ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MASTER'S DEGREE IN ARTIFICIAL INTELLIGENCE

$\sim \cdot \sim$

ACADEMIC YEAR 2024–2025

LANGUAGES AND ALGORITHMS FOR AI
Module 3

Prof.
Ugo Dal Lago

Author
Niccolò Zanotti

Version: February 8, 2025
Latest version, Source

Summary

Notes for the Languages and Algorithms for AI module 3 course taught by Prof. Ugo dal Lago in a.y. 2023/2024.

The course gives a brief introduction to the Theory of Computation and primarily deals with Computational Complexity Theory. Finally an introduction to Computational Learning Theory is given.

For lack of time, Chapter 3 on Computational Learning theory is not fully complete. If you wish to give a second life to this notes integrating updates to reflect newer changes in the Course, fork the repo on Github and contribute!

Contents

Summary i

0 Mathematical Preliminaries 1
- 0.1 Representing objects as strings . . . 2
- 0.2 Decision problems/languages . . . 2
- 0.3 Big-Oh notation . . . 3

1 The Computational Model 7
- 1.1 The Turing Machine . . . 7
  - 1.1.1 Machine computations . . . 9
  - 1.1.2 Machine efficiency, runtime . . . 9
  - 1.1.3 Machines as strings . . . 10
- 1.2 Uncomputability . . . 10
  - 1.2.1 The halting problem . . . 11
- 1.3 Semantic Languages . . . 11
- 1.4 Complexity classes . . . 12
- 1.5 Polynomial time computable problems . . . 13
- 1.6 The Church-Turing Thesis . . . 13
- 1.7 The complexity class FP . . . 14
- 1.8 The class EXP . . . 14

2 NP and NP–completeness 17
- 2.1 The Complexity class NP . . . 17
  - 2.1.1 Relation between **NP** and **P** . . . 17
  - 2.1.2 Nondeterministic Turing machines . . . 18
- 2.2 Reductions and NP–completeness . . . 19
- 2.3 The Cook-Levin Theorem . . . 20
- 2.4 Proving the hardness of a problem . . . 21
- 2.5 Examples of **NP**-complete problems . . . 22
  - 2.5.1 `INDSET` . . . 22
  - 2.5.2 `CLIQUE` . . . 22
  - 2.5.3 `kSAT` . . . 22
  - 2.5.4 Vertex Cover or Node Cover . . . 23
  - 2.5.5 Universal Sink (US) . . . 23
  - 2.5.6 Set Packing (SP) . . . 23
  - 2.5.7 Subgraph Isomorphism (SI) . . . 23
  - 2.5.8 SUBSETSUM (SSP) . . . 23

3 Elements of Computational Learning Theory **25**
3.1 Terminology 25
3.2 The General Model 25
3.3 PAC learnable and efficiently PAC learnable 26
3.4 The PAC learning model 26

4 Exercises **29**
4.1 Turing Machines 29
4.2 Undecidability and Rice's theorem 30
4.3 Polynomial-time computable problems 31
4.4 **NP** complete problems 37

Bibliography **43**

CHAPTER ZERO

Mathematical Preliminaries

The notation we will adopt closely follows the one in Arora and Barak (2009)[1].

[1] Sanjeev Arora and Boaz Barak (2009). *Computational Complexity: A Modern Approach*. Cambridge: Cambridge University Press

Standard notation

We let $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$ denote the set of integers, and $\mathbb{N}$ denote the set of natural numbers (i.e., nonnegative integers). A number denoted by one of the letters i, j, k, ℓ, m, n is always assumed to be an integer. If $n \geq 1$, then $[n]$ denotes the set $\{1, \ldots, n\}$. For a real number x, we denote by $\lceil x \rceil$ the smallest $n \in \mathbb{Z}$ such that $n \geq x$ (ceiling function) and by $\lfloor x \rfloor$ the largest $n \in \mathbb{Z}$ such that $n \leq x$. Whenever we use a real number in a context requiring an integer, the operator $\lceil\ \rceil$ is implied.

We denote by $\log x$ the logarithm of x to the base 2.

We say that a condition $P(n)$ holds for *sufficiently large* n if there exists some number N such that $P(n)$ holds for every $n > N$ (for example, $2^n > 100n^2$ for sufficiently large n). We use expressions such as $\sum_i f(i)$ (as opposed to, say, $\sum_{i=1}^{n} f(i)$) when the range of values i takes is obvious from the context. If u is a string or vector, then u_i denotes the value of the i^{th} symbol/coordinate of u.

Strings

Definition 0.1 (String)**.** Let S be a finite set. A string over the alphabet S is a finite, possibly empty, tuple of elements of S. We employ the following notation

$S^n \doteq \{\text{set of all strings over } S \text{ of length exactly } n \in \mathbb{N}\}$;

$S^0 \doteq \varepsilon$ is the empty string;

$S^* \doteq \{\text{set of all strings}\} = \cup_{n \geq 0} S^n$.

In this course we will typically consider strings over the *binary* alphabet $S = \{0, 1\}$. If x and y are strings, then we denote their concatenation (the tuple that contains first the elements of x and then the elements of y) by $x \circ y$ or sometimes simply xy. The set of strings forms a monoid with the

concatenation operation (associativity + ε as identity element). If x is a string and $k \geq 1$ is a natural number, then x^k denotes the concatenation of k copies of x. For example, 1^k denotes the string consisting of k ones. The length of a string x is denoted by $|x|$.

Additional notation

If S is a distribution then we use $x \in_R S$ to say that x is a random variable that is distributed according to S; if S is a set then this denotes that x is distributed uniformly over the members of S. We denote by U_n the uniform distribution over $\{0,1\}^n$. For two length-n strings $x, y \in \{0,1\}^n$, we denote by $x \odot y$ their dot product modulo 2; that is $x \odot y = \sum_i x_i y_i \pmod 2$. In contrast, the inner product of two n-dimensional real or complex vectors $\boldsymbol{u}, \boldsymbol{v}$ is denoted by $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ (see Section A.5.1). For any object x, we use $\lfloor x \rfloor$ (not to be confused with the floor operator $\lfloor x \rfloor$) to denote the representation of x as a string (see Section 0.1).

0.1 Representing objects as strings

The basic computational task considered in this book is *computing a function*. In fact, we will typically restrict ourselves to functions whose inputs and outputs are finite *strings of bits* (i.e., members of $\{0,1\}^*$).

Representation

Considering only functions that operate on bit strings is not a real restriction since simple encodings can be used to *represent* general objects—integers, pairs of integers, graphs, vectors, matrices, etc.—as strings of bits. For example, we can represent an integer as a string using the binary expansion (e.g., 34 is represented as 100010) and a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix A such that $A_{i,j} = 1$ iff the edge $\overline{ij}$ is present in G). We will typically avoid dealing explicitly with such low-level issues of representation and will use $\lfloor x \rfloor$ to denote some canonical (and unspecified) binary representation of the object x. Often we will drop the symbols $\lfloor \rfloor$ and simply use x to denote both the object and its representation.

Computing functions with nonstring inputs or outputs

The idea of representation allows us to talk about computing functions whose inputs are not strings (e.g., functions that take natural numbers as inputs). In all these cases, we implicitly identify any function f whose domain and range are not strings with the function $g : \{0,1\}^* \to \{0,1\}^*$ that given a representation of an object x as input, outputs the representation of $f(x)$. Also, using the representation of pairs and tuples, we can also talk about computing functions that have more than one input or output.

0.2 Decision problems/languages

Definition 0.2 (Language)**.** We call a *language* any subset of S^* where S is an alphabet.

An important special case of functions mapping strings to strings is the case of *Boolean* functions[2], whose output is a single bit

[2] They are also called characteristic functions.

Definition 0.3 (Boolean function)**.** We call Boolean function any function f with $f : \{0,1\}^* \to \{0,1\}$. Any boolean function naturally identifies a language:

$$\mathcal{L}_f = \{x \in \{0,1\}^* : f(x) = 1\} \subseteq \{0,1\}^* . \quad (1)$$

We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of deciding the language $\mathcal{L}_f$ (i.e., given x, decide whether $x \in \mathcal{L}_f$):

Definition 0.4 (Decision problem)**.** A decision problem for a given language $\mathcal{M}$ can be seen as the task of computing f such that $\mathcal{M} = \mathcal{L}_f$.

Example 0.5 (INDSET)**.** By representing the possible invitees to a dinner party with the vertices of a graph having an edge between any two people who don't get along, the dinner party computational problem from the introduction becomes the problem of finding a maximum sized *independent set* (set of vertices without any common edges) in a given graph. The corresponding language is:

$$\text{INDSET} = \{(G,k) : \exists S \subseteq V(G) \text{ s.t. } |S| \geq k \text{ and } \forall u,v \in S, (u,v) \notin E(G)\}$$

An algorithm to solve this language will tell us, on input a graph G and a number k, whether there exists a conflict-free set of invitees, called an *independent set*, of size at least k. It is not immediately clear that such an algorithm can be used to actually find such a set, but we will see this is the case in Chapter 2. For now, let's take it on faith that this is a good formalization of this problem.

0.3 Big-Oh notation

We will typically measure the computational efficiency of an algorithm as the number of basic operations it performs as *a function of its input length*. That is, the efficiency of an algorithm can be captured by a function T from the set $\mathbb{N}$ of natural numbers to itself such that $T(n)$ is equal to the maximum number of basic operations that the algorithm performs on inputs of length n. However, this function T is sometimes overly dependant on the low-level details of our definition of a basic operation. For example, the addition algorithm will take about three times more operations if it uses addition of single digit *binary* (i.e., base 2) numbers as a basic operation, as opposed to *decimal* (i.e., base 10) numbers. To help us ignore these low-level details and focus on the big picture, the following well-known notation is very useful.

Definition 0.6 (Big-Oh notation)**.** Let $f, g : \mathbb{N} \to \mathbb{N}$. Then we say that

$$f = O(g) \text{ if } \exists c \in \mathbb{R}^+ \text{ s.t. } f(n) \leq c \cdot g(n)\,, \tag{2}$$

$$f = \Omega(g) \text{ if } \exists c \in \mathbb{R}^+ \text{ s.t. } f(n) \geq c \cdot g(n) \tag{3}$$

for sufficiently large n, and

$$f = \Theta(g) \text{ if } f = O(g) \text{ and } f = \Omega(g)\,. \tag{4}$$

Exercises

Exercise 1. One maybe interested in evaluating the cardinality $|\mathcal{L}|$ of a set $\mathcal{L} \leq S^n$. What is the cardinality of S^n , namely the set of all strings of length $n \in N$.

Solution. $|S^n| = |S|^n$ this can be proved by induction. If one wants to be sure that a statement P(n) about natural numbers holds for all $n \in N$ one can prove that:

- P(0) holds (**BASE CASE**)
- $\forall n.P(n) \Rightarrow P(n+1)$ (**INDUCTIVE CASE**)

In the context of our exercise, then:

- The base case consists in proving that $|S^0| = |S|^0$, this is very easy: $|S^0| = |\{\varepsilon\}| = 1$ (the cardinality of all the function of length zero). Then we have that $|S|^0 = 1$ because the cardinality of S is surely different from zero, so to the power of 0 it is 1.
- The inductive case. We suppose $|S^n| = |S|^n$ and we prove that $|S^{n+1}| = |S|^{n+1}$: if we start from $|S^{n+1}| = |S||S^n|$ there are S many ways of choosing the first symbol and $|S^n|$ ways of choosing the rest. Then we substitute $|S^n| = |S|^n$ and we get the result.

So we have proved that the number of element in the set of Strings of length n is exponential, where the base of the exponential is the cardinality of the alphabet.

Exercise 2. Relate the following pair of functions by way of the asymptotic operators O, Ω, Θ :

- $f_1(n) = n\log(n)$
- $g_1(n) = 10n\log(\log(n))$
- $f_2(n) = 1000n$
- $g_2(n) = \frac{1}{100}n\log(n)$

Solution. Let use consider the first pair of functions (f_1 and g_1):

$$\lim_{n\to\infty} \frac{f_1(n)}{g_1(n)} = \lim_{n\to\infty} \frac{n\log(n)}{10n\ \log(\log(n))}$$

the n goes away, also we can say that $\log(n)$ grows asymptotically faster than log $(\log(n))$ so the limit will will be $+\infty$. As a consequence we can

say that $f_1(n)$ is Ω of g_1. And dually $g_1(n) = O\ (f_1(n))$.
For the second pair of functions (f_2 and g_2) :

$$\lim_{n\to\infty} \frac{f_1(n)}{g_1(n)} = \lim_{n\to\infty} \frac{1000n}{\frac{1}{100} n \log(n)}$$

the n goes away, so it is a constant divided by $\log(n)$, so for n that goes to infinity the ration of f_2 and g_2 will go to 0. So we have that f_2 is $(O(g_2))$

Exercise 3. We would like to find appropriate encodings for the following discrete sets:

- A. The set $\mathbb{Q}$ of rational numbers
- B. Disjoint union of $\mathbb{N}$ and $\{0,1\}^*$ which is $\{(l,n)|n \in N\} \cup \{(r,s)|s \in \{0,1\}\}$
- C. Graphs, namely pairs in the form (V,E) such that V is a finite set of nodes and $E \subset V \times V$ is a finite set of edges

Solution (A)**.** In the first it is sufficient to observe that elements of $\mathbb{Q}$ can be encoded as

$$\mathbb{Q} = \{(z/n) : z \in \mathbb{Z}, n \in \mathbb{N}, n > 0\}\,.$$

So all together (z,n) becomes the string: S_z (sign of z) m_z (binary encoding of the modulus of z) $\#$ m_n(binary encoding of n):

$$\llcorner(z/n)\lrcorner = S_z\ m_z \#\ m_n$$

of course appropriately encoded in $\{0,1\}^*$

Solution (B)**.** This is relative easy because in representing disjoint unions we can encode

$$N \biguplus \{0,1\}$$

as follow :

$$(l,n) \Rightarrow 0 \cdot \llcorner n \lrcorner$$

$$(r,s) \Rightarrow 1 \cdot S$$

Solution (C)**.** There are **many** ways of representing graphs, one possibility being the following:

- Nodes are identified with the natural numbers between 1 and $|V|$;
- Each edge can be seen as a pair of nodes, by definition;
- The whole set of edges can be represented as: $(v_1^1, v_2^1), (v_1^2, v_2^2), \ldots, (v_1^m, v_2^m)$ can be encoded as $\llcorner v_1^1 \lrcorner \# \llcorner v_2^1 \lrcorner \# \ldots \# \llcorner v_1^m \lrcorner \# \llcorner v_2^m \lrcorner$. Then the whole graph then becomes the encoding of: $(|V|, \llcorner E \lrcorner)$ where $\llcorner E \lrcorner$ is the encoding of the edges. Is sufficient to have only $|V|$ because the nodes are identified by all the natural numbers between 1 and $|V|$.

Exercise 4. Given

$$S = \{w \in \{0,1\}^* |\ |w| : n, n \geq 2, w \text{ begin and ends with } 1\}$$

what is the cardinality of S ($|S|$)?

Solution. We know that the cardinality of a set of String of length n with an alphabet with only two element is 2^n. If we know that the first and the last element of the string are always 1 then I have to subtract 2^2 so in the end the cardinality will be : $|S| = 2^{n-2}$

Exercise 5. Given

$$S = \{w \in \{0,1\}^* | \ |w| \leq n\}$$

what is the cardinality of S?

Solution. The cardinality of S will be the union of the cardinality of of the set $\{0,1\}^m$ with $m \leq n$. The cardinality of the union of two set is the sum of the cardinality of the sets:

$$|A \cup B| = |A| + |B|$$

so we will have that

$$\left| \bigcup_{m \leq n} \{0,1\}^m \right| = \sum_{m \leq n} |\{0,1\}^m| = \sum_{m=0}^{n} 2^m .$$

Exercise 6. Given

$$\mathcal{S} = \{w \in \{0,1\}^* : |w| = n, \ n \geq 0, \ w \text{ is palindrome}\}$$

find the cardinality of S.

Solution. We analyze two distinct cases, the first is when the length of the string is even and the second is when is odd.
In the even case we have that the string will be formed by two parts: $w = u \odot \tilde{u}$ where $\tilde{u}$ is the reversed version of string u (e.g. 0110). In this case if we call k the length of u then we have that $u \in \{0,1\}^k$ and $n = 2k$. So the cardinality of S in this case will be $|S| = 2^{n/2}$.
In the odd case we have that the string will be formed as: $w = u \odot 0 \odot \tilde{u}$ or $w = u \odot 1 \odot \tilde{u}$. So we will have to find the cardinality of the union of these two sets. Each of them has a cardinality of 2^k so if we sum them we obtain 2×2^k that is 2^{k+1} . But $n = 2k + 1$ so $k = \frac{n-1}{2}$ so the cardinality of S will be : $|S| = 2^{(n-1)/2+1} = 2^{(n+1)/2}$.

CHAPTER ONE

The Computational Model

Giving positive results about the feasibility of a computational task[3] is, in principle, straightforward: implement an algorithm solving the task on a machine powerful enough. However this reveals to be a shortsighted approach as one would not be able to prove the opposite, i.e. the impossibility to solve a certain task of a computational task by showing it is not feasible on the available concrete machines. We therefore want to develop a *model* of computation.

[3] We recall from Section 0.1 that for us a task is thought of as the problem of computing a function $f : \{0,1\}^* \to \{0,1\}$

Such model is defined in the form of an abstract machine (1) as simple as possible and (2) able to simulate all physically realistic machines. The universally accepted model of computation is the so-called "Turing Machine" originated from Alan Turing.

1.1 The Turing Machine

In order to give a formal definition of the Turing model of computation we need to lay out some concepts and notation.

The **scratchpad**(s) consist(s) of k *tapes*, where a tape is an infinite 1−directional line of cells, each holding a symbol from a finite alphabet Γ. We call Γ the *alphabet* of the machine. Each of the tapes has a head, i.e. a pointer to a cell or, mathematically, a natural number specifying how far from the left-most cell we currently are (at that stage of the computation). The head can read or write one symbol at a time from or to the tape. It can move left or right. Some tapes are read-only, the input tapes. The last tape(s) can be taken as the output tape(s) containing the result of the computation. If those tapes are absent, the result of the computation is either 0 or 1 based on the final state reached by the machine.

As for the **instructions**, we call Ω the *finite* set of states the machine can be in. A state provides the internal description of part of the machine. At each step the machine performs the following operations

1. Reads the symbols under the k tape heads;
2. For the $k-1$ read-write tapes, replaces the symbol with a new one or leaves it unchanged;

3. Changes its state to a new one;

4. Moves each of the k tape heads to the left/right/stay in place.

This is a very simple yet expressive model.

Note. The set of states $\mathfrak{Q}$ is different from the set of configurations which can be infinite (see Def. 1.2).

We now give the formal definition of a Turing Machine.

Definition 1.1 (Turing Machine)**.** A Turing Machine (TM) working on k tapes is described as a triple $(\Gamma, \mathfrak{Q}, \delta)$ containing

- A finite set Γ of tape symbols, which we assume contains the blank symbol $\square$, the start symbol $\triangleright$, and the binary digits 0 and 1;
- A finite set $\mathfrak{Q}$ of states which includes a designated initial state q_{init} and a designated final state q_{halt};
- A **transition function**[4] δ that defines the instructions regulating the functioning of the machine at each step:

$$\delta : \mathfrak{Q} \times \Gamma^k \to \mathfrak{Q} \times \Gamma^{k-1} \times \{L, S, R\}^k . \tag{1.1}$$

  where Γ^k specifies the symbols under the k heads, Γ^{k-1} indicates the next symbols under the writeable heads (input tape stays the same) and the tuple $\{L, S, R\}^k$ specifies the heads movements (left, right or stay in place).

Note. When the first parameter is the halting state q_{halt}, then δ cannot touch the tapes nor the heads:

$$(q_{halt}, (\sigma_1, \ldots, \sigma_k)) \overset{\delta}{\mapsto} (q_{halt}, (\sigma_2, \ldots, \sigma_k), (S, \ldots, S))$$

and the machine is stuck.

Definition 1.2 (Configuration of a TM)**.** Given a Turing Machine, $\mathcal{M} = (\Gamma, \mathfrak{Q}, \delta)$ working on k tapes, a configuration consists of

- The current state q;
- The contents of the k tapes;
- The positions of the k tape heads;

one such configuration will be denoted with C.

Definition 1.3 (Initial/Final configuration)**.** We indicate with $\mathfrak{I}_x$ the *initial* configuration for input $x \in \{0,1\}^*$ consisting of

- q_{init};
- first tape: $\triangleright x\ \square \ldots \square$, other tapes: $\triangleright \square \ldots \square$;
- tapes heads all positioned on the $\triangleright$ symbol.

A *final* configuration for output $y \in \{0,1\}^*$ is any configuration whose current state is q_{halt} and output tape $\triangleright y \square \ldots \square$.

1.1.1 Machine computations

The transition function naturally determines the unfolding of successive configurations.

Definition 1.4. We say that the TM $\mathcal{M}$ returns $y \in \{0,1\}^*$ on input $x \in \{0,1\}^*$ in t *steps* if

$$\mathfrak{I}_x \overset{\delta}{\mapsto} C_1 \overset{\delta}{\mapsto} \ldots \overset{\delta}{\mapsto} C_t$$

where C_t is a final configuration for y. We then write $y = \mathcal{M}(x)$.

Definition 1.5 (Computable function)**.** Let $\mathcal{M}$ be a Turing Machine. We say $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if and only if $\forall x \in \{0,1\}^* \mathcal{M}(x) = f(x)$. If that is the case, f is said to be *computable*.

1.1.2 Machine efficiency, runtime

Definition 1.6 (Computing a function and running time)**.** A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ in time $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input x in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$, in this case f is said to be **computable** in time T.

Definition 1.7. A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time T iff f is computable in time f.

Example 1.8. The set of palindrome words is decidable in time $T(n) = 3n$. Computing the parity of binary strings requires time $T(n) = n + 2$.

Definition 1.9 (Time-constructible function)**.** A function $f : \mathbb{N} \to \mathbb{N}$ is time constructible if $T(n) \geq n$ and there is a TM $\mathcal{M}$ that computes the function $x \mapsto \llcorner T(|x|) \lrcorner$ in time $T(n)$.

Remark 1.10 (Definition robustness)**.** The definition of the model given here

may be subject to many tweaks. It is a simple exercise to see that most changes to the definition do not yield a substantially different model, since our model can simulate any of these new models. In context of computational complexity, however, we have to verify not only that one model can simulate another,but also that it can do so efficiently. However as long as we are willing to ignore polynomial factors in the running time (which might be raised if restricting the alphabet,reducing to one tape or allowing the tapes to be infinite in both directions[5] we are just fine with the model 1.1.

[4] δ has a table representation. This can be huge for certain machines, nevertheless finite.

1.1.3 Machines as strings

Since the behavior of a TM $\mathcal{M}$ is determined by δ we may well use the list of all inputs and outputs of this function and encode those as as $\{0,1\}^*$effectively yielding a representation $\llcorner\mathcal{M}\lrcorner$.

[5] See Section 1.3.1. of Arora and Barak (2009) for proofs.

It is of technical usefulness[6] to make the following assumptions

- $\forall x \in \{0,1\}^*, \exists \mathcal{M} : x = \llcorner M \lrcorner$;
- Every TM $\mathcal{M}$ is represented by infinitely many strings; however one is the "canonical" representation which we indicate with $\llcorner\mathcal{M}\lrcorner$.

Theorem 1.11 (Efficient Universal Turing Machine)**.** There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*, \mathcal{U}(x,\alpha) = \mathcal{M}_\alpha(x)$ where $\mathcal{M}_\alpha$ denotes the TM represented by α.
Moreover if $\mathcal{M}_\alpha$ halts on input x within T steps then $\mathcal{U}(x,\alpha)$ halts withing $CT\log(T)$ steps, where C is independent of $|x|$ and depends only on $\mathcal{M}_\alpha$.

To give a general proof one has to encode configurations of Turing Machines as strings, and prove that $\mathcal{U}$ can simulate $\mathcal{M}_\alpha$ for every *alpha*.

1.2 Uncomputability

Theorem 1.12 (Uncomputable functions exist)**.** There exists a function $\mathtt{UC} : \{0,1\}^* \to \{0,1\}$ that is not computable by any Turing Machine.

Proof. It suffices to consider the function

$$\mathtt{UC}(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha = 1 \\ 1 & \text{otherwise} \end{cases} . \tag{1.2}$$

If $\mathtt{UC}$ were computable, there would be a TM $\mathcal{M}$ such that $\mathcal{M}(\alpha) = \mathtt{UC}(\alpha), \forall \alpha \in \{0,1\}^*$ according to Def. 1.5 and, in particular, when $\alpha = \llcorner\mathcal{M}_\alpha\lrcorner$. This, however, would be a contradiction since, by Equation 1.2

$$\mathtt{UC}(\llcorner\mathcal{M}\lrcorner) = 1 \Leftrightarrow \mathcal{M}(\llcorner\mathcal{M}\lrcorner) \neq 1 \Leftrightarrow \mathtt{UC}(\llcorner\mathcal{M}\lrcorner) = 0$$

□

1.2.1 The halting problem

Let's consider a more interesting function, the *halt* function

$$\texttt{HALT}(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{halts on input } x \\ 0 & \text{otherwise} \end{cases} \tag{1.3}$$

Being able to compute such a function would mean being able to check whether an algorithm terminates or not.

Theorem 1.13 (Uncomputability of HALT)**.** The halt function 1.3 is not computable by any Turing Machine.

Proof. Suppose, for the sake of contradiction, that there was a TM $\mathcal{M}_{\texttt{HALT}}$ computing HALT. We will use $\mathcal{M}_{\texttt{HALT}}$ to show a TM $\mathcal{M}_{\texttt{UC}}$ computing UC, contradicting Theorem 1.12.

The TM $\mathcal{M}_{\texttt{UC}}$ is simple: On input α, $\mathcal{M}_{\texttt{UC}}$ runs $\mathcal{M}_{\texttt{HALT}}(\alpha, \alpha)$. If the result is 0 (meaning that M_α does not halt on α), then $\mathcal{M}_{\texttt{UC}}$ outputs 1. Otherwise, $\mathcal{M}_{\texttt{UC}}$ uses the universal TM $\mathcal{U}$ to compute $b = \mathcal{M}_\alpha(\alpha)$. If $b = 1$, then $M_{\texttt{UC}}$ outputs 0; otherwise, it outputs 1.

Under the assumption that $\mathcal{M}_{\texttt{HALT}}(\alpha, \alpha)$ outputs $\texttt{HALT}(\alpha, \alpha)$ within a finite number of steps, the TM $\mathcal{M}_{\texttt{UC}}(\alpha)$ will output $\texttt{UC}(\alpha)$.

□

Remark 1.14 (Reduction technique)**.** The proof's technique to prove Theorem 1.13 is called *reduction*. We have shown that the computing UCis reducible to computing HALT. In other terms, we showed that if there were an algorithm computing HALT then there would be one for UC.

Remark 1.15. The result 1.13 can be seen as a way to reinterpret Gödel's first incompleteness theorem computationally (See Sec. 1.5.2 of Arora and Barak (2009)).

1.3 Semantic Languages

Definition 1.16 (Semantic language)**.** A language $\mathcal{L} \subseteq \{0,1\}^*$ is said to be *semantic* when

- $\forall \alpha \in \mathcal{L}, \exists \mathcal{M}$ TM such that $\alpha = \llcorner M \lrcorner$; and
- If $\llcorner \mathcal{M} \lrcorner \in \mathcal{L}$ and $\mathcal{N}$ is a TM computing the same function as $\mathcal{M}$ then $\llcorner \mathcal{N} \lrcorner \in \mathcal{L}$.

Remark 1.17. Semantic languages can be seen as extensional properties of programs, e.g. the set of all machines computing a certain function, the set of all terminating programs, ...

Definition 1.18 (Property)**.** If a language $\mathcal{L}$ belongs to a semantic language $\mathcal{P}$, we can say that $\mathcal{L}$ satisfies the property P.
P is said to be *non-trivial* if

1. $\mathcal{P} \neq \emptyset$; and
2. $\mathcal{P} \neq \{\llcorner \mathcal{M} \lrcorner | \mathcal{M} \text{ is a TM}\}$

P is said to be *extensional* if

$$\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{N}) \Rightarrow \llcorner \mathcal{M} \lrcorner \in P \iff \llcorner \mathcal{N} \lrcorner \in P\,.$$

We have the following result regarding semantic languages:

Theorem 1.19 (Rice's Theorem)**.** Any decidable semantic language $\mathcal{L}$ is trivial, i.e. either $\mathcal{L} = \emptyset$ or $\mathcal{L} = \{0,1\}^*$.

Remark 1.20. This is a generalization of the halting problem. It implies that all non-trivial and extensional properties of a language are undecidable.

To prove that a language is decidable one can

- Construct a TM computing the function or deciding the given language;
- Describe the algorithm, in an informal way, computing the function / deciding the language using only elementary operation. However evaluating the performance precisely is often hard.

To prove that a language $\mathcal{L}$ is *undecidable* one can show that there exists a *computable* mapping $\phi : \{0,1\}^* \to \{0,1\}^*$ such that

$$s \in \mathcal{G} \Leftrightarrow \phi(s) \in \mathcal{L}$$

where $\mathcal{G}$ is a language known to be undecidable. This way an hypothetical algorithm for deciding $\mathcal{L}$ would be turned into one for $\mathcal{G}$, which cannot exist. Alternatively one can prove $\mathcal{L}$ is semantic and use Theorem 1.19.

1.4 Complexity classes

Definition 1.21. A *Complexity class* is a set of tasks which can be computed within some prescribed resource bounds.

It is not a set of TMs, however its definition is based on TMs.

Definition 1.22 (**DTIME** class)**.** Let $T : \mathbb{N} \to \mathbb{N}$. A language $\mathcal{L}$ is in the class **DTIME**$(T(n))$ iff there is a TM deciding $\mathcal{L}$ and running in time $n \mapsto c \cdot T(n)$ for some constant c.

Remark 1.23. The D in **DTIME** stands for "deterministic". The class **DTIME**$(T(n))$ would define classes of problems that are not robust, i.e. they depend too much on the underlying model. Therefore the need for a larger class to study efficiently solvable tasks is clear.

1.5 Polynomial time computable problems

Now we can define a robust class of task that is called Class **P**.

Definition 1.24 (The class **P**)**.** The class **P** is defined as

$$P = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c) \tag{1.4}$$

The class **P** (Polynomial) includes all those languages $\mathcal{L}$ which can be decided by a TM working in polynomial time. Indeed for any polynomial $P(n)$

$$\exists c, d > 0, \bar{n} \in \mathbb{Z} \text{ s.t. } n > \bar{n} \Rightarrow P(n) \leq c \cdot n^d \,.$$

Remark 1.25. P is generally considered as the class of efficiently decidable languages.

Remark 1.26 (Robustness of **P**)**.** The class **P** is closed under most of operations performed on programs, e.g. composition, bounded loops, . . . As a consequence it is fairly easy to prove a problem to be in **P**: it is sufficient to provide an algorithm solving it working in polynomial time (without the need to construct the TM explicitly).

The exponents c bounding the running time of any machine deciding a language $\mathcal{L} \in \mathbf{P}$ can be be, in principle, huge. For most of problems of practical interest, however, c is relatively small.

1.6 The Church-Turing Thesis

The Church-Turing Thesis is an hypothesis that has near-universal acceptance (but it cannot be formally proven):

Claim 1.27 (Church–Turing Thesis)**.** Every physically realizable computer can be simulated by a TM with overhead in time. The class of computable tasks would not be larger (actually, equal!) if formalized in a realistic way, but different.

Remark 1.28. (Why the computation model does not matter) The Church–Turing statement suggests that every physically realizable computational device – whether it's based on silicon, DNA, neurons or some other alien technology – can be simulated by a Turing machine. This implies that the set of *computable* problems would be no larger on any other computational model that on the TM. Thus the underlying model of computation does not matter in this sense.

However when it comes to *efficiently* computable problems the situation is less clear. The stronger version of the statement 1.27 reads:

Claim 1.29 (Strong Church-Turing Thesis)**.** Every physically realizable computer can be simulated by a TM with a polynomial overhead in time. In this view, the class **P** would stay the same even with a different model of computation.

The statement 1.29 attributes a strong robustness to the class **P**. However it is more controversial due to possible developments in e.g. quantum computation.

1.7 The complexity class FP

Sometimes one would like to classify functions rather than languages. This can be done by slightly generalizing a couple of concepts we have previously introduced.

Definition 1.30 (**FDTIME** class)**.** A function f is said to belong to the class **FDTIME**$(T(n))$ iff exists a TM computing f and running in time $c \cdot T(n)$ for some constant c and with $T : \mathbb{N} \to \mathbb{N}$.

Definition 1.31 (**FP** class)**.** The class **FP** is defined as:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c) \tag{1.5}$$

Remark 1.32. **FP** is a kind of generalization of **P**. If a language $\mathcal{L} \in \mathbf{P}$ then the characteristic function f of $\mathcal{L}$ is trivially in **FP**. However the contrary is not necessarily true: if $f \in \mathbf{FP}$ does not imply $\mathcal{L}_f \in \mathbf{P}$. There are canonical ways of turning a function f into a language $\mathcal{L}_f$ based on which class of functions f belongs to, e.g. optimization function, ...

1.8 The class EXP

The next class of functions, beyond the polynomials, that have nice closure properties is the class of exponential functions.

Definition 1.33 (**EXP** class)**.** The class **EXP** is defined as

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \tag{1.6}$$

Definition 1.34 (**FEXP** class)**.** The class **FEXP** is defined as follows:

$$\mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c}) \tag{1.7}$$

Of course it holds that: $\mathbf{P} \subseteq \mathbf{EXP}$ and $\mathbf{FP} \subseteq \mathbf{FEXP}$.

CHAPTER TWO

NP and NP–completeness

Between **P** and **EXP**, one can define many other classes. This is useful to classify those problems in EXP for which we do not know whether they are in P.

2.1 The Complexity class NP

We now formalize the class of problems with an *efficiently verifiable solutions*, where the notion of efficiency is the one of polynomial time computability, as outlined in Chapter 1.

Definition 2.1 (The class **NP**)**.** A language $\mathcal{L} \subseteq \{0,1\}^*$ is in **NP**if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM $\mathcal{M}$ (called the *verifier* for $\mathcal{L}$) such that for every $x \in \{0,1\}^*$,

$$x \in \mathcal{L} \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } \mathcal{M}(x,u) = 1 \tag{2.1}$$

If $x \in \mathcal{L}$ and $u \in \{0,1\}^{p(|x|)}$ satisfy $\mathcal{M}(x,u) = 1$, then we call u a *certificate* for x (with respect to the language $\mathcal{L}$ and machine $\mathcal{M}$).

Remark 2.2. From Def. 2.1 we have that **P** $\subseteq$ **NP** since $p(|x|)$ can be 0, i.e. u can be ε.

Crafting a solution (i.e. looking for the appropriate y) for x can potentially be more difficult than just *checking* y to be a solution to x.

Note. Differently from **P** and **EXP**, the class **NP** does not have a natural counterpart as a class of functions.

2.1.1 Relation between NP and P

We have the following trivial relationships between **NP**and the classes **P**and **DTIME**$(T(n))$.

Theorem 2.3. **P**$\subseteq$ **NP**$\subseteq$ **EXP**.

Proof. (**P**$\subseteq$ **NP**): Suppose $\mathcal{L} \in$ **P** is decided in polynomial-time by a TM $\mathcal{N}$. Then $\mathcal{L} \in$ **NP**, since we can take $\mathcal{N}$ as the machine $\mathcal{M}$ in Definition 2.1 and make $p(x)$ the zero polynomial (in other words, u is an empty string).

(**NP**$\subseteq$ **EXP**): If $\mathcal{L} \in$ **NP** and $M, p()$ are as in Definition 2.1, then we can decide $\mathcal{L}$ in time $2^{O(p(n))}$ by enumerating all possible strings u and using $\mathcal{M}$ to check whether u is a valid certificate for the input x. The machine accepts iff such a u is ever found. Since $p(n) = O(n^c)$ for some $c > 1$, the number of choices for u is $2^{O(n^c)}$, and the running time of the machine is similar. □

Currently, we do not know of any stronger relation between **NP**and deterministic time classes than the trivial ones stated in Claim 2.4. The question whether or not **P**= **NP** is considered *the* central open question of complexity theory and is also an important question in mathematics and science at large. Most researchers believe that **P**$\neq$ **NP** since years of effort have failed to yield efficient algorithms for **NP**-complete problems.

2.1.2 Nondeterministic Turing machines

The class **NP** can also be defined using a variant of Turing machines called *nondeterministic* Turing machines (abbreviated NDTM). In fact, this was the original definition, and the reason for the name **NP**, which stands for *nondeterministic polynomial time*. The only difference between an NDTM and a standard TM (as defined in Def. 1.1) is that an NDTM has

- two transition functions δ_0 and δ_1; and
- has a special state denoted by q_{accept}.

When an NDTM $\mathcal{M}$ computes a function, we envision that at each computational step $\mathcal{M}$ makes an arbitrary choice as to which of its two transition functions to apply. For every input x, we say that $\mathcal{M}(x) = 1$ if there *exists* some sequence of these choices (which we call the *nondeterministic choices* of $\mathcal{M}$) that would make $\mathcal{M}$ reach q_{accept} on input x. Otherwise—if *every* sequence of choices makes $\mathcal{M}$ halt without reaching q_{accept}—then we say that $\mathcal{M}(x) = 0$. We say that $\mathcal{M}$ runs in $T(n)$ time if for every input $x \in \{0,1\}^*$ and every sequence of nondeterministic choices, $\mathcal{M}$ reaches either the halting state or q_{accept} within $T(|x|)$ steps.

Definition 2.4 (**NTIME** class)**.** For every function $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$, we say that $L \in$ **NTIME**$(T(n))$ if there is a constant $c > 0$ and a $c \cdot T(n)$-time NDTM $\mathcal{M}$ such that for every $x \in \{0,1\}^*$, $x \in L \Leftrightarrow \mathcal{M}(x) = 1$.

The next theorem gives an alternative characterization of **NP** as the set of languages computed by polynomial-time *nondeterministic* Turing machines.

Theorem 2.5. **NP** $= \bigcup_{c \in \mathbb{N}}$**NTIME**$(n^c)$.

Proof. The main idea is that the sequence of nondeterministic choices made by an accepting computation of an NDTM can be viewed as a certificate

that the input is in the language, and vice versa.

Suppose $p : \mathbb{N} \to \mathbb{N}$ is a polynomial and L is decided by a NDTM N that runs in time $p(n)$. For every $x \in L$, there is a sequence of nondeterministic choices that makes N reach q_{accept} on input x. We can use this sequence as a *certificate* for x. This certificate has length $p(|x|)$ and can be verified in polynomial time by a *deterministic* machine, which simulates the action of N using these nondeterministic choices and verifies that it would have entered q_{accept} after using these nondeterministic choices. Thus, $L \in \mathbf{NP}$ according to Definition 2.1.

Conversely, if $L \in \mathbf{NP}$ according to Definition 2.1, then we describe a polynomial-time NDTM N that decides L. On input x, it uses the ability to make nondeterministic choices to write down a string u of length $p(|x|)$. (Concretely, this can be done by having transition δ_0 correspond to writing a 0 on the tape and transition δ_1 correspond to writing a 1.) Then it runs the deterministic verifier $\mathcal{M}$ of Definition 2.1 to verify that u is a valid certificate for x, and if so, enters q_{accept}. Clearly, N enters q_{accept} on x if and only if a valid certificate exists for x. Since $p(n) = O(n^c)$ for some $c > 1$, we conclude that $L \in \mathbf{NTIME}(n^c)$. □

As is the case with deterministic TMs, NDTMs can be easily represented as strings, and there exists a *universal* nondeterministic Turing machine. In fact, using nondeterminism, we can even make the simulation by a universal TM slightly more efficient.

One should note that, unlike standard TMs, NDTMs are not intended to model any physically realizable computation device.

2.2 Reductions and NP–completeness

What can we conclude from the fact that a language $\mathcal{L}$ is in the class NP? We can only conclude that it is not too complicated to solve (not much). We need a way to establish a relation between two languages so we can know something about the relative difficulty of deciding them.

Definition 2.6 (Reductions, **NP**-hardness and **NP**-completeness)**.** A language $\mathcal{L} \subseteq \{0,1\}^*$ is *polynomial-time Karp reducible to a language* $\mathcal{L}' \subseteq \{0,1\}^*$ (sometimes shortened to just "polynomial-time reducible"), denoted by $\mathcal{L} \leq_p \mathcal{L}'$, if there is a *polynomial-time computable function* $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in \mathcal{L}$ if and only if $f(x) \in \mathcal{L}'$.

We say that $\mathcal{L}'$ is **NP**-hard if $\mathcal{L} \leq_p \mathcal{L}'$ for every $\mathcal{L} \in \mathbf{NP}$. We say that $\mathcal{L}'$ is **NP**-complete if $\mathcal{L}'$ is **NP**-hard and $\mathcal{L}' \in \mathbf{NP}$.

The important (and easy to verify) property of polynomial-time reducibility is that if $\mathcal{L} \leq_p \mathcal{L}'$ and $\mathcal{L}' \in \mathbf{P}$then $\mathcal{L} \in \mathbf{P}$—see Figures 2.1 and 2.2. This is why we say in this case that $\mathcal{L}'$ is *at least as hard as* $\mathcal{L}$, as far as polynomial-time algorithms are concerned. Note that $\leq_p$ is a *relation* among languages, and

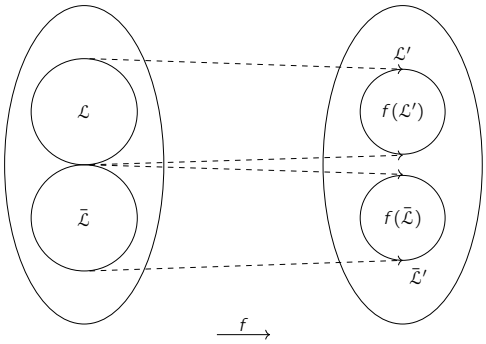


Figure 2.1

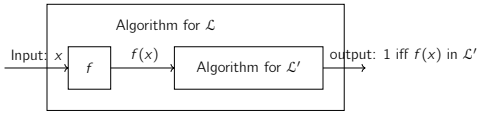


Figure 2.2

part 1 of Theorem 2.8 shows that this relation is *transitive*. Later we will define other notions of reduction, and many will satisfy transitivity. Part 2 of the theorem suggests the reason for the term **NP**-hard—namely, an **NP**-hard languages is *at least as hard* as any other **NP** language. Part 3 similarly suggests the reason for the term **NP**-complete: to study the **P** versus **NP** question it suffices to study whether any **NP**-complete problem can be decided in polynomial time.

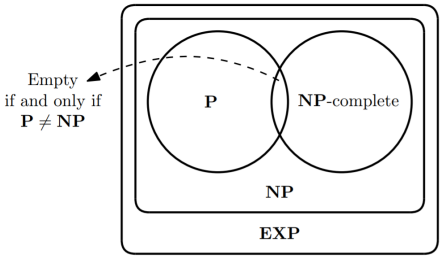


Figure 2.3: **P** $\subseteq$ **NP** $\subseteq$ **EXP**. It is not known whether **NP** and **P** are equal or different, same for **NP** and **EXP**. We only know that **P** $\neq$ **EXP**.

Theorem 2.7. The following is true:

1. $\leq_p$ is a pre-order, i.e. it is reflexive and transitive.
2. If language $\mathcal{L}$ is **NP**-hard and $\mathcal{L} \in$ **P**, then **P** = **NP**.
3. If language $\mathcal{L}$ is **NP**-complete, then $\mathcal{L} \in$ **P** if and only if **P**= **NP**.

Proof. The main observation underlying all three parts is that if p, q are two functions that grow at most as n^c and n^d, respectively, then composition $p(q(n))$ grows as at most n^{cd}, which is also polynomial. We now prove part 1 and leave the others as simple exercises.

If f_1 is a polynomial-time reduction from $\mathcal{L}$ to $\mathcal{L}'$ and f_2 is a reduction from $\mathcal{L}'$ to $\mathcal{L}''$, then the mapping $x \mapsto f_2(f_1(x))$ is a polynomial-time reduction from $\mathcal{L}$ to $\mathcal{L}''$ since $f_2(f_1(x))$ takes polynomial time to compute given x. Finally, $f_2(f_1(x)) \in \mathcal{L}''$ iff $f_1(x) \in \mathcal{L}'$, which holds iff $x \in \mathcal{L}$.

For the second point, think that if a language $\mathcal{L}$ is **NP**-hard it means that is harder than any other **NP** problem. But we know that $\mathcal{L}$ is in **P**so we know for sure that any other problem could also be solved in **P**. So **P** = **NP**.

For the third point:
($\Rightarrow$) If $\mathcal{L} \in$ **P** then any language $\mathcal{H}$ in **NP** is such that $\mathcal{H} \leq_p \mathcal{L}$, but since $\mathcal{L}$ is also in **P** it follows that $\mathcal{H}$ is also in **P**.
($\Leftarrow$) If **P** = **NP**, and we know that $\mathcal{L}$ is **NP**-complete (which means that it is **NP**-hard and also is **NP**) then $\mathcal{L} \in$ **NP** implies **P** = **NP**.

□

2.3 The Cook-Levin Theorem

Boolean formulas, CNF, and SAT

Some of the simplest examples of **NP**-complete problems come from propositional logic. A *Boolean formula* over the variables $u_1, \ldots, u_n$ consists of the variables and the logical operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). For example, $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$ is a Boolean formula. If φ is a Boolean formula over variables $u_1, \ldots, u_n$, and $z \in \{0,1\}^n$, then $\varphi(z)$ denotes the value of φ when the variables of φ are assigned the values z (where we identify 1 with True and 0 with False). *A formula φ is satisfiable if there exists some assignment z such that $\varphi(z)$ is* True. *Otherwise, we say that φ is unsatisfiable.*

The above formula $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$ is satisfiable, since the assignment $u_1 = 1, u_2 = 0, u_3 = 1$ satisfies it. In general, an assignment $u_1 = z_1, u_2 = z_2, u_3 = z_3$ satisfies the formula iff at least two of the z_i's are 1.

A Boolean formula over variables $u_1, \ldots, u_n$ is in *CNF form* (shorthand for *Conjunctive Normal Form*) if it is an AND of OR's of variables or their negations. For example, the following is a 3CNF formula: (here and elsewhere, $\overline{u_i}$ denotes $\neg u_i$)

$$(u_1 \vee \overline{u_2} \vee u_3) \wedge (u_2 \vee \overline{u_3} \vee u_4) \wedge (\overline{u_1} \vee u_3 \vee \overline{u_4})$$

More generally, a CNF formula has the form

$$\bigwedge_i \left(\bigvee_j v_{ij} \right)$$

where each v_{ij} is either a variable u_k or its negation $\overline{u_k}$. The terms v_{ij} are called the *literals* of the formula and the terms $(\bigvee_j v_{ij})$ are called its *clauses*. A kCNF is a CNF formula in which all clauses contain at most k literals. We denote by SATthe language of all satisfiable CNF formulae and by 3SAT the language of all satisfiable 3CNF formulae.

Theorem 2.8 (Cook-Levin)**.** The following two languages

$$\texttt{SAT} = \{\llcorner F \lrcorner | F \text{ is a satisfiable CNF}\}$$
$$\texttt{3SAT} = \{\llcorner F \lrcorner | F \text{ is a satisfiable 3CNF}\}$$

are **NP-complete**.

sketch. The steps for proving Theorem 2.8 are the following. Both SAT and 3SAT are clearly in **NP**, since a satisfying assignment can serve as the certificate that a formula is satisfiable. Thus we only need to prove that they are **NP**-hard. We do so by (a) proving that SAT is **NP**-hard and then (b) showing that SATis polynomial-time Karp reducible to 3SAT. This implies that 3SAT is **NP**-hard by the transitivity of polynomial-time reductions. See e.g. Section 2.3 of Arora and Barak (2009). □

2.4 Proving the hardness of a problem

The only way to prove that a problem is hard is to prove the language is **NP**-complete, in this way we have proven that the problem is not so hard (being in **NP**) but not so easy either (unless **P** = **NP** of course).

But how can we prove that a language $\mathcal{L}$ is **NP**-complete?

If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:

1. That $\mathcal{L}$ is in **NP**.

- This amounts to showing that there are p polynomial and $\mathcal{M}$ polytime TM such that $\mathcal{L}$ can be written as

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1\}$$

  which is typically rather easy.

2. That any other language $\mathcal{H} \in$ **NP** is such that $\mathcal{H} \leq_p \mathcal{L}$.
   - We can of course prove the statement directly.
   - More often (e.g. when showing 3SAT **NP**-complete), one rather proves that $\mathcal{J} \leq_p \mathcal{L}$ for a language $\mathcal{J}$ which is already known to be **NP**-complete.
   - This is correct, simply because $\leq_p$ is transitive (Theorem 2.7)

$$\vdots$$
$$\mathcal{H} \xrightarrow{\leq_p} \mathcal{J} \xrightarrow{\leq_p} \mathcal{L}$$
$$\vdots$$

2.5 Examples of NP-complete problems

2.5.1 INDSET

The **Independent Set** of a graph $\mathbb{G}$ with vertex set V and edges set E defined as follows

$$\texttt{INDSET} = \{\langle \mathbb{G}, k\rangle : \exists S \subseteq V \text{ s.t. } |S| \geq k \text{ and } \forall u, v \in S, (u,v) \notin E\}.$$

The Independent Set Problem is the problem of determining whether a graph $\mathbb{G}$ admits an independent set of size at least k.

input: $(\mathbb{G}, k)$
output: 1 iff $\exists S \subseteq V$ s.t. $|S| \geq k \wedge \forall v, u \in S\ (v,u) \notin E$

2.5.2 CLIQUE

$\texttt{CLIQUE} = \{\langle \mathbb{G}, k\rangle : \exists S \subseteq V \text{ s.t. } |S| \geq k \text{ and } \forall u, v \in S, u \neq v \Rightarrow (u,v) \in E\}$

input: $(\mathbb{G}, k)$
output: 1 iff $\mathbb{G}$ contains clique of size k
A clique is a subset W ⊆ V of its vertices such that any pair v, w ∈ W of distinct vertices is such that (v, w) ∈ E . In other words given an undirected graph a clique is a subset of this graph where all the vertices are connected, effectively yielding a complete subgraph.

2.5.3 kSAT

input: a formula F
output: 1 iff F is a satisfiable kCNF

2.5.4 Vertex Cover or Node Cover

input: $(\mathbb{G}, k)$
output: 1 iff exists a subset of the vertices $\exists C \subseteq V$ so that $|C| \leq b$ and $\forall (v, u) \in E . v \in C \ or \ u \in C$
It means that a node cover of a graph G is a part of the vertices for which we know that at least one vertices for each node is in the cover.

2.5.5 Universal Sink (US)

For having an universal sink we need to have a *directed* graph $\mathbb{G} = (V, E)$. Since we have a directed graph we do not require E to be symmetric. We can represent graphs using the so-called adjacency matrix. So we can write the graph as the pair $\mathbb{G} = (V, A)$ where A is a $x \times x$ matrix over $\{0, 1\}$. If we put a 1 in the position i, j then we have a edge between the vertex i and j.

input: $\mathbb{G}$
output: 1 iff exists a vertex v_i such that $\forall j, k \leq n$ with $j \neq i$ we have $A_{i,k} = 0$ and $A_{j,i} = 1$.
It means that a directed graph has universal sink, (at max 1) if it has a vertex so that all the nodes goes into the vertex and nothing comes out of it.

2.5.6 Set Packing (SP)

input: $(S_1, ..., S_n, k)$
output: 1 iff $\exists (P_1, P_2..., P_k) \in (S_1, ..., S_n)$ s.t. $\forall i, j \ i \neq j \ P_i \cap P_j = \emptyset$
It means that given a set of sets the problems gives 1 only if exists a subset of size k of this set so that every element (set) is independent from one another.

2.5.7 Subgraph Isomorphism (SI)

Given two undirected graphs $\mathbb{G}_1 = (V_1, E_1)$ and $\mathbb{G}_2 = (V_2, E_2)$ we say that:

1. $\mathbb{G}_1$ is a subgraph of $\mathbb{G}_2$ if and only if $V_1 \subseteq V_2$ and $E_1 = E_2 \cap (V_1 \times V_1)$
2. A function $h : V_1 \to V_2$ is a homomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ if and only if (v,u)$\in E_1$ implies $(h(v), h(u)) \in E_2$.
3. $\mathbb{G}_1$ is isomorphic to $\mathbb{G}_2$ if and only if there exists a bijective (function between the elements of two sets, where each element of one set is paired with exactly one element of the other set, and each element of the other set is paired with exactly one element of the first set.) homomorphism h from $\mathbb{G}_1$ to $\mathbb{G}_2$.

input: $\mathbb{G}_1 = (V_1, E_1)$ and $\mathbb{G}_2 = (V_2, E_2)$
output: 1 iff $\mathbb{G}_1$ has a subgraph isomorphic to $\mathbb{G}_2$.

2.5.8 SUBSETSUM (SSP)

In its most general formulation, there is a multiset S of integers and a target-sum T, and the question is to decide whether any subset of the integers sum to precisely T.

CHAPTER THREE

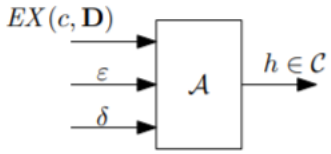
Elements of Computational Learning Theory

3.1 Terminology

- **Instance space** X $\rightarrow$ is the space from which we take the data. Is the set of instances of objects the learner wants to classify. The data from the instance space are generated through a distribution **D**, unknown to the learner.
- **Concepts** $\rightarrow$ it's a collection of objects subsets of the instance space X. This should be thought of as properties of objects.
- **Concepts class** $\rightarrow$ Collection of concepts
- **Target Concept** $\rightarrow$ it's the concept the learner wants to build a classifier for
- **Representation class** $\rightarrow$ we talk about representation class when we have a concept with non constant space to represent it, it requires size(e) bits with $e \in C$.

3.2 The General Model

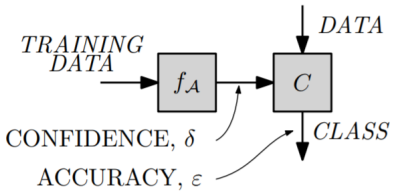
Every learning algorithm is designed to learn concepts from a concept class C but it does not know the target concept c, nor the associated distribution D. The interface of every learning algorithm can be described in the following way



- ε is the **error parameter**, while δ is the **confidence parameter**.
- EX(c,D) is called the *oracle*, a procedure that A can call as many times she wants, and which return an element x from the distribution D, labelled according to whether it is in c or not. Think of it as labeled data.

- The error of any $h \in C$ is defined as $error_{D,c} = Pr_{x \sim D}[h(x) \neq c(x)]$

So the general model will be:



1-δ is the probability for which we obtain a classifier with accuracy <ε .

3.3 PAC learnable and efficiently PAC learnable

PAC learnable is the equivalent of a function that can be computed, of a language that can be decided. A concept class is PAC learnable if there is an algorithm that can learn it. If the rime complexity of A is bounded by a polynomial in $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$ we say that C is efficiently PAC learnable. The complexity of A is measured taking into account the number of calls to the oracle.

3.4 The PAC learning model

Note. A Reference is given by Chapter 2 of Mohri et al. (2018)[7]

[7] Mehryar Mohri et al. (2018). *Foundations of Machine Learning*. 2nd ed. MIT Press

We first introduce several definitions and the notation needed to present the PAC model.

We denote by $\mathcal{X}$ the set of all possible examples or instances. $\mathcal{X}$ is also sometimes referred to as the **input space** or **instant space**. The set of all possible **labels** or target values is denoted by $\mathcal{Y}$. For the purpose of this introductory treatment, we will limit ourselves to the case where is reduced to two labels, $\mathcal{Y} = \{0, 1\}$, which corresponds to the so-called binary classification. The results can then be extended to generalized version of Y.

A **concept** $c : \mathcal{X} \to \mathcal{Y}$ is a mapping from $\mathcal{X}$ to $\mathcal{Y}$. Since $\mathcal{Y} = \{0, 1\}$, we can identify c with the subset of $\mathcal{X}$ over which it takes the value 1. Thus, in the following, we equivalently refer to a concept to learn as a mapping from $\mathcal{X}$ to $\{0, 1\}$, or as a subset of $\mathcal{X}$. As an example, a concept may be the set of points inside a triangle or the indicator function of these points. In such cases, we will say in short that the concept to learn is a triangle.

A **concept class** is a set of concepts we may wish to learn and is denoted by $\mathcal{C}$. This could, for example, be the set of all triangles in the plane. We assume that examples are independently and identically distributed (i.i.d.) according to some fixed but unknown distribution $\mathcal{D}$.

The learning problem is then formulated as follows. The learner considers a fixed set of possible concepts $\mathcal{H}$, called a **hypothesis set**, which might

not necessarily coincide with C. It receives a sample $S = (x_1, ..., x_m)$ drawn i.i.d. according to $\mathcal{D}$ as well as the labels $(c(x_1), ..., c(x_m))$, which are based on a specific target concept $c \in C$ to learn. The task is then to use the labeled sample S to select a hypothesis $h \in \mathcal{H}$ that has a small generalization error with respect to the concept c. The generalization error of a hypothesis $h \in \mathcal{H}$, also referred to as the risk or true error (or simply error) of h is denoted by $R(h)$ and defined as follows.

Definition 3.1 (Generalization error)**.** Given a hypothesis $h \in \mathcal{H}$, a target concept $c \in \mathcal{C}$, and an underlying distribution $\mathcal{D}$, the generalization error or risk of h is defined by

$$R(h) = \mathbb{P}_{x\sim\mathcal{D}}[h(x) \neq c(x)] = \mathbb{E}_{x\sim\mathcal{D}}[1_{h(x)\neq c(x)}], \tag{3.1}$$

where 1_ω is the indicator function of the event ω.

The generalization error of a hypothesis is not directly accessible to the learner since both the distribution $\mathcal{D}$ and the target concept care unknown. However, the learner can measure the empirical error of a hypothesis on the labeled sample S.

Definition 3.2 (Empirical error)**.** Given a hypothesis $h \in \mathcal{H}$, a target concept $c \in \mathcal{C}$, and a sample $S = (x_1, \ldots, x_m)$, the empirical error or empirical risk of h is defined by

$$\hat{R}_S(h) = \frac{1}{m}\sum_{i=1}^{m} 1_{h(x_i)\neq c(x_i)}. \tag{3.2}$$

Thus, the empirical error of $h \in \mathcal{H}$ is its average error over the sample S, while the generalization error is its expected error based on the distribution $\mathcal{D}$.

Definition 3.3 (PAC-learning)**.** A concept class $\mathcal{C}$ is said to be PAC-learnable if there exists an algorithm $\mathcal{A}$ and a polynomial function $poly(\cdot,\cdot,\cdot,\cdot)$ such that for any $\varepsilon > 0$ and $\delta > 0$, for all distributions $\mathcal{D}$ on $\mathcal{X}$ and for any target concept $c \in \mathcal{C}$, the following holds for any sample size $m \geq poly(1/\varepsilon, 1/\delta, n, size(c))$:

$$\mathbb{P}_{S\sim\mathcal{D}^m}[R(h_S) \leq \varepsilon] \geq 1 - \delta. \tag{3.3}$$

CHAPTER FOUR

Exercises

4.1 Turing Machines

Exercise 7. Define an efficient 1-tape Turing Machine computing the function inverse : $\{0,1\}^* \to \{0,1\}^*$ such that inverse(x) is the binary string obtained by flipping all bits in x, e.g. inverse(01011) is 10100. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Solution. The alphabet Γ can be defined as $\{\triangleright, \square, 0, 1\}$, while the set of states Q is $\{q_{\text{init}}, q_s, q_r\}$. The transition function is specified as follows:

$$
\begin{aligned}
(q_{\text{init}}, \triangleright) &\longmapsto (q_s, \triangleright, S) \\
(q_s, \triangleright) &\longmapsto (q_2, \triangleright, R) \\
(q_s, 0) &\longmapsto (q_s, 1, R) \\
(q_s, 1) &\longmapsto (q_s, 0, R) \\
(q_s, \square) &\longmapsto (q_r, \square, S) \\
(q_r, \square) &\longmapsto (q_r, \square, L) \\
(q_r, 0) &\longmapsto (q_r, 0, L) \\
(q_r, 1) &\longmapsto (q_r, 1, L) \\
(q_r, \triangleright) &\longmapsto (q_{\text{halt}}, \triangleright, S)
\end{aligned}
$$

In all the other cases (e.g. when the state is q_{init} and the symbol is not $\triangleright$, the behavior of the machine is not relevant, i.e., δ can be defined arbitrarily defined.

Exercise 8. Define an efficient 2-tape Turing Machine accepting only words $w \in \{0,1\}^*$ having the same number of 0's and 1's. For example, 011100 is accepted and 101 is rejected. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Solution. The general idea is that we write every 1 we encounter in the input tape into the working tape until we reach the end of the word. We then go back and move left on the working tape only when there is a matching 0 in the input tape.

We take $\Gamma = \{\triangleright, \square, 0, 1\}$ for the alphabet, $Q = \{q_{\text{init}}, q_{\text{halt}}, q_r, q_l\}$ for the set of states and the transition function $\delta : Q \times \Gamma^2 \to Q \times \Gamma \times \{L, S, R\}^2$ as follows:

$$
\begin{aligned}
(q_{\text{init}}, (\triangleright, \triangleright)) &\xmapsto{\delta} (q_r, \triangleright, (R, R)) \\
(q_r, (0, \square)) &\longmapsto (q_r, \square, (R, S)) \\
(q_r, (1, \square)) &\longmapsto (q_r, 1, (R, R)) \\
(q_r, (\square, \square)) &\longmapsto (q_l, \square, (L, L)) \\
(q_l, (0, 1)) &\longmapsto (q_l, \square, (L, L)) \\
(q_l, (1, 1)) &\longmapsto (q_l, 1, (L, S)) \\
(q_l, (\triangleright, \triangleright)) &\longmapsto (q_{\text{halt}}, \triangleright, (S, S)) \quad \checkmark \text{same number of 0's and 1's} \\
(q_l, (0, \triangleright)) &\longmapsto (q_l, \triangleright, (S, S)) \quad \times \text{more 0's, the TM is stuck} \\
(q_l, (\triangleright, 1)) &\longmapsto (q_l, 1, (S, S)) \quad \times \text{more 1's, the TM is stuck}
\end{aligned}
$$

Exercise 9. Construct a deterministic TM of the kind you prefer, which decides a language $\mathcal{L}$ containing all the $w \in \{0,1\}^*$ such that between any pair of occurrences of 0 in w there's an odd number of 1s. Study the complexity of TM you have defined.

Solution. We define a 1-tape Turing Machine with alphabet $\Gamma = \{\triangleright, \square, 0, 1\}$ and with a set of states $Q = \{q_{\text{init}}, q_1, q_2, q_3, q_4, q_{halt}\}$. The transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{S, L, R\}$ is specified as follows:

$$
\begin{aligned}
(q_{init}, \triangleright) &\xmapsto{\delta} (q_1, \triangleright, R) \\
(q_1, 1) &\longmapsto (q_1, 1, R) \\
(q_1, 0) &\longmapsto (q_2, 0, R) \\
(q_1, \square) &\longmapsto (q_{halt}, \square, S) \quad \checkmark\ w = 1^n \\
(q_2, 1) &\longmapsto (q_3, 1, R) \\
(q_2, 0) &\longmapsto (q_3, 0, S) \\
(q_2, \square) &\longmapsto (q_{halt}, \square, S) \quad \checkmark w = 1^k 0 \\
(q_3, 1) &\longmapsto (q_4, 1, S) \\
(q_3, \square) &\longmapsto (q_{halt}, \square, S) \\
(q_3, 0) &\longmapsto (q_2, 0, R) \quad \text{back to 1st 0 occurrence} \\
(q_4, 1) &\longmapsto (q_3, 1, S) \quad \text{even n of 1s} \\
(q_4, 0) &\longmapsto (q_{halt}, 0, S) \quad \times \\
(q_4, \square) &\longmapsto (q_{halt}, \square, S) \quad \checkmark
\end{aligned}
$$

The TM has to go through the tape only one time, so the complexity is linear $O(n)$.

4.2 Undecidability and Rice's theorem

Exercise 10. Prove the undecidability of the language

$$\mathcal{L}_E = \{\llcorner \mathcal{M} \lrcorner | \mathcal{M} \text{ is a TM and } \varepsilon \in \mathcal{L}(\mathcal{M})\}$$

by showing that (a) $\mathcal{L}_E$ is reducible to $\mathcal{L}_{halt}$ and (b) via Rice's Theorem.

Solution. As discussed in Section 1.3, we look for a computable mapping ϕ, which we define here as the function $\phi : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such

that

$$\phi(\alpha, x) = \varepsilon \Leftrightarrow \mathcal{M}_\alpha \text{ halts on input } x\,.$$

The TM corresponding to ϕ can be defined as follows: run $\mathcal{M}_\alpha$ on input x and erase the output tape so that ε is yielded.

If we assume $\mathcal{L}_E$ to be decidable, then there exists a Turing machine $\mathcal{M}_E$ such that for an input (α, x), it returns 1 if $\mathcal{M}_\alpha$ outputs ε on input x and returns 0 otherwise.
We now define a TM $\mathcal{M}_{halt}$ which for an input (α, x) returns 1 if $\mathcal{M}_E(\phi(\alpha), x)$ returns 1 and returns 0 otherwise. Therefore, $\mathcal{M}_{halt}(\alpha, x)$ returns 1 if and only if α halts on x.

We now show that $\mathcal{L}_E$ is undecidable by using Rice's theorem:

- $\mathcal{L}_E$ is not trivial: the "constant" Turing machine which always outputs 1 is not it $\mathcal{L}_E$ and the "constant" Turing machine which always outputs ε is in $\mathcal{L}_E$.
- $\mathcal{L}_E$ is semantic: for two Turing machines $\mathcal{M}$ and $\mathcal{N}$ with $\llcorner\mathcal{M}\lrcorner$ in $\mathcal{L}_E$ and for all input x, M and $\mathcal{N}$ have the same output, we must have $[N]$ in $\mathcal{L}_E$. If $\llcorner\mathcal{M}\lrcorner$ in $\mathcal{L}_E$, then it ouputs ε for some input x so by hypothesis, $\mathcal{N}$ must also output ε for x and therefore $\llcorner\mathcal{N}\lrcorner$ is in $\mathcal{L}_E$.

Therefore, we can apply Rice's theorem 1.19 and conclude that $\mathcal{L}_E$ is undecidable. □

4.3 Polynomial-time computable problems

How can we prove that an algorithm works in polynomial time? We do that in 4 steps:

1. We *encode* the input as a binary string. Our analysis of the complexity of the algorithm will be given with respect to the length (call it ℓ) of such a string.
2. We prove that the number of instructions of the algorithm is *bounded by a polynomial* in ℓ.
3. We argue that each instruction can be simulated by a Turing machine in polynomial time.
4. We show that all *'intermediate' data and results* of the algorithm are bounded by a polynomial in ℓ.

Exercise 11 (Linear search)**.** Given a list $A = [a_1, \ldots, a_n]$ such that $\forall i,\ a_i \in \mathbb{N}$ return an index $i \in \{1, \ldots, n\}$ such that $A[i] = v$, where $v \in \mathbb{N}$, if any, otherwise return -1. Input: list A and natural number v Output: natural number

Solution. One algorithm solving is Exercise 11 is Algorithm 1:

(*Input encoding*). In order to encode $A = [a_1, \ldots, a_n]$ as a binary string, we first need to encode its components a_i. For that, we can choose one standard encoding of the natural numbers in $\{0,1\}^*$. Let us write $\llcorner a_i \lrcorner$ for the encoding of a_i in binary. Since using n bits we can encode $2^n - 1$ (natural) numbers, the encoding of a number $a \in \mathbb{N}$ requires $\log a + 1$ bits[8].

[8] Actually, we should take the floor of $\log a$, $\lfloor \log a \rfloor$.

Algorithm 1: Linear Search

Data: $A = [a_1, \ldots, a_n]$, v
Result: An index $i \in \{1, \ldots, n\}$ s.t. $A[i] = v$, if any, -1 otherwise
$i \leftarrow 1$;
while $i \leq n$ **do**
  if $A[i] = v$ **then**
    return i;
  else
    $i \leftarrow i + 1$;
return -1;

Therefore, we have $|\llcorner a_i \lrcorner| = \log a_i + 1$. The next step is to understand how to encode the whole A. For that, we regard a list of elements $[b_1, \ldots, b_n]$ as a 'pair of pairs' of the form $(((b_1, b_2), b_3), \ldots b_n)$. Recall that given a pair (b_1, b_2) of bitstrings, we can define the string $\llcorner (b_1, b_2) \lrcorner \in \{0,1\}^*$ by first translating (b_1, b_2) as the string $b_1 \# b_2 \in \{0, 1, \#\}$ and then encoding $b_1 \# b_2$ as a string $\llcorner (b_1, b_2) \lrcorner \in \{0,1\}^*$. For the latter point, we simply map 0 to 00, 1 to 11, and # to 01. As a consequence, we see that $|\llcorner (b_1, b_2) \lrcorner| = 2|b_1| + 2|b_2| + 2$. Now, given a list $[b_1, \ldots, b_n]$ of bitstring by regarding it as a pair $(((b_1, b_2), b_3), \ldots b_n)$, we see that we obtain an encoding $\llcorner [b_1, \ldots, b_n] \lrcorner$ of length $\sum_{i=1}^{n} 2|b_i| + 2(n-1)$. Applying these general considerations to A (and recalling that $|\llcorner a_i \lrcorner| = \log a_i + 1$), we obtain:

$$\llcorner A \lrcorner = \llcorner [\llcorner a_1 \lrcorner, \ldots, \llcorner a_n \lrcorner] \lrcorner \qquad |\llcorner A \lrcorner| = \sum_{i=1}^{n} 2(\log a_i + 1) + 2(n-1)$$

Finally, we pair A with v (recall that both A and v are input of our algorithm), so $\llcorner (\llcorner A \lrcorner, \llcorner v \lrcorner) \lrcorner$ gives an encoding of the input of our algorithm in binary notation. Notice that

$$\ell = |\llcorner (\llcorner A \lrcorner, \llcorner v \lrcorner) \lrcorner| = 2 \left(\sum_{i=1}^{n} 2(\log a_i + 1) + 2(n-1) \right) + 2(\log v + 1) + 2.$$

We now move to step 2. The latter is straightforward. Our algorithm consists of:

- 1 assignment ($i \leftarrow 1$).
- n iteration of:
  - An inequality check ($i \leq n$).
  - A conditional branching performing:
    * An equality check ($A[i] = v$),
    * Either a return instruction or an assignment ($i \leftarrow i + 1$).
- 1 return instruction

Therefore, the number of the instruction is of the form $b + c \cdot n$, for suitable constants b, c, and thus it is bounded by a polynomial in ℓ.

(*TM simulation*). In order to prove step 3, we have to argue that all the aforementioned instructions can be simulated by a TM in polynomial time.

For instance, an equality check can be simulated as follows. Say we have two values a and b stored in different portions of a tape of a TM. In order to check whether a is equal to b, the machine simply moves back and forth between a and b checking whether they are bitwise equal. This can be done in polynomial time with respect to the length of a and b, provided that the 'distance' between a and b in the tape is itself bounded by a polynomial in the length of a and b. This will be indeed ensured by step 4. Similar arguments can be used to show that all other instructions can be simulated efficiently by a TM.

(*Intermediate data/results size*). Finally, in order to prove step 4 we simply observe that the only intermediate value computed by our algorithm is i, which can be at most n (and therefore it is bounded by a polynomial in ℓ).

Exercise 12 (Universal sink)**.** Recall that a *directed* graph is a pair $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is the 'edge' relation between vertices. Notice that we do not require E to be symmetric. We represent graphs using the so-called adjacency matrices. Formally, we regard graphs as pairs (V, A) where $V = \{v_1, \ldots, v_n\}$ is a set of vertices and A is an $n \times n$-matrix over $\{0, 1\}$. Intuitively, A encodes the edge relation according to the convention that $A_{i,j} = 1$ if and only if there is an edge from v_i to v_j. A *universal sink* is a vertex v_i such that for all $j, k \leq n$ with $j \neq i$ we have

$$A_{i,k} = 0 \qquad A_{j,i} = 1$$

Notice that if a graph has a universal sink, then the latter is unique. Show that determining whether a graph has a universal sink is computable in polynomial time.

Solution. Let $G = (V, A)$ be a graph with $V = \{v_1, \ldots, v_n\}$. We design Algorithm 2 to solve Exercise 12. The key observation is noticing that if v_i is a universal sink, then the i-th row in A contains only 0s, whereas the i-th column contains all 1s (except in the entry $A_{i,i}$). Graphically:

$$\begin{pmatrix} & & i & & & & \\ & & 1 & & & & \\ & & 1 & & & & \\ & & 1 & & & & \\ i & 0 & 0 & 0 & 0 & 0 & 0 \\ & & 1 & & & & \\ & & 1 & & & & \end{pmatrix}$$

Let us write $US(v)$ for the property "v is a universal sink". Then, we see that for all vertices $v_j, v_k \in V$ we have that

$$A_{j,k} = 1 \Rightarrow \neg US(v_j);$$
$$A_{j,k} = 0 \Rightarrow \neg US(v_k).$$

We can thus proceed as follows. Let POS be a list of potential universal sinks. Obviously, we begin assuming $POS = V$. We sequentially pick pairs of vertices (v_i, v_j) in POS and look at $A_{j,k}$. If $A_{j,k} = 1$, then we know that v_j cannot be universal sink, and thus we remove it from POS. Otherwise, $A_{j,k} = 0$ meaning that v_k cannot be universal sink, and thus we remove

it from POS. Proceeding this way, we will end up with POS containing a single vertex v_i. We then check whether v_i is universal sink by checking whether the i-th row of A contains 0s only, and whether the i-th column of A contains all 1s (except for $A_{i,i}$). If we succeed then v_i is a universal sink. Otherwise, there is no universal sink.

Algorithm 2: Universal Sink Detection

```
Data: V = [v_1, ..., v_n], A
Result: An index i ∈ {1, ..., n} s.t. US(v_i), if any, −1 otherwise
POS ← [1, ..., n];
// We use only labels of vertexes
i ← 1;
j ← 2;
while j ≤ n do
    if A_{i,j} = 1 then
        POS = POS.remove(i);
        i ← j;
        j ← j + 1;
    else
        POS = POS.remove(j);
        j ← j + 1;
i = POS.fst;
// We have POS = [i] for some i
// Check row
j ← 1;
while j ≤ n do
    if A_{i,j} = 0 then
        j ← j + 1;
    else
        return −1;
// Check column
j ← 1;
while j ≤ n do
    if A_{j,i} = 1 or j = i then
        j ← j + 1;
    else
        return −1;
return i;
```

Notice that we might have been more efficient in checking columns and rows. However, our implementation is closer to what we would do when programming a TM and makes our analysis easier.

We now show that the above algorithm works in polynomial time. First, the encoding of the input is standard. The input, in fact, consists of a natural number n (the number of vertices[1]) and of an $n \times n$-matrix of bits. We encode the latter as a list made of n elements each of which consisting of n bits. Therefore, such an encoding has length $2n^2 + n - 1$. Pairing the latter with the encoding of n (which has length $\log n$), we obtain an input of length

[1] Recall that we actually work with labels $1, \ldots, n$ of vertices, rather than with vertices themselves.

$2(2n^2 + n - 1) + 2\log n + 1$.

How many instructions our algorithm has? The first loop consists of $n - 1$ iterations, each of which essentially consists of assignments, checking values of the matrix, and removing elements from a list. After that, we have loops for checking row and columns, which consist of n iteration each (and thus a total of $2n$ iterations). Summing up, we have $3n - 1$ iterations, plus a fixed number of equality checking, assignments (plus arithmetic operations), and basic operations on lists. The algorithm thus has running time $O(n)$, and thus it is polynomial in $2(2n^2 + n - 1) + 2\log n + 1$. The last two points to prove in order to conclude that our algorithm runs in polynomial time are showing that each instructions can be simulated by a TM in polynomial time, and that all intermediate values/results have length polynomially bounded by the length of the input. The latter point is straightforward, whereas for the former we essentially proceed as in the solution of Exercise 11.

Exercise 13 (FP function)**.** Given two strings v, w, determine whether v is a substring of w, i.e., whether w contains an exact occurrence of v.

Algorithm 3: Substring Check

```
Data: Strings v, w
Result: True if v is substring of w, False otherwise
n ← size(w);
m ← size(v);
k ← 0;
for i ∈ 0..n − 1 do
    if v[k] = w[i] then
        k ← k + 1;
    else
        k ← 0;
return k = m;
```

Solution. Input Encoding: Let n be the number of distinct symbols in strings v, w. Each symbol requires $\log n + 1$ bits. Using alphabet $\Sigma = \{0, 1, \#, @\}$ encoded as $\{00, 01, 11, 10\}$, where:

- $\#$ separates characters
- @ separates strings

For $k = |v| + |w|$, total input size is:

$$I = 2k(\log n + 1) + 2k + 2$$

which is polynomial.

Basic Instructions Count: Algorithm contains:

- Three initial assignments
- Loop with n iterations containing:
  - Equality check
  - Assignment and increment or

  - Assignment to zero

- Final equality check

Total instructions: $a + bn = \mathcal{O}(n)$ where $n < l$, thus polynomial in input size.

Intermediate Results: Counter variables are bounded:

- k ranges from 0 to $|v|$
- i ranges from 0 to $|w|$

Basic Operations: All operations are polynomial-time on a Turing Machine:

- Assignment
- Increment
- Equality check

Exercise 14 (FP function)**.** Given a list $L = [L_1, \ldots, L_n]$, return its inverse $[L_n, L_{n-1}, \ldots, L_1]$.

Algorithm 4: List Inversion

```
Data: List L of length n
Result: Inverted list Inv_L
k ← size(L);
j ← 0;
for i ← k − 1 to 0 do
    Inv_L[j] ← L[i];
    j ← j + 1;
return Inv_L;
```

Solution. Input Encoding: For natural numbers L_i, using alphabet $\Sigma = \{0, 1, \#\}$ encoded as $\{00, 01, 11\}$:

$$\text{length} = 2n(\log L + 1) + 2n$$

where # separates elements.

Basic Instructions Count: Total instructions: $c + na = \mathcal{O}(n)$, where n is input size.

Basic Operations: All operations are polynomial-time on TM:

- Assignment
- Increment
- Equality check

Intermediate Results: All variables polynomially bounded:

- k: input size
- j: ranges from 0 to k
- Inv_L: same size as input

Exercise 15 (FP function)**.** Given two lists $L = [L_1, \ldots, L_n]$ and $P = [P_1, \ldots, P_n]$ of rational numbers, return their scalar product.

Algorithm 5: Scalar Product

```
Data: Lists L, P of rational numbers, length n
Result: Scalar product of L and P
n ← size(P);
dot ← 0;
for i ← 0 to n do
    dot ← dot + L[i] · P[i];
return dot;
```

Solution. Input Encoding: Using alphabet $\Sigma = \{+, -, 0, 1, \#, @, /\}$ requiring 3 bits, where:

- @ separates lists
- # separates numbers
- / separates numerator/denominator
- $+, -$ for signs

Rational numbers encoded as Sign num/den. Total length:

$$3 \cdot 2 \cdot 2n(\log n + 1) + 3 \cdot 2n + 3$$

Basic Instructions Count: Total instructions: $c + n$, polynomial in input length.

Basic Operations: All operations polynomial-time on TM:

- Assignment
- Addition
- Multiplication

Intermediate Results: dot is polynomially bounded by input size.

4.4 NP complete problems

Theorem 4.1. The following set is NP-complete:

$\text{CLIQUE} = \{(\mathcal{G}, k) \mid \mathcal{G} \text{ is an undirected graph containing a clique of size at least } k\}$

Proof. First, we define a clique as a subset $W \subseteq V$ of vertices such that any pair $v, w \in W$ of distinct vertices satisfies $\{v, w\} \in E$. In other words, a clique is a subset where all vertices are pairwise connected.

CLIQUE $\in$ **NP**: The certificate is a subset $W \subseteq V$ containing a k-clique because:

- $|W| \leq |\mathcal{G}|$
- Verifying W is a clique of size $\geq k$ takes quadratic time: check each pair $v, w \in W$ for edge connectivity

Reduction 3SAT $\leq_p$ CLIQUE: We construct f such that for every 3CNF formula F, $f(F) = (\mathcal{G}, k)$ where:

- Create a vertex for each literal occurrence in F
- For literals l_s^i and l_q^j where $i \neq j$ and $s, q \in \{1, 2, 3\}$: Add edge (l_s^i, l_q^j) iff literals are logically compatible (e.g., no edge between x and $\neg x$)
- Set k equal to number of clauses in F

Correctness: F is satisfiable iff $f(F)$ has a clique of size k (number of clauses in F)

($\Rightarrow$) If F is satisfiable:

- Some truth assignment makes all n clauses true
- For each clause $i \in \{1, \ldots, n\}$, $\exists s \in \{1, 2, 3\}$ where l_s^i is true
- These literals are pairwise consistent (no x and $\neg x$)
- Corresponding vertices form a clique of size n in $f(F)$

($\Leftarrow$) If $f(F)$ has n-clique:

- Each column (clause) contributes one vertex (no edges within columns)
- Chosen literals preserve logical consistency
- This selection yields a satisfying assignment for F
- At least one true literal per clause implies F is satisfiable

□

Exercise 16. Prove that the following language L is in NP, where:

$L = \{(G, k) \mid G = (V, E)$ is a graph whose nodes can be partitioned into $\leq k$ indepen

Equivalently:

$$L = \{(G, k) \mid \exists N : V \to \{1, \ldots, k\} \text{ s.t. } \forall (u, v) \in E, N[u] \neq N[v]\}$$

where $G = (V, E)$ is a graph and k is a natural number.

Solution. Certificate Structure: Let $N : V \to \{1, \ldots, k\}$ be the certificate, where $N[v]$ represents the assigned number (color) for vertex v.

Algorithm 6: Graph Coloring Verifier

```
Data: Graph G = (V, E), number k, assignment N
Result: True if N is valid k-coloring, False otherwise
for (u, v) ∈ E do
    if N[u] = N[v] then
        return false;
return true;
```

Certificate Size:

- N contains $|V|$ numbers

- Each number is $\leq k$, requiring $\log k$ bits
- Total size: $\mathcal{O}(|V| \log k)$, polynomial in input size

Verifier Complexity:

- Checks each edge once
- Time complexity: $\mathcal{O}(|E|) = \mathcal{O}(|V|^2)$
- Polynomial in input size

Therefore, $L \in$ **NP**.

Exercise 17 (1SAT)**.** Consider the following problem:

$$\texttt{1SAT} = \{\llcorner A \lrcorner | A \text{is a satisfiable 1CNF}\}$$

To which complexity class does 1SAT belong? Prove your claim.

Solution. 1SAT belongs to the complexity class **P**.

Algorithm 7: 1SAT

```
Data: Matrix A containing literals and their assignments
Result: True if formula is satisfiable, False otherwise
n ← size(A)[0];
for i ∈ 0..n do
    for j ∈ 0..n do
        if A[i][0] = A[j][0] ∧ A[i][1] ≠ A[j][1] then
            return false;
return true;
```

The input can be encoded as a list of literals, which may be negated. The encoding requires:

- Symbol to separate literals
- Symbol to denote negation

Using alphabet $\Sigma = \{0, 1, \#, @\}$ encoded as $\{00, 01, 10, 11\}$, where:

- k is the number of different literals
- n is the total number of literals
- $w < n$ is the number of negated literals

The total length of the input encoding is:

$$l = 2(n(\log k + 1) + n + w)$$

which is polynomial in the length of the input.

The number of basic instructions is:

$$c + b \cdot n^2$$

which is polynomially bounded with respect to the length of the encoding of the input.

Every basic instruction can be modeled by a Turing Machine that works in polynomial time. In particular, equality checking can be performed in polynomial time.

The partial results (counter i) are natural numbers ranging from 0 to the input size, thus polynomially bounded by the length of the input encoding.

Therefore, 1SAT $\in$ **P**.

Exercise 18 (ONEINDSET)**.** Given INDSET, classify its subset ONEINDSET whose elements are pairs (G, k) where:

- G is an undirected graph where each node is in at most one edge
- k is a natural number

Solution. Formal Definition:

$$\texttt{ONEINDSET} = \{(G,k) \mid \exists I \subseteq V : |I| \geq k \wedge \forall u, v \in I : (u,v) \notin E$$

$$\wedge \forall u, v, w \in V : (u,v) \in E \Rightarrow (u,w) \notin E \text{ for } v \neq w\}$$

> **Theorem 4.2.** Let $G = (V, E)$ be an undirected graph. If every $v \in V$ is contained in at most one edge, then G has an independent set of size $|I| = |V| - |E|$.

Proof. For each edge $(u, v) \in E$:

- Neither u nor v can be in any other edge by assumption
- Each edge eliminates exactly one vertex from potential independent set
- After excluding one vertex per edge, remaining vertices form independent set
- Size of independent set is $|V| - |E|$

□

Complexity Classification: ONEINDSET $\in$ P because:

- Can compute $|V| - |E|$ in polynomial time
- Compare with input k
- Return true iff $|V| - |E| \geq k$

Exercise 19 (3CLIQUE)**.** We studied the problem CLIQUE. You are required to classify the subset 3CLIQUE of CLIQUEconsisting of all the pairs $(G, 3)$. To which class does 3CLIQUE belong?

Solution. THREECLIQUE belongs to the complexity class P.

The input graph $G = (V, E)$ has V encoded as a natural number and E as pairs of natural numbers, with separator symbol # between natural numbers. Using alphabet $\Sigma = \{0, 1, \#\}$ encoded as $\{00, 01, 11\}$, the total bits required are:

$$I = 2(2 \cdot |E|(\log n + 1) + 2|E| + (\log |V| + 1) + 1) = \mathcal{O}(|E|) = \mathcal{O}(|V|^2)$$

Algorithm 8: 3CLIQUE

```
Data: Graph G = (V, E)
Result: True if graph contains a 3-clique, False otherwise
for i ∈ V do
    for j ∈ V do
        for k ∈ V do
            if (i, j) ∈ E ∧ (i, k) ∈ E ∧ (j, k) ∈ E then
                return true;
return false;
```

With three nested loops iterating over V, the number of basic instructions is:

$$c + b|V|^3 = \mathcal{O}(|V|^3)$$

which is polynomial in the length of the input encoding.

The most complex basic operation (checking if a pair is connected by an arc) runs in:

$$\mathcal{O}(|E|) = \mathcal{O}(|V|^2) = \mathcal{O}(I)$$

which is polynomial with respect to the input encoding size.

Variables i, j, k are natural numbers with maximum size $\log|V| + 1$ bits, which is polynomially bounded with respect to the input encoding size.

Therefore, THREECLIQUE $\in$ P.

Exercise 20 (kSAT4)**.** Consider the following problem:

$$\texttt{4SAT} = \{\llcorner A \lrcorner | A \text{is a satisfiable 4CNF}\}$$

To which complexity class does 1SAT belong? Prove your claim.

to which complexity class does 4SAT belong? Prove your claim.

Solution. 4SAT is **NP**-complete. We prove this in two steps:

1. 4SAT $\in$ **NP**

We have 4SAT = $\{A \mid \exists P$ such that $A(P) = \text{true}\}$, where P is the certificate consisting of a truth assignment for each variable. The length of P is polynomial with respect to the length of A. The verifying Turing Machine needs only to substitute the truth values in A, which can be done in polynomial time. Therefore, 4SAT $\in$ NP.

2. 3SAT $\leq_p$ 4SAT

Problem definitions:

3SAT:

- Input: Formula A
- Output: 1 iff A is satisfiable and in 3CNF

4SAT:

- Input: Formula A

- Output: 1 iff A is satisfiable and in 4CNF

The reduction $\texttt{3SAT} \leq_p \texttt{4SAT}$ means that $A \in \texttt{3SAT}$ iff $M(A) \in \texttt{4SAT}$.

Reduction: For each clause of the form $(A \vee B \vee C)$, create two clauses:

$$(A \vee B \vee C \vee D) \wedge (A \vee B \vee C \vee \neg D)$$

where D is a new variable. This makes D's truth value irrelevant to the satisfiability of the 4CNF formula. The reduction runs in polynomial time as we only add one new variable per clause.

Correctness:

($\Rightarrow$) If $A \in \texttt{3SAT}$: Let $(A \vee B \vee C)$ be satisfiable. Then both $(A \vee B \vee C \vee D)$ and $(A \vee B \vee C \vee \neg D)$ are satisfiable by construction, as D's value is irrelevant.

($\Leftarrow$) If $M(A) \in \texttt{4SAT}$: If $(A \vee B \vee C \vee D) \wedge (A \vee B \vee C \vee \neg D)$ is satisfiable, since we have both D and $\neg D$ in different clauses, their values are irrelevant. Therefore, $(A \vee B \vee C)$ must be satisfiable in 3SAT.

Since 3SAT is NP-complete and we have shown both $\texttt{4SAT} \in$ NP and $\texttt{3SAT} \leq_p \texttt{4SAT}$, we conclude that 4SAT is NP-complete.

Exercise 21 (2SAT)**.** Consider the following problem: 2SAT = A|A is a satisfiable 2CNF To which complexity class does 2SAT belong? Prove your claim.

Solution. This is a quite complex problem to solve, but it has been solve and 2SAT belong to P. The dimonstration is based on the fact that I can rewrite (A or B) as ((notA => B) and (notB => A))

Exercise 22 (NODECOVER)**.** see exercise sheet #TODO place it here

Exercise 23 (SET PACKING)**.** see exercise sheet #TODO place it here

Bibliography

Arora, Sanjeev and Boaz Barak (2009). *Computational Complexity: A Modern Approach*. Cambridge: Cambridge University Press.

Mohri, Mehryar, Afshin Rostamizadeh, and Ameet Talwalkar (2018). *Foundations of Machine Learning*. 2nd ed. MIT Press.