificate.

Let $m = |V(G)|$ and let $R_0, \ldots, R_m$ be the sets of vertices reachable from s with maximum $i \in [0, m-1]$ steps:

$$R_i = \{v \in V(G) \mid s \to v \text{ with maximum } i \text{ edges}\}$$

Let us also assume that the vertices of G are identified by a number. For each vertex $v \in V(G)$, verifying that $v \in R_i$ is very easy: all it takes is a certificate $\langle v_0, \ldots v_k \rangle$ describing the path, so that the verifier can perform the following procedure:

`certify_v_in_R_i` $(\langle v_0, \ldots, v_k \rangle, i, v)$:

1. Initialize a counter $h = 0$
2. Verify if $v_0 = s$. If true, increment h, otherwise the procedure *rejects*.
3. Verify that for each $0 < j < k$ it is true that $(v_{j-1}, v_j) \in E(G)$. If true, increment h, otherwise the procedure *rejects*.

4. Check if $v_k = v$. If true, increment h (at this point we will have that $h = k$), otherwise the procedure *rejects*.

5. Check if $h \leq i$. If true, the procedure *accepts*, otherwise *rejects*.

With this order of operations, the certificate is read-once. Furthermore, since the vertices of G are identified by a number, the maximum length of this certificate is $O(m \log n)$. The procedure above, however, only allows us to certify that $v \in R_i$, but not that $v \notin R_i$. This operation is in fact more complex, but it can be structured with recursive copies of the certificate.

Suppose we know that $|R_i| = \ell_i$ and we want to verify that $v \notin R_i$. Consider the sub-certificate $\left\langle c_{u_1}, \ldots, c_{u_{\ell_i}} \right\rangle$, where c_{u_k} is in turn a sub-certificate that verifies whether $u_k \in R_i$. Furthermore, since the vertices of G are identified by a number, in order for each sub-certificate to be different it is sufficient to verify that $u_1 < \ldots u_{\ell_i}$. We therefore define the following procedure:

`certify_v_not_in_R`$_i$`_with_` $|R_i|$ $\left(\left\langle c_{u_1}, \ldots, c_{u_{\ell_i}} \right\rangle, i, v, \ell_i\right)$:

1. Initialize $w = 0$ and $k = 1$.

2. Repeat until the end of the sub-certificate:

    3. Check whether $u_k \in R_i$ using the procedure `certify_v_in_R`$_i$ (c_{u_k}, i, u_k). If the procedure rejects, this procedure *rejects* as well.

    4. If $w \neq 0$, check if $w < u_k$. If true, set $w = u_k$ and increment k, otherwise *rejects*.

    5. Check $u_k \neq v$. If false, *rejects*.

5. Check if $k = \ell_i$. If true, *accept*, otherwise *rejects*.

With this order of operations, this certificate is read-once. Furthermore, since for each i we have that $|R_i| \leq m$, the certificate contains at most m sub-certificates, which we know have length $O(m \log n)$. Therefore, this certificate has length $O(m^2 \log n)$.

Suppose instead that we know that $|R_{i-1}| = \ell_{i-1}$ and we want to verify that $v \notin R_i$. Similarly to the previous case, consider the sub-certificate $\left\langle c_{u_1}, \ldots, c_{u_{\ell_{i-1}}} \right\rangle$, where c_{u_k} this time is a sub-certificate that verifies whether $u_k \in R_{i-1}$. The procedure is similar to the previous one, but with a fundamental difference: we verify that none of the vertices in R_{i-1} are adjacent vertices to v.

`certify_v_not_in_R`$_i$`_with_` $|R_{i-1}|$ $\left(\left\langle c_{u_1}, \ldots, c_{u_{\ell_{i-1}}} \right\rangle, i, v, \ell_{i-1}\right)$:

1. Initialize $w = 0$ and $k = 1$.

2. Repeat until end of the sub-certificate:

    3. Check if $u_k \in R_i$ using the procedure `certify_v_in_R`$_i$ $(c_{u_k}, i-1, u_k)$. If the procedure rejects, this procedure also *rejects*.

4. If $w \neq 0$, check if $w < u_k$. If true, set $w = u_k$ and increment k, otherwise *rejects*.
5. Check if $w \neq v$ and $w \neq u$ for every vertex adjacent to v. If false, *rejects*.

5. Check if $k = \ell_{i-1}$. If true, *accept*, otherwise *reject*.

This sub-certificate is also read-once and its length is $O(m^2 \log n)$. At this point, we need to define the last step: knowing that $|R_{i-1}| = \ell_{i-1}$, we want to verify that $|R_i| = \ell_i$. Let's consider the sub-certificate $\langle c_{v_0}, \ldots, c_{v_{m-1}} \rangle$. For each vertex v of G we have only two options: the vertex belongs to R_i, so we can verify this by giving a certificate to the first procedure, or it does not belong to R_i, so knowing $|R_{i-1}|$ we can verify this by giving a certificate to the third procedure.

`certify_` $|\mathtt{R_{i-1}}|$ `_with_` $|\mathtt{R}|$

1. Initialize a counter $h = 0$.
2. Repeat for $k = 0, \ldots, m-1$:
3. Check whether $v_k \in R_i$ using the procedure `certify_v_in_R`$_\mathtt{i}$ (c_{v_k}, i, v_k). If the procedure accepts, increment h.
4. Otherwise, check whether $v_k \notin R_i$ via the procedure `certify_v_not_in_R`$_\mathtt{i}$`_with_` $|\mathtt{R_{i-1}}|$ $(c_{v_k}, i, v_k, \ell_{i-1})$. If the procedure rejects, this procedure *rejects* too.
5. If $h = \ell_i$ then *accept*, otherwise *reject*.

This procedure does nothing more than calculate the cardinality of R_i, to then verify whether it is equal to the given value ℓ_i. Intuitively, we note that the certificate is composed of m other sub-certificates each of length $O(m^2 \log n)$. Therefore, this certificate has length $O(m^3 \log n)$.

Once all the necessary procedures have been defined, we define the following TM V:

V = "Given the string $\langle \langle G, s, t \rangle, c \rangle$ as input:

1. Interpret $c = \langle \ell_0, \ldots, \ell_{m-1}, c_0, \ldots, c_{m-1}, c_t \rangle$
2. Repeat for $i = 0, \ldots, m-1$:
3. Execute the procedure `certify_` $|\mathtt{R_{i-1}}|$ `_with_` $|\mathtt{R_{i-1}}|$ $(c_i, i, \ell_i, \ell_{i-1})$. If the procedure rejects, this procedure *rejects* too.
4. Execute the procedure `certify_v_not_in_R`$_\mathtt{i}$`_with_` $|\mathtt{R_m}|$ (c_t, m, t, ℓ_m). If the procedure accepts, this procedure also *accepts*, otherwise *rejects*.

In other words, the TM V uses the values $\ell_0, \ldots, \ell_{m-1}$ and the sub-certificates $c_0, \ldots, c_{m-1}$ to to certify that $|R_m| = \ell_m$, then use the sub-certificate c_t to determine whether $t \in R_m$, thus concluding that V is a verifier for $\overline{\text{PATH}}$. Since each of the procedures is read-once and the final certificate is composed of m integers each requiring $\log n$ bits, $m+1$ certificates each requiring $O(m^3 \log n)$ bits, and since m is in $O(n)$, we conclude that the space cost of the verifier V is $O(m \log n + (m+1)(m^3 \log n)) = O(n^4 \log n)$ and therefore that $\overline{\text{PATH}} \in \mathsf{NL}$. □

Bibliography

[Adl78] Leonard Adleman. "Two theorems on random polynomial time". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. 1978. DOI: 10.1109/SFCS.1978.37.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES is in P". In: *Annals of Mathematics* (2004). DOI: 10.4007/annals.2004.160.781.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, et al. "Time bounds for selection". In: *Journal of Computer and System Sciences* (1973). DOI: https://doi.org/10.1016/S0022-0000(73)80033-9.

[Imm88] Neil Immerman. "Nondeterministic Space is Closed under Complementation". In: *SIAM Journal on Computing* (1988). DOI: 10.1137/0217058.

[Lau83] Clemens Lautemann. "BPP and the polynomial hierarchy". In: *Information Processing Letters* (1983). DOI: https://doi.org/10.1016/0020-0190(83)90044-3.

[Sav70] Walter J. Savitch. "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of Computer and System Sciences* (1970). DOI: https://doi.org/10.1016/S0022-0000(70)80006-X.

[Sip83] Michael Sipser. "A complexity theoretic approach to randomness". In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. 1983. DOI: 10.1145/800061.808762.

[Sze88] R. Szelepcsényi. "The method of forced enumeration for nondeterministic automata". In: (1988). DOI: 10.1007/BF00299636.