Computability and Computational Complexity
Academic year 2023–2024, first semester
Lecture notes

Mauro Brunato

Version: 2024-01-06

Caveat Lector

These very schematic lecture notes have been drafted during the 2018-2019 and 2019-2020 editions of the course. They were adjusted and updated during the Fall 2023 Semester.
Their main purpose is to keep track of what is being said during the lectures.
Reading this document is not enough to pass the exam.
You should also see the linked citations and footnotes, the additional material and the references provided on the following webpage, which will also contain the up-to-date version of these notes:

`https://comp3.eu/`

To check for new version of this document, please compare the version date on the title page to the one reported on the webpage.

Changelog

This log (in reverse temporal order, most recent change first) highlights the differences between the current edition of the notes and the previous ones.

2024-01-06

- Improved the answer to Exercise 17.

2024-01-04

- More errors pointed out by students.
  - Section 5.2 kept referring to paths instead of cycles;
  - Lots and lots of confusing typos.

2024-01-01

- Fixed mistakes (thanks to a student for pointing them out).
  - Table 4.1: inverted meaning of "false negatives" and "false positives" in the **RP** row;
  - section 4.3.4: corrected "$\mathbf{P} \setminus \mathbf{BPP}$" to "$\mathbf{BPP} \setminus \mathbf{P}$";
  - statement of Theorem 40.

2023-12-28

- Added Section 4.1.1 about the "restricted" halting problem and its **EXP** completeness.

2023-12-19

- Created Part II to collect topics that were discussed in the previous years but are not in the current edition's syllabus.

2023-12-18

- Added exercises (23 and following) from later written exams.

2023-12-16

- Removed the theorem "$\mathbf{NEXP} \neq \mathbf{EXP} \Rightarrow \mathbf{NP} \neq \mathbf{P}$" from Section 4.1.
- Corrected the probabilistic time class diagram in Fig. 4.2 (in the previous version, **PP** did not include **NP**), added $\mathbf{PP} \supseteq \mathbf{NP}$ as a theorem.
- Added Section 4.3.4 on quantum computing.

2023-11-23

- Expanded Section3.7 about **coNP** according to what has been said in the last lesson:
  - definition of the TAUTOLOGY language;
  - definition of universal-mode NDTMs (as opposed to existential-mode);
  - properties of the FACTORING language.

2023-10-26

- Added Section 2.3 with some final thoughts about the computability part of the course:
  - actual Turing machines that cannot be proved to halt;
  - infinite-state Turing machines;
  - Oblivious Turing machines.

2023-09-05

- Initial version from the 2019-2020 course.

Contents

Part I

Lecture notes

Chapter 1

Computability

1.1 Basic definitions and examples

In computer science, every problem instance can be represented by a finite sequence of symbols from a finite alphabet, or equivalently as a natural number. In the following, let Σ denote a finite set of *symbols*. Σ will be the *alphabet* we are going to use to represent things. Pairs, triplets, n-tuples of symbols are represented by the usual cartesian product notations:

$$\Sigma^2 = \Sigma \times \Sigma = \{(s,t)|s,t \in \Sigma\}, \quad \Sigma^3 = \Sigma \times \Sigma \times \Sigma, \dots, \Sigma^n = \overbrace{\Sigma \times \Sigma \times \cdots \times \Sigma}^{n \text{ times}}$$

As a shorthand, instead of representing tuples of symbols in the formal notation $(s_1, s_2, \dots, s_n)$ we will use the simpler "string" notation $s_1 s_2 \cdots s_n$. As a particular case, let $\varepsilon = ()$ represent the empty tuple (with $n = 0$ elements). Therefore, the set of strings of length n can be defined by induction:

$$\Sigma^n = \begin{cases} \{\varepsilon\} & \text{if } n = 0 \\ \Sigma \times \Sigma^{n-1} & \text{if } n > 0. \end{cases}$$

Finally, the Kleene closure of this sequence is the set of all *finite* strings on the alphabet Σ:

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n.$$

It is worthwile to note that Σ^*, while being infinite in itself, only contains *finite* sequences of symbols. Moreover, for every $n \in \mathbb{N}$, Σ^n is finite ($|\Sigma^n| = |\Sigma|^n$, where $| \cdot \|$ represents the cardinality of a set).

Our main focus will be on functions that map input strings to output strings on a given alphabet,

$$f : \Sigma^* \to \Sigma^*,$$

or in functions that map strings onto a "yes"/"no" decision set,

$$f : \Sigma^* \to \{0, 1\};$$

in such case, we talk about a *decision problem*.

Examples:

- Given a natural number n, is n prime?
- Given a graph, what is the maximum degree of its nodes?
- From a customer database, select the customers that are more than fifty years old.

- Given a set of pieces of furniture and a set of trucks, can we accommodate all the furniture in the trucks?

As long as the function's domain and codomain are finite, they can be represented as sequences of symbols, hence of bits, therefore as strings (although some representations make more sense than others); observe that some problems among those listed are decision problems, others aren't.

Decision functions and sets

There is a one-to-one correspondence between decision functions on Σ^* and subsets of Σ^*. Given $f : \Sigma^* \to \{0,1\}$, its obvious set counterpart is the subset of strings for which the function answers 1:

$$S_f = \{s \in \Sigma^* : f(s) = 1\}.$$

Conversely, given a string subset $S \subseteq \Sigma^*$, we can always define the function that decides over elements of the set:

$$f_S(s) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{if } s \notin S. \end{cases}$$

Given a function, or equivalently a set, we say that it is **computable**[1] (or **decidable**, or **recursive**) if and only if a procedure can be described to compute the function's outcome in a finite number of steps. Observe that, in order for this definition to make sense, we need to define what an acceptable "procedure" is; for the time being, let us intuitively consider any computer algorithm.

Examples of computable functions and sets are the following:

- the set of even numbers;
- a function that decides whether a number is prime or not;
- any finite or cofinite[2] set, and any function that decides on them;
- any function studied in a basic Algorithms course (sorting, hashing, spanning trees on graphs. . .).

We will see that the set of "computer programs" is too small with respect to the set of all decision functions; therefore, for some functions there is no program able to compute them. In the first part of this course we will first define what is a "computer program" and we will proceed to identify a non-computable decision function.

1.1.1 A set which we are (currently) unable to enumerate

In order to build some more intuition, let's start with a very simple problem that has been baffling mathematicians for centuries.

Collatz sequences

Given $n \in \mathbb{N} \setminus \{0\}$, let the *Collatz sequence* starting from n be defined as follows:

$$\begin{aligned} a_1 &= n \\ a_{i+1} &= \begin{cases} a_i/2 & \text{if } a_i \text{ is even} \\ 3a_i + 1 & \text{if } a_i \text{ is odd,} \end{cases} \qquad i = 1, 2, \dots \end{aligned}$$

In other words, starting from n, we repeatedly halve it while it is even, and multiply it by 3 and add 1 if it is odd.

[1] `https://en.wikipedia.org/wiki/Recursive_set`

[2] A set is *cofinite* when its complement is finite.

```
function collatz (n ∈ ℕ \ {0}) ∈ {0,1}
    repeat
        if n = 1 then return 1
        if n is even
            then n ← n/2
            else n ← 3n + 1
    return 0
```

```
function collatz (n ∈ ℕ \ {0}) ∈ {0,1}
    return 1
```

Figure 1.1: Left: the only way I know to decide whether n is a Collatz number isn't guaranteed to work. Right: a much better way, but it is correct if and only if the conjecture is true.

The *Collatz conjecture*[3] states that every Collatz sequence eventually reaches the value 1. While most mathematicians believe it to be true, nobody has been able to prove it.

Suppose that we are asked the following question:

> "Given $n \in \mathbb{N} \setminus \{0\}$, does the Collatz sequence starting from n reach 1?"

If the answer is "yes," let us call n a *Collatz number*. Let $f : \mathbb{N} \setminus \{0\} \to \{0,1\}$ be the corresponding decision function:

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is a Collatz number} \\ 0 & \text{if } n \text{ is not a Collatz number,} \end{cases} \qquad n = 1, 2, \dots$$

Then the Collatz conjecture simply states that all positive integers are Collatz numbers or, equivalently, that $f(n) = 1$ on its whole domain.

Decidability of the Collatz property

Let us consider writing a function, in any programming language, to answer the above question, i.e., a function that returns 1 if and only if its argument is a Collatz number. Figure1.1 details two possible ways to do it, and both have problems: the rightmost one requires us to have faith in an unproven mathematical conjecture; the left one only halts when the answer is 1 (the final **return** is never reached).

In more formal terms, we are admitting that we are **not** able to prove that the Collatz property is *decidable* (i.e., that there is a computer program that always terminates with the correct answer[4]). However, we have provided a procedure that terminates with the correct answer when the answer is "yes" (the function is not *total*, in the sense that it doesn't always provide an answer). We call such set **recursively enumerable**[5] (or RE, in short).

Having a procedure that only terminates when the answer is "yes" might not seem much, but it actually allows us to enumerate all numbers having the property. The function in Fig. 1.2 shows the basic trick to enumerate a potentially non-recursive set, applied to the Collatz sequence: the **diagonal method**[6]. Rather than performing the whole decision function on a number at a time (which would expose us to the risk of an endless loop), we start by executing the first step of the decision function for the first input ($n = 1$), then we perform the second step for $n = 1$ and the first step of $n = 2$; at the i-th iteration, we perform the i-th step of the first input, the $(i-1)$-th for the second, down to the first step for the i-th input. This way, every Collatz number will eventually hit 1 and be printed out.

The naïf approach of following the table rows is not guaranteed to work, since it would loop indefinitely, should a non-Collatz number ever exist.

Observe that the procedure does not print out the numbers in increasing order.

---

[3]`https://en.wikipedia.org/wiki/Collatz_conjecture`
[4]To the best of my knowledge, which isn't much.
[5]`https://en.wikipedia.org/wiki/Recursively_enumerable_set`
[6]See `https://comp3.eu/collatz.py` for a Python version.

```
procedure enumerate_collatz
    queue ← []
    for n ← 1 ... ∞                                   Repeat for all numbers
        queue_n ← n                                   Add n to queue with itself as starting value
        for i ← 1 ... n:                              Iterate on all numbers up to n
            if queue_i = 1                            i is Collatz, print and forget it
                print i
                delete queue_i                        deleted means "Already taken care of"
            else if queue_i is not deleted            if current number wasn't printed and forgotten yet
                if queue_i is even                    Advance i-th sequence in the queue by one step
                    then queue_i ← queue_i / 2
                    else queue_i ← 3 · queue_i + 1
```

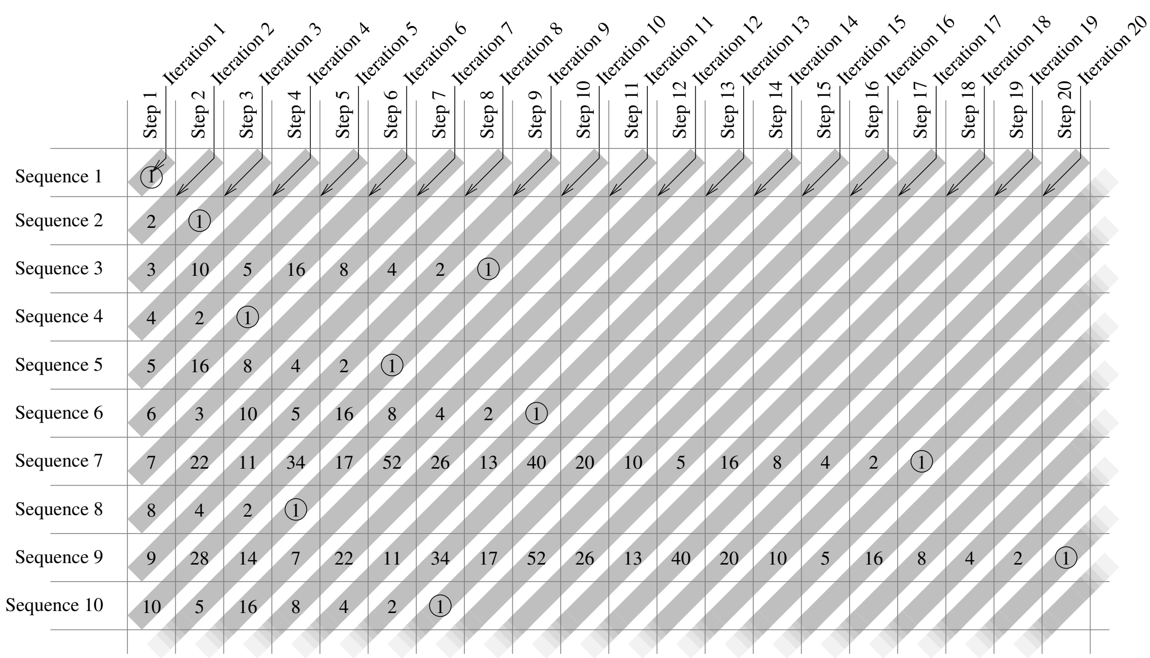


Figure 1.2: Enumerating all Collatz numbers: top: the algorithm; bottom: a working schematic

1.2 A computational model: the Turing machine

Among the many formal definition of computation proposed since the 1930s, the Turing Machine (TM for short) is by far the most similar to our intuitive notion. A Turing Machine[7] is defined by:

- a finite alphabet Σ, with a distinguished "default" symbol (e.g., "␣" or "0") whose symbols are to be read and written on an infinitely extended tape divided into cells;
- a finite set of states Q, with a distinguished initial state and one or more distinguished halting states;
- a set of rules R, described by a (possibly partial) function that associates to a pair of symbol and state a new pair of symbol and state plus a direction:

$$R : Q \times \Sigma \to \Sigma \times Q \times \{L, R\}.$$

Initially, all cells contain the default symbol, with the exception of a finite number; the non-blank portion of the tape represent the *input* of the TM. The machine also maintains a *current position* on the tape. The machine has an initial state $q_0 \in Q$. At every step, if the machine is in state $q \in Q$, and the symbol $\sigma \in \Sigma$ appears in the current position of the tape, the machine applies the rule set R to (q, σ):

$$(\sigma', q', d) = R(q, \sigma).$$

The machine writes the symbol σ' on the current tape cell, enters state q', and moves the current position by one cell in direction d. If the machine enters one of the distinguished halting states, then the computation ends. At this point, the content of the (non-blank portion of) the tape represents the computation's *output*.

Observe that the input size for a TM is unambiguously defined: the size of the portion of tape that contains non-default symbols. Also the "execution time" is well understood: it is the number of steps before halting. Therefore, when we say that the computational complexity of a TM for inputs of size n is $T(n)$ then we mean that $T(n)$ is the worst-case number of steps that a TM performs before halting when the input has size n.

1.2.1 Examples

In order to experiment with Turing machines, many web-based simulators are available. The two top search results for "turing machine demo" are

- `http://morphett.info/turing/turing.html`
- `https://turingmachinesimulator.com/`.

Students are invited to read the simplest examples and to try implementing a TM for some simple problem (e.g., some arithmetic or logical operation on binary or unary numbers). Also, see the examples provided in the course web page.

1.2.2 Computational power of the Turing Machine

With reference to more standard computational models, such as the Von Neumann architecture of all modern computers, the TM seems very limited; for instance, it lacks any random-access capability. The next part of this course is precisely meant to convince ourselves that a TM is exactly as powerful as any other (theoretical) computational device. To this aim, let us discuss some possible generalizations.

[7] `https://en.wikipedia.org/wiki/Turing_machine`

Multiple-tape Turing machines

A k-tape Turing machine is a straightforward generalization of the basic model, with the following variations:

- the machine has k unlimited tapes, each with an independent current position;
- the rule set of the machine takes into account k symbols (one for each tape, from the current position) both in reading and in writing, and k movement directions (each current position is independent), with the additional provision of a "stay" direction S in which the position does not move:
$$R : Q \times \Sigma^k \to \Sigma^k \times Q \times \{L, R, S\}^k.$$

Multiple-tape TMs are obviously more practical for many problems. For example, try following the execution of the binary addition algorithms below:

- 1-tape addition from `http://morphett.info/turing/turing.html`: select "Load an example program/Binary addition";
- 3-tape addition from `https://turingmachinesimulator.com/`: select "Examples/3 tapes/Binary addition".

However, it turns out that any k-tape Turing machine can be "simulated" by a 1-tape TM, in the sense that it is possible to represent a k-tape TM on one tape, and to create a set of 1-tape rules that simulates the evolution of the k-tape TM. Of course, the 1-tape machine is much slower, as it needs to repeatedly scan its tape back and forth just to simulate a single step of the k-tape one.

Theorem 1 (k-tape Turing machine emulation)**.** *If a k-tape Turing machine $\mathcal{M}$ takes time $T(n)$ on inputs of time n, then it is possible to program a 1-tape Turing machine $\mathcal{M}'$ that simulates it (i.e., essentially performs the same computation) in time $O(T(n)^2)$.*

Proof. See Arora-Barak, Claim 1.9 in the public draft.

Basically, the k tapes of $\mathcal{M}$ are encoded on the single tape of $\mathcal{M}'$ by alternating the cell contents of each tape; in order to remember the "current position" on each tape, every symbol is complemented by a different version (e.g., a "hatted" symbol) to be used to mark the current position. To emulate a step of $\mathcal{M}$, the whole tape of $\mathcal{M}'$ is first scanned in order to find the k symbols in the current positions; then, a second scan is used to replace each symbol in the current position with the new symbol; then a third scan performs an update of the current positions.

Since $\mathcal{M}$ halts in $T(n)$ steps, no more that $T(n)$ cells of the tapes will ever be visited; therefore, every scan performed by $\mathcal{M}'$ will take at most $kT(n)$ steps. Given some more details, cleanup tasks and so on, the simulation of a single step of $\mathcal{M}$ will take at most $5kT(n)$ steps by $\mathcal{M}'$, therefore the whole simulation takes $5kT(n)^2$ steps. Since $5k$ is constant wrt the input size n, the result follows. □

Size of the alphabet

The number of symbols that can be written on a tape (the size of the alphabet Σ) can make some tasks easier; for instance, in order to deal with binary numbers a three-symbol alphabet ("0", "1", and the blank as a separator) is convenient, while working on words is easier if the whole alphabet is available.

While a 1-sized alphabet $\Sigma = \{\textvisiblespace\}$ is clearly unfit for a TM (no way to store information on the tape), a 2-symbol alphabel is enough to simulate any TM:

Theorem 2 (Emulation by a two-symbol Turing Machine)**.** *If a Turing machine $\mathcal{M}$ with a k-symbol alphabet Σ takes time $T(n)$ on an input of size n, then it can be simulated by a Turing machine $\mathcal{M}'$ with a 2-symbol alphabet $\Sigma' = \{0, 1\}$ in time $O(T(n))$ (i.e., with a linear slowdown).*

Proof. See Arora-Barak, claim 1.8 in the public draft, where for convenience machine $\mathcal{M}'$ is assumed to have 4 symbols and the tape(s) extend only in one direction.

Every symbol from alphabet Σ can be encoded by $\lceil \log_2 k \rceil$ binary digits from Σ'. Every step of machine $\mathcal{M}$ will be simulated by $\mathcal{M}'$ by reading $\lceil \log_2 k \rceil$ cells in order to reconstruct the current symbol in $\mathcal{M}$; the symbol being reconstructed bit by bit is stored in the machine state (therefore, $\mathcal{M}'$ requires many more states that $\mathcal{M}$). This scan is followed by a new scan to replace the encoding with the new symbol (again, all information needed by $\mathcal{M}'$ will be "stored" in its state), and a third (possibly longer) scan to place the current position to the left or right encoding. Therefore, a step of $\mathcal{M}$ will require less than $4\lceil \log_2 k \rceil$ steps of $\mathcal{M}'$, and the total simulation time will be

$$T'(n) \leq 4\lceil \log_2 k \rceil T(n).$$

□

Simulating other computational devices

Although they are very simple devices, we can convince ourselves quite easily that Turing machines can emulate a simple CPU/RAM architecture: just replace random access memory with sequential search on a tape (tremendous slowdown, but we are not concerned by it now), the CPU's internal registers can be stored in separate tapes, and every opcode of the CPU corresponds to a separate set of states of the machine. Operations such as "load memory to a register," "perform an arithmetic or logical operation between registers," "conditionally junp to memory" and so on can be emulated.

1.2.3 Universal Turing machines

The main drawback of TMs, as described up to now, with respect to our modern understanding of computational systems, is that each serves one specific purpose, encoded in its rule set: a machine to add numbers, one to multiply, and so on.

However, it is easy to see that a TM can be represented by a finite string in a finite alphabet: each transition rule can be seen as a quintuplet, each from a finite set, and the set of rules is finite. Therefore, it is possible to envision a TM $\mathcal{U}$ that takes another TM $\mathcal{M}$ as input on its tape, properly encoded, together with an input string s for $\mathcal{M}$, and simulates $\mathcal{M}$ step by step on input s. Such machine is called a Universal Turing machine (UTM).

One such machine, using a 16 symbol encoding and a single tape, is described in

`https://www.dropbox.com/sh/u7jsxm232giwown/AADTRNqjKBIe_QZGyicoZWjYa/utm.pdf`

and can be seen in action at the aforementioned link `http://morphett.info/turing/turing.html`, clicking "Load an example program / Universal Turing machine."

1.2.4 The Church-Turing thesis

We should be convinced, by now, that TMs are powerful enough to be a fair computational model, at least on par with any other reasonable definition. We formalize this idea into a sort of "postulate," i.e., an assertion that we will assume to be true for the rest of this course.

Postulate 1 (Church-Turing thesis)**.** *Turing machines are at least as powerful as every physically realizable model of computation.*

This thesis allows us to extend every result about TMs to every physical computational device.

1.3 Uncomputable functions

It is easy to understand that, even if we restrict our interest to decision functions, almost all functions are not computable by a TM. In fact, as the following Lemmata 1 and 2 show, there are simply too many functions to be able to define a TM for each of them.

Lemma 1. *The set of decision functions $f : \mathbb{N} \to \{0,1\}$ (or, equivalently, $f : \Sigma^* \to \{0,1\}$), is uncountable.*

Proof. By contradiction, suppose that a complete mapping exists from the naturals to the set of decision functions; i.e., there is a mapping $n \mapsto f_n$ that enumerates all functions. Define function $\hat{f}(n) = 1 - f_n(n)$. By definition, function $\hat{f}$ differs from f_n on the value it is assigned for n (if $f_n(n) = 0$ then $\hat{f}(n) = 1 - f_n(n) = 1 - 0 = 0$, and vice versa). Therefore, contrary to the assumption, the enumeation is not complete because it excluded function $\hat{f}$. □

Lemma 1 is an example of *diagonal argument*, introduced by Cantor in order to prove the uncountability of real numbers: focus on the "diagonal" values (in our case $f_n(n)$, by using the same number as function index and as argument), and make a new object that systematically differs from all that are listed.

Lemma 2. *Given a finite alphabet Σ, the number of TMs on that alphabet is countable.*

Proof. As shown in the Universal TM discussion, every TM can be encoded in some appropriate alphabet. As shown by Theorem 2, every alphabet with at least two symbols can emulate and be emulated by every other alphabet. Therefore, it is possible to define a representation of any TM in any alphabet.

We know that strings can be enumerated: first we count the only string in Σ^0, then the strings in Σ^1, then those in Σ^2 (e.g., in lexicographic order), and so on. Since every string $s \in \Sigma^*$ is finite ($s \in \Sigma^{|s|}$), sooner or later it will be enumerated. Therefore there is a mapping $\mathbb{N} \to \Sigma^*$, i.e., Σ^* is countable.

Since TMs can be mapped on a subset of Σ^* (those strings that define TMs according to the chosen encoding), and are still infinite, it follows that TMs are countable. □

Therefore, whatever way we choose to enumerate TMs and to associate them with decision functions, we will inevitably leave out some functions. Hence, given that TMs are our definition of computing,

Corollary 1. *There are uncomputable decision functions.*

1.3.1 Finding an uncomputable function

Let us introduce a little more notation. As already defined, the alphabet Σ contains a distinguished, "default" symbol, which we assume to be "␣". Before the computation starts, only a finite number of cell tapes have non-blank symbols. Let us define as "input" the smallest, contiguous set of tape cells that contains all non-blank symbols at the beginning of the computation.

A Turing machine transforms an input string into an output string (the smallest contiguous set of tape cells that contain all non-blank symbols at the *end* of the computation), but it might never terminate. In other words, if we see a TM machine as a function from Σ^* to Σ^* it might not be a *total* function.

As an alternative, we may introduce a new value, ∞, as the "value" of a non-terminating computation; given a Turing machine $\mathcal{M}$, if its compuattion on input s does not terminate we will write $\mathcal{M}(s) = \infty$.

While TM encodings have a precise syntax, so that not all strings in Σ^* are syntactically valid encodings of some TM, we can just accept the convention that any such invalid string encodes the TM that immediately halts (think of s as a program, executed by a UTM that immediately stops if there is a syntax error). This way, all strings can be seen to encode a TM, and most string just encode the "identity function" (a machine that halts immediately leaves its input string unchanged). Let us therefore call $\mathcal{M}_s$ the TM whose encoding is string s, or the machine that immediately terminates if s is not a valid encoding.

With this convention in mind, we can design a function whose outcome differs from that of any TM. We employ a diagonal technique akin to the proof of Lemma 1: for any string $\alpha \in \Sigma^*$, we define our function to differ from the output of the TM encoded by α on input α itself.

Theorem 3. *Given an alphabet Σ and an encoding $\alpha \mapsto \mathcal{M}_\alpha$ of TMs in that alphabet, the function*

$$UC(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases} \qquad \forall \alpha \in \Sigma^*$$

is uncomputable.

Proof. Let $\mathcal{M}$ be any TM, and let $m \in \Sigma^*$ be its encoding (i.e., $\mathcal{M} = \mathcal{M}_m$). By definition, $UC(m)$ differs from $\mathcal{M}(m)$: the former outputs one if and only if the latter outputs anything else (or does not terminate).
See also Arora-Barak, theorem 1.16 in the public draft. □

What is the problem that prevents us from computing UC? While the definition is quite straightforward, being able to emulate the machine $\mathcal{M}_\alpha$ on input α is not enough to always decide the value of $UC(\alpha)$. We need to take into account also the fact that the emulation might never terminate. This allows us to prove, as a corollary of the preceding theorem, that there is no procedure that always determines whether a machine will terminate on a given input.

Theorem 4 (Halting problem)**.** *Given an alphabet Σ and a encoding $\alpha \mapsto \mathcal{M}_\alpha$ of TMs in that alphabet, the function*

$$HALT(s,t) = \begin{cases} 0 & \text{if } \mathcal{M}_s(t) = \infty \\ 1 & \text{otherwise} \end{cases} \forall (s,t) \in \Sigma^* \times \Sigma^*$$

(i.e., which returns 1 if and only if machine $\mathcal{M}_s$ halts on input t) is uncomputable.

Proof. Let's proceed by contradiction. Suppose that we have a machine $\mathcal{H}$ which computes $\text{HALT}(s,t)$ (i.e., when run on a tape containing a string s encoding a TM and a string t, always halts returning 1 if machine $\mathcal{M}_s$ would halt when run on input t, and returning 0 otherwise). Then we could use $\mathcal{H}$ to compute function UC.
For convenience, let us compute UC using a machine with two tapes. The first tape is read-only and contains the input string $\alpha \in \Sigma^*$, while the second will be used as a work (and output) tape. To compute UC, the machine will perform the following steps:

- Create two copies of the input string α onto the work tape, separated by a blank (we know we can do this because we can actually write the machine);
- Execute the machine $\mathcal{H}$ (which exists by hypothesis) on the work tape, therefore calculating whether the computation $\mathcal{M}_\alpha(\alpha)$ would terminate or not. Two outcomes are possible:
  - If the output of $\mathcal{H}$ is zero, then we know that the computation of $\mathcal{M}_\alpha(\alpha)$ wouldn't terminate, therefore, by definition of function UC, we can output 1 and terminate.
  - If, on the other hand, the output of $\mathcal{H}$ is one, then we know for sure that the computation $\mathcal{M}_\alpha(\alpha)$ would terminate, and we can emulate it with a UTM $\mathcal{U}$ (which we know to exist) and then "inverting" the result à la UC, by executing the following steps:
    - As in the first step, create two copies of the input string α onto the work tape, separated by a blank;

* Execute the UTM $\mathcal{U}$ on the work tape, thereby emulating the computation $\mathcal{M}_\alpha(\alpha)$;
* At the end, if the output of the emulation was 1, then replace it by a 0; if it was anything other than 1, replace it with 1.

This machine would be able to compute UC by simply applying its definition, but we know that UC is not computable by a TM; all steps, apart from $\mathcal{H}$, are already known and computable. We must conclude that $\mathcal{H}$ cannot exist.
See also Arora-Barak, theorem 1.17 in the public draft. □

This proof employs a very common technique of CS, called *reduction*: in order to prove the impossibility of HALT, we "reduce" the computation of UC to that of HALT; since we know that the former is impossible, we must conclude that the latter is too.

The Haliting Problem for machines without an input

Consider the special case of machines that do not work on an input string; i.e., the class of TMs that are executed on a completely blank tape. Asking whether a computation without input will eventually halt might seem a simpler question, because we somehow restrict the number of machines that we are considering.

Let us define the following specialized halting function:

$$HALT_\varepsilon(s) = \mathrm{HALT}(s,\varepsilon) = \begin{cases} 0 & \text{if } \mathcal{M}_s(\varepsilon) = \infty \\ 1 & \text{otherwise} \end{cases} \forall s \in \Sigma^*.$$

It turns out that if we were able to compute $HALT_\varepsilon$ then we could also compute HALT:

Theorem 5. *$HALT_\varepsilon$ is not computable.*

Proof. By contradiction, suppose that there is a machine $\mathcal{H}'$ that computes $HALT_\varepsilon$. Such machine would be executed on a string s on the tape, and would return 1 if the machine encoded by s would halt when run on an empty tape, 0 otherwise.
Now, suppose that we are asked to compute $\mathrm{HALT}(s,t)$ for a non-empty input string t. We can transform the computation $\mathcal{M}_s(t)$ on a computation $\mathcal{M}_{s'}(\varepsilon)$ on an empty tape where s' contains the whole encoding s, but prepended with a number of states that write the string t on the tape. In other words, we transform a computation on a generic input into a computation on an empty tape that writes the desired input before proceeding.
After modifying the string s into s' on tape, we can run $\mathcal{H}'$ on it. The answer of $\mathcal{H}'$ is precisely $\mathrm{HALT}(s,t)$, which would therefore be computable. □

Again, the result was obtained by reducing a known impossible problem, HALT to the newly introduced one, $HALT_\varepsilon$.

1.3.2 Recursive enumerability of halting computations

Although HALT is not computable, it is clearly recursively enumerable. In fact, we can just take a UTM and modify it to erase the tape and write "1" whenever the emulated machine ends, and we would have a partial function that always accepts (i.e., returns 1) on terminating computations.

It is also possible to output all $(s,t) \in \Sigma^* \times \Sigma^*$ pairs for which $\mathcal{M}_s(t)$ halts by employing a diagonal method similar to the one used in Fig. 1.2[8].

Function HALT is our first example of R.E. function that is provably not recursive.

Observe that, unlike recursivity, R.E. does *not* treat the "yes" and "no" answer in a symmetric way. We can give the following:

[8]See the figure at `https://en.wikipedia.org/wiki/Recursively_enumerable_set#Examples`

Definition 1. *A decision function $f : \Sigma^* \to \{0, 1\}$ is* co-R.E. *if it admits a TM $\mathcal{M}$ such that $\mathcal{M}(s)$ halts with output 0 if and only if $f(s) = 0$.*

In other words, co-R.E. functions are those for which it is possible to compute a "no" answer, while the computation might not terminate if the answer is "yes". Clearly, if f is R.E., then $1 - f$ is co-R.E.

Theorem 6. *A decision function $f : \Sigma^* \to \{0, 1\}$ is recursive if and only if it is both R.E. and co-R.E.*

Proof. Let us prove the "only if" part first. If f is recursive, then there is a TM $\mathcal{M}_f$ that computes it. But $\mathcal{M}_f$ clearly satisfies both the R.E. definition ($\mathcal{M}_f(s)$ halts with output 1 if and only if $f(s) = 1$) and the co-R.E. definition ($\mathcal{M}_f(s)$ halts with output 0 if and only if $f(s) = 0$).
About the "if" part, if f is R.E., then there is a TM M_1 such that $M_1(s)$ halts with output 1 iff $f(s) = 1$; since f is also co-R.E., then there is also a TM $\mathcal{M}_0$ such that $M_1(s)$ halts with output 0 iff $f(s) = 0$. Therefore, a machine that alternates one step of the execution of $\mathcal{M}_1$ with one step of $\mathcal{M}_0$, halting when one of the two machines halts and returning its output, will eventually terminate (because, whatever the value of f, at least one of the two machines is going to eventually halt) and precisely decides f. $\square$

Observe that, as already pointed out, any definition given on decision functions with domain Σ^* also works on domain $\mathbb{N}$ (and on any other discrete domain), and can be naturally extended on subsets of strings or natural numbers. We can therefore define a set as recursive, recursively enumerable, or co-recursively enumerable.

Decision and acceptance

In the following, we will use the following terms when speaking of languages.

Definition 2.
- *If language S is recursively enumerable, i.e. there is a TM $\mathcal{M}$ such that $\mathcal{M}(s) = 1 \Leftrightarrow s \in S$, then we say that $\mathcal{M}$* accepts *S (or that it "recognizes" it).*
- *Given a TM $\mathcal{M}$, the language recognized by it (i.e., the set of all inputs that are accepted by the machine) is represented by $L(\mathcal{M})$.*
- *If language S is recursive, i.e. there is a TM $\mathcal{M}$ that accepts it and always halts, then we say that $\mathcal{M}$* decides *S.*

In the case of functions transforming strings, we will use the following terms.

Definition 3. *If a function $f : \Sigma^* \to \Sigma^*$ is computable, i.e. there is a TM $\mathcal{M}$ that always halts and such that $\mathcal{M}(s) = f(s)$, then we say that $\mathcal{M}$* computes *f.*

We generalize the notion to functions outside the realm of strings by considering suitable representations. E.g., a machine $\mathcal{M}$ *computes* an integer function $f : \mathbb{N} \to \mathbb{N}$ if it transforms a representation of $n \in \mathbb{N}$ (e.g., its decimal, binary or unary notation) into the corresponding representation of $f(n)$. Since all representations of integer numbers can be converted to each other by a TM, the choice of a specific one is arbitrary and does not impact on the definition. Therefore, we can resort to unary notation and say that

Theorem 7. *A function $f : \mathbb{N} \to \mathbb{N}$ is computable if and only if there is a TM $\mathcal{M}$ on alphabet $\Sigma = \{1, \text{␣}\}$ such that*

$$\forall n \in \mathbb{N} \quad \mathcal{M}(1^n) = 1^{f(n)}.$$

I.e., the TM $\mathcal{M}$ maps a string of n ones into a string of $f(n)$ ones.

1.3.3 Another uncomputable function: the Busy Beaver game

Since we might be unable to tell at all whether a specific TM will halt, the question arises of how complex can machine's output be for a given number of states.

Definition 4 (The Busy Beaver game)**.** *Among all TMs on alphabet* $\{0,1\}$ *and with* $n = |Q|$ *states (not counting the halting one) that halt when run on an empty (i.e., all-zero) tape:*

- *let* $\Sigma(n)$ *be the largest number of (not necesssarily consecutive) ones left by any machine upon halting;*
- *let* $S(n)$ *be the largest number of steps performed by any such machine before halting.*

Function $\Sigma(n)$ is known as the *busy beaver* function for n states, and the machine that achieves it is called the Busy Beaver for n states.

Both functions grow very rapidly with n, and their values are only known for $n \leq 4$. The current Busy Beaver candidate with $n = 5$ states writes more than 4K ones before halting after more than 47M steps.

Theorem 8. *The function* $S(n)$ *is not computable.*

Proof. Suppose that $S(n)$ is computable. Then, we could create a TM to compute $HALT_\varepsilon$ (the variant with empty input) on a machine encoded in string s as follows:

on input s
  count the number n of states of $\mathcal{M}_s$
  compute $\ell \leftarrow S(n)$
  emulate $\mathcal{M}_s$ for at most ℓ steps
  if the emulation halts before ℓ steps
    then $\mathcal{M}_s$ clearly halts: **accept** and **halt**
    else $\mathcal{M}_s$ takes longer than the BB: **reject** and **halt**

□

Observe that, by construction, $\Sigma(n) \leq S(n)$ (a TM cannot write more than a symbol per step). The next result is even stronger. Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that f "eventually outgrows" g, written $f >_E g$, if $f(n) \geq g(n)$ for a sufficiently large value of n:

$$f >_E g \Leftrightarrow \exists N : \forall n > N f(n) \geq g(n).$$

Theorem 9. *The function* $\Sigma(n)$ *eventually outgrows any computable function.*

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be computable. Let us define the following function:

$$F(n) = \sum_{i=0}^{n} \left[f(i) + i^2\right].$$

By definition, F clearly has the following properties:

$$F(n) \geq f(n) \quad \forall n \in \mathbb{N}, \tag{1.1}$$

$$F(n) \geq n^2 \quad \forall n \in \mathbb{N}, \tag{1.2}$$

$$F(n+1) > F(n) \quad \forall n \in \mathbb{N} \tag{1.3}$$

the latter because $F(n+1)$ is equal to $F(n)$ plus a strictly positive term. Moreover, since f is computable, F is computable too. Suppose that M_F is a TM on alphabet $\{0,1\}$ that, when positioned on the rightmost symbol of an input string of x ones and executed, outputs a string of $F(x)$ ones (i.e., computes the function $x \mapsto F(x)$ in unary representation) and halts below the rightmost one. Let C be the number of states of M_F.

Given an arbitrary integer $x \in \mathbb{N}$, we can define the following machine $\mathcal{M}$ running on an initially empty tape (i.e., a tape filled with zeroes):

- Write x ones on the tape and stop at the rightmost one (i.e., the unary representation of x: it can be done with x states, see Exercise 2 at page 90);
- Execute M_F on the tape (therefore computing $F(x)$ with C states);
- Execute M_F again on the tape (therefore computing $F(F(x))$ with C more states).

The machine $\mathcal{M}$ works on alphabet $\{0,1\}$, starts with an empty tape, ends with $F(F(x))$ ones written on it and has $x+2C$ states; therefore it is a busy beaver candidate, and the $(x+2C)$-state busy beaver must perform at least as well:

$$\Sigma(x+2C) \geq F(F(x)). \tag{1.4}$$

Now,

$$F(x) \geq x^2 >_E x + 2C;$$

the first inequality comes from (1.2), while the second stems from the fact that x^2 eventually dominates any linear function of x. By applying F to both the left- and right-hand sides, which preserves the inequality sign because of (1.3), we get

$$F(F(x)) >_E F(x+2C). \tag{1.5}$$

By concatenating (1.4), (1.5) and (1.1), we get

$$\Sigma(x+2C) \geq F(F(x)) >_E F(x+2C) \geq f(x+2C).$$

Finally, by replaxing $n = x+2C$, we obtain

$$\Sigma(n) >_E f(n).$$

□

This proof is based on the original one given by Tibor Radó in 1962[9].

1.3.4 Reductions

Note that a few results in the past sections (Theorems 4, 5 and 8) made use of similar arguments: "If A were computable, then we could use it to solve B; however, we know that B is uncomputable, therefore A is too." Now we want to formalize such reasoning scheme.

Definition 5. *Let $L_1 \subset \Sigma_1^*$ and $L_2 \subset \Sigma_2^*$ be two languages (on possibly different alphabets). A function*

$$f : \Sigma_1^* \to \Sigma_2^*$$

is said to be a reduction *from L_1 to L_2 if*

$$\forall s \in \Sigma_1^* \quad s \in L_1 \Leftrightarrow f(s) \in L_2.$$

Basically, we can use a reduction to transform the question "Does s belong to L_1?" into the equivalent question "Does $f(s)$ belong to L_2?"

Clearly, to be useful in computability results, f has to be computable (meaning, as usual, that there is a TM $\mathcal{M}_f$ that computes f).

Definition 6. *We say that $f : \Sigma_1^* \to \Sigma_2^*$ is a* Turing reduction *from $L_1 \subset \Sigma_1^*$ to $L_2 \subset \Sigma_2^*$ if it is a reduction from L_1 to L_2 and it is computable.*

[9] See for instance:
`http://computation4cognitivescientists.weebly.com/uploads/6/2/8/3/6283774/rado-on_non-computable_functions.pdf`

If f is a reduction from L_1 to L_2 we write $L_1 <_f L_2$. In general, if there is a Turing reduction from L_1 to L_2 we say that L_1 is Turing reducible to L_2 and write $L_1 <_T L_2$.

Note that we do *not* require f to have any specific property such as being injective or surjective: just that it "does its work" by transforming any element of L_1 into an element of L_2 and every string that is not in L_1 into a string that is not in L_2.

All computability proofs by reduction follow one of the schemes listed in the following theorem:

Theorem 10. *Let languages L_1 and L_2 and function f be such that $L_1 <_f L_2$; then*

1. *if L_2 is decidable and f is computable, then L_1 is decidable too;*
2. *if L_1 is undecidable and f is computable, then L_2 is undecidable too;*
3. *lf L_1 is undecidable and L_2 is decidable, then f is uncomputable.*

Proof. The first point is proven by showing that, if we have a machine for f and a machine for L_2 we can build a machine for L_1. Let $\mathcal{M}_{L_2}$ be a TM that decides L_2, and let $\mathcal{M}_f$ be a TM that computes f. Then the machine $\mathcal{M}$ that concatenates an execution of $\mathcal{M}_f$ and an execution of $\mathcal{M}_{L_2}$, i.e. computes $\mathcal{M}(s) = \mathcal{M}_{L_2}\big(\mathcal{M}_f(s)\big)$, decides L_1 by definition of f.

The other two points follow by contradiction.

□

In other words, by writing $L_1 <_T L_2$ we mean that L_1 is "less uncomputable" than L_2.

Observe that the proofs of Theorems 4 and 5 follow the second scheme of Theorem 10, while the proof of Theorem 8 follows the third scheme, where the function $S(n)$ is part of the reduction.

Consequences of the Halting Problem incomputability

If HALT were computable, we would be able to settle any mathematical question that can be disproved by a counterexample (on a discrete set), such as the Collatz conjecture, Goldbach's conjecture[10], the non-existence of odd perfect numbers[11]... We would just need to write a machine that systematically search for one such counterexample and halts as soon as it finds one: by feeding this machine as an input to $\mathcal{H}$, we would know whether a counterexample exists at all or not.

More generally, for every proposition P in Mathematical logic we would know whether it is provable or not: just define a machine that, starting from pre-encoded axioms, systematically generates all their consequences (theorems) and halts whenever it generates P. Machine $\mathcal{H}$ would tell us whether P is ever going to be generated or not.

Note that, in all cases described above, we would only receive a "yes/no" answer, not an actual counterexample or a proof.

1.4 Rice's Theorem

Among all questions that we may ask about a Turing machine $\mathcal{M}$, some of them have a *syntactic* nature, i.e., they refer to its actual implementation: "does $\mathcal{M}$ halt within 50 steps?", "Does $\mathcal{M}$ ever reach state q?", "Does $\mathcal{M}$ ever print symbol σ on the tape?"...

Other questions are of a *semantic* type, i.e., they refer to the language accepted by $\mathcal{M}$, with no regards about $\mathcal{M}$'s behavior: "does $\mathcal{M}$ only accept even-length strings?", "Does $\mathcal{M}$ accept any string?", "Does $\mathcal{M}$ accept at least 100 different strings?"...

Definition 7. *A* property *of a TM is a mapping P from TMs to $\{0,1\}$, and we say that $\mathcal{M}$ has property P when $P(M) = 1$.*

[10] Every even number (larger than 2) can be expressed as the sum of two primes, see `https://en.wikipedia.org/wiki/Goldbach%27s_conjecture`

[11] `https://en.wikipedia.org/wiki/Perfect_number`

Definition 8. *A property is* semantic *if its value is shared by all TMs recognizing the same language: if $L(\mathcal{M}) = L(\mathcal{M}')$, then $P(\mathcal{M}) = P(\mathcal{M}')$.*

By extension, we can say that a language S has a property P if the machine that recognizes S has it. Finally, we define a property as *trivial* if all TMs have it, or if no TM has it. A property is *non-trivial* if there is at least one machine having it, and one not having it.

The two trivial properties (the one possessed by all TMs and the one posssessed by none) are easy to decide, respectively by the machine that always accepts and by the one that always rejects. On the other hand:

Theorem 11 (Rice's Theorem)**.** *All non-trivial semantic properties of TMs are undecidable.*

Proof. As usual, let's work by contradiction via reduction from the Halting Problem.
Suppose that a non-trivial semantic property P is decidable; this means that there is a TM $\mathcal{M}_P$ that can be run on the encoding of any TM $\mathcal{M}$ and returns 1 if $\mathcal{M}$ has property P, 0 otherwise.
Let us also assume that the empty language $\emptyset$ does not have the property P (otherwise we can work on the complementary property), and that the Turing machine $\mathcal{N}$ has the property P (we can always find $\mathcal{N}$ because P is not trivial).
Given the strings $s, t \in \Sigma^*$, we can then check whether $\mathcal{M}_s(t)$ halts by building the following auxiliary TM $\mathcal{N}'$ that, on input u, works as follows:

- move the input u onto an auxiliary tape for later use, and replace it with t;
- execute $\mathcal{M}_s$ on input t;
- when the simulation halts (which, as we know, might not happen), restore the original input u on the tape by copying it back from the auxiliary tape;
- run $\mathcal{N}$ on the original input u.

The machine $\mathcal{N}'$ we just defined accepts the same language as $\mathcal{N}$ if $\mathcal{M}_s(t)$ halts, otherwise it runs forever, therefore accepting the empty language. Therefore, running our hypothetical decision procedure $\mathcal{M}_P$ on machine $\mathcal{N}'$ we obtain "yes" if $\mathcal{M}_s(t)$ halts (since in this case $L(\mathcal{N}) = L(\mathcal{N}')$) ,and "no" if $\mathcal{M}_s(t)$ doesn't halt (and thus the empty language, which doesn't have the property P, is recognized). □

Observe that we simply use $\mathcal{N}$, which has the property, as a sort of Trojan horse for computation $\mathcal{M}_s(t)$. See also the Wikipedia entry for Rice's Theorem[12].

[12] `https://en.wikipedia.org/wiki/Rice%27s_theorem`

Chapter 2

Some undecidable problems

2.1 Post Correspondence Problem

The following is an example of a problem that, while not immediately related to a computational device, can be proved to be uncomputable[1]:

Definition 9 (Post Correspondence Problem — PCP)**.** *Given two sets of n strings, $\{A_1, \dots, A_n\} \subset \Sigma^*$ and $\{B_1, \dots, B_n\} \subset \Sigma^*$, is it possible to find a finite sequence of k indices $1 \le i_1, \dots, i_k \le n$ (in no particular order and possibly with repetitions) such that $A_{i_1} A_{i_2} \cdots A_{i_k} = B_{i_1} B_{i_2} \cdots B_{i_k}$?*

In other words, if $\{(A_1, B_1), (A_2, B_2), \dots, (A_n, B_n)\} \subset \Sigma^* \times \Sigma^*$ is a finite list of *pairs* of strings, is it possible to select a finite sequence of pairs (possibly with repetitions) so that the concatenation of the first members (the A's) is equal to the concatenation of the B's?

As a trivial example, if for a specific index j $A_j = B_j$, then the positive answer to PCP is just the sequence of length $k = 1$ where $i_1 = j$. Another example with a positive answer is the following:

i	A_i	B_i
1	xxy	yxxy
2	xyxy	xyxyxxx
3	xxxyyy	yy
4	yx	yxx
5	xy	yx
6	xx	x

A solution is the index sequence $2, 3, 1, 4, 5, 5, 6$, giving the following concatenations:

i	2	3	1	4	5	5	6
A	xyxy	xxxyyy	xxy	yx	xy	xy	xx
B	xyxyxxx	yy	yxxy	yxx	yx	yx	x

Note that this is actually the concatenation of two simpler solutions: $2, 3, 1$ and $4, 5, 5, 6$.

Some problems have no solution. For instance:

i	A_i	B_i
1	10	101
2	011	11
3	101	011

[1]See the Wikipedia article
https://en.wikipedia.org/wiki/Post_correspondence_problem
and also
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.721.2199&rep=rep1&type=pdf

The sequence must start with $i_1 = 1$, but then the only way to proceed is to keep concatenating (A_3, B_3), but this way the B sequence is always 1 symbol longer than A:

i	1	3	3	3	3	3	···
A	`10`	`101`	`101`	`101`	`101`	`101`	...
B	`101`	`011`	`011`	`011`	`011`	`011`	...

Let us consider another, simpler variant of the PCP:

Definition 10 (Modified Post Correspondence Problem — MPCP)**.** *In the same conditons of PCP, we furthermore require that the first chosen index is $i_1 = 1$ (i.e., pair 1 is initially laid out).*

2.1.1 Undecidability of the Modified PCP

Let us consider a TM $\mathcal{M}$ with the following limitations:

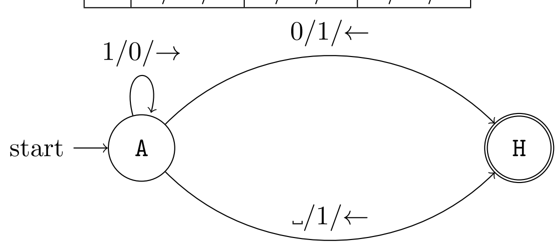
- $\mathcal{M}$ has a 3-symbol alphabet $\Sigma = \{␣, 0, 1\}$, where the default symbol is ␣;
- $\mathcal{M}$ never moves left of its starting position (i.e., the tape only extends indefinitely to the right);
- $\mathcal{M}$ never writes a ␣ (however it still has two symbols to write).

As we have seen, none of these limitations actually impair the universality of $\mathcal{M}$.

A small example

As an example, consider the following 1-state TM $\mathcal{M}$ that increments a binary number whose LSB is at the starting position (`A` is the state name, `H` is the halting state):

	␣	0	1
`A`	1/←/`H`	1/←/`H`	0/→/`A`



We use letters for states in place of the more customary $q_0, q_1, \dots$ or descriptive names like `start`, `change` because we will need to represent them as symbols in an MPCP instance.

We want to build a Modified PCP instance in which individual strings represent "pieces" of the TM's configuration, while the (A_i, B_i) string pairs "force" the construction of the solution in a way that represents the evolution of the machine's configuration from one step to the other. We will use the following alphabet:

$$\Sigma = \{␣, \texttt{0}, \texttt{1}, \texttt{A}, \texttt{H}, \texttt{\#}, \texttt{\$}\},$$

i.e., all symbols in $\mathcal{M}$'s alphabet, one symbol per state including the halting one, and two separator symbols, "`#`" to separate subsequent steps of $\mathcal{M}$'s execution, and "`$`" to represent the end of the execution.

Remember that a "configuration" of a TM consists of three pieces of information:

- the content of the tape,

- the current position, and
- the current state.

Suppose that in its initial configuration $\mathcal{M}$'s tape contains the string `11`, then its representation will be "`#A11#`", i.e., the tape's content with the state's symbol to the left of the current position, and delimiters`#` to enclose it. Since the first steps involves replacing the leftmost `1` on the tape with a `0` and moving right, the representation of $\mathcal{M}$'s evolution will be "`#A11#0A1#`".

We want to design the Modified PCP instance so that, every time we need to choose a string pair, the choice is (almost) forced, and in a way that the concatenation of the B_i's is always one $\mathcal{M}$'s step further than the concatenation of the A_i's.

We start by forcing the initial pair $A_1 =$ `#`, $B_1 =$ `#A11#`:

i	1
A	`#`
B	`#A11#`

Note that the next character to match in B is a state name, followed by a symbol. Since our transition rule requires the machine to replace the symbol and move right, whenever we find the string "`A1`" we know that the next configuration will need to contain "`0A`". That will be our second string pair ($A_2 =$ `A1`, $B_2 =$ `0A`), and we will be able to proceed with the string composition as follows:

i	1	2
A	`#`	`A1`
B	`#A11#`	`0A`

In order to complete the first step, all other symbols that are not in the current position must be copied, therefore we will need a bunch of other "copying" rules (one per tape symbol, one for the state separator)

$$A_3 = B_3 = \texttt{0}, \quad A_4 = B_4 = \texttt{1}, \quad A_5 = B_5 = \texttt{␣}, \quad A_6 = B_6 = \texttt{\#}.$$

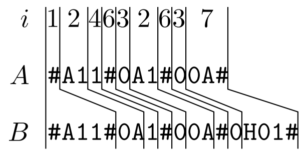
Note that these rules would make the original PCP trivial, but we are working with the modified version where an initial string is forced. With these new rules we can advance the matching:

i	1	2	4	6
A	`#`	`A1`	`1`	`#`
B	`#A11#`	`0A`	`1`	`#`

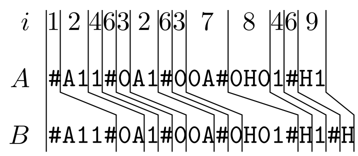
Note that the existing rules allow us to take the matching still further:

i	1	2	4	6	3	2	6	3
A	`#`	`A1`	`1`	`#`	`0`	`A1`	`#`	`0`
B	`#A11#`	`0A`	`1`	`#`	`0`	`0A`	`#`	`0`

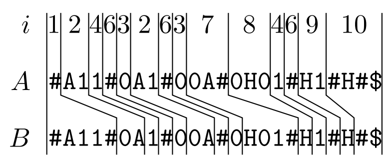
Note that in the configuration that we are currently trying to match the state letter is on the right of all tape symbols. This means that the current symbol is a blank, and $\mathcal{M}$'s transition rule requires to write "`1`", move left and halt. Since we need to move left, we cannot use the pair $i = 3$ to proceed, because the next symbol to appear in B should be the state letter "`H`"; to move left, we introduce the pair $A_7 =$ `0A#`, $B_7 =$ `H01#`, and the matching becomes:



Now string B represents the whole execution of $\mathcal{M}$ on input "`11`". Still, $A \neq B$. We will now introduce a few ad-hoc steps that, upon reaching the halting state H, get rid of all tape symbols and keep the state as the only useful information. Whenever we need to match anything in the form "σH" or "$H\sigma$", where σ is a tape symbol, we can proceed by leaving only "`H`" on B. We can add a shortcut by also matching any string in the form "$\sigma_1 H \sigma_2$". In this case, the two following pairs will do the trick: $A_8 = \texttt{0H0}$, $B_8 = \texttt{H}$ and $A_9 = \texttt{H1}$, $B_9 = \texttt{H}$.



Note how, as we clean out tape symbols, string A starts catching up to B. When B's configuration is reduced to just the Halting state symbol, we can finally close the matching by adding the following final pair to the instance, where we use the finalization marker "$": $A_{10} = \texttt{\#H\#\$}$, $B_{10} = \texttt{\#\$}$.



We have therefore constructed a Modified PCP instance that mimicks the evolution of $\mathcal{M}$ and that has a solution precisely because the machine halts.

General case

Based on the previous example, consider a TM $\mathcal{M}$ on an alphabet $\Sigma_{\mathcal{M}}$ and stateset Q, including the halting states, with the limitations discussed above. Given the initial tape content (input) $x \in \Sigma_{\mathcal{M}}^*$, we can simulate the machine's execution by building a MPCP instance on the alphabet

$$\Sigma = \Sigma_{\mathcal{M}} \cup Q \cup \{\texttt{\#}, \texttt{\$}\}$$

with the following string pairs:

- Initial pair: $A_1 = \texttt{\#}$, $B_1 = \texttt{\#A}x\texttt{\#}$.
  B_1 represents the machine in its initial configuration. With the rules below, any attempt to match string B as it grows will result in following $\mathcal{M}$'s evolution past the initial configuration.
- Copy pairs: for every symbol $\sigma \in \Sigma_{\mathcal{M}}$, add pair $A_i = B_i = \sigma$. Also add the pair $A_i = B_i = \texttt{\#}$ to propagate the "end of step" symbol.
  These pairs are needed to propagate the symbols on the tape outside the current position, in the sense that every time we add one such A_i to extend string A, the same symbol will be added to B by means of the corresponding B_i.
- Rule pairs: Add string pairs that represent the transition rules at the current position:

For every state of the form:	Add the following string pairs:
$(\sigma, S) \mapsto (\sigma', S', \rightarrow)$	$A_i = S\sigma$, $B_i = \sigma' S'$
$(\sigma, S) \mapsto (\sigma', S', \leftarrow)$	$A_i = \mu S \sigma$, $B_i = S' \mu \sigma'$ for every $\mu \in \Sigma_{\mathcal{M}}$
	(if $\sigma = \text{␣}$, then add a pair with $\sigma = \texttt{\#}$).

These pairs are the only

ones such that a non-halting state appears in a string A_i. Therefore, in order to extend the matching we will be forced to use them whenever a non-halting state symbol appears in B, enforcing the application of the transition function to the next step.
Note that in the initial pair $|A_1| < |B_1|$, and that for all other pairs listed up to now $|A_i| \leq |B_i|$; therefore, string B will always be longer than string A.

- Final cleanup: for all halting states $H \in Q$ and all tape symbols $\sigma, \sigma' \in \Sigma_{\mathcal{M}}$ add the following string pairs:

$$A_i = \sigma H, B_i = H; \qquad A_i = H\sigma, B_i = H; \qquad A_i = \sigma H \sigma', B_i = H.$$

  As said before, these pairs apply to the halting state and "consume" all tape symbols appearing in B until the halting state alone appears. Note that these are the only pairs up to now where $|A_i| > |B_i|$; therefore, until a halting state appears, there is no hope to get string A as long as string B.

- Closing pair: for all halting states H, add pair $A_i = \texttt{\#}H\texttt{\#\$}$, $B_i = \texttt{\#\$}$.
  This puts an end to the matching rush: string A is matched to the remaining part of string B.

By the considerations about matching and string lengths, one should remain convinced that the MPCP with the proposed set of string pairs has a solution if and only if $\mathcal{M}(x)$ halts, therefore proving the following theorem:

Theorem 12. *The Modified Post Correspondence Problem is undecidable.*

2.1.2 Undecidability of the Post Correspondence Problem

So far, we have been considering the "modified" case in which an initial pair is enforced. Now we need a way to transform an instance of the MPCP into an equivalent instance (i.e., with the same solution or lack thereof) of the PCP.

Suppose that the n pairs $(A_1, B_1), (A_2, B_2), \dots, (A_n, B_n)$ are an instance of the Modified PCP.

We want to transform it into a PCP instance (i.e., an instance that does not explicitly require the first chosen pair to be $i = 1$). Let * be a symbol not present in the strings. Then we can create the pairs (A'_i, B'_i) by putting a "*" *before* every symbol in A_i and *after* every symbol in B_i. So far, the PCP would have no solution: all strings in A start with the new symbol, while no string in B does.

In order to enforce the first original pair, let us introduce the new pair (A'_0, B'_0) where $A'_0 = A_1$ and $B'_0 = *B_1$. Being (so far) the only pair starting with the same symbol, (A'_0, B'_0) is the only viable first choice.

Let $1, i_2, i_3, \dots, i_k$ be a solution to the original MPCP, i.e., $A_1 A_{i_2} \cdots A_{i_k} = B_1 B_{i_2} \cdots B_{i_k}$. Then, the sequence of indices $0, i_2, \dots, i_k$ is *almost* a solution to the PCP problem that we are trying to build, in the sense that $B'_0 B'_{i_2} \cdots B'_{i_k}$ is one "*" longer than $A'_0 A'_{i_2} \cdots A'_{i_k}$. Therefore we add one last pair (A'_{n+1}, B'_{n+1}) to "absorb" the asterisk: $A'_{n+1} = *\$$, $B'_{n+1} = \$$.

As an example, here is a conversion from a MPCP instance to an equivalent PCP instance:

MPCP

i	A_i	B_i
1	11	1
2	1	111
3	0111	10
4	10	0

$\Rightarrow$

PCP

i	A_i	B_i
0	*1*1	*1*
1	*1*1	1*
2	*1	1*1*1*
3	*0*1*1*1	1*0*
4	*1*0	0*
5	*$	$

A solution to the MPCP is the $i_1 = 1$, $i_2 = 3$, $i_3 = 2$, $i_4 = 2$, $i_5 = 4$:

$$\begin{array}{r|l}
i & 1 \mid 3 \mid 2 \mid 2 \mid 4 \\
A & \texttt{11}\mid\texttt{01111}\mid\texttt{1}\mid\texttt{1}\mid\texttt{10} \\
B & \texttt{1}\mid\texttt{10}\mid\texttt{111}\mid\texttt{111}\mid\texttt{0}
\end{array}$$

The corresponding solution to the PCP problem is $i_1 = 1$, $i_2 = 3$, $i_3 = 2$, $i_4 = 2$, $i_5 = 4$, $i_6 = 5$:

$$\begin{array}{r|l}
i & 0 \mid 3 \mid 2 \mid 2 \mid 4 \mid 5 \\
A & \texttt{*1*1}\mid\texttt{*0*1*1*1}\mid\texttt{*1}\mid\texttt{*1}\mid\texttt{*1*0}\mid\texttt{*\$} \\
B & \texttt{*1*}\mid\texttt{1*0*}\mid\texttt{1*1*1*}\mid\texttt{1*1*1*}\mid\texttt{0*}\mid\texttt{\$}
\end{array}$$

The above described construction provides a PCP instance that is solvable if and only if the original MPCP instance was solvable. In addition, the construction is clearly computable and is therefore a Turing reduction from MPCP to PCP.

This proves the following

Theorem 13. *The Post correspondence problem is uncomputable.*

2.2 Kolmogorov complexity

We are quite used to programs that "compress" our files in order to save space on our mass storage media. Programs such as WinZip, WinRAR, gzip, bzip2, xzip, 7z basically operate by identifying predictable patterns in the sequence of symbols that compose the original file and replacing them with shorter descriptions according to a predefined language.

Since a file is just a string of symbols, we can ask ourselves "how much can a given string be compressed?"

To better formalize the question, let us consider the following setting:

Definition 11. *Let Σ be a suitable alphabet (e.g., ASCII or Unicode), let $\mathcal{U}$ be a universal turing machine working on Σ and let $x \in \Sigma^*$ be a string. We say that the pair of strings $D = (s, t) \in \Sigma^* \times \Sigma^*$ is a* description *of x if s encodes a TM which, when simulated by $\mathcal{U}$ on input t, produces x on the tape and halts:*

$$\mathcal{U}(s, t) = \mathcal{M}_s(t) = x.$$

In other words, we are formalizing in terms of Turing machines a very common scenario: $\mathcal{U}$ is our computer (with its operating system), while $D = (s, t)$ is a *self-extracting* executable file where s is the code that performs the decompression and t is the actual string being decompressed. We usually "run" the decompression code by double-clicking on its icon.

Another way of looking at the definition is to think of $\mathcal{U}$ as a programming language, s as a program written in that language and t as its input.

Of course, every string has many possible descriptions.

We are interested, once $\mathcal{U}$ is fixed, in finding the "most compressed" description for a string x.

Definition 12. *Given a UTM $\mathcal{U}$ and a string x, we define its* Kolmogorov complexity[2] *$K_{\mathcal{U}}(x)$ to be the size of its smallest description in $\mathcal{U}$:*

$$K_{\mathcal{U}}(x) = \min\{|(s, t)| : \mathcal{U}(s, t) = x\}.$$

We assume that $|(s, t)| = |s| + |t|$ (i.e., the size of the description is the size of the input string t plus the size of the decompressing program s).

[2]See the Wikipedia article
`https://en.wikipedia.org/wiki/Kolmogorov_complexity`

2.2.1 Dependence on the underlying computational model

Note that the definition of Kolmogorov complexity depends on the chosen computational substrate (the UTM $\mathcal{U}$). Different machines have different encodings, with different sizes, in the same way that different languages can express the same algorithm in more or less concise ways.

Theorem 14. *Given two UTMs $\mathcal{U}$ and $\mathcal{V}$, there is a* constant *value $c_{\mathcal{U}\mathcal{V}}$ such that, for every x,*

$$|K_{\mathcal{U}}(x) - K_{\mathcal{V}}(x)| \leq c_{\mathcal{U}\mathcal{V}}.$$

Note that the constant is independent of the specific string x.

Proof. Let $x \in \Sigma^*$.
Let $D_{\mathcal{U}} = (s_{\mathcal{U}}, t_{\mathcal{U}})$ be a shortest description of x in $\mathcal{U}$ (i.e., such that $\mathcal{U}(D_{\mathcal{U}}) = x$ and $|D_{\mathcal{U}}| = K_{\mathcal{U}}(x)$).
Conversely, let $D_{\mathcal{V}} = (s_{\mathcal{V}}, t_{\mathcal{V}})$ be a shortest description of x in $\mathcal{V}$ (i.e., such that $\mathcal{V}(D_{\mathcal{V}}) = x$ and $|D_{\mathcal{V}}| = K_{\mathcal{V}}(x)$).
Since $\mathcal{U}$ is a UTM, it can be used to simulate $\mathcal{V}$. Let v be the representation of $\mathcal{V}$ in $\mathcal{U}$. Therefore, $(v, D_{\mathcal{V}})$ is a description of x in $\mathcal{U}$. In fact,

$$\mathcal{U}(v, D_{\mathcal{V}}) = \mathcal{V}(D_{\mathcal{V}}) = x.$$

Therefore, $|(v, D_{\mathcal{V}})| \geq K_{\mathcal{U}}(x)$, and thus

$$K_{\mathcal{U}}(x) - K_{\mathcal{V}}(x) \leq |v|.$$

By exchanging $\mathcal{U}$ and $\mathcal{V}$, let u be an encoding of $\mathcal{U}$ that allows us to simulate it with $\mathcal{V}$; we obtain the symmetric inequality:

$$K_{\mathcal{V}}(x) - K_{\mathcal{U}}(x) \leq |u|.$$

By combining the two constants, $c_{\mathcal{U}\mathcal{V}} = \max\{u, v\}$, we obtain the thesis. □

The theorem tells us that the specific computing substrate is not very influent, as the size of x grows, because the difference is constant.

This corresponds to having a self-extracting executable created for a specific OS, say Linux, and asking if a similar compression level would be achievable on Windows. The answer is yes because, given any Linux executable of any size, we can transform it into a Windows executable by prepending to it a Linux simulator for Windows: with a fixed overhead (the Linux simulator for Windows), every self-extracting file for Linux becomes a valid self-extracting file for Windows.

2.2.2 Uncomputability of Kolmogorov complexity

However, we can prove that we cannot compute the Kolmogorov complexity of a generic string. In other words, we cannot be sure that a given description is the most compressed.

Theorem 15. *Given the UTM $\mathcal{U}$, the function $K_{\mathcal{U}} : \Sigma^* \to \mathbb{N}$ is uncomputable.*

Proof. By contradiction, suppose that $\mathcal{M}$ is a TM that computes $K_{\mathcal{U}}$. Suppose that $\mathcal{M}$ is represented by string m in $\mathcal{U}$.
Let us create the following Turing machine $\mathcal{N}$:

for all $s \in \Sigma^*$
  if $\mathcal{M}(s) \geq |m| + 1000000$
    write s
    halt

Observe the following:

- $\mathcal{N}$ does not take an input, and outputs a string whose Kolmogorov complexity (wrt $\mathcal{U}$) is greater or equal to $|m| + 1000000$.
- $\mathcal{N}$ contains $\mathcal{M}$ as a "subroutine", but we can safely assume that its description does not add more than a million symbols to that of $\mathcal{M}$.

Let x be the string written by $\mathcal{N}$ starting from the empty input. From the first point, we know that

$$K_{\mathcal{U}}(x) \geq |m| + 1000000.$$

On the other hand, let n be the string that represents $\mathcal{N}$; from the second point we know that (n, ε) is a description of x, therefore

$$K_{\mathcal{U}}(x) \leq |n| < |m| + 1000000,$$

therefore we have a contradiction. Observe that, if 1000000 looks too small an overhead, we can increase it as much as we want. □

We have searched for a string x of high Kolmogorov complexity, and in the process we have been able to generate it with a machine whose size is smaller than the (alleged) complexity of x.

This theorem is a formal rendition of the famous Berry paradox:

> "The smallest positive integer not definable with less than thirteen Englishwords."

defines such an integer with twelve words.

2.3 "Fun facts" about computability

2.3.1 Provably undecidable machines

So we know that HALT is non-recursive: there is no TM that can decide whether a given machine $\mathcal{M}$ halts on a given input t, and give the correct answer for all machines $\mathcal{M}$ and all inputs t.
However, this leaves the following question open: "Are we aware of a *specific* TM which we are *demonstrably* incapable to decide whether it halts or not?"

Consider the following facts:

1. A formal system, such as Arithmetic, can be seen as a collection of basic truths (the *axioms*) and a few inference rules (such as the *Modus Ponens*, which tells us that if A is true and $A \Rightarrow B$ is true then B is true).

2. Starting from the list of axioms of a sound Arithmetic Theory, such as Zermelo-Fraenkel's axioms, which we assume to be true, we can systematically scan the list to find statements to which we can apply our inference rules. For instance, a pair of statements in the form "A" and "$A \Rightarrow B$" can be used to *prove* B as follows: "Since A is on the list, then it is true; the same can be said for $A \Rightarrow B$; therefore, by Modus Ponens, B must be true too", so we can add B to the list. Once a statement is on the list, it can be used to infer new theorems.

3. By playing this game, every theorem (i.e., provable statement) will sooner or later appear on the list.

4. This game can be played by a TM; namely, it is possible to write a TM $\mathcal{M}_{\mathrm{ZF}}$ that writes all Zermelo-Fraenkel axioms on the tape[3] and, by repeated application of inference rules, combines all possible known truths together to generate new true statements, that it also writes on the tape. Once a statement is on the tape, it can be used as an established truth to generate new true statements.
   Every theorem in the Theory will be written down on $\mathcal{M}_{\mathrm{ZF}}$'s tape, sooner or later.

[3] There are a few additional complications: for example, Z.-F. has an infinite set of axioms; however, they belong to a finite set of parametric *axiom schemes*, and we only need to write those on the tape; every time we select an axiom scheme, we pair it with numeric values for its parameters in a systematic way

5. Let us make $\mathcal{M}_{\text{ZF}}$ halt if and only if it generates the statement $0 = 1$; the statement is blatantly false, so every Mathematician hopes that $\mathcal{M}_{\text{ZF}}$ will *never* generate it and therefore will never halt. Note that if $\mathcal{M}_{\text{ZF}}$ generates the statement $0 = 1$ it means that it is possible to prove it; since the opposite ($0 \neq 1$) can also be proved, we would conclude that the Zermelo-Fraenkel set of axioms is inconsistent: it can prove two contradictory statements, from which every statement can be proved, no matter if true or false.

6. Here comes the problem: due to Gödel's First Incompleteness Theorem, no formal theory that can define Arithmetic (such as Zermelo-Fraenkel's set of axioms) can prove its own consistency.

7. Therefore, it's impossible to prove whether $\mathcal{M}_{\text{ZF}}$ halts (if not by making the Theory stronger by adding some specialized axiom such as "$\mathcal{M}_{\text{ZF}}$ halts" or "$\mathcal{M}_{\text{ZF}}$ never halts" to it).

Note that $\mathcal{M}_{\text{ZF}}$ isn't just a theoretical machine that one is allowed to "suppose" to exist; it is a very specific example that researchers were able to create using 745 states and a 2-symbol alphabet[4].

The existence of $\mathcal{M}_{\text{ZF}}$ sets an upper bound on the number of states for which the busy beaver function $S(n)$ is computable. In particular, it proves that $S(745)$ cannot be determined. Most researchers think, however, that $S(n)$ is uncomputable for much lower values.

What about smaller machines?

The largest number of states n for which we know everything is $n = 4$. There are many 5-state, 2-symbol TMs for which we are still unable to decide whether they halt or not, but it is commonly believed that they will be settled in a few years.

However, $S(n)$ grows so fast that bets are being placed on a very small value of n as the real "uncomputability threshold" for 2-symbol TMs.

For machines with more than two symbols, the threshold might be even lower.

A very interesting starting point to explore this topic is Scott Aaronson's survey "The Busy Beaver Frontier"[5].

2.3.2 Enhancing and restricting TMs

Allowing infinite states

The main reason why we restrict TMs to a finite number of states is that we want to define a practical computational model. Having an infinite number of states would be equivalent to having a computer with an infinitely long program. It is easy to prove that such assumption would make every function computable: we would be able to encode the complete input in the machine's state.

Theorem 16. *Every decision function is computable by a Turing Machine with a countable set of states.*

Proof. Let $\Sigma = \{\text{␣}, 0, 1\}$; let $f : \{0,1\}^* \to \{0,1\}$ be an arbitrary decision function on binary strings. We can define the TM that decides f as follows.
Since Σ^* is countable, let us index the states with strings: $Q = \{q_s | s \in \Sigma^*\}$, with q_ε as initial state. Let the initial position be the leftmost symbol in the input.
Our machine will sweep the input string, erasing it and collecting the bit sequence in the state's index. As soon as the string is fully scanned, the machine reads a blank and replaces it with the value $f(s)$, leaving it as its final output before halting.
Here is the transition function that implements the machine:

$$\begin{aligned} F : \Sigma \times Q &\to \Sigma \times Q \times \{\leftarrow, \rightarrow\} \\ (\sigma, q_s) &\mapsto \begin{cases} (\text{␣}, q_{s\sigma}, \rightarrow) & \text{if } \sigma \in \{0,1\} \\ (f(s), \text{HALT}, \rightarrow) & \text{if } \sigma = \text{␣}. \end{cases} \end{aligned}$$

[4]see for instance `https://www.ingo-blechschmidt.eu/assets/bachelor-thesis-undecidability-bb748.pdf`
[5]`https://www.scottaaronson.com/papers/bb.pdf`

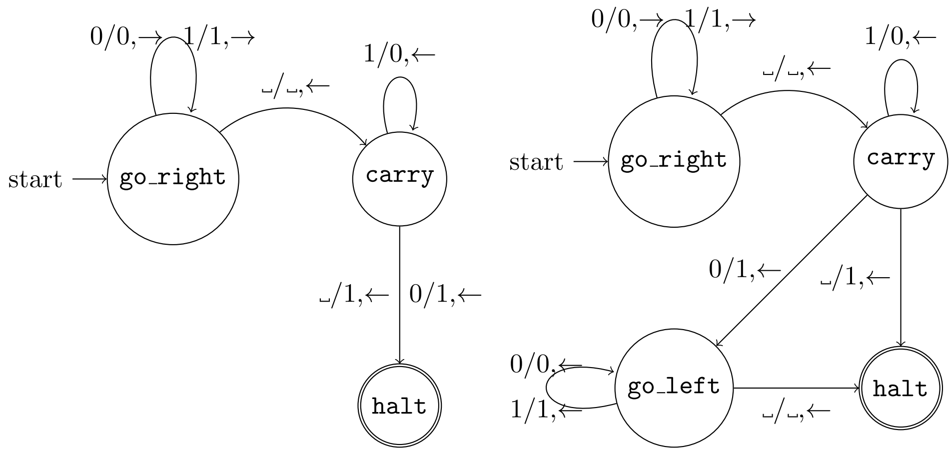


Figure 2.1: A non-oblivious TM (left) and an equivalent oblivious version (right).

□

Two observations:

- Basically, the machine we just described implements a binary tree with unbounded depth, where every node is associated to the function's value for the corresponding sequence of edge labels. In other words, an infinite-state machine would allow us to build a program with an infinite chain of `if` statements.
- Note that we needn't be able to "build" the machine. What we can say is that for every possible decision function, even the Halting problem, there is an infinite-state TM that computes it, even though we might not be able to identify it.

In conclusion, machines with an infinite number of states are quite uninteresting for our purposes.

Oblivious Turing Machines

Definition 13. *A (one-tape) Turing Machine is said to be* oblivious *if the sequence of left-right moves it performs does not depend on the specific input string, but only on its length.*

The term is mutuated from computer security and cryptography: it refers to the inability for an observer who can only see the machine's movements, but not the actual tape contents, to learn anything about the machine's input — aside from its length.

As an example, consider the machine represented in the left-hand side of Fig. 2.1. It works on $\Sigma = \{\textvisiblespace, 0, 1\}$ and, assuming an input string $s \in \{0,1\}^*$ and start position in the leftmost non-blank symbol, increments the value represented by s by 1. It scans the input left to right (state "go_right") until it finds a blank, then (state "carry") overwrites 1's with 0's until it finda the first non-1 symbol; then it writes a 1 and halts. The initial ("go_right") phase of the machine is oblivious: the motion does not depend on whether the symbol is 0 or 1, and it only ends when a blank is found (in this context, we interpret blanks as end-of-input markers). However, the carry phase ends as soon as a non-1 symbol is found, and therefore depends on the input symbols. An onlooker who only cares about the machine's moves learns something about the input string, namely how many 1's it ended with (i.e., the position of the rightmost 0).

To make the machine fully oblivious, it can simply keep moving left after the carry phase, leaving the rest of the input unchanged, and halting only when it finds the left blank delimiter. To do this (right-hand side of Fig. 2.1), we just add a state "go_left" that completes the left sweep before halting.

At this point, for every input s of size $n = |s|$, the machine performs precisely n right moves, followed by $n + 2$ left moves before halting, regardless of the input symbols: an onlooker only learns the string length, but none of its bits.

This restriction does not impair the model's computational power: all computable functions can be computed by an oblivious machine (with a quadratic time loss).

Theorem 17. *Every 1-tape TM $\mathcal{M}$ with worst-case time $T_{\mathcal{M}}(n)$ on input size n can be simulated by an oblivious TM $\mathcal{M}_{obl}$ with a quadratic increase on the worst-case time: $T_{\mathcal{M}_{obl}}(n) = O(T_{\mathcal{M}}(n)^2)$.*

Proof. Consider the proof of Theorem 1, concerning the emulation of a multiple-tape TM. The emulation technique presented in that proof is almost oblivious, as it requires a sequence of full input sweeps in order to emulate a single step of the k-tape machine.

Fundamentally, $\mathcal{M}_{\mathrm{obl}}$ will use an extended alphabet with marked symbols to record the current position of $\mathcal{M}$ on the tape. Every step of $\mathcal{M}$ is emulated by sweeping the whole input left-to-right in order to acquire the symbol in the current position, then right-to-left to update the tape.

With a few more precautions, such as marking the beginning and end of the (potentially increasing) visited portion of the tape with special symbols, all tape sweeps can be made exactly similar to each other, with the exception of their length, which will increase by 1 at each end at every sweep.

The quadratic slowdown is due to the fact that we replace every step of $\mathcal{M}$ with a number of tape sweeps, and the size of the (visited portion of the) tape might grow by 1 at every step, and is therefore bounded by $O(T_{\mathcal{M}}(n))$. □

Chapter 3

Complexity classes: P and NP

From now on, we will be only dealing with computable functions; the algorithms that we will analize will always terminate, and our main concern will be about the amount of resources (time, space) required to compute them.

3.1 Definitions

When discussing complexity, we are mainly interested in the relationship between the size of the input and the execution "time" of an algorithm executed by a Turing machine. We still refer to TMs because both input size and execution time can be defined unambiguously in that model.

Input size

By "size" of the input, we mean the number of symbols used to encode it in the machine's tape. Since we are only concerned in asymptotic relationships, the particular alphabet used by a machine is of no concern, and we may as well just consider machines with alphabet $\Sigma = \{0, 1\}$.
We require that the input data are encoded in a reasonable way. For instance, numbers may be represented in base-2 notation (although the precise base does not matter when doing asymptotic analysis), so that the size of the representation $r_2(n)$ of integer n in base 2 is logarithmic with respect to its value:

$$|r_2(n)| = O(\log n).$$

In this sense, unary representations (representing n by a string of n consecutive 1's) is not to be considered reasonable because its size is exponential with respect to the base-2 notation.

Execution time

We dub "execution time," or simply "time," the number of steps required by a TM to get to a halting state. Let $\mathcal{M}$ be a TM that always halts. We can define the "time" function

$$\begin{aligned} t_{\mathcal{M}} : \Sigma^* &\to \mathbb{N} \\ x &\mapsto \#\text{ of steps before } \mathcal{M} \text{ halts on input } x \end{aligned}$$

that maps every input string x onto the number of steps that $\mathcal{M}$ performs upon input x before halting. $\mathcal{M}$ always halts, so it is a well-defined function. Since the number of strings of a given size n is finite, we can also define (and actually compute, if needed) the following "worst-case" time for inputs of size n:

$$\begin{aligned} T_{\mathcal{M}} : \mathbb{N} &\to \mathbb{N} \\ n &\mapsto \max\{t_{\mathcal{M}}(x) : x \in \Sigma^n\}, \end{aligned}$$

i.e., $T_{\mathcal{M}}(n)$ is the longest time that $\mathcal{M}$ takes before halting on an input of size n.

3.2 Polynomial languages

Let us now focus on decision problems.

Definition 14. *Let $f : \mathbb{N} \to \mathbb{N}$ be any computable function. We say that a language $L \subseteq \Sigma^*$ is of class DTIME(f), and write $L \in$ DTIME(f), if there is a TM $\mathcal{M}$ that decides L and its worst-case time, as a function of input size, is dominated by f:*

$$L \in DTIME(f) \quad \Leftrightarrow \quad \exists \mathcal{M} : \quad L(\mathcal{M}) = L \wedge T_{\mathcal{M}} = O(f).$$

In other words, DTIME(f) is the class of all languages that can be decided by some TM in time eventually bounded by function $c \cdot f$, where c is constant.
Saying $L \in$ DTIME(f) means that there is a machine $\mathcal{M}$, a constant $c \in \mathbb{N}$ and an input size $n_0 \in \mathbb{N}$ such that, for every input x with size larger than n_0, $\mathcal{M}$ decides $x \in L$ in at most $c \cdot f(|x|)$ steps.

Languages that can be decided in a time that is polynomial with respect to the input size are very important, so we give a short name to their class:

Definition 15.

$$\boldsymbol{P} = \bigcup_{k=0}^{\infty} DTIME(n^k).$$

In other words, we say that a language $L \in \Sigma^*$ is polynomial-time*, and write $L \in \boldsymbol{P}$, if there are a machine $\mathcal{M}$ and a polynomial $p(n)$ such that for every input string x*

$$x \in L \quad \Leftrightarrow \quad \mathcal{M}(x) = 1 \,\wedge\, t_{\mathcal{M}}(x) \leq p(|x|). \tag{3.1}$$

3.2.1 Examples

Here are some examples of polynomial-time languages.

CONNECTED — Given an encoding of graph G (e.g., the number of nodes followed by an adjacency matrix or list), $G \in$ CONNECTED if and only if there is a path in G between every pair of nodes.

PRIME — Given a base-2 representation of a natural number N, we say that $N \in$ PRIME if and only if N is, of course, prime.
Observe that the naive algorithm "divide by all integers from 2 to $\lfloor\sqrt{N}\rfloor$" is *not* polynomial with respect to the size of the input string. In fact, the input size is $n = O(\log N)$ (the number of bits used to represent a number is logarithmic with respect to its magnitude), therefore the naive algorithm would take $\lfloor\sqrt{N}\rfloor - 1 = O(2^{n/2})$ divisions in the worst case, which grows faster than any polynomial[1].
Anyway, it has recently been shown[2] that PRIME $\in$ **P**.

(Counter?)-examples

On the other hand, we do not know of any polynomial-time algorithm for the following languages:

SATISFIABILITY or SAT — Given a Boolean expression $f(x_1, \dots, x_n)$ (usually in conjunctive normal form, CNF[3]) involving n variables, is there a truth assignment to the variables that satisfies (i.e., makes true) the formula[4]?

---

[1] An algorithm that is polynomial with respect to the *magnitude* of the numbers instead than the size of their representation is said to be "pseudo-polynomial." In fact, the naive primality test would be polynomial if we chose to represent N in unary notation (N consecutive 1's).

[2] `https://en.wikipedia.org/wiki/Primality_test#Fast_deterministic_tests`

[3] `https://en.wikipedia.org/wiki/Conjunctive_normal_form`

[4] `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

CLIQUE — Given an encoding of graph G and a number k, does G contain k nodes that are all connected to each other[5]?

TRAVELING SALESMAN PROBLEM or TSP — Given an encoding of a complete *weighted* graph G (i.e., all pairs of nodes are connected, and pair i, j is assigned a "weight" w_{ij}) and a "budget" k, is there an order of visit (permutation) σ of all nodes such that

$$\left(\sum_{i=1}^{n-1} w_{\sigma_i \sigma_{i+1}}\right) + w_{\sigma_n \sigma_1} \leq k, \tag{3.2}$$

i.e., the total weight along the corresponding closed path in that order of visit (also considering return to the starting node) is within budget[6]?

However, we have no proof that these languages (and many others) are not in **P**. In the following section, we will try to characterize these languages.

3.2.2 Example: Boolean formulas and the conjunctive normal form

To clarify the SAT example, let us specify how a typical SAT instance is represented.

Given n boolean variables $x_1, \dots, x_n$, we can define the following:

- a *term*, or *literal*, is a variable x_i or its negation $\neg x_i$;
- a *clause* is a disjunction of terms;
- finally, a *formula* or *expression* is a conjunction of clauses.

Definition 16 (Conjunctive Normal Form (CNF))**.** *A formula f is said to be in* conjunctive normal form *with n variables and m clauses if it can be written as*

$$f(x_1, \dots, x_n) = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} g_{ij},$$

where clause i has l_i terms, every literal g_{ij} is in the form x_k or in the form $\neg x_k$.

For instance, the following is a CNF formula with $n = 5$ variables and $m = 4$ clauses:

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= (x_1 \vee \neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ &\quad \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \tag{3.3}$$

Asking about the satisfiability of a CNF formula f amount at asking for a truth assignment such that every clause has at least one true literal. For example, the following assignment, among many others, satisfies (3.3):

$$x1 = x2 = \text{true}; \quad x_3 = x_4 = x_5 = \text{false}.$$

We can therefore say that $f \in \text{SAT}$.

Note that CNF is powerful enough to express any (unquantified) statement about boolean variables. For instance, the following 2-variable formula is satisfiable only by variables having the same truth value:

$$(\neg x \vee y) \wedge (x \vee \neg y).$$

It therefore "captures" the idea of equality in the sense that it is true whenever $x = y$. In fact, the clause $(\neg x \vee y)$ means "x implies y."

[5] `https://en.wikipedia.org/wiki/Clique_(graph_theory)`

[6] `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

Moreover, there are standard ways to convert *any* Boolean formula to CNF, based on some simple transformation rules, easily verifiable by testing all possible combinations of values — or just by reasoning:

$$
\begin{aligned}
a \vee (b \wedge c) &\equiv (a \vee b) \wedge (a \vee c) \\
a \wedge (b \vee c) &\equiv (a \wedge b) \vee (a \wedge c) \\
\neg(a \vee b) &\equiv \neg a \wedge \neg b \\
\neg(a \wedge b) &\equiv \neg a \vee \neg b \\
a \to b &\equiv \neg a \vee b.
\end{aligned}
$$

3.3 NP languages

While the three languages listed above (SAT, CLIQUE, TSP) cannot be decided by any known polynomial algorithm, they share a common property: if a string is in the language, there is an "easily" (polynomially) verifiable proof of it:

- If $f(x_1, \dots, x_n) \in$ SAT (i.e., boolean formula f is satisfiable), then there is a truth assignment to the variables $x_1, \dots, x_n$ that satisfies it. If we were given this truth assignment, we could easily check that, indeed, $f \in$ SAT. Note that the truth assignment consists of n truth values (bits) and is therefore shorter than the encoding of f (which contains a whole boolean expression on n variables), and that computing a Boolean formula can be reduced to a finite number of scans.
- If $G \in$ CLIQUE, then there is a list of k interconnected nodes; given that list, we could easily verify that G contains all edges between them. The list contains k integers from 1 to the number of nodes in G (which is polynomial with respect to the size of G's representation) and requires a presumably quadratic or cubic time to be checked.
- If $G \in$ TSP, then there is a permutation of the nodes in G, i.e., a list of nodes. Given that list, we can easily sum the weights as in (3.2) and check that the inequality holds.

In other words, if we are provided a *certificate* (or *witness*), it is easy for us to check that a given string belongs to the language. What's important is that both the certificate's size and the time to check are polynomial with respect to the input size. The class of such problems is called **NP**. More formally:

Definition 17. *We say that a language $L \subseteq \Sigma^*$ is of class **NP**, and write $L \in$ **NP**, if there is a TM $\mathcal{M}$ and two polynomials $p(n)$ and $q(n)$ such that for every input string x*

$$x \in L \quad \Leftrightarrow \quad \exists c \in \Sigma^{q(|x|)} : \mathcal{M}(x, c) = 1 \wedge t_{\mathcal{M}}(x, c) \le p(|x|). \tag{3.4}$$

Basically, the two polynomials are needed to bound both the size of certificate c and the execution time of $\mathcal{M}$.

Observe that the definition only requires a (polynomially verifiable) certificate to exist only for "yes" answers, while "no" instances (i.e., strings x such that $x \notin L$) might not be verifiable.

3.3.1 Non-deterministic Turing Machines

An alternative definition of **NP** highlights the meaning of the class name, and will be very useful in the future.

Definition 18. *A non-deterministic Turing Machine (NDTM) is a TM with two different, independent transition functions. At each step, the NDTM makes an arbitrary choice as to which function to apply. Every sequence of choices defines a possible* computation *of the NDTM. We say that the NDTM* accepts *an input x if at least one computation (i.e., one of the possible arbitrary sequences of choices) terminates in an accepting state.*

There are many different ways of imagining a NDTM: one that flips a coin at each step, one that always makes the right choice towards acceptance, one that "doubles" at each step following both choices at once. Note that, while a normal, deterministic TM is a viable computational model, a NDTM is not, and has no correspondence to any current or envisionable computational device[7].

Alternate definitions might refer to machines with more than two choices, with a subset of choices for every input, and so on, but they are all functionally equivalent.

We can define the class NTIME(f) as the NDTM equivalent of class DTIME(f), just by replacing the TM in Definition 14 with a NDTM:

Definition 19. *Let $f : \mathbb{N} \to \mathbb{N}$ be any computable function. We say that a language $L \subseteq \Sigma^*$ is of class NTIME(f), and write $L \in$ NTIME(f), if there is a NDTM $\mathcal{M}$ that decides L and its worst-case time, as a function of input size, is dominated by f:*

$$L \in \mathit{NTIME}(f) \quad \Leftrightarrow \quad \exists \quad \mathit{NDTM} \quad \mathcal{N}: \quad L(\mathcal{N}) = L \wedge T_{\mathcal{N}} = O(f).$$

Indeed, the names "DTIME" and "NTIME" refer to the deterministic and non-deterministic reference machine. Also, the name **NP** means "non-deterministically polynomial (time)," as the following theorem implies by setting a clear parallel between the definition of **P** and **NP**:

Theorem 18.

$$\boldsymbol{NP} = \bigcup_{k=0}^{\infty} \mathit{NTIME}(n^k).$$

Proof. See also Theorem 2.6 in the online draft of Arora-Barak. We can prove the two inclusione separately.

Let $L \in$ **NP**, as in Definition 17. We can define a NDTM $\mathcal{N}$ that, given input x, starts by non-deterministically appending a certificate $c \in \Sigma^{q(|x|)}$: every computation generates a different certificate. After this non-deterministic part, we run the machine $\mathcal{M}$ from Definition 17 on the tape containing (x, c). If $x \in L$, then at least one computation has written the correct certificate, and thus ends in an accepting state. On the other hand, if $x \notin L$ then no certificate can end in acceptance. Therefore, $\mathcal{N}$ accepts x if and only if $x \in L$. The NDTM $\mathcal{N}$ performs $q(|x|)$ steps to write the (non-deterministic) certificate, followed by the $p(|x|)$ steps due to the execution of $\mathcal{M}$, and is therefore polynomial with respect to the input. Thus, $L \in \mathrm{NTIME}(n^k)$ for some $k \in \mathbb{N}$.

Conversely, let $L \in \mathrm{NTIME}(n^k)$ for some $k \in \mathbb{N}$. This means that x can be decided by a NDTM $\mathcal{N}$ in time $q(|x|) = O(|x|^k)$, during which it performs $q(|x|)$ arbitrary binary choices. Suppose that $x \in L$, then there is an accepting computation by $\mathcal{N}$. Let $c \in \{0,1\}^{q(|x|)}$ be the sequence of arbitrary choices done by the accepting computation of $\mathcal{N}(x)$. We can use c as a certificate in Definition 17, by creating a deterministic TM $\mathcal{M}$ that uses c to emulate $\mathcal{N}(x)$'s accepting computation by performing the correct choices at every step. If $x \notin L$, then no computation by $\mathcal{N}(x)$ ends by accepting the input, therefore all certificates fail, and $\mathcal{M}(x, c) = 0$ for every c. Thus, all conditions in Definition 17 hold, and $L \in$ **NP**. □

3.4 Reductions and hardness

Nobody knows if **NP** is a proper superset of **P**, yet. In order to better assess the problem, we need to set up a hierarchy within **NP** in order to identify, if possible, languages that are harder than others. To do this, we resort again to *reductions*.

Definition 20. *Given two languages $L, L' \in$ **NP**, we say that L is* polynomially reducible *to L', and we write $L \leq_p L'$, if there is a function $R : \Sigma^* \to \Sigma^*$ such that*

$$x \in L \quad \Leftrightarrow \quad R(x) \in L'$$

and R halts in polynomial time wrt $|x|$.

[7]Not even quantum computing, no matter what popular science magazines write.

In other words, R maps strings in L to strings in L' and strings that are not in L to strings that are not in L'. Note that we require R to be computable in polynomial time, i.e., there must be a polynomial $p(n)$ such that $R(x)$ is computed in at most $p(|x|)$ steps. If $L \leq_p L'$, we say that L' is at least as hard as L. In fact, if we have a procedure to decide L', we can apply it to decide also L with "just" a polynomial overhead due to the reduction.

3.4.1 Simple examples

Reductions between versions of SAT

Definition 21 (k-CNF)**.** *If all clauses of a CNF formula have at most k literals in them, then we say that the formula is k-CNF (conjunctive normal form with k-literal clauses).*

For instance, (3.3) is 4-CNF and, in general, k-CNF for all $k \geq 4$. It is not 3-CNF because it has some 4-literal clauses. Sometimes, the definition of k-CNF is stricter, and requires that every clause has *precisely* k literals. Nothing changes, since we can always write the same literal twice in order to fill the clause up.

Definition 22. *Given $k \in \mathbb{N}$, the language k-SAT is the set of all (encodings of) satisfiable $k-CNF$ formulas.*

Let us start with a "trivial" theorem:

Theorem 19. *Given $k \in \mathbb{N}$,*

$$k\text{-}SAT \leq_p SAT.$$

Proof. Define the reduction $R(x)$ as follows: given a string x, if it encodes a k-CNF formula, then leave it as it is; otherwise, return an unsatisfiable formula. □

The simple reduction takes into account the fact that k-SAT $\subseteq$ SAT, therefore if we are able to decide SAT, we can a fortiori decide k-SAT.

The following fact is less obvious:

Theorem 20.

$$SAT \leq_p 3\text{-}SAT.$$

Proof. Let f be a CNF formula. Suppose that f is not 3-CNF. Let clause i have $l_i > 3$ literals:

$$\bigvee_{j=1}^{l_i} g_{ij} \tag{3.5}$$

Let us introduce a new variable, h, and split the clause as follows,

$$\left(h \vee \bigvee_{j=1}^{l_i - 2} g_{ij} \right) \wedge (\neg h \vee g_{i,l_i-1} \vee g_{il_i}), \tag{3.6}$$

by keeping all literals, apart from the last two, in the first clause, and putting the last two in the second one. By construction, the truth assignments that satisfy (3.5) also satisfy (3.6), and viceversa. In fact, if (3.5) is satisfied then at least one of its literals are true; but then one of the two clauses of (3.6) is satisfied by the same literal, while the other can be satisfied by appropriately setting the value of the new variable h. Conversely, if both clauses in (3.6) are satisfied, then at least one of the literals in (3.5) is true, because the truth value of h alone cannot satisfy both clauses.

The step we just described transforms an l_i-literal clause into the conjunction of an $(l_i - 1)$-literal clause and a 3-literal clause which is satisfiable if and only if the original one was; by applying it recursively, we end up with a 3-CNF formula which is satisfiable if and only if the original f was. □

As an example, the 4-CNF formula (3.3) can be reduced to the following 3-CNF with the two additional variables h and k used to split its two 4-clauses:

$$\begin{aligned} f'(x_1,x_2,x_3,x_4,x_5,h,k) \quad = \quad & (h \vee x_1 \vee \neg x_2) \wedge (\neg h \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ & \wedge (k \vee \neg x_1 \vee \neg x_2) \wedge (\neg k \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \tag{3.7}$$

Theorem 20 is interesting because it asserts that a polynomial-time algorithm for 3-SAT would be enough for the more general problem. With the addition of Theorem 19, we can conclude that all k-SAT languages, for $k \geq 3$, are equivalent to each other and to the more general SAT.

On the other hand, it can be shown that 2-SAT $\in \mathbf{P}$.

Simple reductions between graph languages

We already met CLIQUE; a strictly related problem is the one of finding an independent set in the graph:

INDEPENDENT SET (or simply INDSET) — Given an encoding of graph G and a number k, does G contain k nodes that are all *disconnected* from each other[8]?

The problem is almost the same, but we require the vertex subset to have *no* edges (while CLIQUE requires the subset to have *all possible* edges). Clearly, INDSET instances can be transformed into equivalent INDSET instances by simply complementing the edge set, which can be attained by negating the graph's adjacency matrix, which is clearly a polynomial time procedure in the graph's size (indeed, linear). Therefore, we can write both

$$\text{CLIQUE} \leq_p \text{INDSET} \qquad \text{and} \qquad \text{INDSET} \leq_p \text{CLIQUE}.$$

3.4.2 Example: reducing 3-SAT to INDSET

Let us see an example of reduction between two problems coming from different domains: boolean logic and graphs.

Theorem 21.

$$\textit{3-SAT} \leq_p \textit{INDSET}.$$

Proof. Let f be a 3-CNF formula. We need to transform it into a graph G and an integer k such that G has an independent set of size k if and only if f is satisfiable.

Let us represent each of the m clauses in f as a separate triangle (i.e., three connected vertices) of G, and let us label each vertex of the triangle as one of the clause's literals. Therefore, G contains $3m$ vertices organized in m triangles.

Next, connect every vertex labeled as a variable to all vertices labeled as the corresponding negated variable: every vertex labeled "$x1$" must be connected to every vertex labeled "$\neg x_1$" and so on. Fig. 3.1 shows the graph corresponding to the 3-CNF formula (3.7): each bold-edged triangle corresponds to one of the six clauses, with every node labeled with one of the literals. The dashed edges connect every literal with its negations.

It is easy to see that the original 3-CNF formula is satisfiable if and only if the graph contains an independent set of size $k = m$ (number of clauses). Given the structure of the graph, no more than one node per triangle can appear in the independent set (nodes in the same triangle are not independent), and if a literal appears in the independent set, then its negation does not (they would be connected by an edge, thus not independent). If the independent set has size m, then we are ensured that one literal per clause can be made true without contradictions. As an example, the six green nodes in Fig. 3.1 form an independent set and correspond to a truth assignment that satisfies f. □

[8] `https://en.wikipedia.org/wiki/Independent_Set_(graph_theory)`

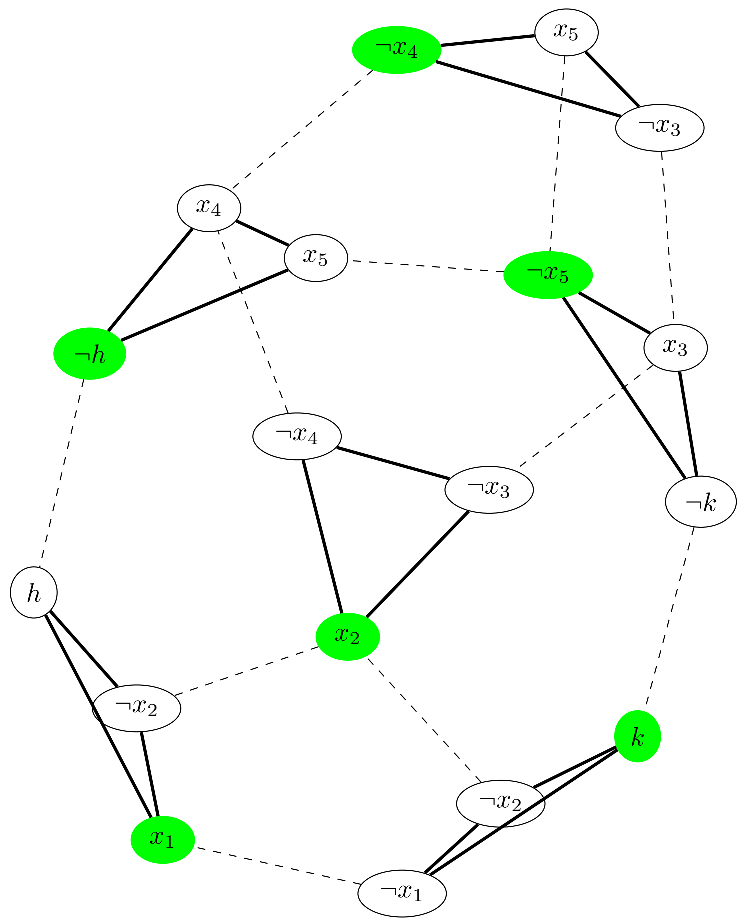


Figure 3.1: Reduction of the 3-CNF formula (3.7) to a graph for INDSET.

3.5 NP-hard and NP-complete languages

Definition 23. *A language L is said to be* **NP**-hard *if for every language $L' \in \textbf{\textit{NP}}$ we have that $L' \leq_p L$.*

In this Section we will show that **NP**-hard languages exist, and are indeed fairly common. The definition just says that **NP**-hard languages are "harder" (in the polynomial reduction sense) than any language in **NP**: if we were able to solve any **NP**-hard language in polynomial time then, by this definition, we would have a polynomial solution to all languages in **NP**.

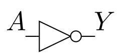
Furthermore, in this Section we shall see that the structure of **NP** is such that it is possible to identify a subset of languages that are "the hardest ones" within **NP**: we will call these languages ***NP**-complete*:

Definition 24. *A language $L \in \textbf{\textit{NP}}$ that is **NP**-hard is said to be* **NP**-complete*.*

In particular, we will show that SAT is **NP**-complete.

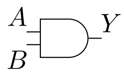
3.5.1 CNF and Boolean circuits

In order to prove the main objective of this part of the course, i.e. that SAT is NP-complete, we want to represent a computation of a NDTM as a CNF expression.



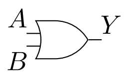
A	Y
1	0
0	1

$$\begin{aligned} & Y = \neg A \\ \equiv\ & (\neg Y \vee \neg A) \wedge (Y \vee A) \end{aligned}$$



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

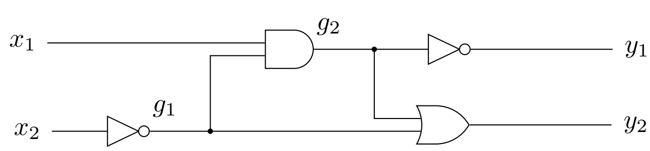
$$\begin{aligned} & Y = A \wedge B \\ \equiv\ & \big(\neg Y \vee (A \wedge B)\big) \wedge \big(Y \vee \neg(A \wedge B)\big) \\ \equiv\ & (\neg Y \vee A) \wedge (\neg Y \vee B) \wedge (Y \vee \neg A \vee \neg B) \end{aligned}$$



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

$$\begin{aligned} & Y = A \vee B \\ \equiv\ & \big(\neg Y \vee (A \vee B)\big) \wedge \big(Y \vee \neg(A \vee B)\big) \\ \equiv\ & (\neg Y \vee A \vee B) \wedge \big(Y \vee (\neg A \wedge \neg B)\big) \\ \equiv\ & (\neg Y \vee A \vee B) \wedge (Y \vee \neg A) \wedge (Y \vee \neg B) \end{aligned}$$

Figure 3.2: A NOT gate (top), an AND gate (middle) and an OR gate (bottom), their truth tables, and derivations of the CNF formulae that are satisfied if and only if their variables are in the correct relation (i.e., only by combinations of truth values shown in the corresponding table).



$$\begin{aligned} & (x_2 \vee g_1) \wedge (\neg x_2 \vee \neg g_1) \\ \wedge\ & (\neg g_2 \vee x_1) \wedge (\neg g_2 \vee g_1) \wedge (g_2 \vee \neg x_1 \vee \neg g_1) \\ \wedge\ & (y_1 \vee g_2) \wedge (\neg y_1 \vee \neg g_2) \\ \wedge\ & (\neg y_2 \vee g_1 \vee g_2) \wedge (y_2 \vee \neg g_1) \wedge (y_2 \vee \neg g_2) \end{aligned}$$

Figure 3.3: A Boolean circuit and its CNF representation: the CNF formula is satisfiable by precisely the combinations of truth values that are compatible with the logic gates.

A way to represent a Boolean formula as dependency of some outputs from some inputs is by means of a Boolean circuit, where logical connectives are replaced by gates. Fig. 3.2 shows the gates corresponding to the fundamental Boolean connectives, together with their truth tables and CNF formulae that are satisfiable by all truth assignments that are compatible with the gate.

We only consider *combinational* Boolean circuits, i.e., circuits that do not preserve states: there are no "feedback loops", and gates can be ordered so that every gate only receives inputs from previous gates in the order.

Any combinational Boolean circuit can be "translated" into a CNF formula, in the sense that the formula is satisfiable by all and only the combinations of truth values that satisfy the circuit. Given a Boolean circuit with n inputs $x_1, \dots, x_n$ and m outputs $y_1, \dots, y_m$ and l gates $G_1, \dots, G_l$:

- add one variable for every gate whose output is not an output of the whole circuit;
- once all gate inputs and outputs have been assigned a variable, write the conjunction of all CNF formulae related to all gates.

Fig. 3.3 shows an example: a Boolean circuit with 2 inputs, 2 outputs and 2 ancillary variables associated to intermediate gates, together with the corresponding CNF formula. This formula completely expresses the dependency between all variables in the circuit, and by replacing truth assignment we can use it to express various questions about the circuit in terms of satisfiability. For example:

1. Is there a truth assignment to inputs x_1, x_2 such that the outputs are both 0?
   We can reduce this question to SAT by replacing $y_1 = y_2 = 0$ (and, of course, $\neg y_1 = \neg y_2 = 1$) in the CNF of Fig. 3.3, and by simplifying we obtain

   $$(x_2 \vee g_1) \wedge (\neg x_2 \vee \neg g_1) \wedge (\neg g_2 \vee x_1) \wedge (\neg g_2 \vee g_1) \wedge (g_2 \vee \neg x_1 \vee \neg g_1) \wedge (g_2) \wedge (\neg g_1) \wedge (\neg g_2),$$

   which is clearly not satisfiable because of the conjunction $g_2 \wedge \neg g_2$.

2. If we fix $x_1 = 1$, is it possible (by assigning a value to the other input) to get $y_2 = 1$?
   To answer this let us replace $x_1 = y_2 = 1$ and $\neg x_1 = \neg y_2 = 0$ into the CNF and simplify:

   $$(x_2 \vee g_1) \wedge (\neg x_2 \vee \neg g_1) \wedge (\neg g_2 \vee g_1) \wedge (g_2 \vee \neg g_1) \wedge (y_1 \vee g_2) \wedge (\neg y_1 \vee \neg g_2) \wedge (g_1 \vee g_2).$$

   The formula is satisfiable by $x_2 = y_1 = 0$, $g_1 = g_2 = 1$, so the answer is "yes, just set the other input to 0".

Note that in this second case we can "polynomially" verify that the CNF is satisfiable by replacing the values provided in the text. In general, on the other hand, verifying the unsatisfiability of a CNF can be hard, because we cannot provide a certificate.

3.5.2 Using Boolean circuits to express Turing Machine computations

As an example, consider the following machine with 2 symbols (0,1) and 2 states plus the halting state, with the following transition table:

	0	1
s_1	$1, s_1, \rightarrow$	$1, s_2, \leftarrow$
s_2	$0, s_1, \leftarrow$	$0, \mathrm{HALT}, \rightarrow$

Suppose that we want to implement a Boolean circuit that, receiving the current tape symbol and state as an input, provides the new tape symbol, the next state and direction as an output. We can encode all inputs of this transition table in Boolean variables as follows:

- the input, being in $\{0, 1\}$, already has a canonical Boolean encoding, let us call it x_1;
- the two states can be encoded in a Boolean variable x_2 with an arbitrary mapping, for instance:

  $$0 \mapsto s_1, \qquad 1 \mapsto s_2.$$

The outputs, that encode the entries of the transition table can be similarly mapped:

- the new symbol on the tape is, again, a Boolean variable y_1;
- the new state requires two bits, because we need to encode the HALT state. Therefore, we will need an output y_2 that encodes the continuation states as before, and an output y_3 that is true when the machine must halt. Therefore, the mapping from y_2, y_3 to the new state is

  $$00 \mapsto s_1, \qquad 10 \mapsto s_2, \qquad 01 \mapsto \mathrm{HALT},$$

  with the combination $y_2 = y_3 = 1$ left unused;
- the direction is arbitrarily mapped on the output variable y_4 ,e.g.,

  $$0 \mapsto \leftarrow, \qquad 1 \mapsto \rightarrow .$$

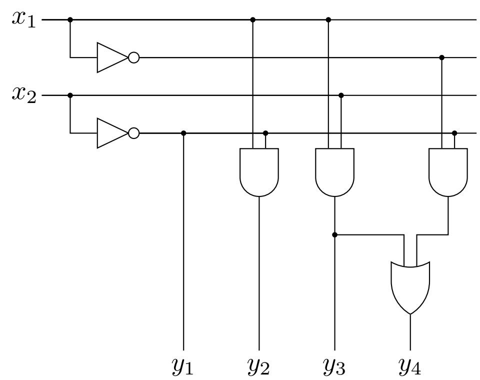


Figure 3.4: The Boolean circuit that implements the transition table of the TM described in the text.

Fig. 3.4 shows the Boolean circuit that outputs the new machine configuration (encodings of state, symbol and direction) based on the current (encoded state and symbol) pair.

The above example suggests that a step of a Turing machine can be executed by a circuit, and that by concatenating enough copies of this circuit we obtain a circuit that executes a whole TM computation:

Lemma 3. *Let $\mathcal{M}$ be a polynomial-time machine whose execution time on inputs of size n is bounded by polynomial $p(n)$. Then there is a polynomial $P(n)$ such that for every input size n there is a Boolean circuit C, whose size (in terms, e.g., of number of gates) bound by $P(n)$, that performs the computation of $\mathcal{M}$.*

Proof outline. Let $\mathcal{M}$ have $|Q| = m$ states. Let us fix the input size n. Then we know that $\mathcal{M}$ halts within $p(n)$ steps. Since every step changes the current position on the tape by one cell, the machine will never visit more than $2p(n) + 1$ cells (considering the two extreme cases of the machine always moving in the same direction). The complete configuration of the machine at a given point in time is therefore described by:

- $2p(n) + 1$ boolean variables (bits) to describe the content of the relevant portion of the tape;
- $|Q|$ bits to describe the state;
- $2p(n)+1$ bits to describe the current position on the tape (one of the bits is 1, the others are 0).

Of course, more compact representations are possible, e.g., by encoding states and positions in base-2 notation. By using building blocks such as the transition table circuit of Fig. 3.4, we can actually build a Boolean circuit C' that accepts as an input the configuration of $\mathcal{M}$ at a given step and outputs the new configuration; this circuit has a number of inputs, outputs and gates that are polynomial with respect to n.

By concatenating $p(n)$ copies of C' (see Fig. 3.5), we compute the evolution of $\mathcal{M}$ for enough steps to emulate the execution on any input of size n. By inserting the initial configuration on the left-hand side, the circuit outputs the final configuration.

If the size of every block C' is bound by polynomial $q(n)$, then the size of the whole circuit is bound by $P(n) = p(n) \cdot q(n)$, therefore it is still polynomial. □

Note that the proof is not complete: in particular, the size of C' is only suggested to be polynomial, but we would need to look much deeper in the structure of C' to be sure of that.

Lemma 4. *Lemma 3 also works if the TM is non-deterministic.*

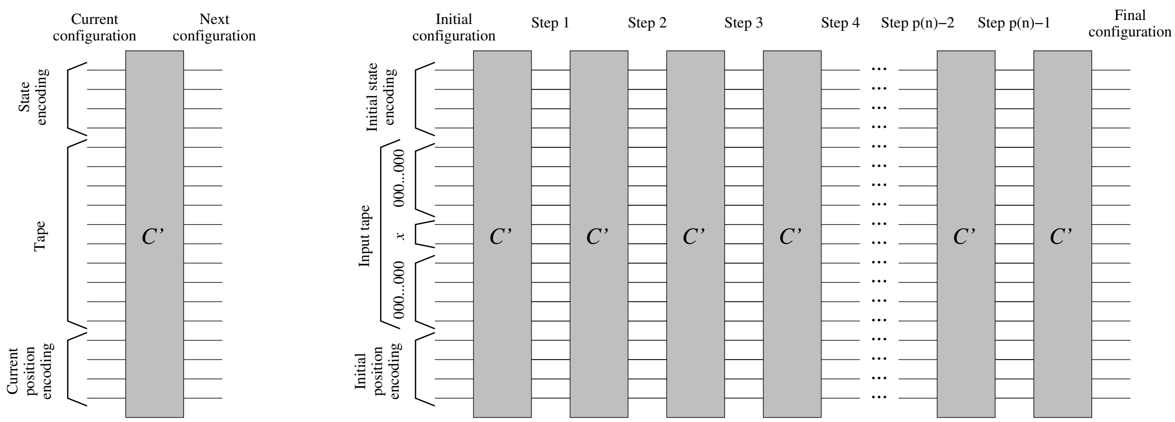


Figure 3.5: (left) C' is a Boolean circuit with a polynomial number of inputs, gates and outputs with respect to the size of the TM's input x. It transforms a Boolean representation of a configuration of the TM into the configuration of the subsequent step. (right) By concatenating $p(|x|)$ copies of C', we get a polynomial representation of the whole computation of the TM on input x.

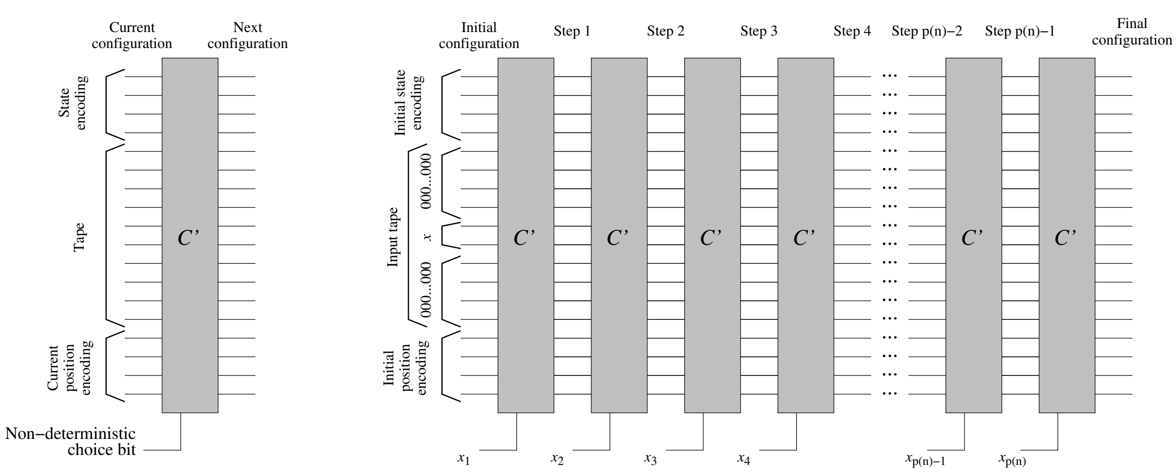


Figure 3.6: Analogous to Fig. 3.5 for a NDTM. (left) Every C' block has an additional input that allows the selection of the non-deterministic choice for the step that it controls. (right) The whole circuit has $p(n)$ additional Boolean inputs $x_1, \ldots, x_{p(n)}$: every combination of choice bits represents one of the $2^{p(n)}$ computations of the NDTM.

Proof outline. See Fig. 3.6: in order to carry on a NDTM's computation, we just need to modify the circuit C' of Lemma 3 to accept one further input, and use it to choose between the two possible outcomes of the transition table. Let us call $x_1, \dots, x_{p(n)}$ the additional input bits of the daisy-chained C' blocks. Each of the $2^{p(n)}$ combinations of these inputs determines one of the possible compuattions of the NDTM. $\square$

Knowing this, we can see how any polynomial computation of a NDTM can be represented by a CNF formula that is only satisfiable if the NDTM accepts its input.

Theorem 22 (Cook's Theorem)**.** *SAT is* ***NP****-hard.*

Proof outline. To prove this, we need to pick a generic language $L \in \mathbf{NP}$ and show that $L \leq_p \mathrm{SAT}$.

Let $\mathcal{N}$ be the NDTM that decides $x \in L$ within the polynomial time bound $p(|x|)$.

Let $x \in \Sigma^n$ be a string of length n. By Lemma 4, we can build a Boolean circuit C with polynomial size that, for any truth value combination of the inputs $x_1, \dots, x_{p(n)}$, performs one of the $2^{p(n)}$ computations of $\mathcal{N}$.

We can transform the Boolean circuit C into a (still polynomial-size) CNF formula f_C by means of the procedure outlined in Sec. 3.5.1.

At this point, the question whether $x \in L$ or not, which can be expressed as "is there at least one computation of $\mathcal{N}(x)$ that ends in an accepting state?", can be answered by assigning the proper truth values to some variables in f_C:

- the "initial state" inputs are set to the representation of the initial state;
- the "input tape" inputs are set to the representation of string x on $\mathcal{N}$'s tape;
- the "initial position" inputs are set to the representation of $\mathcal{N}$'s initial position on the tape;
- the variables corresponding to the "final state" outputs are set to the representation of the accepting halting state.

After simplifying for these preset values, the resulting CNF formula f'_C still has a lot of free variables, among which are the choice bits $x_1, \dots, x_{p(n)}$.

By construction, the CNF formula f'_C is satisfiable if and only if there is a computation of $\mathcal{N}$ that starts from the initial configuration with x on the tape and ends in an accepting state. Therefore,

$$x \in L \quad \leftrightarrow \quad f'_C \in \mathrm{SAT}.$$

$\square$

Of course, we already know that $\mathrm{SAT} \in \mathbf{NP}$, hence the following:

Corollary 2. *SAT is* ***NP****-complete.*

3.6 More NP-complete languages

NP-complete languages have an important role in complexity theory: they provide an upper bound for how hard can a language in **NP** be.

Since the composition of two polynomial-time reductions is still a polynomial-time reduction, we have the following:

Lemma 5. *If L is* ***NP****-hard and $L \leq_p L'$, then also L' is* ***NP****-hard too.*

So, whenever we reduce an **NP**-complete language to any other language $L \in \mathbf{NP}$, we can conclude that L' is **NP**-complete too.

From Theorem 20, and from the fact that $3\text{-SAT} \in \mathbf{NP}$, we get:

Lemma 6. *3-SAT is* ***NP****-complete.*

Next, from Theorem 21, and from the fact that INDSET $\in$ **NP**, we get:

Lemma 7. *INDSET is* ***NP****-complete.*

We have already established the equivalence between INDSET and CLIQUE, therefore

Lemma 8. *CLIQUE is* ***NP****-complete.*

Let us introduce a few more problems in **NP**.

VERTEX COVER — Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, is there a vertex subset $V' \subseteq V$ of size (at most) k such that every edge in E has at least one endpoint in V'?

INTEGER LINEAR PROGRAMMING (ILP) — Given a set of m linear inequalities with integer coefficients on n integer variables, is there at least a solution? In other terms, given $n \times m$ coefficients $a_{ij} \in \mathbb{Z}$ and m bounds $b_i \in \mathbb{Z}$, does the following set of inequalities

$$\begin{cases} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & \leq & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & \leq & b_m \end{cases}$$

have a solution with $x_1, \dots, x_n \in \mathbb{Z}$?

VERTEX COLORING — Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, is there an assignment from V to $\{1, \dots, k\}$ ("k colors") such that two connected vertices have different colors?

It is easy to see that all such languages are in **NP**.

Theorem 23. *VERTEX COVER is* ***NP****-complete.*

Proof. First of all, VERTEX COVER $\in$ **NP**, since a cover V' of size k is polynomially-sized wrt the problem instance and is verifiable in polynomial time.

Observe that if V' of size k is an independent set in $G = (V, E)$, then its complement $V \setminus V'$ is a vertex cover of size $|V| - k$ and viceversa. □

Here are a few slightly more complex reductions.

Theorem 24. *ILP is* ***NP****-complete.*

Proof. First, ILP is clearly in **NP**.

To prove **NP**-hardness, we reduce INDEPENDENT SET to ILP. Given a graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, we can set up some constraints such that we create an integer program whose solutions imply an independent set of size k for G and vice versa.

Let's create an integer program vith one variable per vertex in V. We want these variables to encode the inclusion of a vertex in the independent set V' ($x_i = 1$ if vertex i is in V, 0 otherwise). Since in INTEGER LINEAR PROGRAMMING all variables can be arbitrary integers, we restrict them between 0 and 1 by setting the inequalities $-x_i \leq 0$ and $x_i \leq 1$ for $i = 1, \dots, |V|$ (i.e., the inequality $0 \leq x_i \leq 1$ translated with only "$\leq$" signs with the x_i's to the left).

The requirement that $x_1, \dots, x_{|V|}$ is an independent set is implemented by introducing a constraint for every edge $i, j \in E$ that requires at most one of the endpoints to be 1: $x_i + x_i \leq 1$. Finally, the requirement that the size of the independent set is (at least) k is encoded in $x_1 + x_2 + \dots + x_{|V|} \geq k$, translated into a "$\leq$" inequality by changing all signs.

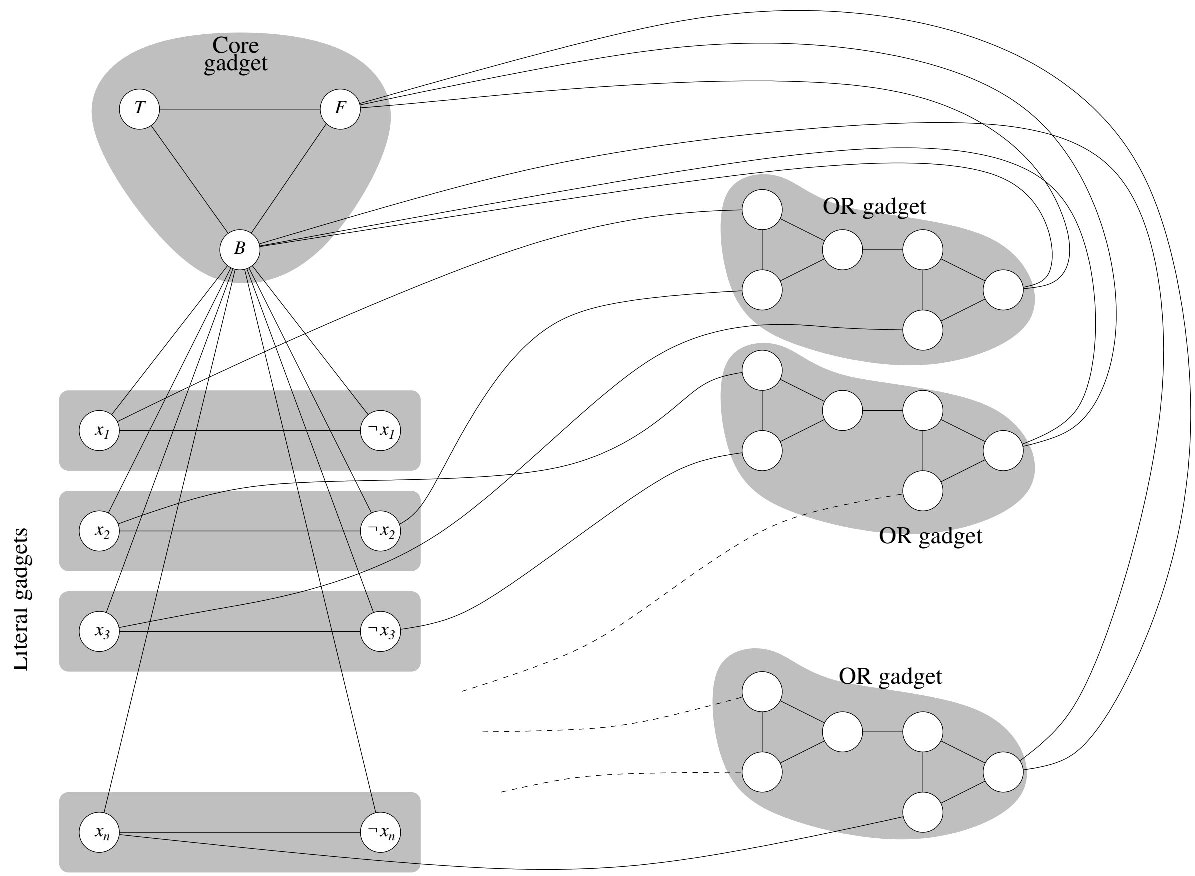


Figure 3.7: Reduction of a 3-CNF formula to the VERTEX COLORING problem with $k = 3$ colors.

In conclusion, the following integer program has a solution if and only if the corresponding graph has an independent set of size k:

$$\begin{cases} -x_i \leq 0 & \forall i \in V \\ x_i \leq 1 & \forall i \in V \\ x_i + x_j \leq 1 & \forall \{i,j\} \in E \\ -x_1 - \dots - x_{|V|} \leq -k & \end{cases}$$

□

Theorem 25. *VERTEX COLORING is* ***NP****-complete.*

Proof. Let's start from a 3-CNF formula f and build a graph that is 3-colorable if and only if f is satisfiable.

The graph will be composed of separate "gadgets" (subgraphs) that capture the semantics of a 3-CNF formula: the construction can be followed in Fig. 3.7.

The first gadget is a triangle whose nodes will be called T ("true"), F ("false") and B ("base"). Among the three colors, the one that will be assigned to node T will be considered to correspond to assigning the value "true" to a node. Same for F. The three nodes are used to "force" specific values upon other nodes of the graph.

The second set of gadgets is meant to assign a node to every literal in the formula. For every variable x_i, there will be two nodes, called x_i and $\neg x_i$. Since we are interested to assigning them truth values, we connect all of them to node B, so that they are forced to assume either the "true" or the "false" color. Furthermore, we connect node x_i to $\neg x_i$ to force them to take different colors.

Next, every 3-literal clause is represented by an OR gadget whose "exit" node is forced to have color "true" by being connected to B and to F. The three "entry" nodes of the gadget are connected to the nodes corresponding to the clause's literals. We can easily verify that every OR gadget is 3-colorable if and only if at least one of the literal nodes it is connected to is not false-colored.

By construction, if f is a satisfiable 3-CNF formula, then it is possible to color the literal nodes so that every OR gadget has at least one true-colored node at its input, and therefore the graph will be colorable. If, otherwise, f is not satisfiable, then every coloring of the literal nodes will result in an OR gadget connected to three false-colored literals, and therefore will not be colorable. □

A special mention goes to the GRAPH ISOMORPHISM language: given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are they isomorphic (e.g., is there a bijection $f : V_1 \to V_2$ such that $f(E_1) = E_2$)? Obviously, GRAPH ISOMORPHISM $\in$ **NP**: the bijection, if it exists, can be checked in polynomial time. However, it is believed that the language is not complete. In fact, there is a *quasi-polynomial*[9] algorithm that decides it.

3.7 An asymmetry in the definition of NP: the class coNP

Observe that the definition of **NP** introduces an asymmetry in acceptance and rejection that is reminiscing of the asymmetry between RE and coRE languages. Namely, while we require only one accepting computation to accept $x \in L$, in order to reject it we require that *all* computations reject it.

This means that, while $x \in L$ admits a polynomial certificate, and therefore is verifiable even by a deterministic polynomial checker, the opposite $x \notin L$ does not: there is no hope for a polynomial checker to become convinced that $x \notin L$.

Definition 25. *The symmetric class to* ***NP*** *is called* ***coNP****: the class of languages that have a polynomially verifiable certificate for strings that do* not *belong to the language.*

$$\boldsymbol{coNP} = \{L \subseteq \Sigma^* : \bar{L} \in \boldsymbol{NP}\}.$$

As an example, consider SAT: given a boolean CNF formula $f : \{0,1\}^n \to \{0,1\}$, is there a truth assignment that satisfies f? As far as we know, there is no way to produce a certificate that polynomially verifies a negative answer.

Given a CNF formula f, its negation is easily represented by a disjunction of conjunctive clauses (a formula in *Disjunctive Normal Form*, DNF), as we can perform a repeated application of De Morgan's laws (invert all $\vee$'s and $\wedge$'s, then negate all terms — remembering that double negations cancel out):

$$\begin{aligned}
\neg f(x_1, \dots, x_n) &= \neg \bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} g_{ij} \\
&= \bigvee_{i=1}^{m} \neg \bigvee_{j=1}^{l_i} g_{ij} \\
&= \bigvee_{i=1}^{m} \bigwedge_{j=1}^{l_i} \neg g_{ij}.
\end{aligned}$$

Asking whether f is satisfiable is equivalent to asking if $\neg f$ is a tautology (i.e., always satisfied), then reversing the answer. We can therefore define the language:

TAUTOLOGY — Given a Boolean DNF formula f, is f satisfied by *all* truth assignments?

[9] more than polynomial, but less than exponential, e.g., DTIME($2^{c_1 (\log n)^{c_2}}$)

Theorem 26. *TAUTOLOGY* $\in$ ***coNP***.

Proof. The above discussion is enough, since we have shown that $f \in \text{TAUTOLOGY} \Leftrightarrow \neg f \not\in \text{SAT}$; however, the discussion can be summarized in a more direct argument: if f is *not* a tautology, then it can be polynomially verified by a truth assignment that falsifies it, while clearly such truth assignment cannot be provided if f is always satisfied. □

We can define **coNP** in terms of non-deterministic TMs as follows. Remember that, up to now, we defined acceptance of input x by a NDTM $\mathcal{N}$ as the existence of at least one accepting computation in the trace of $\mathcal{N}(x)$. We call such a machine an *exixstential-mode* NDTM:

Definition 26. *An* existential-mode *NDTM is a non-deterministic TM* $\mathcal{N}^\exists$ *which is said to accept its input* x *iff the non-deterministic computation* $\mathcal{N}^\exists(x)$ *has at least one accepting branch*[10].
Conversely, a universal-mode *NDTM is a non-deterministic TM* $\mathcal{N}^\forall$ *which is said to accept its input* x *iff* all *branches of the non-deterministic computation* $\mathcal{N}^\forall(x)$ *are accepting.*

Therefore, a universal-mode NDTM $\mathcal{N}^\forall$ rejects an input x iff at least one branch of $\mathcal{N}^\forall(x)$ ends in rejection.

We now have a symmetrical landscape: definition 25 mirrors definition 17, and universal-mode NDTMs "mirror" existential-mode NDTMs, so that the following result can be proved by, again, mirroring the proof of Theorem 18:

Theorem 27. ***coNP*** *is the class of languages that are decided in polynomial time by some universal-mode NDTM.*

3.7.1 Relationship between P, NP and coNP

Clearly, $\mathbf{P} \subset \mathbf{NP} \cap \mathbf{coNP}$ because in **P** positive and negative answers are both polynomially verifiable. Currently, we don't know if the inclusion is strict or not.

Consider the decision version of the well-known integer factorization problem:

FACTORING — Given two positive integers $n, k \in \mathbb{N}$ does n have a prime factor $p \geq k$?

In other words, in order to have a decision problem we do not ask for the factor itself, we just compare it with a target value.

We do not know of any polynomial algorithm for integer factorization. Most numbers have small prime factors, and are easily decomposable, but some (namely, products of two large primes) are hard. Therefore, we cannot prove (yet) that FACTORING $\in \mathbf{P}$. However:

Theorem 28. *FACTORING* $\in$ ***NP*** $\cap$ ***coNP***.

Proof. Suppose we are given the input instance $n, k \in \mathbb{N}$. Observe that, if we assume the usual positional representation for integers (e.g., base-2 or base-10 representation), then the input size is $O(\log n + \log k)$; moreover, since the question makes sense only if $k < n$, then the input size is $O(\log n)$.

- If the answer is yes, meaning that there is a prime number $p \geq k$ that divides n, such prime is an acceptable polynomial certificate. To verify it:

    1. the size of p is smaller than the size of n (therefore, the certificate is polynomially-sized wrt the instance);
    2. check that p is actually prime (there is a polynomial-time algorithm for that),
    3. check that $p \geq k$,

[10] Again, this is what we usually refer to as NDTM, unless specified otherwise.

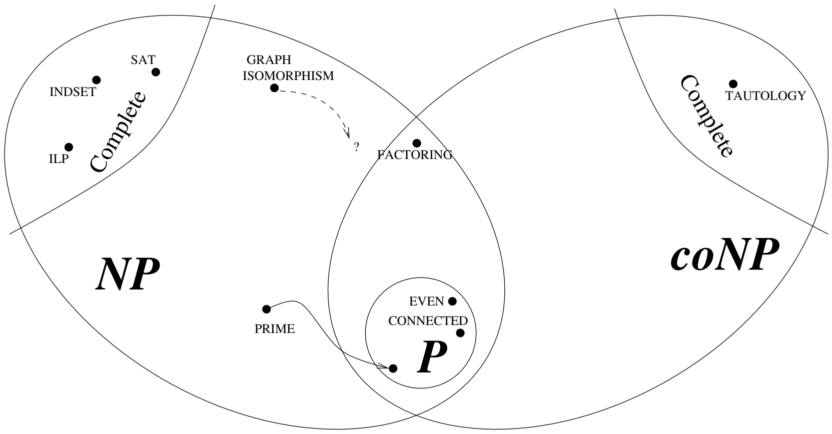


Figure 3.8: What we know up to now. If any of the **NP** or **coNP**-complete problems were to be proven in **P**, then all sets would collapse into it.

4. check that p divides n.

- If the answer is no, meaning that all prime factors of n are smaller than k, the list of all m prime factors $p_1, p_2, \ldots, p_m$ of n is a polynomial certificate for the negative answer. In fact, we can verify it as follows:

  1. the list is composed of $m \le \log_2 n$ numbers, each requiring $O(logn)$ symbols for its representation; the size of the list is therefore $O((\log n)^2)$, polynomial wrt the input size;
  2. check that all numbers in the list $p_1, p_2, \ldots, p_m$ are prime (again, we use $m \le \log_2 n$ applications of the polynomial-time primality algorithm);
  3. check that all $p_1, p_2, \ldots, p_m$ are less than k;
  4. check that $p_1 p_2 \ldots p_m = n$.

□

Fig. 3.8 summarizes what has been said in this chapter.

Chapter 4

Other complexity classes

Not all languages are **NP** or **coNP**. It is possible to define languages with higher and higher complexity.

4.1 The exponential time classes

It is possible to define classes that are analog to **P** and **NP** for exponential, rather than polynomial, time bounds:

Definition 27.

$$\boldsymbol{EXP} = \bigcup_{c=1}^{\infty} DTIME(2^{n^c}), \qquad \boldsymbol{NEXP} = \bigcup_{c=1}^{\infty} NTIME(2^{n^c}),$$

and, of course,

$$\boldsymbol{coNEXP} = \{L \subseteq \Sigma^* : \bar{L} \in \boldsymbol{NEXP}\}.$$

In short, **EXP** is the set of languages that are decidable by a deterministic Turing machine in exponential time (where "exponential" means a polynomial power of a constant, e.g., 2); **NEXP** is the same, but decidable by a NDTM. In other words, a language L is in **NEXP** when $x \in L$ iff there is an exponential-sized (wrt x) certificate verifiable in exponential time. Finally, **coNEXP** is the set of exponentially-disprovable languages.

Cominq up with languages that are in these classes, but not in **NP** or **coNP** is harder. One "natural" language is the equivalence of two regular expressions under specific limitations to their structure.

The following result should be immediate:

Lemma 9.

$$\boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{EXP} \subseteq \boldsymbol{NEXP}.$$

Proof. The only non trivial inclusion should be $\mathbf{NP} \subseteq \mathbf{EXP}$, but we just need to note that a non-deterministic machine with polynomial time bound can clearly be simulated by a deterministic machine in exponential time by performing all computations one after the other. □

Fig. 4.1 summarizes the addition of the exponential classes.

4.1.1 The "Restricted" Halting Problem

We already know (since Theorem 4) that the problem whether a deterministic TM $\mathcal{M}$ will halt on a given input x is in general undecidable. However, we also know that, due to the existence of universal

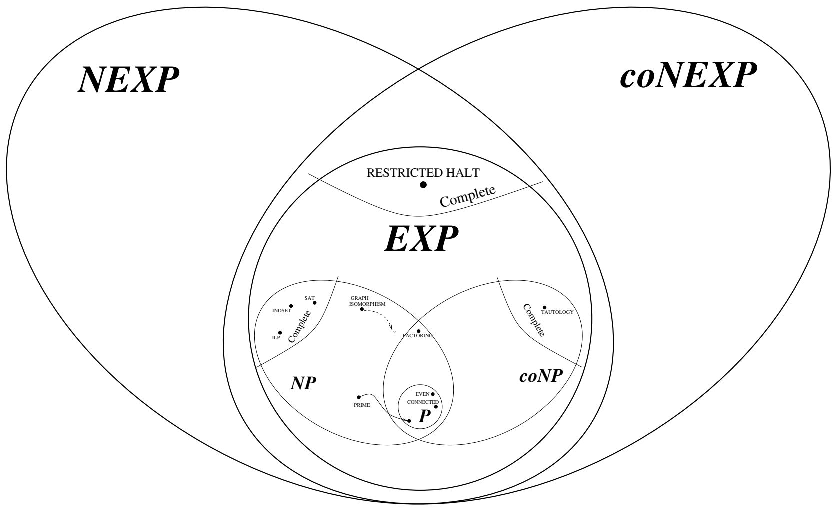


Figure 4.1: The exponential classes. The inner part is shown in greater detail in Fig. 3.8.

TMs, we are able to simulate any computation $\mathcal{M}(x)$ for an arbitrary number of steps. Thus, the following language is computable:

$$\text{RESTRICTED HALT} = \left\{ \begin{array}{lcr} (\mathcal{M}, x, t) & : & \mathcal{M} \text{ is a DTM} \wedge x \text{ is a string in } \mathcal{M}\text{'s alphabet} \\ & & \wedge T \in \mathbb{N} \wedge T_{\mathcal{M}}(x) \leq t \end{array} \right\}.$$

In other words, RESTRICTED HALT is the set of all TMs that halt on a given input within a given number of steps.

Then we can easily verify the following:

Theorem 29.

$$\textit{RESTRICTED HALT} \in \textbf{\textit{EXP}}.$$

Proof. In order to check whether the triplet $(\mathcal{M}, x, t)$ belongs to RESTRICTED HALT, we can emulate the computation of $\mathcal{M}(x)$ for at most t steps (less, if it halts sooner) using a universal TM $\mathcal{U}$.

Let m be the size of $\mathcal{M}$'s representation in $\mathcal{U}$, $n = |x|$ the size of the input string, and $s = O(\log t)$ the size of the representation of the number of steps, then the triplet $(\mathcal{M}, x, t)$ is represented as an input string of $O(m + n + s)$ symbols.

Since the emulation must be carried on for at most $t = 2^{O(s)}$ steps, each requiring possibly some scans of $\mathcal{M}$'s representation, then the whole simulation will take time $O(m \cdot 2^{ks})$ for some constant k, therefore requiring exponential time with respect to the input size. □

In order to better understand this proof, note that the number of steps that we need to emulate is, of course, linear with respect to the *value* represented by the input (t), but it is *exponential* with respect to the number s of symbols required to represent t on the tape. We have already discussed this issue in Section 3.2.1 where we named these problems "pseudo-polynomial".

It is very unlikely that RESTRICTED HALT $\in$ **NP**: for that, we would need a polynomial-size certificate that allows us to "skip" the exponential number t of simulated steps required to prove that $\mathcal{M}(x)$ halts within time t.

In fact, we can see that RESTRICTED HALT is "the hardest" language in its class:

Theorem 30. *RESTRICTED HALT is* ***EXP****-complete (with respect to polynomial-time reductions).*

Sketch of the proof. We need to prove prove that every language in **EXP** has a polynomial-time reduction to RESTRICTED HALT.

Let $L \in \mathbf{EXP}$ be one such generic language, and let $\mathcal{M}_L$ be an exponential-time TM that decides L. In particular, let p be a polynomial such that $\mathcal{M}_L(x)$ halts within $2^{p(|x|)}$ steps for every input string x. We can tweak $\mathcal{M}_L$ into a new machine $\mathcal{M}'_L$ that, instead of rejecting, runs forever (we turn a machine that "decides" L into a new one that merely "accepts" it):

$\mathcal{M}'_L$ **on input** x:
if $\mathcal{M}(x)$ **accepts**
  then accept
  else run forever

Therefore, we have created a machine $\mathcal{M}'_L$ that halts within $t = 2^{p(|x|)}$ steps if and only if the original machine halted in an accepting state, otherwise won't halt within that time limit:

$$x \in L \qquad \Leftrightarrow \qquad (\mathcal{M}'_L, x, 2^{p(|x|)}) \in \text{RESTRICTED HALT}.$$

Since the reduction $x \mapsto (\mathcal{M}'_L, x, 2^{p(|x|)})$ can be computed in polynomial time with respect to $|x|$, we have

$$L \leq_p \text{RESTRICTED HALT}.$$

□

4.2 Space complexity classes

Up to this point, we considered time (expressed as the number of TM transitions) as the only valuable resource. Still, one may envision cases in which space constraints are more important. In order to provide a significant definition of space, we need to just consider *additional* space with respect to the input. In this Section we will use Turing machines with at least two tapes, the first one being a read-only tape containing the input string, which won't count towards space occupation.

Definition 28. *Given a computable function $f : \mathbb{N} \to \mathbb{N}$, DSPACE$\bigl(f(n)\bigr)$ is the class of languages L that are decidable in space bounded by $O\bigl(f(|x|)\bigr)$, where n is the size of the input; i.e., $L \in$ DSPACE$\bigl(f(n)\bigr)$ if there is a multi-tape TM $\mathcal{M}$, with a read-only input tape, such that $\mathcal{M}$ decides $x \in L$ by using $O\bigl(f(|x|)\bigr)$ cells in the read/write tape(s).*

Note that, since we exclude the input tape from the computation, we allow for space complexities that are less than linear, such as DSPACE(1) or DSPACE($\log n$). This contrasts with time complexity classes which assume at least linear time because of the time needed to read the input.

We can introduce the equivalent non-deterministic class:

Definition 29. *Given a computable function $f : \mathbb{N} \to \mathbb{N}$, $L \in$ NSPACE$\bigl(f(n)\bigr)$ if there is a multi-tape non-deterministic TM $\mathcal{N}$, with a read-only input tape, such that $\mathcal{N}$ decides $x \in L$ by using $O\bigl(f(|x|)\bigr)$ cells in the read/write tape(s).*

4.2.1 Logarithmic space classes: L and NL

Definition 30.

$$\boldsymbol{L} = \mathit{DSPACE}(\log n)$$

is the class of languages that are decidable by a deterministic TM using logarithmic read-write space;

$$\boldsymbol{NL} = \mathit{NSPACE}(\log n)$$

is the same if non-deterministic compuattions are allowed.

Note that if the input encodes a data structure, such as a graph or a Boolean formula, then a counter or a pointer referring to it has size $O(\log n)$ (in order to write numbers up to n we need $O(\log n)$ symbols), therefore **L** contains all languages decidable by a constant number of pointers/counters.

Observe that if space is bounded by $c \log n$, then the machine can have at most $O(2^{c\log n}) = O(n^c)$ configurations, and therefore it must halt within that number of steps. Therefore

Theorem 31.

$$\boldsymbol{L} \subseteq \boldsymbol{P}, \qquad \boldsymbol{NL} \subseteq \boldsymbol{NP}.$$

Examples

The language

$$\text{POWER OF TWO} = \{1^{2^i} : i \in \mathbb{N}\}$$

of sequences of ones whose length is a power of two is in **L**. In fact, in order to determine the length of a string we just need a counter, whose size if logarithmic with respect to the input string.

Definition 31. *A triplet composed of a* directed *graph $G = (V, E)$ and two nodes $s, t \in V$ belongs to the CONNECTIVITY (or ST-CONNECTIVITY, or STCON) language if there is a path in G from s to t.*

Note that the definition is about a directed graph, and it requires a path from a specified source node s to a specified target node t.

Observe that a non-deterministic TM can simply keep in its working tape a "current" node (initially s), and non-deterministically jump from the current node to any connected node following the graph's adjacency matrix:

on input $G = (V, E)$; $s, t \in V$
- `current` $\leftarrow s$ — *"current" is a counter on the working tape*
- **repeat** $|V|$ **times**
  - **if** `current` $= t$
    - **then accept** and **halt**
  - **non deterministically** — *here the computation splits among all adjacent nodes*
    - `current` $\leftarrow$ a node adjacent to `current`
- **reject** and **halt**

If there is a path from s to t, then one of the computations will be lucky enough to follow it and terminate in an accepting state within $|V|$ computations; otherwise, no computation will be able to reach t and all will terminate in a rejecting state after $|V|$ iterations. Note that an actual NDTM implementation will require space for a current node, an iteration counter and possibly some auxiliary variables which all need to contain numbers from 1 to $|V|$. Therefore, the amount of space needed is bounded by $c \cdot \log |V|$. Since the input must contain an adjacency matrix, which is quadratic with respect to $|V|$, its size is $|x| = O(|V|^2)$. Therefore,

Theorem 32.

$$\mathit{STCON} \in \boldsymbol{NL}.$$

Any known efficient deterministic algorithm for STCON requires linear space (we have to maintain a queue of nodes, or at least be able to mark nodes as visited). While we don't conclusively know if STCON also belongs to **L**, we can prove the following:

Theorem 33.

$$\mathit{STCON} \in \mathit{DSPACE}\big((\log n)^2\big).$$

Proof. The following algorithm only requires $(\log n)^2$ space, even though it is extremely inefficient in terms of time:

on input $G = (V, E)$; $s, t \in V$

```
path_exists ← function (v, w, l)
    if l = 0
        return false                                    No more steps allowed.
    if l = 1                                            If only one step remains,
        return (v, w) ∈ E                               either v points directly to w, or nothing.
    for all v′ ∈ V
        if path_exists(v,v',⌊l/2⌋) ∧ path_exists(v',w,⌈l/2⌉)
            return true
    return false
if path_exists (s, t, |V|)
    accept and halt
else
    reject and halt
```

The function path_exists tells us if there is a path from the generic node $v \in V$ to the generic node $w \in V$ having length at most l. It is recursive: in the base cases, it tests if v and w are directly connected or are the same node (in which case the path obviously exists). Otherwise, the following property is true: *if a path of length l exists from v to w, then we can find a node v' in the middle of it*, in the sense that the paths from v to v' and from v' to w have length $l/2$ (give or take one if l is odd). The function searches for this middle node v' by iterating through all nodes in V; this is extremely inefficient in terms of time, but it allows the application of a divide-et-impera strategy that keeps the recursion depth to $\log l$.

Since the function requires only a constant number of variables to work, each of size $O(\log |V|)$, and that the call depth, starting from $l = |V|$, is again $O(\log |V|)$, remembering that the input size is $n = O(|V|^2)$, the conclusion follows. □

Observe that the proof outline doesn't explicitly define a TM; however, a recursive call stack can be stored in a TM tape as contiguous blocks of cells.

Another way to understand the algorithm in the proof above is the following: if we replaced the recursive calls with the following pair, we would obtain the usual step-by-step path search (albeit still rather inefficient):

$$\texttt{path\_exists}(v,v',1) \wedge \texttt{path\_exists}(v',w,l-1).$$

While Savitch's solution requires a large number of steps, but is able to keep the recursive depth logarithmic with respect to the problem size, the step-by-step alternative takes much fewer steps, but the recursion depth is linear[1].

4.2.2 Polynomial space: PSPACE and NPSPACE

As we did for **P** and **NP**,

Definition 32.

$$\begin{aligned}\textbf{\textit{PSPACE}} &= \bigcup_{c=0}^{\infty} \mathit{DSPACE}(n^c),\\ \textbf{\textit{NPSPACE}} &= \bigcup_{c=0}^{\infty} \mathit{NSPACE}(n^c).\end{aligned}$$

[1]See `https://comp3.eu/savitch.py` for a small python script that lets you compare the two approaches

The following inequalities should be obvious enough: **PSPACE** $\subseteq$ **NPSPACE** (as determinism is a special case of nondeterminism), **P** $\subseteq$ **PSPACE** (as having polynomial time allows us to touch at most a polynomial chunk of tape), **NP** $\subseteq$ **NPSPACE** (same reason).

A very important result shows that nondeterminism is less important for space-bounded computations: renouncing nondeterminism causes at most a quadratic loss.

Theorem 34 (Savitch's theorem)**.** *Given a function $f(n)$,*

$$\mathit{NSPACE}\big(f(n)\big) \subseteq \mathit{DSPACE}\big(f(n)^2\big).$$

Proof. Consider a language $L \in \text{NSPACE}\big(f(n)\big)$ and a generic input x. Then there is a NDTM $\mathcal{N}$ that decides $x \in L$ by using at most $O\big(f(|x|)\big)$ tape cells. The number of different configurations of the machine is therefore bounded by $N_c = 2^{O(f(|x|))}$. Let us consider the directed graph $G = (V, E)$ having all possible N_c configurations as the set V of nodes, and with an edge $(c_1, c_2) \in E$ if there is a transition rule in the $NDTM$ that allows transition from c_1 to c_2. Every path in G represents a possible computation of the machine, starting from an arbitrary configuration.

Let us call $s \in V$ the initial configuration of $\mathcal{N}$ with input x. Obviously, $x \in L$ if and only if there is an accepting computation of $\mathcal{N}$ with input x, i.e., if and only if there is a path in G from s to an accepting configuration. Let us add a new node t to V, and an edge from every accepting state to t. At this point, $x \in L$ if and only if there is a path from s to t in G, therefore if and only if $(G, s, t) \in$ STCON.

From Theorem 33, this STCON problem can be decided in space $O\big((\log N_c)^2\big) = O\Big(\big(\log 2^{O(f(|x|))}\big)^2\Big) = O\big(f(|x|)^2\big)$. □

This is an immediate consequence:

Corollary 3.

$$\boldsymbol{PSPACE} = \boldsymbol{NPSPACE}.$$

Proof.

$$\textbf{PSPACE} \subseteq \textbf{NPSPACE} = \bigcup_{c=0}^{\infty} \text{NSPACE}(n^c) \subseteq \bigcup_{c=0}^{\infty} \text{DSPACE}(n^{2c}) = \textbf{PSPACE}.$$

□

4.3 Randomized complexity classes

Observe that the definition of **NP** just requires one non-deterministic computation out of exponentially many to accept the input. Although only one computation might be accepting, there might be better cases in which we are guaranteed that a given fraction of the computations accept the input (if it belongs to the language).

4.3.1 The classes RP and coRP

Ley us define the following complexity class:

Definition 33. *Let $L \in$ **NP**, and let $0 < \varepsilon < 1$. We say that L is* randomized polynomial time*, and write $L \in$ **RP**, if there is a NDTM $\mathcal{M}$ that decides L in polynomial time and, whenever $x \in L$,*

$$\frac{\textit{Number of accepting computations of } \mathcal{M}(x)}{\textit{Number of computations of } \mathcal{M}(x)} \geq \varepsilon. \tag{4.1}$$

Obviously, if $x \notin L$ then there are no accepting computations. In other words, if $L \in \mathbf{RP}$ we are guaranteed that, whenever $x \in L$, a sizable number of computations accept it[2].

Theorem 35.

$$\boldsymbol{P} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{NP}.$$

Proof. The second inclusion derives from the definition; for the first one, just observe that a deterministic machine can be seen as a NDTM where all computation coincide, therefore either all computations accept (and the bound (4.1) is satisfied) or all reject. □

Equivalently, if we define **NP** in terms of a deterministic TM $\mathcal{M}$ and polynomial-size certificates $c \in \{0,1\}^{p(|x|)}$, we can define $L \in \mathbf{RP}$ if

$$\frac{|\{c \in \{0,1\}^{p(|x|)} : \mathcal{M}(x,c) = 1\}|}{2^{p(|x|)}} \geq \varepsilon.$$

We can see this definition in terms of probability of acceptance: suppose that $x \in L$, and let us generate a random certificate c. Then, $\Pr\bigl(\mathcal{M}(x,c) = 1\bigr) \geq \varepsilon$. Conversely, if $x \notin L$ then $\Pr\bigl(\mathcal{M}(x,c) = 1\bigr) = 0$, because x has no acceptance certificates.

This fact suggests a method to improve the probability of acceptance at will:

```
on input x
  repeat N times
    c ← random certificate in {0,1}^p(|x|)
    if M(x,c) = 1
      then accept and halt
  reject and halt
```

In other words, if the machine keeps rejecting x for many certificates, keep trying for N times, where N is an adjustable parameter.

The probability that, given $x \in L$ the machine rejects it N times (and therefore x is finally rejected) is

$$\Pr(\text{REJECT } x | x \in L) \leq (1 - \varepsilon)^N.$$

Therefore, by increasing the number N of repetitions, the probability of an error (rejecting x even though $x \in L$) can be made arbitrarily small. Of course, the opposite error (accepting x when $x \notin L$) is not possible because if $x \notin L$ there are no accepting certificates.

This results suggests that the definition of **RP** does not depend on the actual value of ε, as long as it is strictly included between 0 and 1. Observe, in fact, that if $\varepsilon = 0$ then we are not setting any lower bound on the number of accepting computation, and therefore the definition would coincide with that of **NP**, while if $\varepsilon = 1$ then we would require that all computations are accepting, thus rendering the certificate useless, and we would be redefining **P**.

As is customary with classes that are asymmetrical wrt acceptance/rejection mechanisms, we can also define its complementary class **coRP** as the class of languages whose complements are in **RP**, i.e., languages that have an NDTM whose computations always accept x whenever $x \in L$ and such that at least a fraction ε of computations reject x if $x \notin L$.

Examples

There are very few "natural" examples of languages in **RP** (or **coRP**) that do not belong to **P** too[3]

[2]See the Arora-Barak draft, Chapter 7, until Section 7.1 included, for definitions based on "probabilistic Turing Machines"

[3]What constituted the main example, PRIMES, is now proved to be in **P**.

Definition 34. *Let* $f : \{0,1\}^n \to \{0,1\}$ *be a computable function mapping* n*-bit strings onto a one-bit value. Then* f *is* constant *if it has the same value for all inputs:*

$$\forall x, y \quad f(x) = f(y),$$

while f *is* balanced *if it takes both values with equal frequency:*

$$|\{x : f(x) = 0\}| = ||\{x : f(x) = 1\}| = 2^{n-1}.$$

Suppose that we are promised that a function f is either constant or balanced. Then, the problem BALANCED FUNCTION is the problem of deciding if f is balanced and it is, in principle, exponential wrt n by virtue of the following algorithm:

on input f
- $f_0 \leftarrow f(0)$
- **for** $x \leftarrow \{1, \dots, 2^{n-1}\}$
  - **if** $f(x) \neq f_0$
    - **then accept** and **halt**
- **reject** and **halt**

In fact, in the worst case we could discover that f is not constant only after evaluating it on half of its possible 2^n input values (remember that if f is not constant then it is necessarily balanced).

The following non-deterministic algorithm, on the other hand, is clearly polynomial:

on input f
- Non-deterministically choose x,$y \in \{0, \dots, 2^n - 1\}$
- **if** $f(x) = f(y)$
  - **then reject**
  - **else accept**

However, observe that we are already assured that the function f is either constant or balanced; therefore, if the non-deterministic algorithm accepts (i.e., at least one of its non-deterministic choices leads to acceptance), then it does so with exactly 50% of its computations. If we make some assumptions on the input size (the algorithm decides a function f: let's assume that both f's representation on the machine's tape and f's execution time are polynomial wrt n), then the algorithm clearly satisfies the rules for **RP**[4].

Definition 35. *Let* $P = \mathbb{Z}/p\mathbb{Z}[x_1, \dots, x_n]$ *be the ring of polynomials on the finite field* $\mathbb{Z}/p\mathbb{Z}$ *(q prime). Suppose that* $f \in P$ *is expressed as a product of low-degree polynomials, e.g.:*

$$f(x_1, \dots, x_n) = (2x_1 + x_2 - 3x_4 + 2x_6 - 1) \cdot (5x_2 + 4x_3 + x_6 + 2) \cdots (x_2 + 4x_5 + x_{n-1} - 3x_n - 5). \quad (4.2)$$

The Polynomial Identity Testing problem *(PIT) is the problem of determining whether* f *is the zero polynomial or not. In out usual notation,*

$$f \in PIT \quad \leftrightarrow \quad f \equiv 0.$$

Observe that PIT could be decided by writing the polynomial f in canonical form (as a sum of monomials in $x_1, \dots, x_n$) and verifying that all coefficients are zero. However, transforming the form (4.2) into the canonical form would require an exponential number of multiplications and sums.

The algorithms to decide PIT rely[5] on evaluating f at a number of random points: if any evaluation gives a non-zero value, then f is obviously non-zero; otherwise, there is a (provably bounded) probability of error. In other words, these algorithms always accept f if $f \in$ PIT, but might also accept $f \notin$ PIT with probability bound by a constant ε, which is precisely the definition of **coRP**.

[4] Although this looks like an artificial problem, it is important because it is one of the earliest examples of languages for which quantum machines have an exponential advantage on classical ones, see `https://en.wikipedia.org/wiki/Deutsch-Jozsa_algorithm`

[5] See `https://en.wikipedia.org/wiki/Polynomial_identity_testing` and `https://en.wikipedia.org/wiki/Schwartz-Zippel_lemma` if interested; the PIT problem is also described in Arora-Barak (draft), Section 7.2.2.

4.3.2 Zero error probability: the class ZPP

An interesting characterization of **RP** and **coRP** is the following:

- $L \in$ **RP** means that there is a machine that, upon random generation of a certificate, never reports false positives (i.e., it only accepts x when $x \in L$), and reports false negatives with probability at most $1 - \varepsilon$;
- $L \in$ **coRP** means that there is a machine that, upon random generation of a certificate, never reports false negatives (i.e., it only rejects x when $x \notin L$), and reports false positives with probability at most $1 - \varepsilon$.

If a language L belongs to both **RP** and **coRP**, then it can benefit of both properties. In other words, if $L \in \mathbf{RP} \cap \mathbf{coRP}$, then there are two polynomial-time TMs M_1 and M_2 and two probability bounds $0 < \varepsilon_1, \varepsilon_2 < 1$ such that

$$\forall x \in \Sigma^* \quad \forall c \in \{0,1\}^{p(|x|)} \qquad \Pr\bigl(M_1(x,c) \text{ accepts}\bigr) \text{ is } \begin{cases} 0 & \text{if } x \notin L \\ \geq \varepsilon_1 & \text{if } x \in L \end{cases} \tag{4.3}$$

and

$$\forall x \in \Sigma^* \quad \forall c \in \{0,1\}^{p(|x|)} \qquad \Pr\bigl(M_2(x,c) \text{ rejects}\bigr) \text{ is } \begin{cases} 0 & \text{if } x \in L \\ \geq \varepsilon_2 & \text{if } x \notin L. \end{cases} \tag{4.4}$$

We can exploit these two machines with the following algorithm:

on input x
  repeat
    $c \leftarrow$ random certificate in $\{0,1\}^{p(|x|)}$
    if $\mathcal{M}_1(x,c)$ accepts
      then accept and **halt**
    if $\mathcal{M}_2(x,c)$ rejects
      then reject and **halt**

Observe that this algorithm does not define an explicit number of iterations. However, if $x \in L$, at every iteration M_1 has probability ε_1 to accept it, after which the algorithm would stop; conversely, if $x \notin L$, at every iteration M_2 has probability ε_2 to reject it, after which the algorithm would stop. with a rejection. If M_1 rejects or M_2 accepts, we know they they might be wrong and just move on with a new certificate. Therefore, the algorithm will eventually halt, and will always halt with the correct answer.

Suppose that $x \in L$: observe that the number of iterations before halting is distributed as a geometric random variable

$$\Pr(\text{the algorithm makes } n \text{ iterations}) = (1-\varepsilon_1)^{n-1}\varepsilon_1,$$

whose mean value, representing the expected number of iterations before halting, is

$$E[\text{iterations before halting}] = \frac{1}{\varepsilon_1},$$

which does not depend on anything but the error probability. The same considerations are valid if $x \notin L$.

Definition 36. ***ZPP*** $=$ ***RP*** $\cap$ ***coRP*** *is the class of problems that admit an algorithm that always gives a correct answer and whose expected execution time is polynomial with respect to the input size.*

The following result should be obvious, given the above definition:

Theorem 36.

$$\boldsymbol{P} \subseteq \boldsymbol{ZPP} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{NP}.$$

Proof. The proof is left as an exercise (exercise 11). □

4.3.3 Symmetric probability bounds: classes BPP and PP

Observe that the probabilistic classes shown up to this point are not very realistic: they require an algorithm that never fails for at least one of the two possible answers. Let us define a class that takes into account errors in both senses.

Definition 37. *A language L is said to be* bounded-error probabilistic polynomial*, written $L \in \boldsymbol{BPP}$, if there is a NDTM $\mathcal{N}$ running in polynomial time with respect to the input size, such that:*

- *if $x \in L$, then at least 2/3 of all computations accept;*
- *if $x \notin L$, then at most 1/3 of all computations accept (i.e., at least 2/3 of all computations reject).*

In other words, a language is **BPP** if it can be decided by a *qualified* majority of computations of a NDTM. We say that the probability of error is "bounded" precisely because there is a wide margin between the acceptance rate in the two cases.

As usual, the algorithm that emulates the NDTM is built as follows by using the deterministic machine $\mathcal{M}$ that emulates $\mathcal{N}$ via certificates:

on input x
- $n \leftarrow 0$
- **repeat** N **times**
  - $c \leftarrow$ random certificate in $\{0,1\}^{p(|x|)}$
  - **if** $\mathcal{M}(x,c)$ accepts
    - **then** $n \leftarrow n+1$
- **if** $n > N/2$
  - **then accept**
  - else **reject**

By making N higher and higher, the probability of error can be reduced at will.

Notice that the 1/3 and 2/3 acceptance thresholds are arbitrary. We just need to have a qualified majority, so an equivalent definition can be given by using any $\varepsilon > 0$ and requiring that the probability of a correct vote (the fraction of correct computations) is greater than $(1/2) + \varepsilon$. In other words, any non-zero separation between the frequencies in the positive and negative case is fine, and provides the same space.

If, on the other hand, we accept *simple majority* votes, then the results are not so nice.

Definition 38. *A language L is said to be* Probabilistic polynomial*, written $L \in \boldsymbol{PP}$, if there is a NDTM $\mathcal{N}$ running in polynomial time with respect to the input size, such that:*

- *if $x \in L$, then at least half of all computations accept;*
- *if $x \notin L$, then at most half of all computations accept (i.e., at least half of all computations reject).*

If the frequency of errors can approach 1/2, then the majority might be attained by one computation out of exponentially many, and reaching a predefined confidence level might require an exponential number of repetition (N in the "algorithm" above might not be constant, rether it could be exponential wrt $|x|$).

Given the above definitions, the following theorem should be obvious:

Theorem 37.

$$\boldsymbol{RP} \subseteq \boldsymbol{BPP} \subseteq \boldsymbol{PP}.$$

Proof. The proof is left as an exercise (exercises 12 and 13). □

The class **BPP** is considered the largest class of "practically solvable" problems, since languages in **BPP** have a polynomial algorithm that, although probabilistic, guarantees an error as small as desired.

No relationship between **NP** and **BPP** is known: it is *unlikely* that **NP** $\subseteq$ **BPP**, because it would imply that all **NP** problems have a satisfactorily probabilistic answer (i.e., heuristics that work very well in all cases); however, the opposite may or may not be the case.

To highlight the impractical nature of **PP**, it is sufficient to show that it contains far too many languages, in particular:

Theorem 38. ***PP*** $\supseteq$ ***NP***

Proof. Let us just take the **NP** $-$ *complete* language SATISFIABILITY and provide a machine that accepts satisfiable CNF formulas by simple majority:

on input f: n-variable CNF formula
- $(x_1, \dots, x_n) \leftarrow$ random truth assignment
- **if** $f(x_1, \dots, x_n)$
  - **then accept**
  - **else accept** with probability $\frac{1}{2} - \frac{1}{2^{n+1}}$, **reject** otherwise

We can easily check that if f is not satisfiable, then the above algorithm accepts or rejects the input at random, with a very slight bias towards rejection, by a $1/2^{n+1}$ margin against equal odds. This margin is so small that, when f is satisfiable even by just one truth assignment, the $1/2^n$ probability of fortuitously stumbling upon it by chance is enough to tip the probability towards acceptance. In fact, the probability that the algorithm accepts a satisfiable CNF formula f is

$$\begin{aligned}
\Pr(f \text{ accepted} \mid f \text{ satisfiable}) &= \Pr(\text{entering "}\textbf{then}\text{" branch}) \cdot \Pr(\text{acceptance in "}\textbf{then}\text{" branch}) + \\
&\quad + \Pr(\text{entering "}\textbf{else}\text{" branch}) \cdot \Pr(\text{acceptance in "}\textbf{else}\text{" branch}) \\
&\geq \overbrace{\frac{1}{2^n} \cdot 1}^{\text{"}\textbf{then}\text{" branch}} + \overbrace{\left(1 - \frac{1}{2^n}\right) \cdot \left(\frac{1}{2} - \frac{1}{2^{n+1}}\right)}^{\text{"}\textbf{else}\text{" branch}} \\
&= \frac{1}{2} + \frac{1}{2^{2n+1}},
\end{aligned}$$

where equality is achieved if there is only one satisfying truth assignment (worst case).

□

The reason why the algorithm described in the above proof is not in **BPP** is the vanishing margin: there is no $\varepsilon > 0$ such that the probability of acceptance is larger that $1/2 + \varepsilon$ for all satisfiable formulas f.

Fig. 4.2 and Table 4.1 summarize what has been said in this Section. Observe that there is no known **BPP** algorithm for FACTORING (both positive and negative certificates seem to be exponentially hard to find).

4.3.4 Quantum computing

As already pointed out, the Turing machine is an abstraction emcompassing our notion of computability (seen as a sequence of deterministic steps), and our understanding of the relationship between problem complexities has been improved by assuming "enhanced" Turing machines with additional capabilities:

- *non-determinism*, i.e. the capability of "following all computations at once", is definitely not realistic, but allows to easily define significant classes of problems (**NP**, **NEXP**, **NL**. . .);

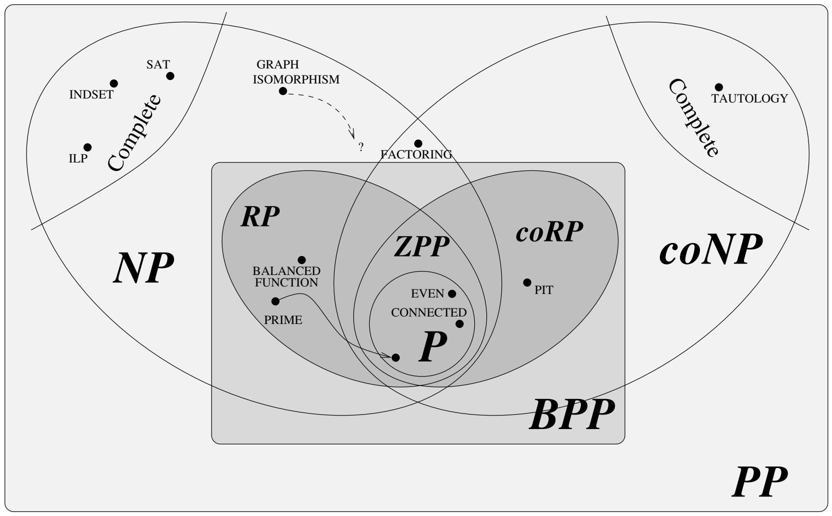


Figure 4.2: What we know about the probabilistic classes introduced in this Section. In particular, the relationship between **BPP** and **NP** is unknown.

Table 4.1: Guaranteed frequency of accepting computations in the various polynomial complexity classes defined in terms of a polynomial NDTM $\mathcal{N}$; $0 < \varepsilon < 1$ is an arbitrary constant value.

Complexity class	Ratio of accepting computations vs. total computations of $\mathcal{N}(x)$		Notes
	if $x \in L$	if $x \notin L$	
P	1	0	Either all computations accept or all reject; $\mathcal{N}$ might as well be deterministic
NP	> 0	0	One computation out of exponentially many is enough to accept
coNP	1	< 1	Reversed roles of acceptance and rejection
RP	$> \varepsilon$	0	No false positives; bound probability of false negatives
coRP	1	$< \varepsilon$	Reversed roles of acceptance and rejection
BPP	$> 1/2 + \varepsilon$	$< 1/2 - \varepsilon$	“Qualified” majority: the ε margin allows us to reduce error probabilities to arbitrarily small values
PP	$> 1/2$	$< 1/2$	“Simple” majority: no guarantee that error probabilities can be reduced to arbitrary values

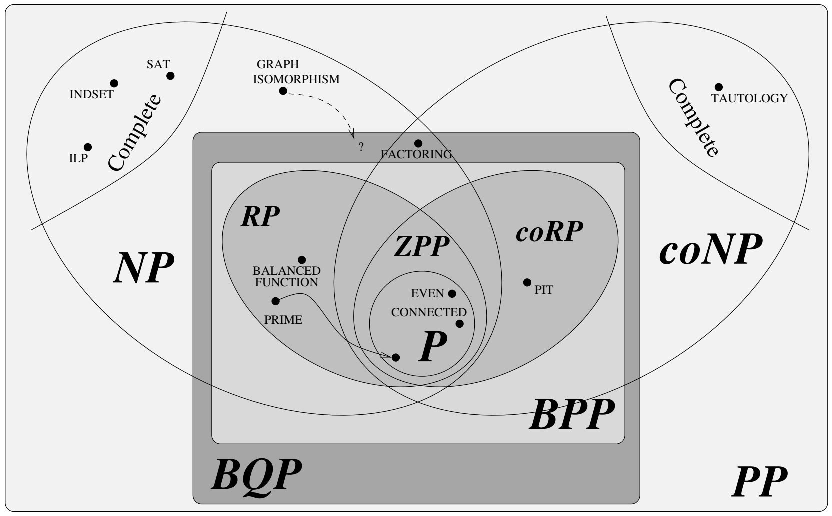


Figure 4.3: Position of the **BQP** quantum class in the previous diagram, with the notable inclusion of the FACTORING language.

- *stochasticity*, i.e. the assumption that the machine can perform random choices, can actually extend the range of problems with an efficient pratical solution (up to class **BPP**), but not many problems appear to belong to $\mathbf{BPP} \setminus \mathbf{P}$: while randomness is extensively used in practical settings, it doesn't lead to much improvement when it comes to theoretical worst-case bounds.

Another promising extension to our notion of computation is given by recent advances in *quantum computing*: from time to time, a "quantum" Turing machine can:

- encode a portion of its tape into a physical system ("quantum circuit"), composed by
  - an array of two-state components at quantum scale (quantum bits or "qubits"), and
  - a sequence of transformations ("quantum gates") acting on the qubits;
- evolve the system through the transformations;
- measure the resulting qubit array and encode the measurement outcome onto the tape.

The strength of quantum computing is given by the fact that the status of an n-qubit system is actually expressed by an array of 2^n complex "amplitudes" (i.e., encoded information is exponential wrt the number of qubits). The weakness of quantum computing is that the amplitude array is not observable, and only determines the probability of a measurement outcome. To this intrinsic weakness we must add lots of engineering problems due to imprecise application of the transformations and to sensibility to external noise.

Definition 39. *A quantum algorithm is assumed to be practical when it uses a number of qubits and of quantum gates which is at most polynomial with respect to the input size. If a language is decidable by such quantum algorithm, it is said to belong to class **BQP** (Bounded-error Quantum Polynomial time).*

Because of the weaknesses reported above, however, very few quantum algorithms are known to actually outperform classical computational models. Integer factorization is the most important problem for which we currently have a polynomial quantum algorithm (Shor's algorithm).

Two significant facts are summarized in Fig. 4.3:

- **BPP** $\subseteq$ **BQP**: quantum systems are true random event generators: any random choice of a stochastic machine can be implemented by a constant-size 1-qubit quantum circuit.
- **BQP** $\subseteq$ **PP**: this is harder to see, however there are ways to emulate quantum circuits with a (possibly exponential) number of classical stochastic computations.

One final note: none of the extensions mentioned above change in any way our notion of computability: the Halting Problem, the Busy Beaver functions, Post's Correspondence Problem, determining the Kolmogorov complexity of a string *remain uncomputable* no matter if we add non-determinism, stochasticity or quantum capabilities. Only their *efficiency* (in terms of time or space) can be improved.

Chapter 5

Selected examples

Let us extend our database of **NP**-complete languages.

5.1 Paths in graphs

Finally, let us consider another important problem we already know to be in **NP**: the Traveling Salesman Problem (TSP). This problem continuously appears in logistics applications, therefore the notion that it is "at least as hard" as any other problem in **NP** has quite negative implications for the real world. To prove completeness, we shall proceed by steps.

5.1.1 Hamiltonian paths

Our first step is to categorize the type of path that the TSP requires on a graph. We will first consider directed graphs.

Definition 40. *Given a directed graph $G = (V, E)$, a path in G is called* Hamiltonian *if it touches every node in V exactly once.*

The computational problem that we want to consider is the following:

Definition 41 (HAMILTONIAN PATH)**.** *Given a directed graph $G = (V, E)$ and two distinct nodes $s, t \in V$, is there a Hamiltonian path in G starting from s and ending in t?*

Unsurprisingly:

Theorem 39. *HAMILTONIAN PATH is **NP**-complete.*

Proof. Clearly, HAMILTONIAN PATH $\in$ **NP**: a certificate is the path itself, expressed as a list on nodes, which can be easily checked for the desired properties: s is the first node, t is the last, every node in V appears exactly once, two consecutive nodes are connected by an edge in the correct direction.

Let us consider a reduction from SAT; i.e., given a generic CNF expression, let us create a graph that has a Hamiltonian path between two specified nodes if and only if the expression is satisfiable.

Let f be a CNF expression on n variables $x_1, \dots, x_n$ organized as the conjunction of n disjunctive clauses $C_1, \dots, C_m$. The main structure of the corresponding graph G is a chain of n "diamonds", one for each variable, as in the left side of Fig. 5.1, with s at the top and t at the bottom. The middle chain of every diamond is doubly linked, and can be traversed by a path in either direction. For this reason, it should be clear that the graph has a multitude of Hamiltonian paths, and we can encode a simple correspondence between the truth assignment to a variable and the direction of traversal of its horizontal chain; e.g., let us assume that a left-to-right traversal corresponds to assigning "true", and

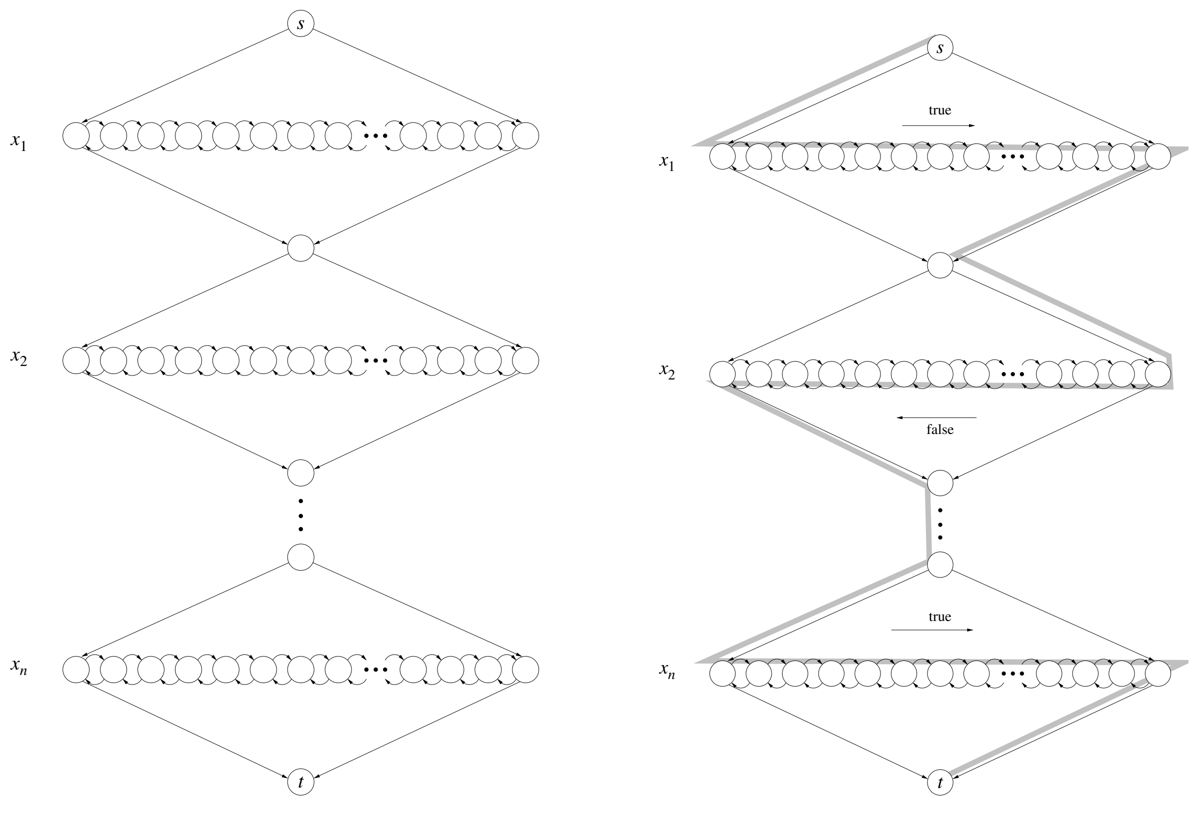


Figure 5.1: Left: directed graph representing Boolean variables $x_1, \ldots, x_n$. Right: Hamiltonian path from s to t encoding a truth assignment to variables ($x_1 = x_n = \top$, $x_2 = \bot$).

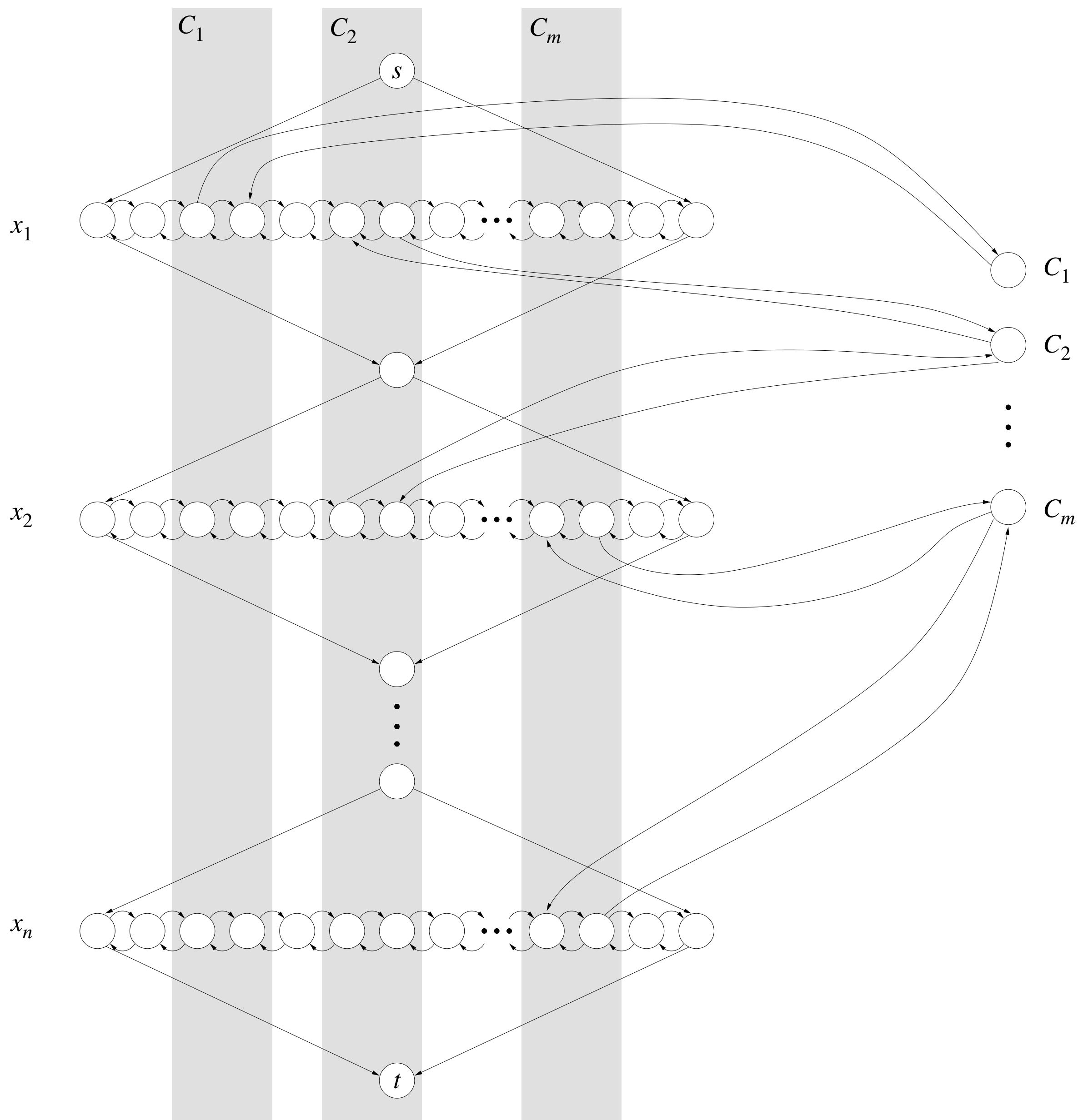


Figure 5.2: Representation of formula $(x_1 \vee \cdots) \wedge (\neg x_1 \vee x_2 \vee \cdots) \wedge \cdots \wedge (\neg x_2 \vee \cdots \neg x_n)$.

right-to-left means “false”. Figure 5.1 (right) shows one such Hamiltonian path and the corresponding truth assigment to variables.

Next, we add a node for each clause $C_1, \dots, C_m$, andwe encode the relationship between variables and clauses as follows; consider the equation

$$\overbrace{(x_1 \vee \cdots)}^{C_1} \wedge \overbrace{(\neg x_1 \vee x_2 \vee \cdots)}^{C_2} \wedge \cdots \wedge \overbrace{(\neg x_2 \vee \cdots \neg x_n)}^{C_m}$$

with reference to Fig. 5.2:

- Every horizontal chain has a consecutive pair of nodes for every clause, each pair separated from the neighboring ones and from the diamond edges by an additional “buffer” node.
- If clause C_i contains the *positive* literal x_j, then we add an edge from the left node in the appropriate pair in the horizontal chain of x_j to node C_i, and an edge from C_i to the right node of the pair. For example, since x_1 appears in clause C_1, we connect the third node in the chain of x_1 to node C_1, and back from C_1 to the fourth node of the chain. This connection allows for a “detour” when traversing the chain from left to right, but not vice versa.
- If clause C_i contains the *negative* literal $\neg x_j$, then we connect the same two nodes in the opposite order; for example, since $\neg x_1$ appears in clause C_2, we connect the seventh node of x_1's chain to C_2, and C_2 back to the sixth node. This allows for a “detour” only when traversing the chain from right to left.

With these provisions, a clause node C_i can be visited in a Hamiltonian path if and only if it is connected to a variable in a way that is compatible with its sense of traversal.

If a truth assignment satisfies a clauses, then the corresponding path through the diamond chain can be extended to visit the corresponding clause node; on the other hand, if an assignment does not satisfy a clause, there is no way to include the clause's node in the path without skipping or revisiting some other nodes, so that the path isn't Hamiltonian anymore. As an example, consider the truth assignment $x_1 = x_n = \top$, $x_2 = \bot$ and Fig. 5.3. Then, no detour can be made to visit node C_2 while keeping the path Hamiltonian. On the other hand, clauses C_1 and C_m can be visited by “cutting” the chain in the eppropriate places.

Therefore, a Hamiltonian path exists in the constructed graph if and only if the equation is satisfiable. □

5.1.2 Directed Hamiltonian cycles

We can remove thw two special nodes s and t by requiring the path to be a cycle:

Definition 42. *Given a directed graph $G = (V, E)$ a Hamiltonian cycle in G is a closed path (i.e., the initial node is also the final one) where every node in V is visited exactly once (clearly, the first and last step, starting and ending at the same node, count as one visit).*

The corresponding problem can be stated as

Definition 43 (DIRECTED HAMILTONIAN CYCLE)**.** *Given a directed graph $G = (V, E)$, does G have a Hamiltonian cycle?*

The reduction discussed above still works just by adding an edge from t to s. Any Hamiltonian cycle must contain that edge, because it is the only way to navigate back once the diamonds have been traversed. Therefore:

Theorem 40. *DIRECTED HAMILTONIAN CYCLE is **NP**-complete.*

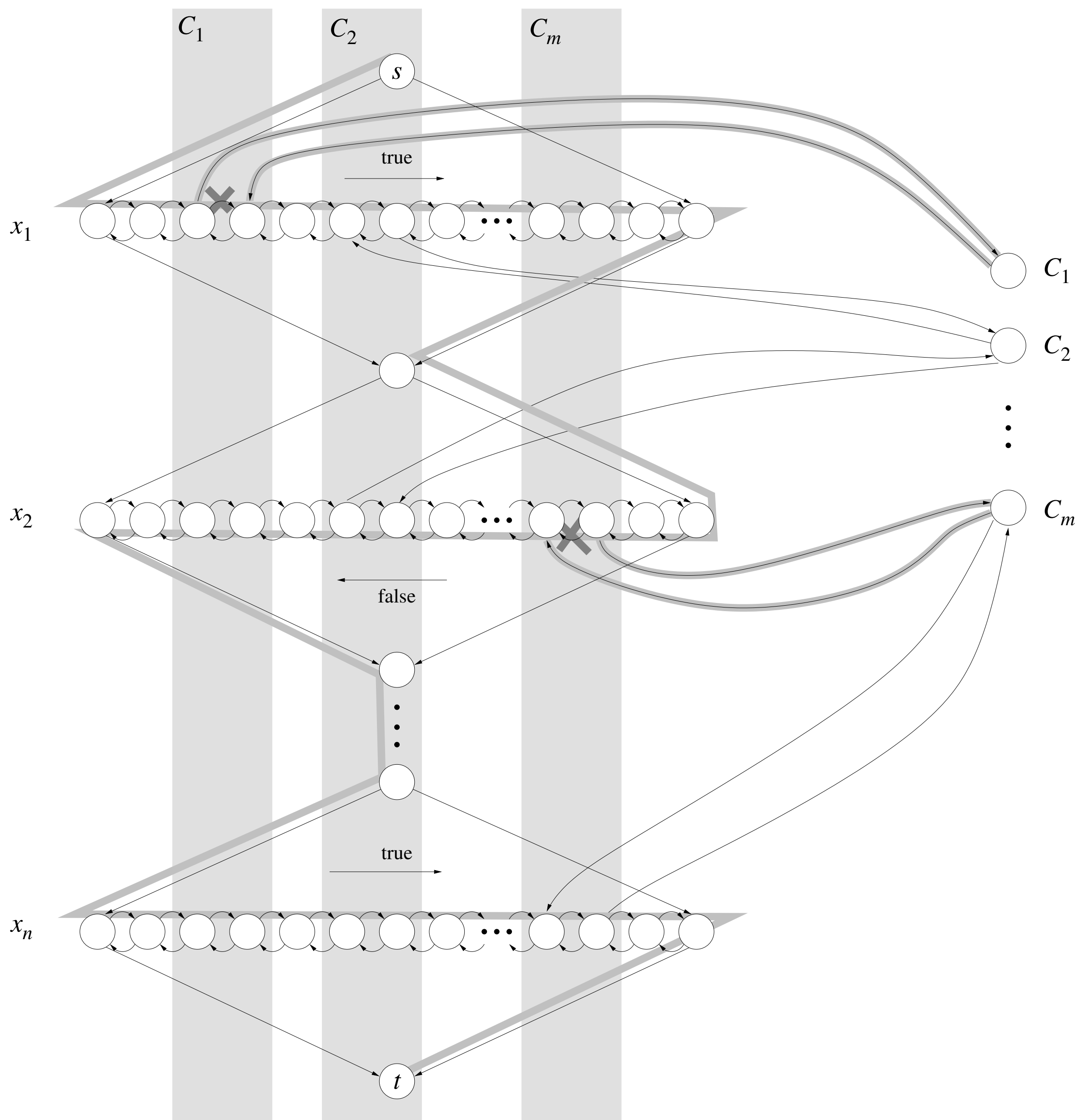


Figure 5.3:

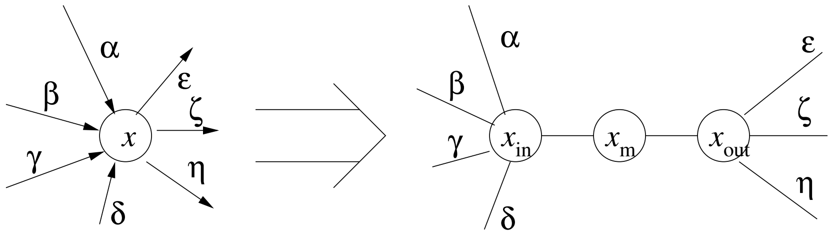


Figure 5.4: Splitting a node in a directed graph into three nodes in the undirected graph.

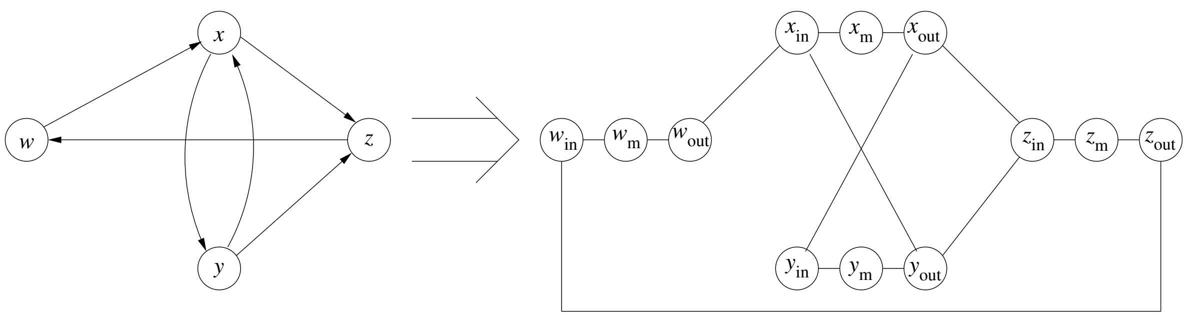


Figure 5.5: Converting a directed graph into an equivalent undirected graph.

5.1.3 Undirected Hamiltonian cycles

The reduction above is strictly dependent on the direction of edges to enforce detours only in precise conditions. However, we can easily reduce DIRECTED HAMILTONIAN CYCLE to its undirected version:

Definition 44 (HAMILTONIAN CYCLE)**.** *Given an undirected graph $G = (V, E)$, does G have a Hamiltonian cycle?*

The problem is somewhat less constrained, since an edge can be traversed in two directions, which might make the problem harder or worse. Anyway, we can easily reduce DIRECTED HAMILTONIAN PATH to it by splitting every node x of the directed graph in three nodes, called x_{in}, x_{m} and x_{out} (see Fig. 5.4) and connect them as follows:

- connect x_{in} to x_{m}, and x_{m} to x_{out};
- for every edge $x \mapsto y$ in the directed graph, connect x_{out} to y_{in}.

Fig. 5.5 shows an example of such reduction.

It is easy to see that if the directed graph has a Hamiltonian cycle, so does the undirected graph: every sequence of edges $x \mapsto y \mapsto z$ that traverses y in the directed graph corresponds to the sequence $x_{\mathrm{out}} - y_{\mathrm{in}} - y_{\mathrm{m}} - y_{\mathrm{out}} - z_{\mathrm{in}}$ that traverses all three nodes corresponding to y; conversely, every path in the undirected graph that traverses y_{in} must procede to y_{m} and y_{out} before exiting to other nodes, otherwise y_{m} could be left out if no node can be visited twice.

Therefore:

Theorem 41. *HAMILTONIAN CYCLE is **NP**-complete.*

5.2 The Traveling Salesman Problem

We can reformulate the TSP in terms of Hamiltonian cycles:

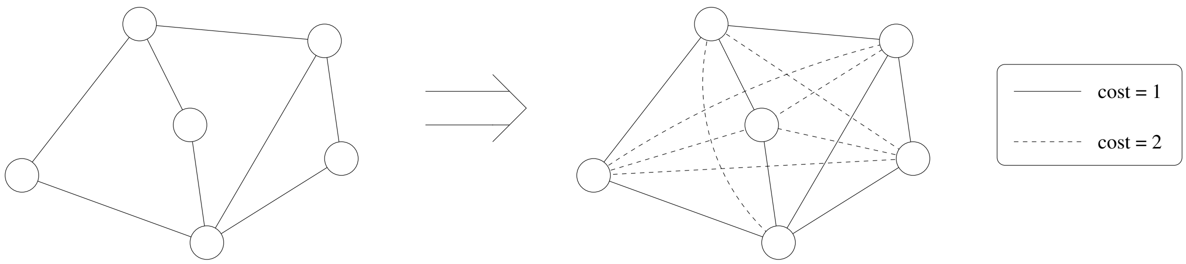


Figure 5.6: Reducing a generic HAMILTONIAN CYCLE instance to an equivalent TSP instance.

Definition 45. *(Traveling Salesman Problem — TSP) Given a complete, undirected graph $G = (V, E)$, with numeric costs associated to edges ($c : E \to \mathbb{N}$) and a budget k, is there a Hamiltonian cycle in G with overall cost not greater than k?*

Theorem 42. *TSP is **NP**-complete.*

Proof. We already know that TSP $\in$ **NP**.

To prove completeness, let us reduce HAMILTONIAN CYCLE to TSP. Given the undirected graph $G = (V, E)$, let us assign cost 1 to all its edges, then complete it adding all missing edges with cost 2. Clearly, the original graph G has a Hamiltonian cycle if and only if the complete version has a Hamiltonian cycle with cost $|V|$ (i.e., composed of $|V|$ edges with cost 1). $\square$

As an illustration, consider Fig. 5.6: the 6-node left-hand side graph has a Hamiltonian cycle if and only if the right-hand side graph has a TSP solution of cost 7 (i.e., visiting all 7 vertices once and not traversing any dashed edge with cost 2).

Part II

Additional material (not in the syllabus)

Chapter 6

Topics from previous editions

This chapter contains material that was presented in previous editions of the course, but has been removed in later editions for reasons of time and clarity.

6.1 A relationship between exponential and polynomial time classes

We can show that the analysis of the relationship between **EXP** and **NEXP** can help wrt the **P** vs. **NP** problem. In particular,

Theorem 43. *If* $\mathbf{EXP} \neq \mathbf{NEXP}$*, then* $\mathbf{P} \neq \mathbf{NP}$*.*

Proof. We will prove the converse. Suppose that $\mathbf{P} = \mathbf{NP}$, and let $L \in \mathbf{NEXP}$. We shall build a deterministic TM that computes L in exponential time.

Since $L \in \mathbf{NEXP}$, there is a NDTM $\mathcal{M}$ that decides $x \in L$ within time bound $2^{|x|^c}$.

We cannot hope to reduce an exponential computation to polynomial time. However, we can exponentially enlarge the input. Consider the language

$$L' = \left\{ \left(x, 1^{2^{|x|^c}}\right) : x \in L \right\}.$$

The language L' is obtained from L by padding all of its strings with an exponentially-sized string of 1's. Now, consider the following NDTM $\mathcal{M}'$ that decides$y \in L'$:

- Check whether y is in the form $\left(x, 1^{2^{|x|^c}}\right)$ for some x (not necessarily in L); if not, REJECT because $y \notin L'$;
- Clean the padding 1's, leaving only x on the tape;
- Execute $\mathcal{M}(x)$ and ACCEPT or REJECT accordingly.

Now, each of the three outlined phases of $\mathcal{M}'$ have an exponential execution time wrt x, but a polynomial time wrt the much larger padded input y. Therefore, $L' \in \mathbf{NP}$.

Since we assumed $\mathbf{P} = \mathbf{NP}$, then $L' \in \mathbf{P}$, therefore there is a deterministic TM $\mathcal{N}'$ that decides L' in polynomial time (wrt the padded size of strings in L', of course).

But then we can define the deterministic TM that, on input x, pads it with $2^{|x|^c}$ ones (in exponential time), then runs $\mathcal{N}'$ on the resulting padded string. This machine is deterministic and accepts L in exponential time, therefore $L \in \mathbf{EXP}$. □

6.2 More NP-complete languages

6.2.1 SET COVER

Definition 46 (SET COVER)**.** *Given a finite set S, n subsets $C_1, \dots, C_n \subseteq S$ and an integer $k \in \mathbb{N}$, is it possible to select k subsets $C_{i_1}, C_{i_2}, \dots, C_{i_k}$ such that their union is S?*

Theorem 44. *SET COVER is **NP**-complete.*

Proof. First, SET COVER is clearly in **NP**.

In order to prove completeness, we start from VERTEX COVER. Given a graph $G = (V, E)$, let $S = E$ in the SET COVER definition, and map every vertex $i \in V$ to set

$$C_i = \{e \in E : i \in e\}$$

of all edges that have vertex i as an endpoint. Solving SET COVER for k subsets amounts to finding k subsets (vertices of G) such that every element of S (every edge of G) belongs to at least one of them (has an endpoint in one of these vertices). □

In other words, we view every vertex as the set of its edges, and we redefine the relation "v is an endpoint of e" as "v contains e".

6.2.2 SUBSET SUM

Now, let us move to the realm of arithmetic.

Definition 47 (SUBSET SUM)**.** *Let $w_1, w_2, \dots, w_n \in \mathbb{N}$, and let $s \in \mathbb{N}$. The problem asks if there is a subset $I \subseteq \{1, \dots, n\}$ such that*

$$\sum_{i \in I} w_i = s. \tag{6.1}$$

Or, equivalently, is there a subset of indices $1 \le i_1 < i_2 < \dots < i_k \le n$ such that $\sum_{j=1}^k w_{i_j} = s$? Or, again, is there an n-bit string $(b_1, b_2, \dots, b_n) \in \{0,1\}^n$ such that $\sum_i b_i w_i = s$?

Theorem 45. *SUBSET SUM is **NP**-complete.*

Proof. As always, let us start by observing that SUBSET SUM $\in$ **NP**. In fact, the input size is $n + 1$ times the representation of the largest of the numbers (plus, possibly, a representation of n itself), therefore it is $|x| = O(n \log \max\{x_1, \dots, x_n, s\})$. A suitable certificate is a list of indices $1 \le i_1 < i_2 < \dots < i_k \le n$, of size $O(k \log n)$, which is linearly bounded by $|x|$. Checking the certificate requires $k \le n$ sums of $(\log s)$-bit numbers, which is again linearly bounded by the input size.

In order to prove that SUBSET SUM is **NP**-hard, let us reduce 3-SAT (which we know to be **NP**-hard) to it. Let F be a 3-CNF boolean formula with n variables and m clauses. For every variable x_i in F, let us build two numbers with the following base-10 representation:

$$\begin{aligned} t_i &= a_1 a_2 \cdots a_n p_1 p_2 \cdots p_m, \\ f_i &= a_1 a_2 \cdots a_n q_1 q_2 \cdots q_m, \end{aligned} \tag{6.2}$$

where the digits of the numbers t_i and f_i are:

$$\begin{aligned} a_j &= \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise,} \end{cases} \\ p_j &= \begin{cases} 1 & \text{if the } j\text{-th clause contains } x_i \text{ without negation} \\ 0 & \text{otherwise,} \end{cases} \\ q_j &= \begin{cases} 1 & \text{if the } j\text{-th clause contains } \neg x_i \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{6.3}$$

For instance, consider the following $n = 3$-variable, $m = 4$-clause formula:

$$F(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

The formula corresponds to the following $n + 3$ pairs of $m + n$-digit numbers:

		a_1	a_2	a_3	p_1	p_2	p_3	p_4
t_1	$=$	1	0	0	1	1	0	1
f_1	$=$	1	0	0	0	0	1	0
t_2	$=$	0	1	0	1	0	1	0
f_2	$=$	0	1	0	0	1	0	1
t_3	$=$	0	0	1	1	0	0	1
f_3	$=$	0	0	1	0	1	0	0

Note that, although the digits are 0 or 1, the numbers are represented in base 10 (e.g., t_1 reads "one million, one thousand, one hundred and one"). The three leftmost digits of each number act as indicators of the variable the number refers to, while the four rightmost ones identify the clauses that would be satisfied if x_i were true (in the case of t_i) or if x_i were false (in the case of f_i).

Let a truth assignment to $x_1, \dots, x_n$ correspond to the choice of one number between each t_i, f_i pair; namely, let us choose t_i if the corresponding x_i is assigned to be true, f_i otherwise. For example, the truth assignment $(x_1, x_2, x_3) = (\top, \bot, \top)$ in the example corresponds to the choice of numbers t_1, f_2 and t_3. Observe that the sum of the three numbers is

$$t_1 + f_2 + t_3 = 1112203$$

The digits of the sum tell us that, for each variable x_i, exactly one number between t_i and f_i has been chosen (the leftmost n digits are 1), and that the four clauses are satisfied by respectively 2, 2, 0 and 3 of their literals. In particular, we get the information that the third clause of F is not satisfied. On the other hand, the assignment $(\bot, \bot, \top)$ corresponds to the choice of variables f_1, f_2 and t_3, whose sum is $f_1 + f_2 + t_3 = 1111112$, so that we know that all clauses are satisfied by at least one of their literals.

We can conclude that F has a satisfying assignment if and only if a subset of the corresponding numbers t_i, f_i can be found whose sum is in the form

$$s = \overbrace{11 \cdots 1}^{n \text{ digits}} s_1 s_2 \cdots s_m, \qquad \text{with } s_1, \dots, s_m \neq 0. \tag{6.4}$$

In order to obtain a proper instance of SUBSET SUM, we need to transform (6.4) into a precise value. Note that, as every clause has at most 3 literals, $s_j \leq 3$. Therefore, we need to provide enough numbers to enable all non-zero s_j's to become precisely 3. We can obtain this by declaring two equal numbers u_i, v_i per clause, with all digits set to zero with the exceptions of the $n + i$-th digit equal to one:

$$u_i = v_i = \overbrace{00 \cdots 0}^{n \text{ digits}} d_1 d_2 \cdots d_m, \tag{6.5}$$

with

$$d_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{6.6}$$

. Therefore, F has a satisfying truth assignment if and only if we can find a subset among the numbers $t_1, \dots, t_n, f_1, \dots, f_n, u_1, \dots, u_m, v_1, \dots, v_m$ as defined in (6.2), (6.3), (6.5), (6.6), whose sum is

$$s = \overbrace{11 \cdots 1}^{n \text{ digits}} \overbrace{33 \cdots 3}^{m \text{ digits}}.$$

$\square$

6.2.3 KNAPSACK

A simple but very important extension of SUBSET SUM is the following, where *two* sets of numbers are involved.

Definition 48 (KNAPSACK)**.** *Given a set of n items with weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$, a knapsack with capacity c and a minimum value s that we want to carry, is there a subset of items that the knapsack would be able to carry and whose overall value is at least s?*
More formally, the problems asks if there is a subset of indices $I \subseteq \{1, \dots, n\}$ such that

$$\sum_{i\in I} w_i \le c \qquad \text{and} \qquad \sum_{i\in I} v_i \ge s. \tag{6.7}$$

The first constraint ensures that the knapsack is not going to break, the second one ensures that we can pack at least the desired value.

Observe that KNAPSACK $\in$ **NP**, since the subset I is smaller that the problem size and the sums can be verified in a comparable number of steps, so that I is a suitable certificate.

Moreover, SUBSET SUM is a special case of KNAPSACK: given $(x_1, \dots, x_n, s)$ as in definition 47, we can reformulate (hence reduce) it as a KNAPSACK instance by letting $w_i = v_i = x_i$ and $c = s$ (so, equating weights and values), which would reduce (6.7) to the equality (6.1). Therefore,

Theorem 46. *KNAPSACK is **NP**-complete.*

The Merkle-Hellman cryptosystem

As a "real-world" application of an **NP**-completeness result, let us consider the following cryptosystem. Now broken, it was one of the earliest public-key cryptosystems[1] together with RSA.

Description Alice generates a sequence of n integers $y_1, \dots, y_n$ which is *super-increasing*, i.e., every item is larger that the sum of all previous ones:

$$y_i > \sum_{j=1}^{i-1} y_j qquad \text{for } i = 2, \dots, n.$$

Notice that, given a super-increasing sequence, there is a simple algorithm to solve the SUBSET SUM problem for a given sum s:

```
function SUPERINCREASING_SUBSET_SUM (y_1, ..., y_n, s)        y_1, ..., y_n is super-increasing
    I ← ∅
    for i ← n..1                                    scan items starting from the largest
        if y_i < s                                  every time an item can be subtracted
            s ← s - y_i                                                      subtract it
            I ← I ∪ {i}                                                record its index
    if s = 0
        return I                                           subtracted items added up to s
    else
        reject                                                the sum was not achievable
```

In order to scramble up her numbers, Alice chooses a positive integer $m > \sum_i y_i$ and another integer $r > 0$ such that r and m are coprime, i.e., $\gcd(r, m) = 1$. Next, she multiplies all the elements in her super-increasing sequence by r, modulo m:

$$x_i \equiv y_i \cdot r \pmod{m}. \tag{6.8}$$

[1] See https://en.wikipedia.org/wiki/Merkle-Hellman_knapsack_cryptosystem

In other words, x_i is the remainder of the division of $y_i r$ by m. She finally publishes the numbers $x_1, \dots, x_n$ as her public key.

When Bob wants to send a message to Alice, he encodes it into an n-bit string $(b_1, \dots, b_n)$. He computes the sum

$$s = \sum_{i=1}^{n} b_i x_i,$$

where the x_i's are the ones published by alice, and sends s to Alice.

Notice that the x_i's have a basically random distribution in $\{0, \dots, m-1\}$. They are not super-increasing, and the SUBSET SUM problem cannot be solved by a simple algorithm.

Alice, however, can move s back to the super-increasing sequence by "undoing" the scrambling operation (6.8). Since she knows r and m, which are kept secret, she can compute the inverse of r modulo m, i.e., the only number $r' \in \{1, \dots, m-1\}$ such that

$$r \cdot r' \equiv 1 \pmod{m}.$$

The algorithm to compute r' is an extension of Euclid's gcd algorithm and is polynomial in the sizes of r and m[2].

Alice can compute $s' \equiv s \cdot r' \pmod{m}$, which yields the same subset defined by Bob's binary string within the super-increasing sequence:

$$s' \equiv sr' \equiv \left(\sum_{i=1}^{n} b_i x_i\right) r' \equiv \sum_{i=1}^{n} b_i x_i r' \equiv \sum_{i=1}^{n} b_i y_i r r' \equiv \sum_{i=1}^{n} b_i y_i \pmod{m}.$$

She can then reconstruct Bob's binary sequence by calling

$$\texttt{SUPERINCREASING\_SUBSET\_SUM}\ (y_1, \dots, y_n, s)$$

which returns the set $I = \{i = 1, \dots, n | b_i = 1\}$.

Observations The system is considerably faster than RSA, because it only requires sums, products and modular remainders, no exponentiation.

Since the encryption and decryption processes are not symmetric (Alice cannot use her private key to encrypt something to be decrypted with her public key), the system is not suitable for electronic signature protocols.

Proposed in 1978, in 1984 a polynomial scheme to reconstruct the super-increasing sequence (and hence Alice's private key) was published by Adi Shamir (the "S" in RSA).

Although based on an instantiation of the SUBSET SUM problem, it is commonly referred to as the "Knapsack" cryptosystem.

6.2.4 k-VERTEX COLORING for $k > 3$

We know that 2-VERTEX COLORING $\in$ **P**, and that 3-VERTEX COLORING is **NP**-complete. What about $k > 3$? Consider, for instance, 4-VERTEX COLORING. On one hand having more colors might seem to relax the problem (more choices mean also more chances of a positive answer); however, we can easily prove that the case $k = 4$ is at least as hard as $k = 3$:

Theorem 47. *4-VERTEX COLORING is* ***NP****-complete.*

Proof. Clearly, 4-VERTEX COLORING $\in$ **NP**.

Let us start with a 3-VERTEX COLORING instance $G = (V, E)$ and let us build an equivalent 4-VERTEX COLORING instance $G' = (V', E')$. To build G', let us start from G and add four new

[2] See `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`

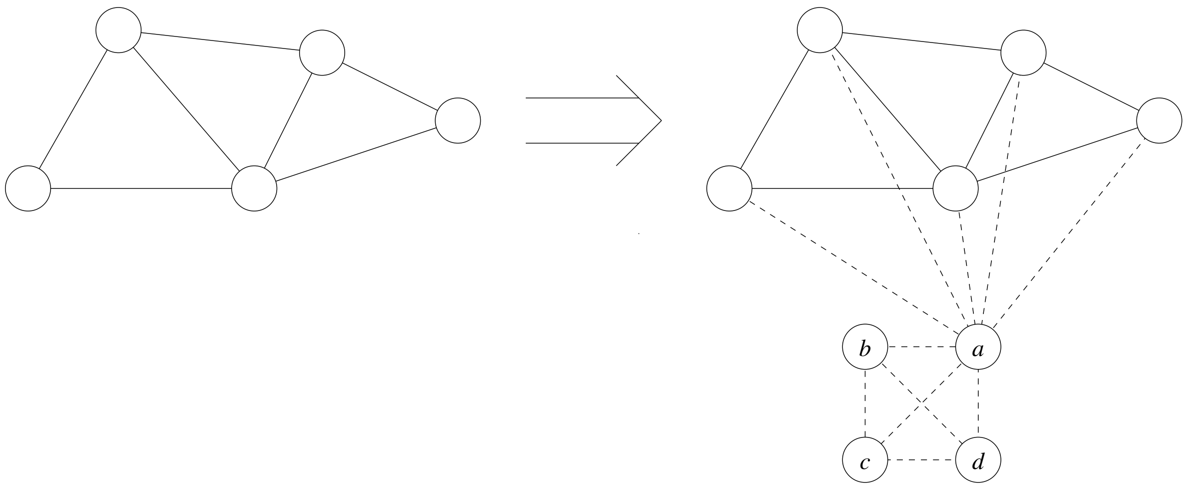


Figure 6.1: Reducing a 3-VERTEX COLORING instance to an equivalent 4-VERTEX COLORING instance.

nodes a, b, c, d, all connected to each other (so that every 4-coloring will need to assign different colors to each node). Then, connect a to all nodes of V.

If G is 3-colorable, then G' 4-colorable; just assign the fourth color to the extra node a.

Conversely, if G' is 4-colorable, then all original nodes in V will have the colors of the extra nodes b, c and d, therefore they have a valid 3-coloring for the original graph G. □

See for example Fig. 6.1: the left-hand side graph is 3-colorable if and only if the right-hand side graph is 4-colorable: the trick consists in wasting the fourth color on node a, forcing the remaining nodes to share three colors.

6.3 Function problems

In this course we mainly discuss decision problems, aka languages, i.e., questions that require a "yes"/"no" answer. Here is a very brief introduction to function problems.

Let us consider the following "functional" versions of already known problems:

F-PATH Given a graph $G = (V, E)$ and two nodes $s, t \in V$, find a path between s and t in G.

F-MINPATH Given a graph $G = (V, E)$ and two nodes $s, t \in V$, find *a path of minimum length* between s and t in G.

F-SAT Given an n-variable CNF formula f, find a satisfying assignment $(x_1, \dots, x_n)$, provided that it exists.

F-INDSET Given a graph $G = (V, E)$ and an integer k, find an independent set $V \subseteq V'$ with size $|V'| \geq k$, if any.

Observe that the satisfying assignment might not be unique. In general, we can model a functional problem as a binary relation

$$R \subseteq \Sigma^* \times \Sigma^*.$$

We write $R(x, y)$ —or, equivalently, $(x, y) \in R$— when x is an instance of the problem and y is a corresponding solution.

Two of the problems listed above, F-PATH and F-MINPATH, have obvious polynomial algorithm based on BFS visits of the graph, and are indeed related to languages in **P**. On the other hand, F-SAT and F-INDSET are the functional equivalent of SAT and INDSET, with the "yes"/"no" answer replaced with the request of an actual solution.

The two following definitions extend the notions of **P** and **NP** to functional problems.

Definition 49 (**FP**). *A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is in class **FP** if there is a polynomial time TM $\mathcal{M}$ that, upon $x \in \Sigma^*$, outputs any $y \in \Sigma^*$ such that $R(x, y)$.*

Note that, given x, there may be many y that satisfy the relation (e.g., many truth assignments may satisfy the same CNF formula): we require $\mathcal{M}$ to output one of them.

Definition 50 (**FNP**). *A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is in class **FNP** if there is a polynomial time TM $\mathcal{M}$ that, upon $x, y \in \Sigma^*$, accepts if and only if $R(x, y)$.*

Observe that we do not actually need non-deterministic machines to define **FNP**: the second member in the relation acts as the certificate[3].

6.3.1 Relationship between functional and decision problems

F-SAT and F-INDSET have the following property: if we were able to solve them, then we would automatically have an answer to the corresponding decision problem. I.e., the decision problems have trivial reductions to their functional versions. Therefore, F-SAT and F-INDSET are **NP**-hard.

What about the converse? Suppose that we had an oracle that gives a solution to the decision problem. In both the SAT and INDSET examples, we could use these oracles to build a solution do the functional version step by step. In the F-SAT case, the algorithm would work by guessing the correct truth values one by one:

```
function FSAT (f)                                              f is in CNF
  if SAT(f) = 0                                                if f is unsatisfiable, stop
      then reject and halt
  n ← number of variables of f
  for i ← 1..n
      if SAT(f|x_i=⊤) = 1                                      Guess the value of x_i
          then x*_i ← ⊤                                        Put the right truth value in the output string x*
          else x*_i ← ⊥
      f ← f|x_i=x*_i                                           Fix x_i in f to the correct truth value and simplify f
  return (x*_1, ..., x*_n)
```

Similar methods can be employed also for other problems.

6.4 Interactive proof systems

In many cases, a problem's solution is not the only aspect of interest in algorithm research. After a solution is found, the problem remains of "proving" the solution to other actors.

A simple example is captured by the definition of **NP**: even if a super-polynomial solver could find a solution, it could be accepted only if a concise (i.e., polynomial) certificate is provided by the solver. We can model **NP** as the class of languages for which, once a very powerful *prover* (usually called P for prover, or M for Merlin, the wizard in the Arthurian saga) not only needs to provide an answer

[3]See also the Wikipedia articles about these classes:
`https://en.wikipedia.org/wiki/FP_(complexity)`
`https://en.wikipedia.org/wiki/FNP_(complexity)`

to a difficult question, but must also provide a proof that can be checked by a less powerful, usually polynomial-time *verifier* (hence called V, or A for Arthur[4]).

While, in **NP**, the purpose of Merlin is to convince Arthur when the answer is positive (no certificate is required if the answer is "no"), we can envision more complex cases in which Arthur is more demanding[5].

6.4.1 An example: GRAPH ISOMORPHISM

Consider the GRAPH ISOMORPHISM language: given two undirected graphs $G_1 = (V_1, E_1)$ and $g_2 = (V_2, E_2)$, are the two graphs isomorphic, i.e., equal after a permutation of the nodes? The language is clearly in **NP**, but what if we (Arthur) also require a proof of the negative answer —a proof that G_1 and G_2 are not isomorphic?

Arthur can prepare a sequence of N graphs $G'_1, \dots, G'_N$, where every G'_i is a random permutation of one of the two original graphs, chosen at random. Arthur sends the graphs $G_1, G_2, G'_1, G'_2, \dots, G'_N$ to Merlin.

After receiving the graphs, Merlin (who has the capability of solving the problem) has two choices:

1. if G_1 is isomorphic to G_2, he sends to Arthur the permutation that maps G_1 onto G_2;
2. if G_1 and G_2 are not isomorphic, then it is possible for Merlin to tell which graphs among $G'_1, \dots, G'_N$ are isomorphic to G_1, and which to G_2.

In the first case, Arthur can check the permutation and get convinced that the two graphs are indeed isomorphic. In the second case, Arthur checks if Merlin has correctly identified the original graph in all N cases; if G_1 and G_2 were isomorphic, Merlin could only answer randomly (because all graphs in the set would be isomorphic to both), and the chance for him to be right would be just 2^{-N}.

Therefore, the described protocol has the following property: if G_1, G_2 are isomorphic, then Merlin can convince Arthur with certainty; on the other hand, Merlin would have only probability 2^{-N} to deceive Arthur by convincing him that they are not. If G_1 and G_2 are not isomorphic, then Merlin has no way to convince Arthur that they are (he would have to provide a node permutation that works, but there is none); on the other hand, Arthur can be convinced with confidence $1 - 2^{-N}$ of the truth.

6.4.2 The Arthur-Merlin protocol

Let us consider, more generally, a protocol where the exchange between Arthur and Merlin requires more than one round. Let L be a language. We assume that Arthur has a polynomial function f that, in order to be convinced whether $x \in L$, generates a "question" $a_1 = f(x)$ for Merlin. We also assume that f is stochastic, in the sense that it can base its output on a finite number of coin tosses.

After receiving the question a_1, Merlin uses his own function g (on which we make no assumptions) to generate an "answer" $a_2 = g(x, a_1)$.

The interaction continues for k "question/answer" rounds, where every message may depend on all

[4] Either Pendragon or Dent — both are fine for our purposes.

[5] For a more comprehensive exposition, see Arora-Barak, first three Sections of Chapter 8 in the online draft.

past history, with the exchange of the following messages:

$$
\begin{aligned}
a_1 &= f(x)\\
a_2 &= g(x, a_1)\\
a_3 &= f(x, a_1, a_2)\\
a_4 &= g(x, a_1, a_2, a_3)\\
\vdots &\quad\ \vdots\\
a_{2k-1} &= f(x, a_1, a_2, \dots, a_{2k-2})\\
a_{2k} &= g(x, a_1, a_2, \dots, a_{2k-1})
\end{aligned}
$$

After receiving the "final" answer a_{2k}, Arthur must decide whether to accept or not the string x. Let $\langle f, g\rangle_k(x)$ be the outcome (acceptance or rejection) after the k-round interaction.

Definition 51. *Given a language L and a positive integer k, we say that L is decided by a k-*round Merlin-Arthur protocol *($L \in \boldsymbol{AM}[k]$) if there is a stochastic, polynomial function f and a (unbounded) function g such that*

- *if $x \in L$, then $\Pr\bigl[\langle f, g\rangle_k(x)$ accepts$\bigr] \geq 2/3$;*
- *if $x \notin L$, then for any function h, $\Pr\bigl[\langle f, h\rangle_k(x)$ accepts$\bigr] \leq 1/3$.*

In other words, if $x \in L$ then Merlin has a way (function g) to convince Arthur to accept with high probability. On the other hand, if $x \notin L$, then whatever method h Merlin uses, he will never be able to convince Arthur to accept with high probability.

6.4.3 The Interactive Polynomial protocol class

As an obvious generalization of the **AM**[k] class, let us consider a case in which the number of rounds is not constant, but bounded by a polynomial in the size of the input (i.e., we allow for a longer chain of interactions if the input is larger). Definition 51 is modified as follows:

Definition 52. *Given a language L, we say that L is decided by a* polynomial interactive proof protocol *($L \in \boldsymbol{IP}$) if there is a polynomial $k(n)$, a stochastic, polynomial function f and a (unbounded) function g such that*

- *if $x \in L$, then $\Pr\bigl[\langle f, g\rangle_{k(|x|)}(x)$ accepts$\bigr] \geq 2/3$;*
- *if $x \notin L$, then for any function h, $\Pr\bigl[\langle f, h\rangle_{k(|x|)}(x)$ accepts$\bigr] \leq 1/3$.*

It turns out that **IP** is just another categorization of **PSPACE**. While one inclusion (**IP** $\subseteq$ **PSPACE**) is quite easy to prove, tho other would require too long. Therefore, we just state the theorem without proving it:

Theorem 48.

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

We can also observe that, if we remove all stochasticity from f (i.e., we make Arthur, the prover, deterministic), then Merlin can determine the whole interaction from the beginnning, providing the sequence $a_1, \dots, a_{2k}$ to Arthur since the beginning. Arthur would just need to check his side of the interaction and take a decision without the need of further rounds; therefore, the sequence $a_1, \dots, a_{2k}$ would be a polynomial certificate of acceptance. Therefore,

Theorem 49. *Let **dIP** be the "deterministic" version of **IP** where we strip stochasticity from Arthur's function f in Definition 52. Then*

$$\textbf{\textit{dIP}} = \textbf{\textit{NP}}.$$

Since it is quite universally believed that **NP** $\subsetneq$ **PSPACE**, then we must conclude that, likely, **dIP** $\subsetneq$ **IP**, i.e., stochasticity plays a fundamental role in interactive proofs. This corresponds to our intuition that being able to make random questions increases our chances of discovering a deceit.

6.5 Zero-knowledge proofs

In contexts such as **NP** and **IP**, we are used to the idea of *verifiable* answers in the form of a certificate or a proof. Such certificate or proof usually provide a solution to the problem, and the polynomial verifier's task is to check whether the solution is correct or not.

Some contexts, mainly related to security and cryptocurrencies, require one party to "prove" that it knows the answer to a question without disclosing information about it.

One simple example of it are challenge-response algorithms in secure handshake protocols: in order to check if Bob has a cryptographic key, Alice doesn't need to directly ask to see it; she will just challenge Bob to encrypt a random piece of information and she will proceed to compare it to the expected result.

In the following scenario, we will assume that Alice has a graph and needs to 3-color it; Bob has a solution, and he wants to prove it to Alice without disclosing any useful information about the coloring (as he might want to sell his coloring, but Alice needs to be sure that she is getting her money's worth).

Given a graph $G = (V, E)$ with $|V| = n$ nodes, a 3-coloring is a sequence $C = (c_1, c_2, \dots, c_n) \in \{1, 2, 3\}^n$ (each node is assigned one "color" out of three) where $\{i, j\} \in E \to c_i \neq c_j$.

Observe that each of the $3! = 6$ 3-color permutations gives raise to a different possible coloring. If $\pi : \{1, 2, 3\} \to \{1, 2, 3\}$ is a permutation, then the permuted coloring $\pi C = (\pi c_1, \dots, \pi c_n)$ is still a valid coloring.

To prove that he has a valid coloring of G, Bob will select a cryptographic hash function H, a random permutation $\pi : \{1, 2, 3\} \to \{1, 2, 3\}$, n random strings $s_1, \dots, s_n$ of suitable length (a few hundred bits) and will send to Alice the n values

$$d_i = H(\pi c_i \oplus s_i), \qquad i = 1, \dots, n, \tag{6.9}$$

where $\oplus$ is the concatenation operator (the bit string representing πc_i followed by the bit string s_i). Observe that s_i acts as a "salt" string, which randomizes the hashing function.

Once Alice receives $d_1, \dots, d_n$, she randomly selects one edge $\{i, j\} \in E$ from the graph and sends i and j to Bob, who sends back the two permuted colors πc_i, πc_j and the two salt values s_i, s_j.

Alice can now check that the two colors are different, that they are in the $\{1, 2, 3\}$ range, and that (6.9) holds for d_i and d_j.

We can make the following observations.

- By knowing the d_i's alone, Alice cannot decode the coloring for two reasons:
  1. the hash function H is cryptographically strong, and therefore one would have to try too many combinations of values for πc_i and s_i before finding one that returns precisely d_i, but more importantly
  2. H is not injective, therefore there might be many combinations that reture exactly the same value d_i.
- Even after receiving the permuted colors and the salt strings for the two vertices i and j, Alice does not get any useful information: all she sees are two different colors, but she has no idea about which permutation was used by Bob, so all combinations are possible (provided that the two colors are different).

- If Bob weren't able to properly 3-color the graph, the colors at the endpoints of at least one edge would be illegal (same color, or out of the allowed range). Since Bob has already sent the hashed values to Alice, and is unable to find a new salt in order to change a node's color depending on Alice's choice, then Alice would discover the problem with probability at least $|E|^{-1}$ (probability of picking the illegal edge, provided that there is at least one).

Therefore, Alice learns nothing about the coloring, and if Bob is cheating then Alice has probability $|E|^{-1}$ to discover it. The probability can be increased at will by repeating the protocol. After N times, the probability for Alice to discover Bob's deceit is at least $1-(1-|E|^{-1})^N$, which can be made arbitrarily close to 1.

Finally, observe that, even though Alice is convinced with very high confidence that Bob knows a 3-coloring for G, she cannot show the proof convincingly to another party.

See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more examples of the technique.

Chapter 7

Further directions

Here are a few topics that might be interesting and that we could not discuss for lack of time.

7.1 About NP

We only analyzed **NP** languages from the (easier) viewpoint of worst-case complexity. A whole line of research is open about *average-case* complexity: given reasonable assumptions on the probability distribution of instances, what is the expected (average) complexity? Even if $\mathbf{P} \neq \mathbf{NP}$, as far as we know the average complexity of some **NP**-complete languages might as well be polynomial. Moreover, for some problems, hard instances might actually exist but be too hard to find. Remember that many applications of **NP**-hardness results (e.g., all public-key cryptography schemes) rely on our ability to actually forge solved instances of hard problems.

Another line of research is *approximability*: for some **NP**-hard optimization (functional) problems, an approximate solution, within a given accuracy bound, might be polynomially achievable.

7.2 Above NP

If $\mathbf{PSPACE} \neq \mathbf{NP}$, as is probably the case, there is a very large gap between complete problems in the two classes, and a whole hierarchy of classes, the *polynomial hierarchy*, tries to characterize the enormous (although possibly void!) gap between the two, based on a quantifier-based generalization of the **NP** definition.

7.3 Other computational models

Extensions of the Turing model to incorporate quantum mechanics are being studied, and a whole lot of complexity classes (all recognizable because of a Q somewhere in their name[1]) has been proposed. If reliable physical devices will ever be able to implement these quantum models, the solutions to some problems, such as integer factoring, some forms of database search, finding the period of modular functions and so on, will become practical.

[1]But in some cases the “Q” stands for “quantifier” — beware of logicians.

Part III

Questions and exercises

Appendix A

Self-assessment questions

This chapter collects a few questions that students can try answering to assess their level of preparation.

A.1 Computability

A.1.1 Recursive and recursively enumerable sets

1. Why is every finite set recursive?
   (Hint: we need to check whether s is in a finite list)

2. Try to prove that if a set is recursive, then its complement is recursive too.
   (Hint: invert 0 and 1 in the decision function's answer)

3. Let S be a recursively enumerable set, and let algorithm $\mathcal{A}$ enumerate all elements in S. Prove that, if $\mathcal{A}$ lists the elements of S in increasing order, then S is recursive.
   (Hint: what if $n \notin S$? Is there a moment when we are sure that n will never be listed by $\mathcal{A}$?)

A.1.2 Turing machines

1. Why do we require a TM's alphabet Σ and state set Q to be finite, while we accept the tape to be infinite?

2. What is the minimum size of the alphabet to have a useful TM? What about the state set?

3. Try writing machines that perform simple computations or accept simply defined strings.

A.2 Computational complexity

A.2.1 Definitions

1. Why introduce non-deterministic Turing machines, if they are not practical computational models?

2. Why do we require reductions to carry out in polynomial time?

3. Am I familiar with Boolean logic and combinational Boolean circuits?

A.2.2 P vs. NP

1. Why is it widely believed that $\mathbf{P} \neq \mathbf{NP}$?
2. Why is it widely hoped that $\mathbf{P} \neq \mathbf{NP}$?

A.2.3 Other complexity classes

1. Why are classes **EXP** and **NEXP** relatively less studied than their polynomial counterparts?
2. What guarantees does **RP** add to make its languages more tractable than generic **NP** languages?

A.2.4 General discussion

1. Worst-case complexity might not lead to an accurate depiction of the world we live in. Read Sections 1 and 2 (up to 2.5 inclusive) of the famous "Five worlds" paper:
   Russell Impagliazzo. *A Personal View of Average-Case Complexity.* UCSD, April 17, 1995.
   `http://cseweb.ucsd.edu/users/russell/average.ps`
   What world do we live in, and which would be the ideal world for the Author?

Appendix B

Exercises

Preliminary observations

Since the size of the alphabet, the number of tapes or the fact that they are infinite in one or both directions have no impact on the capabilities of the machine and can emulate each other, unless the exercise specifies some of these details, students are free to make their choices.

As for accepting or deciding a language, many conventions are possible. The machine may:

- erase the content of the tape and write a single “1” or “0”;
- write “1” or “0” and then stop, without bothering to clear the tape, with the convention that acceptance is encoded in the last written symbol;
- have two halting states, `halt-yes` and `halt-no`;
- any other unambiguous convention;

with the only provision that the student writes it down in the exercise solution.

Exercise 1

For each of the following classes of Turing machines, decide whether the halting problem is computable or not. If it is, outline a procedure to compute it; if not, prove it (usually with with a reduction from the general halting problem). Unless otherwise stated, always assume that the non-blank portion of the tape is bounded, so that the input can always be finitely encoded if needed.

1.1) TMs with 2 symbols and at most 2 states (plus the halting state), starting from an empty (all-blank) tape.

1.2) TMs with at most 100 symbols and 1000000 states.

1.3) TMs that only move right;

1.4) TMs with a circular, 1000-cell tape.

1.5) TMs whose only tape is read-only (i.e., they always overwrite a symbol with the same one);

Hint — *Actually, only one of these cases is uncomputable...*

Solution 1

The following are minimal answers that would guarantee a good evaluation on the test.

1.1) The definition of the machine meet the requirements for the Busy Beaver game; Since we know the BB for up to 4 states, it means that every 2-state, 2-symbol machine has been analyzed on an empty tape, and its behavior is known. Therefore the HP is computable for this class of machines.

1.2) As we have seen in the lectures, 100 symbols and 1,000,000 states are much more than those needed to build a universal Turing machine $\mathcal{U}$. If this problem were decidable by a machine, say $\mathcal{H}_{1,000,000}$, then we could solve the general halting problem "does $\mathcal{M}$ halt on input s" by asking $\mathcal{H}_{1,000,000}$ whether $\mathcal{U}$ would halt on input (M, s) or not. In other words, we could reduce the general halting problem to it, therefore it is undecidable.

1.3) If the machine cannot visit the same cell twice, the symbol it writes won't have any effect on its future behavior. Let us simulate the machine; if it halts, then we output 1. Otherwise, sooner or later the machine will leave on its left all non-blank cells of the tape: from now on, it will only see blanks, therefore its behavior will only be determined by its state. Take into account all states entered after this moment; as soon as a state is entered for the second time, we are sure that the machine will run forever, because it is bound to repeat the same sequence of states over and over, and we can interrupt the simulation and output 0; if, on the other hand, the machine halts before repeating any state, we output 1.

1.4) As it has a finite alphabet and set of states (as we know from definition), the set of possible configurations of a TM with just 1000 cells is fully identified by (i) the current state, (ii) the current position, and (iii) the symbols on the tape, for a total of $|Q| \times 1000 \times |\Sigma|^1 000$ configurations. While this is an enormous number, a machine running indefinitely will eventually revisit the same configuration twice. So we just need to simulate a run of the machine: as soon as a configuration is revisited, we can stop simulating the machine and return 0. If, on the other hand, the simulation reaches the halt state, we can return 1.

1.5) Let $n = |Q|$ be the number of states of the machine. Let us number the cells with consecutive integer numbers, and consider the cells a and b that delimit the non-null portion of the tape. Let us simulate the machine. If the machine reaches cell $a-(n+1)$ or $b+n+1$, we will know that the machine must have entered some state twice while in the blank portion, therefore it will go on forever: we can stop the simulation and return 0. If, on the other hand, the machine always remains between cell $a-n$ and $b+n$, then it will either halt (then we return 1) or revisit some already visited configuration in terms of current cell and state; in such case we know that the machine won't stop because it will deterministically repeat the same steps over and over: we can then stop the simulation and return 0.

Exercise 2

2.1) Complete the proof of Theorem 9 by writing down, given a positive integer n, an n-state Turing machine on alphabet $\{0,1\}$ that starts on an empty (i.e., all-zero) tape, writes down n consecutive ones and halts below the rightmost one.
2.2) Test it for n=3.

Solution 2

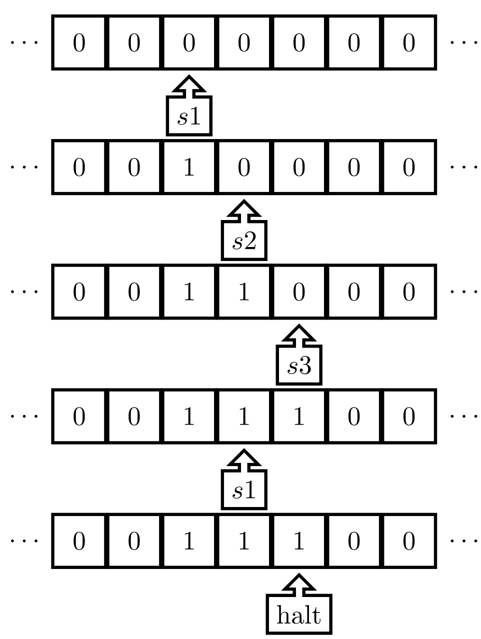
2.1) Here is a possible solution:

	0	1
s_1	1, right, s_2	1, right, halt
s_2	1, right, s_3	—
	$\vdots$	
s_i	1, right, s_{i+1}	—
	$\vdots$	
s_{n-1}	1, right, s_n	—
s_n	1, left, s_1	—

Entries marked by "—" are irrelevant, since they are never used. Any state can be used for the final move.
2.2) For $n = 3$, the machine is

	0	1
s_1	1, right, s_2	1, right, halt
s_2	1, right, s_3	—
s_3	1, left, s_1	—

Here is a simulation of the machine, starting on a blank (all-zero) tape:

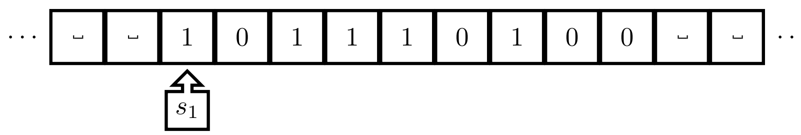


Exercise 3

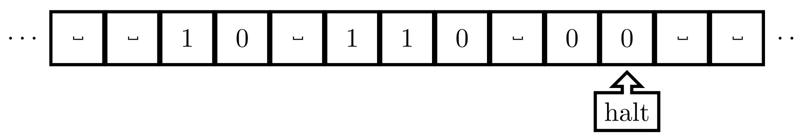
3.1) Write a Turing machine according to the following specifications:

- the alphabet is $\Sigma = \{␣, 0, 1\}$, where '␣' is the default symbol;
- it has a single, bidirectional and unbounded tape;
- the input string is a finite sequence of symbols in $\{0, 1\}$, surrounded by endless '␣' symbols on both sides;
- the initial position of the machine is on the leftmost symbol of the input string;
- every '1' that immediately follows '0' must be replaced with '␣' (i.e., every sequence '01' must become '0␣').
- the final position of the machine is at the righmost symbol of the output sequence.

For instance, in the following input case



the final configuration should be



You can assume that there is at least one non-'␣' symbol on the tape, but considering the more general case in which the input might be the empty string is a bonus.

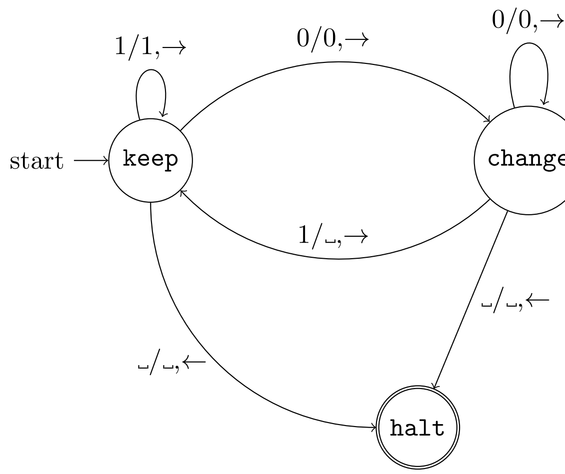
3.2) Show the sequence of steps that your machine performs on the input

"010011000111"

Solution 3

Two possible representations of the Turing machine are shown below; many other representations and transition rule sets are possible.

	␣	0	1
`keep`	␣/←/`halt`	0/→/`change`	1/→/`keep`
`change`	␣/←/`halt`	0/→/`change`	␣/→/`keep`



1/1,→
0/0,→
0/0,→
start
keep
change
1/␣,→
␣/␣,←
␣/␣,←
halt

Exercise 4

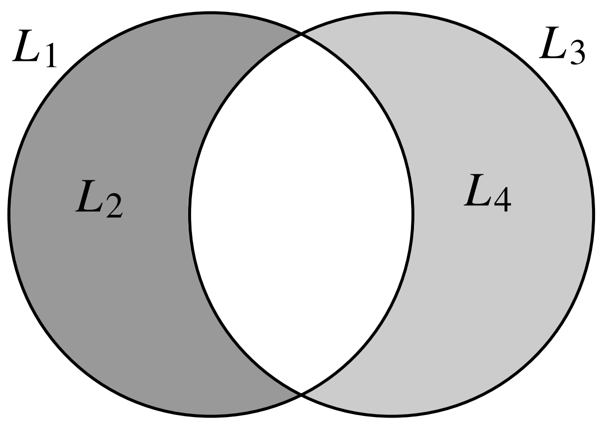
Let $\mathcal{M}$ represent a Turing Machine, let there be an encoding $s \to \mathcal{M}_s$ mapping string $s \in \Sigma^*$ to the TM $\mathcal{M}_s$ encoded by it. Finally, remember that in our notation $\mathcal{M}(x) = \infty$ means "$\mathcal{M}$ does not halt when executed on input x". Consider the following languages:

$$\begin{aligned}
L_1 &= \{s \in \Sigma^* \,|\, \exists x \mathcal{M}_s(x) \neq \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ halts on some inputs}\} \\
L_2 &= \{s \in \Sigma^* \,|\, \forall x \mathcal{M}_s(x) \neq \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ halts on all inputs}\} \\
L_3 &= \{s \in \Sigma^* \,|\, \exists x \mathcal{M}_s(x) = \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ doesn't halt on some inputs}\} \\
L_4 &= \{s \in \Sigma^* \,|\, \forall x \mathcal{M}_s(x) = \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ doesn't halt on any input}\}
\end{aligned}$$

4.1) Provide examples of TMs $\mathcal{M}_1, \dots, \mathcal{M}_4$ such that $\mathcal{M}_1 \in L_1, \dots, \mathcal{M}_4 \in L_4$.
4.2) Describe the set relationships between the four languages (i.e., which languages are subsets of others, which are disjoint, which have a non-empty intersection).

Solution 4

Observe that this exercise has very little to do with computability; however, being able to understand and answer it is a necessary prerequisite to the course. **4.1)** The machine that immediately halts ($s_0 = \texttt{HALT}$) is an example for L_1 and L_2. The machine that never halts (e.g., always moving right and staying in state s_0) is an example for L_3 and L_4.
4.2) If a machine always halts, it clearly halts on some inputs; therefore, $L_2 \subset L_1$ (equality is ruled out by the fact that there are machines that halt on some inputs and don't on others: $L_1 \cap L_3 \neq \emptyset$). With similar considerations, we can say that $L_4 \subset L_3$.
L_2 is disjoint from both L_3.
Also, observe that $L_2 = L_1 \setminus L_3$ and $L_4 = L_3 \setminus L_1$.
The relationship among the sets can be shown in the following diagram:



Exercise 5

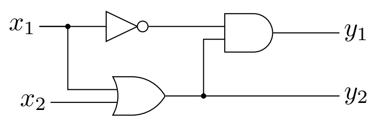
For each of the following properties of TMs, say whether it is semantic or not, and prove whether it is decidable or not.
5.1) $\mathcal{M}$ decides words with an 'a' in them.
5.2) $\mathcal{M}$ always halts within 100 steps.
5.3) $\mathcal{M}$ either halts within 100 steps or never halts.
5.4) $\mathcal{M}$ decides words from the 2018 edition of the Webster's English Dictionary.
5.5) $\mathcal{M}$ never halts in less than 100 steps.
5.6) $\mathcal{M}$ is a Turing machine.
5.7) $\mathcal{M}$ decides strings that encode a Turing machine (according to some predefined encoding scheme).
5.8) $\mathcal{M}$ is a TM with at most 100 states.

Solution 5

5.1) The property is semantic, since it does not depend on the specific machine but only on the language that it recognizes. The property is also non-trivial (it can be true for some machines, false for others), therefore it satisfies the hypotheses of Rice's theorem. We can safely conclude that it is uncomputable.
Note: the language "All words with an 'a' in them" is *computable. What we are talking about here is the "language" of all Turing machines that recognize it.*
5.2) Since we can always add useless states to a TM, given a machine M that satisfies the property, we can always modify it into a machine M' such that $L(M) = L(M')$, but that runs for more than 100 steps. Therefore the property is not semantic. It is also decidable: in order to halt within 100 steps, the machine will never visit more than 100 cells of the tape in either direction, therefore we "just" need to simulate it for at most 100 steps on all inputs of size at most 200 (a huge but finite number) and see whether it always halts within that term or not.
5.3) Again, the property is not semantic: different machines may recognize the same language but stop in a different number of steps. In this case, it is clearly undecidable: just add 100 useless states at the beginning of the execution and the property becomes "M never halts".
5.4) The property is semantic, since it only refers to the language recognized by the machine, and is clearly non-trivial. Therefore it satisfies Rice's Theorem hypotheses and is uncomputable. *Note: as in point 5.1, the language "all words in Webster's"* is *computable, but we aren't able to always decide whether a TM recognizes it or not.*
5.5) This is the complement of property 5.2, therefore not semantic and decidable.
5.6) The property is trivial, since all TMs trivially have it. Therefore, it is decidable by the TM that always says "yes" with no regard for the input.
5.7) The property is semantic because it refers to a specific language (strings encoding TMs). It is not trivial: even if the encoding allowed for all strings to be interpreted as a Turing machine, the only machines that possess the property would be those that recognize every string.
5.8) Deciding whether a machine has more or less than 100 states is clearly computable by just scanning the machine's definition and counting the number of different states. The property is not semantic.

Exercise 6

Consider the following Boolean circuit:



6.1) Write down the CNF formula that is satisfied by all and only combinations of input and output values compatible with the circuit.
6.2) Is it possible to assign input values to x_1, x_2 such that $y_1 = 0$ and $y_2 = 1$? Provide a CNF formula that is satisfiable if and only if the answer is yes.
6.3) Is it possible to assign input values to x_1, x_2 such that $y_1 = 1$ and $y_2 = 0$? Provide a CNF formula that is satisfiable if and only if the answer is yes.

Solution 6

6.1) Let g_1 be the variable associated to the NOT gate; the other two gates are already associated to the circuit's outputs. The formula, obtained by combining the equations in Fig. 3.2 is therefore:

$$\begin{aligned}
f(x_1,x_2,g_1,y_1,y_2) \quad = \quad & (\neg y_2 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (\neg x_2 \vee y_2) \\
& \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g_1) \\
& \wedge (\neg y_2 \vee \neg g_1 \vee y_1) \wedge (\neg y_1 \vee y_2) \wedge (\neg y_1 \vee g_1).
\end{aligned}$$

The first line describes the OR gate, the second the NOT, the thirs the AND.
6.2) Let us set $y_1 = 0$ and $y_2 = 1$ in f and simplify:

$$\begin{aligned}
f'(x_1,x_2,g_1) \quad = \quad & f(x_1,x_2,g_1,0,1) \\
= \quad & (\emptyset \vee x_1 \vee x_2) \wedge \cancel{(\neg x_1 \vee 1)} \wedge \cancel{(\neg x_2 \vee 1)} \\
& \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g1) \\
& \wedge (\emptyset \vee \neg g_1 \vee \emptyset) \wedge \cancel{(1 \vee 1)} \wedge \cancel{(1 \vee g_1)} \\
= \quad & (x_1 \vee x_2) \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g1) \wedge \neg g_1.
\end{aligned}$$

Note that f' is satisfiable: the last clause obviously requires $g_1 = 0$, after which the second clause implies $x_1 = 1$ and the value of x_2 becomes irrelevant. Therefore, by just setting $x_1 = 1$ the circuit will provide the required output.
6.3) Let us perform the substitution:

$$\begin{aligned}
f''(x_1,x_2,g_1) \quad = \quad & f(x_1,x_2,g_1,0,1) \\
= \quad & \cancel{(1 \vee x_1 \vee x_2)} \wedge (\neg x_1 \vee \emptyset) \wedge (\neg x_2 \vee \emptyset) \\
& \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g_1) \\
& \wedge \cancel{(1 \vee \neg g_1 \vee 1)} \wedge \overbrace{(0 \vee 0)}^{\text{unsatisfiable}} \wedge (\emptyset \vee g_1),
\end{aligned}$$

which, because of the second-to-last clause, cannot be satisfied. Therefore, the circuit cannot have the required output.

Exercise 7

Show that SAT $\leq_p$ ILP by a direct reduction.

Hint — *Given a CNF formula f, represent its variables as variables in an integer program. Use constraints to force every variable in $\{0,1\}$ and other constraints to force every clause to have at least one true literal.*

Solution 7

Let $x_1, \dots, x_n$ be the variables of f. We can directly map them to n variables of an ILP.
The first constraint is that every variable must be 0 or 1; this can be translated into two constraints per variable:

$$-x_i \leq 0, \quad x_i \leq 1 \qquad \text{for } i = 1, \dots, n;$$

Next, every clause must be true. We can translate this into one constraint per clause, where we require that the sum of the literals that compose it is not zero. Literal x_i is mapped onto itself, while a negated literal $\neg x_i$ can be translated into the "arithmetic" equivalent of negation $1 - x_i$. For instance, clause $(x_3 \vee \neg x_9 \vee x_{16})$ is rendered into

$$x_3 + (1 - x_9) + x_{16} \geq 1, \qquad \text{i.e.,} \qquad - x_3 + x_9 - x_{16} \leq 0.$$

Therefore, a CNF formula with n variables and m clauses is mapped onto a ILP problem with n variables and $2n + m$ constraints.

$\boxed{\textbf{Exercise 8}}$

Consider the SET PACKING problem: given n sets $S_1, \dots, S_n$ and an integer $k \in \mathbb{N}$, are there k sets $S_{i_1}, \dots, S_{i_k}$ that are mutually disjoint?
8.1) Prove that SET PACKING $\in$ **NP**.
8.2) Prove that SET PACKING is **NP**-complete.

Hint — *You can prove the completeness by reduction of INDEPENDENT SET.*

$\boxed{\textbf{Solution 8}}$

8.1) The certificate is the subset of indices $i_1, \dots, i_k$; we just need to check that they are k different indices and that the corresponding sets are disjoint, and both tests are clearly polynomial in the problem size.
8.2) Let $G = (V, E)$ a graph, and we are asked if it has an independent set of size k.
For every node $i \in V$, consider the set of its edges $S_i = \big\{\{i, j\} \in E\big\}$. Given two vertices $i, j \in V$, the only element that can be shared between the corresponding sets S_i and S_j is a common edge, i.e., edge $\{i, j\}$. Therefore, two vertices are disconnected in G (there is no edge between them) if and only if the corresponding sets S_i and S_j are disjoint. Thus, k mutually independent vertices $i_1, i_2, \dots, i_k$ correspond to k mutually disjoint sets $S_{i_1}, S_{i_2}, \dots, S_{i_k}$.
This reduction is clearly polynomial.

Exercise 9

Tweak the proof of Theorem 24 in order to reduce VERTEX COVER (in place of INDEPENDENT SET) to ILP.

Hint — *We need to transform the condition "every edge has at most one endpoint in the set", used in the aforementioned theorem, into the condition "every edge has* at least *one endpoint in the set"; the condition "there must be at least k 1's" must become "there must be* at most *k 1's".*

Solution 9

Following the suggestion, the condition that the selected vertices must be part of a vertex cover becomes $x_i + x_j \geq 1$, therefore $-x_i - x_j \leq -1$; equivalently, the size condition becomes $x_1 + \cdots + x_{|V|} \leq k$. The whole reduction becomes therefore:

$$\begin{cases} -x_i \leq 0 & \forall i \in V \\ x_i \leq 1 & \forall i \in V \\ -x_i - x_j \leq -1 & \forall \{i,j\} \in E \\ x_1 + \ldots + x_{|V|} \leq k & \end{cases}$$

Exercise 10

A *tautology* is a formula that is always true, no matter the truth assignment to its variables.
A Boolean formula is in *disjunctive normal form* (DNF) if it is written as the disjunction of clauses, where every clause is the conjunction of literals (i,e,, like CNF but exchanging the roles of connectives).
Let TAUTOLOGY be the language of DNF tautologies. Prove that TAUTOLOGY $\in$ **coNP**.

Hint — *You can do it directly (by applying any definition of* ***coNP****), or by observing that a tautology is the negation of an unsatisfiable formula, and that the negation of a CNF leads to a DNF.*

Solution 10

The suggestion says it all: a **coNP** certificate is any truth assignment that falsifies the DNF formula (thus proving that it is not a tautology).
Alternatively, let f be a CNF formula: f is unsatisfiable if and only if $\neg f$ is a tautology, and by applying the De Morgan rules we can write $\neg f$ as a DNF formula.

Exercise 11

Show that $\mathbf{P} \subseteq \mathbf{ZPP}$.

Hint — *A polynomial-time language is automatically in **ZPP** because…*

Solution 11

We can obviously define $\mathbf{P}$ as the class of languages for which either all computations accept or all reject. Therefore, the fraction of accepting computations (for $\mathbf{RP}$) and of rejecting computations (for $\mathbf{coRP}$) satisfies any threshold ε.
Or we can say that there is a TM $\mathcal{M}$ such that

$$\forall x \in \Sigma^* \qquad \Pr\bigl(M(x) \text{ accepts}\bigr) \text{ is } \begin{cases} 0 & \text{if } x \notin L \\ 1 & \text{if } x \in L, \end{cases}$$

which clearly falls into the characterization of $\mathbf{RP}$ given by (4.3). Same for the rejection probability in $\mathbf{coRP}$.

Exercise 12

Show that $\mathbf{RP} \subseteq \mathbf{BPP}$.

Hint — *The condition for a language to be in* $\boldsymbol{RP}$ *can be seen as a further restriction on those imposed on* $\boldsymbol{BPP}$.

Solution 12

The characterization (4.3) of $\mathbf{RP}$ implies the characterization from Definition 37 as soon as $\varepsilon \geq 2/3$. However, we know by application of the probability boosting algorithm, that all thresholds $0 < \varepsilon < 1$ define the same class.

Exercise 13

Show that **BPP** $\subseteq$ **PP**.

Hint — *The conditions for a language to belong to a class automatically satisfy those for the other.*

Solution 13

The suggestion says it all.

Exercise 14

The following statement is actually true, but what's wrong with the proof provided here?

Theorem: **NPSPACE** $\subseteq$ **PSPACE**

Proof — Let $L \in$ **NPSPACE**, let $\mathcal{N}$ be the NDTM that decides it in polynomial space, and let x be an input string. Since we have no time bounds, we can emulate all possible computations of $\mathcal{N}(x)$, one after the other, until one of them accepts x or we exhaust all of them. Of course, there is an exponential number of computations, but we have no time bounds, and each computation only requires polynomial space by definition: the tape can be reused between computations.
Therefore, we can emulate $\mathcal{N}$ by a deterministic, polynomial-space TM $\mathcal{M}$, and $L \in$ **PSPACE**, thus proving the assertion. □

Solution 14

The "proof" doesn't take into account the fact that a polynomial space-bounded computation can have exponential time; while time is not a problem per se, emulating non-deterministic computations requires to keep track of the non-deterministic choices by maintaining one bit at every step, therefore a deterministic stepwise emulator requires an exponential amount of space.

Exercise 15

In a population of n people, every individual only talks to individuals whom he knows. Steve needs to tell something to Tracy, but he doesn't know her directly.
We are provided with a (not necessarily symmetric) list of "who knows whom" in the group, and we are asked to tell whether Steve will be able to pass his information to Tracy via other people.
15.1) Is there a polynomial-time algorithm to give an answer (assume any realistic computing model you like)? If yes, describe it; if not (or if you cannot think of any), explain what is the main obstacle.
15.2) Is there a polynomial-space algorithm? Can we do any better? Describe the most space-efficient implementation that you can think of.

Solution 15

Just STCON in disguise (Steve is s, Tracy is t).
15.1) Graph connectivity is polynomial. For instance, we could create a spanning tree starting from s and see if it ever reaches t.
15.2) Any reasonable graph exploration algorithm is polynomial space-bounded: we just need to keep track of what nodes have already been visited and, possibly, a queue of "current" nodes.
However, we have seen a much more space-efficient $O\big((\log n)^2\big)$ implementation when proving Savitch's theorem.

Exercise 16

A graph $G = (V, E)$ is connected if there is a path between every pair of nodes. Show that the language of connected graphs is in **NL**.

Solution 16

An implementation just needs to iterate between pairs of nodes in V (i.e., two logarithmic-space counters) and run STCON on each pair (which we already know to be **NL**).

Exercise 17

Let L be a language, and let $\mathcal{N}$ be a non-deterministic Turing Machine that decides $x \in L$ in time $O(|x|^3 \log |x|)$.

17.1) Suppose that, whenever $x \in L$, at least 15 computations of $\mathcal{N}(x)$ accept; what probabilistic complexity classes does L belong to, and why?

17.2) Suppose that, whenever $x \in L$, at most 15 computations of $\mathcal{N}(x)$ do not accept; what probabilistic complexity classes would L belong to, and why?

Hint — *Consider the following classes:* ***RP****,* ***coRP****,* ***ZPP****,* ***BPP****,* ***PP****. Bonus points if you also consider* ***P*** *and* ***NP****.*

Solution 17

17.1) Clearly, $L \in \mathbf{NP}$, because a non-deterministic TM decides it in polynomial time, and therefore $L \in \mathbf{PP}$ (but $\mathcal{N}$ must be tweaked in order to meet the definition). Observe that, if $x \in L$, the guaranteed ratio of accepting computations (which is a constant 15) to the total number tends to zero as the input size grows: the number of possible computations grows exponentially with the computation time. Therefore, there is no $\varepsilon > 0$ such that

$$\frac{15}{\text{Number of computations}} > \varepsilon;$$

this means that the existence of $\mathcal{N}$ alone does not guarantee that L belongs to any other probabilistic class (they all require a finite, nonzero bound).

17.2) Again, $L \in \mathbf{NP}$ for the same reason as above. This time, if $x \in L$, *almost all computations accept*: only a small, residual number (15 against an exponentially growing number) keep rejecting valid inputs. Since a very large fraction of computations (almost 100%) accepts valid inputs, and all invalid ones are rejected, the machine satisfies the definition of **RP**.

Observations

- Actually, in the case 17.2 we could say even more: $L \in \mathbf{P}$. In fact, we just need to emulate $16 = 15 + 1$ computations of the NDTM (each being in polynomial time): if $x \in L$, even in the worst case one of the computations will accept, otherwise all of them will reject. As a consequence, L belongs to *all* probabilistic classes that we defined.

- Saying "suppose that the total number of computations is 30, then the ratio is 1/2" doesn't make sense: as said above, the number of computations is unbounded, and grows very quickly.

- $O(n^3 \log n)$ *is* polynomial, since $\log n = O(n)$.

Exercise 18

An examiner must plan an oral exam for N students, where every student is asked one, and one only, question.
The examiner has the following information:

- a list of pairs of students who know each other (suppose that the relation is symmetric, but not transitive), and
- a number, k, of questions that she can ask.

We must determine whether the number of questions, k, is sufficient to avoid that two students knowing each other are asked the same question.
18.1) Describe a polynomial-time algorithm that decides the decision problem defined above when $k = 2$.
18.2) Prove that the problem is **NP**-complete in the general case (you can assume the **NP**-completeness of a language if it has been discussed in class).

Solution 18

The problem is equivalent to k-VERTEX COLORING, where students are vertices, edges are pairs of students who know each other, colors are questions.
18.1) Any polynomial solution for 2-coloring (or, equivalently, to verify if a graph is bipartite) is fine. For every connected component, start by assigning the first color to an arbitrary node; pick any node that has already been colored, and give the opposite color to its neighbors; if this is impossible (a neighbor already has the same color), halt and reject. Whenever all nodes are colored, accept.
18.2) A polynomially verifiable certificate could be, for instance, a question assignment to students. Reduction from k-VERTEX COLORING: for every vertex, let there be a student; for every edge, let the two correspooinding students know each other. Let there be k questions. There is a color assignment if and only if there is a question assignment.

Observations

- Note that the first point asked for an algorithm (in any form, even a verbal description). Therefore, simply answering "2-coloring is **P**" wouldn't grant full marks.
- Other reductions are possible, of course, provided that the answer is motivated.

Exercise 19

The police must intercept all cellphone communications within a group of n people and have the following information:

- their identities and phone numbers (plus any other info that is needed in such cases);
- which pairs of people know each other (people who don't know each other will not directly communicate).

They want to know if there is a way to be sure to intercept all calls within group members while putting only k phones under surveillance (we assume that a communication can be intercepted if at least one of the two phones is under surveillance).

19.1) Prove that the problem is **NP**.

19.2) Prove that the problem is **NP**-complete by reduction from some other known problem.

Solution 19

19.1) The certificate is the list of k people to be put under surveillance. It is clearly polynomial wrt the problem size (it is a subset of the n people), and we just need to check that each of the n people is either in the list, or knows someone in the list. We can run this check in quadratic time (on a computer, a little more on a TM).

19.2) We can reduce VERTEX COVER to this problem: given an undirected graph $G = (V, E)$, let us build a set of $n = |V|$ people, and let persons i and j know each other iff $\{i, j\} \in E$. Then, G has a vertex cover of size k iff we can intercept all communications by putting k people under surveillance.

Observations

- Basically, the stated problem is VERTEX COVER under disguise. The observation that the problem is VERTEX COVER and therefore it is **NP**-complete would guarantee maximum marks.
- As usual, pay attention to the sense of the reduction. Reducing our problem to something else would prove nothing.

Exercise 20

Let n-SUBSET SUM be a restriction of SUBSET SUM to only instances of precisely n numbers. The problem size can still be arbitrarily large, because the numbers may be as large as we want.
Would 1000000-SUBSET SUM be still in **NP**? Would it be **NP**-complete? Would it fall back to **P**?

Solution 20

Clearly, the problem is still in **NP** because checking a restricted version is never harder than checking the unrestricted one.
However, the language is not complete (unless $\mathbf{P} = \mathbf{NP}$) because it is actually polynomial.
Consider the naive algorithm that iterates through all 2^n subsets of numbers and for each computes the corresponding sum, comparing it to s. Let l be the maximum length of the numbers' representation. Then every som requires time $O(nl)$ (adding at ost n l-bit numbers), therefore the complete algorithm runs within a $O(2^n nl)$ time bound (give or take some polynomial slowdown due to TM quirks).
Since n is constant, the complexity becomes

$$O(2^n nl) = O(1000000 \cdot 2^{1000000} l) = O(\text{constant} \cdot l) = O(l),$$

which is clearly linear in the problem size.

Exercise 21

Let M-SUBSET SUM be a restriction of SUBSET SUM to only instances where all numbers $x_1, \ldots, x_n$ in the set (including the sum s) are not larger than M. The problem size can still be arbitrarily large, because the set can contain as many numbers as we want.
Would 1000000-SUBSET SUM be still in **NP**? Would it be **NP**-complete? Would it fall back to **P**?

Solution 21

Observe that a previous version of this exercise allowed for an unbounded s, however the answer becomes more complex.
Whatever the number of elements n is, since the sum is bounded by M, we only need to iterate among all sets of size M or less. The number of subsets of size M is

$$\binom{n}{M} = O(n^M),$$

therefore we need to iterate among $O(Mn^M)$ subsets, considering also smaller set sizes. Considering the $O(N \log M)$ calculation of the sum (observe that it is constant with respect to the problem size, which in our case is only driven by n), the overall complexity is therefore

$$O(Mn^M N \log M) = O(\text{constant} \cdot n^M) = O(n^M),$$

which is polynomial wrt n (even though M might be a very large exponent).

Exercise 22

As an embedded system programmer, you are asked to design an algorithm that solves SUBSET SUM on a (deterministic!) device with a $O(\log n)$ additional space constraint.

22.1) Should you start coding right away, should you argue that the solution is probably beyond your capabilities, or should you claim that the task is infeasible?

22.2) What space constraint would you be comfortable with, among $O(\log n)$, $O\big((\log n)^2\big)$, $O(n)$, $O(n^2)$, $O(2^n)$?

Solution 22

22.1) You have been asked to solve a notoriously **NPSPACE**-complete problem in logarithmic space! We know that any algorithm that decides a language in logarithmic space must terminate in polynomial time (there is a polynomial number of distinct configuration, and if configuration is repeated the algorithm does not terminate). Therefore, you can only succeed if $\mathbf{P} = \mathbf{NP}$ (and even in that case you cannot be sure, because non all polynomial time-bounded algorithms run in logarithmic space). You should just point out that the general consensus is that the task is infeasible.

22.2) After excluding $O(\log n)$, observe that $O\big((\log n)^2\big)$ allows for $O(2^{(\log n)^2}) = O(n^{\log n})$ different configuration (hence steps) which, although superpolynomial (not bounded by any n^c), is less than exponential — a time bound still too small to be in your comfort zone for a potentially exponential **NP**-complete problem.

Having a set of n bits to iterate through all subsets and a little more space to accumulate sums is, however, more than enough. I would ask for linear space.

Exercise 23

Consider the following language on the two-symbol alphabet $\{\texttt{0}, \texttt{1}\}$:

$$L = \{\texttt{0}^n\texttt{1}^m \mid n, m \in \mathbb{N} \land n > m\}.$$

In plain terms, a string is in L if and only if it starts with a sequence of 0's followed by a (possibly empty) sequence of 1's and nothing else, with strictly more 0's than 1's.
Some examples:

$$\begin{array}{ccc} \texttt{00011} \in L & \texttt{00111} \notin L & \texttt{0} \in L \\ \texttt{1} \notin L & \texttt{10} \notin L & \texttt{11000} \notin L \\ \texttt{0110100} \notin L & \texttt{0000} \in L & \texttt{0011} \notin L \\ & \varepsilon \notin L. & \end{array}$$

23.1) Write down a one-tape deterministic Turing Machine $\mathcal{M}$ on the three-symbol alphabet $\{\texttt{0}, \texttt{1}, \sqcup\}$ that, given an input string $s \in \{\texttt{0}, \texttt{1}\}^*$, decides $s \in L$.
You may assume that the input string s is surrounded by infinite blank cells $\sqcup$ in both directions, and that the initial current position is the leftmost symbol of s.
23.2) What is the time complexity of your machine $\mathcal{M}$?
More precisely: if n is the input size, what is the smallest exponent k such that $\mathcal{M} \in \mathrm{DTIME}(n^k)$? Explain briefly.

Solution 23

23.1) A TM could, for example, keep erasing the leftmost zero and the rightmost one, until only zeroes remain. Any other unexpected condition (zero following a one, no zeroes, and so forth) must cause rejection.
As an example, here is a description of a machine that can be copied and pasted on `http://morphett.info/turing/turing.html`:

```
; Recognize 0^m1^n with m>n.

; Go back and forth, repeatedly removing leftmost 0 and rightmost 1
; until just excess 0's remain. Any other outcome causes rejection.
; The initial state is erase_leftmost_0

; Initially, erase the obligatory leftmost 0
erase_leftmost_0  0 _ r skip_0_right  ; skip all other 0's to the right
erase_leftmost_0  * * r halt-reject   ; Any other symbol, reject

; Keep skipping all 0's to the right
skip_0_right      0 0 r skip_0_right  ; As long as 0, skip it
skip_0_right      1 1 r skip_1_right  ; Upon 1, start skipping 1's
skip_0_right      _ _ l halt-accept   ; All 0's skipped, no 1's: OK

; After all 0's were skipped, start skippping 1's to the right
skip_1_right      1 1 r skip_1_right  ; Keep moving as long as it's 1
skip_1_right      _ _ l erase_rightmost_1 ; move back to erase the last 1
skip_1_right      0 0 r halt-reject   ; After the 1's, there shouldn't be 0's

; Erase the rightmost 1
erase_rightmost_1 1 _ l skip_1_left   ; then start moving left

; Move to the left skipping all 1's
skip_1_left       1 1 l skip_1_left   ; keep skipping 1's
skip_1_left       0 0 l skip_0_left   ; all 1's were skipped, start with 0's
skip_1_left       _ _ l halt-reject   ; no 0's means error
```

```
; Keep moving to the left skipping all 0's
skip_0_left       0 0 l skip_0_left   ; as long as there are 0's
skip_0_left       _ _ r erase_leftmost_0 ; passed the whole string, restart
```

Clearly, any description is acceptable, and some missed halting conditions can be forgiven.

23.2) The machine performs back and forth passes on the n-symbol input string, and removes a symbol with every pass. Therefore, its time complexity is $O(n^2)$.

Exercise 24

Prove that the language L defined in Exercise 23 belongs to the complexity class **L**.

Solution 24

A two-tape Turing Machine just needs to initialize a counter to zero on the second tape, then scan the input string; it increments the counter as long as it finds a zero, then decrements it as long as it finds ones, halting in any other case or when the counter returns to zero.
Since the counter never counts more than the number of input symbols, its size is logarithmic with respect to the size of the input.

$\boxed{\textbf{Exercise 25}}$

Is it always possible for an instructor to correctly evaluate a student's answer to Exercise 23? Explain.

$\boxed{\textbf{Solution 25}}$

Giving a positive or negative evaluation to the student's answer amounts to deciding the properties "$\mathcal{M}$ decides L" and "$\mathcal{M}$ doesn't decide L" for the machine $\mathcal{M}$ that he described.
Both properties are semantic, therefore by Rice's Theorem they cannot be decided: there will be some machines for which the instructor won't be able to say whether they correctly answer the question or not.
In other words, the student could embed in the solving machine another TM whose halting property the instructor could not be able to prove.

Exercise 26

26.1) Let L be a language on a finite alphabet, and let $\mathcal{N}$ be a non-deterministic Turing machine on the same alphabet with the following properties:

- $\mathcal{N}(x)$ takes at most $|x|^2$ non-deterministic steps before halting ($|x|$ is the size of x).
- If $x \notin L$, then $\mathcal{N}(x)$ rejects the input.
- If $x \in L$, then $\mathcal{N}(x)$ accepts the input.

Are these properties sufficient for us to say that $L \in \mathbf{NP}$?

26.2) Suppose that $\mathcal{N}$ has the following additional property:

- At every step, $\mathcal{N}$ performs at most one binary non-deterministic choice (i.e., $\mathcal{N}$ has two transition functions)

Given this property and those listed in the previous point, determine an upper bound for the number $C_{\mathcal{N}}(x)$ of non-deterministic computations performed by $\mathcal{N}$ on input x as a function of the input size $|x|$.

26.3) Suppose that $\mathcal{N}$ has the following additional property:

- If $x \in L$, out of the $C_{\mathcal{N}}(x)$ computations of $\mathcal{N}(x)$, at least $\sqrt{C_{\mathcal{N}}(x)}$ end in an accepting state.

Is this additional property (and the previous ones) sufficient for us to say that $L \in \mathbf{RP}$?

Solution 26

26.1) Yes, the three properties are precisely the ones that define the class **NP** (Non-deterministic Polynomial). The first property ensures that $\mathcal{N}$ always halts within a polynomial number of steps with respect to the input size; the second and third property just say that $\mathcal{N}$ decides L.

26.2) A computation of $\mathcal{N}(x)$ takes at most $|x|^2$ steps. At every step, a computation can take two alternative paths, "branching" into two computations; i.e., the computational paths split into two (at most) at every step until completion, therefore ending in $2^{|x|^2}$ leaves.

26.3) Let $x \in L$ and $|x| = n$. All that we know about the ratio of accapting computations is

$$\frac{\text{number of accepting computations}}{\text{number of computations}} \geq \frac{\sqrt{C(n)}}{C(n)} = \frac{1}{\sqrt{C(n)}},$$

but this bound tends to zero as $C(n)$ increases. Therefore, there is no constant $\varepsilon > 0$ bounding the ratio from below. Another way to say it is that the ratio of accepting computations is vanishingly small as the input size increases. Therefore, the property isn't enough for us to say that $L \in \mathbf{RP}$.

Exercise 27

Consider the following language in $\{0,1\}^*$:

$$K = \{0^n 1^n : n \in \mathbb{N}\} = \{\varepsilon, 01, 0011, 000111, 00001111, 0000011111, \dots\}$$

i.e., all strings composed by a sequence of zeroes followed by the *same* number of ones.
27.1) Write a single-tape Turing Machine with alphabet $\Sigma = \{\textvisiblespace, 0, 1\}$ that recognizes K.
27.2) Prove or disprove the decidability of each of the following properties of TMs:

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M} \text{ decides } K\}, \\
\mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ decides } K \text{ in less than 100 steps}\}, \\
\mathcal{P}_3 &= \{\mathcal{M} : \mathcal{M} \text{ decides } K \cap \Sigma^{100} \text{ (i.e., strings in } K \text{ not longer that 100 symbols)}\}.
\end{aligned}$$

Hint — *For 27.1 use any notation you like, and encode acceptance and rejection as you prefer (0/1 on tape, two different halting states, etc.).*

Solution 27

27.1) The simplest, although, not the most efficient, machine just keeps erasing the leftmost 0 and the rightmost 1 until the input is empty or some unexpected symbol appears (e.g., leftmost 1, righmost 0, blank when a 1 should be erased).
We assume that the input is a contiguous string of 0's and 1's, surrounded by blanks, and that the machine starts on the leftmost input symbol. Here is the transition table:

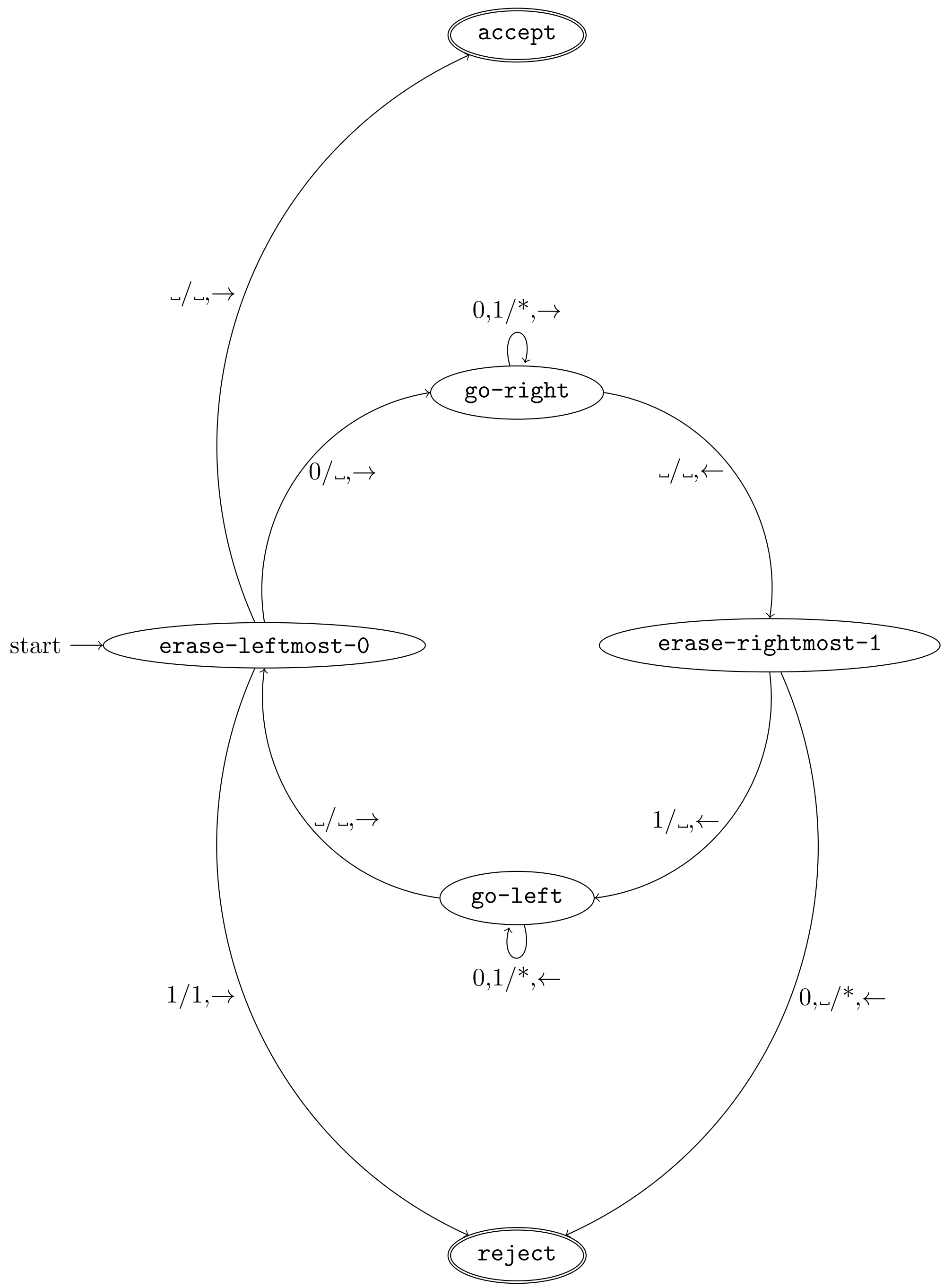
	␣	0	1
`erase-leftmost-0`	␣/→/`accept`	␣/→/`go-right`	1/→/`reject`
`go-right`	␣/←/`erase-rightmost-1`	0/→/`go-right`	1/→/`go-right`
`erase-rightmost-1`	␣/←/`reject`	0/←/`reject`	␣/←/`go-left`
`go-left`	␣/→/`erase-leftmost-0`	0/←/`go-left`	1/←/`go-left`

An encoding suitable for the TM simulator seen in class[1] is:

```
erase-leftmost-0 0 _ r go-right    ; found and erased a 0
erase-leftmost-0 1 1 r halt-reject ; unexpected 1
erase-leftmost-0 _ _ r halt-accept ; the input has been consumed

go-right 0 0 r go-right          ; keep skipping the input
go-right 1 1 r go-right
go-right _ _ l erase-rightmost-1 ; found the end of the input

erase-rightmost-1 1 _ l go-left     ; found and erased a 1
erase-rightmost-1 0 0 l halt-reject ; unexpected 0
erase-rightmost-1 _ _ l halt-reject ; unexpected blank

go-left 0 0 l go-left          ; keep skipping the input
go-left 1 1 l go-left
go-left _ _ r erase-leftmost-0 ; found the beginning of the input
```

The same machine as an automaton form:

[1] `http://morphett.info/turing/turing.html`



27.2)

- Property $\mathcal{P}_1$ is clearly semantic ($\mathcal{M} \in \mathcal{P}_1 \Leftrightarrow L(\mathcal{M}) = K$) and is not trivial (there is at least one machine that decides K and at least one that doesn't); therefore, by Rice's Theorem, it is undecidable.

- A TM limited to 100 steps cannot decide K. Consider, e.g., the string $s_1 = 0^{1000}1^{1000} \in K$. A TM limited to 100 steps wouldn't be able to read the whole input, therefore it wouldn't be able to tell s_1 from $s_2 = 0^{1000}1^{1001} \notin K$. Therefore, $\mathcal{P}_2 = \emptyset$, hence it is trivially computable by a TM that always rejects.

- Again, $\mathcal{P}_3$ is semantic and non-trivial, thus uncomputable.

Observations

- Many other TMs are possible for 27.1.
- Observe that, since 27.1 requires the TM to just *recognize* K, rejection could be replaced by a non-halting computation.
- As usual, there is a significant distinction between the computability of K and the computability of the property "This machine decides K".
- Property $\mathcal{P}_2$ doesn't just require the TM to halt after 100 steps, but also to decide K. Therefore, simulating the TM for 100 steps isn't enough: we also need to consider which inputs it should be simulated on.
- The fact that the language defining $\mathcal{P}_3$ is finite doesn't matter: Rice's theorem is still valid, because we wouldn't be able to always assert whether a TM would halt or not.

Exercise 28

Let $L_1, L_2 \in \mathbf{NP}$. Does $L_1 \cup L_2 \in \mathbf{NP}$? Does $L_1 \cap L_2 \in \mathbf{NP}$? Why?

Hint — *Be as formal as you can, e.g.: "Since $L_1 \in \boldsymbol{NP}$, then there is a TM $\mathcal{M}_1$ such that…"*

Solution 28

Since $L_1 \in \mathbf{NP}$, then there is a NDTM $\mathcal{N}_1$ that decides L_1 in polynomial time. Same for L_2.
Given input x, to decide whether $x \in L_1 \cup L_2$ we just need a NDTM that accepts x whenever $\mathcal{N}_1$ *or* $\mathcal{N}_2$ accepts it:

- Store x for future use.
- Run $\mathcal{N}_1$ on input x. If $\mathcal{N}_1$ accepts, then accept and halt.
- Restore input x.
- Run $\mathcal{N}_2$ on input x.

This machine runs in time that is, in the worst case, the sum of the times of $\mathcal{N}_1(x)$ and $\mathcal{N}_2(x)$ plus the time to copy and restore x, therefore it is polynomial in $|x|$.
Likewise, to decide whether $x \in L_1 \cap L_2$ we need a NDTM that accepts x whenever $\mathcal{N}_1$ *and* $\mathcal{N}_2$ accepts it:

- Store x for future use.
- Run $\mathcal{N}_1$ on input x. If $\mathcal{N}_1$ rejects, then reject and halt.
- Restore input x.
- Run $\mathcal{N}_2$ on input x.

The worst-case runtime is the same of the previous machine.

Observations

- The fact that $L_1 \cap L_2$ is (in some sense) "smaller" than both L_1 and L_2 doesn't mean that it is "easier", nor that $L_1 \cup L_2$ is "harder".
- Also remember that the exercise doesn't cite completeness.

Exercise 29

Consider the following classical **NP**-complete languages:

$$\begin{aligned}\text{CLIQUE} &= \{(G,k) : \text{Undirected graph } G \text{ has a completely connected subgraph of size } k\},\\ \text{INDSET} &= \{(G,k) : \text{Undirected graph } G \text{ has a completely disconnected subgraph of size } k\}.\end{aligned}$$

29.1) Describe a polynomial-time reduction from one language to the other.
29.2) Show that $\text{CLIQUE} \cap \text{INDSET} \neq \emptyset$.

Hint — *For 29.1, choose the direction you like. In 29.2, don't be afraid of simple answers: to show that a set is not empty, you just need to find an element in it.*

Solution 29

29.1) See the notes: $G = (V, E)$ has a clique of size k if and only if $\bar{G} = (V, \bar{E})$ (same vertex set, complementary edge set) has an independent set of the same size.
29.2) We need to show that there is a graph G and an integer k such that G has both a clique of size k and an independent set of size k. Just take any nonempty graph G and $k = 1$:

$$(G, 1) \in \text{CLIQUE} \cap \text{INDSET}.$$

Exercise 30

Consider the UNIVERSITY HIRING decision problem:

> A university needs to hire the teaching staff for a new degree, for which a set T of topics must be taught. The executive board received n applications from prospective teachers, and every applicant $i \in \{1, \dots, n\}$ has knowledge of a subset $S_i \subseteq T$ of the required topics. The budget allows for hiring at most $k \le n$ teachers. Is there a choice of k applicants so that all teaching topics are covered?

An instance of the problem consists of parameters $n, k, T, S_1, \dots, S_n$.
30.1) Prove that UNIVERSITY HIRING $\in$ **NP**.
30.2) Prove by reduction that UNIVERSITY HIRING is complete in the class **NP**. The reduction can refer to any language discussed during the course.
30.3) Prove that if k is kept constant (e.g., $k = 10$), then the problem's asymptotic complexity is polynomial wrt input size.

Solution 30

30.1) Proving that an instance has positive answer only requires to provide the k indexes $(i_1, \dots, i_k)$ of the hired professors. The certificate is clearly polynomial wrt the input size (actually, much smaller). To verify that the certificate is valid, we just need to check (i) that the certificate contains (no more than) k numbers; (i) that all such numbers are different and between 1 and n; (iii) that every element in T apears in at least one of the topic subsets $S_{i_1}, \dots, S_{i_n}$.
30.2) UNIVERSITY HIRING is just a rephrasing of SUBBSET COVER. Formally, given an instance of SUBSET COVER, we map the union of all sets to T, and each of the sets to a different S_i.

30.3) If k is constant, rather than being a parameter in the instance, then the naïf algorithm that consists of generating and checking all possible certificates must iterate through

$$\binom{n}{k} = O(n^k)$$

different certificates, each of which can be checked in polynomial time.

Observations

- Beware of the direction of the reduction! You have to start from a *generic* instance of a known **NP**-complete problem and map it to a UNIVERSITY HIRING instance, not vice versa. Otherwise, you are only proving that your particular idea is not feasible, but you are not ruling out all possible methods. E.g., by reducing UNIVERSITY HIRING to SAT you just prove that using SAT is a bad idea, but you are not excluding the possibility that there are other, better reductions to problems in **P**!
- It was also possible to start from other known **NP**-complete problems. For example, starting from VERTEX COVER and then mimicking the reduction proposed in the course to SUBSET COVER.

$\boxed{\textbf{Exercise 31}}$

31.1) When is a language recursive? When is it recursively enumerable?
31.2) Prove that the following property $\mathcal{P}$ of Turing machines $\mathcal{M}$ is not recursive:

$$\mathcal{P} = \{\mathcal{M} : \mathcal{M}(\varepsilon) \text{ halts after an even number of steps}\}$$

where ε is the empty input string.
31.3) Prove that $\mathcal{P}$ is recursively enumerable.

Hint — *Point 31.3 can be proved by explicitly outlining an enumeration algorithm.*

$\boxed{\textbf{Solution 31}}$

31.1) See the definitions in the literature
31.2) The property is not semantic, therefore Rice's Theorem cannot be invoked.
Since we are dealing with machines running on an empty input, let us reduce the halting problem on empty input (that we called HALT_ε) to property $\mathcal{P}$, and suppose that. Clearly, $\mathcal{P}(\mathcal{M}) \Rightarrow \text{HALT}_\varepsilon(\mathcal{M})$. On the other hand, if $\mathcal{P}(\mathcal{M})$ is false, $\mathcal{M}$ might still be halting in an odd number of steps. In this case, let us add one dummy step to machine $\mathcal{M}$ before the halting state. Formally:

$$\text{HALT}_\varepsilon(\mathcal{M}) \quad \Leftrightarrow \quad \mathcal{P}(\mathcal{M}) \vee \mathcal{P}(\mathcal{M} + \text{dummy step before halting}).$$

Otherwise, consider any machine $\mathcal{M}$: if it halts, then the machine $\mathcal{M}+\mathcal{M}$ (consisting of two executions of $\mathcal{M}$) halts in an even number of steps:

$$\text{HALT}_\varepsilon(\mathcal{M}) \quad \Leftrightarrow \quad \mathcal{P}(\mathcal{M} + \mathcal{M}).$$

31.3) Consider a UTM that simulates a machine $\mathcal{M}$ and counts the number of steps. If the simulation halts, then $\mathcal{U}$ accepts $\mathcal{M}$ iff the number of steps was even. This machine satisfies the definition of recursive enumerability.
For an explicit enumeration algorithm, let us just employ the usual diagonalization method, but before writing out a halting machine we additionally check whether it halted in an even number of steps.

Observations

- Saying that a language is recursive if "we" can decide it is not the best way to formulate a definition, since "we" (or "you," or "I") is not a well defined computational model.
- Again, the direction of the reduction is fundamental. Reducing $\mathcal{P}$ to HALT proves nothing, we need to reduce HALT to $\mathcal{P}$, i.e., assume that $\mathcal{P}$ is decidable and try to solve HALT with it.
- For the reason above, just saying "in order to decide $\mathcal{P}(\mathcal{M})$ we would need to know if $\mathcal{M}$ halts, but this is not possible" doesn't prove anything, because you are only ruling out methods that use HALT to decide $\mathcal{P}$; however, that line of reasoning is not excluding other possible ways of deciding $\mathcal{P}$ that don't use HALT at all.

Exercise 32

Consider the following language:

$$S = \left\{ \left(x \in \{0,1\}^*, k \in \mathbb{N} \right) : x \text{ contains a subsequence of } k \text{ adjacent 0's} \right\}.$$

For example, $(101000101, 3) \in S$ because the binary string contains 3 consecutive zeroes, while $(101000101, 4) \notin S$ because the binary string does not contain 4 consecutive zeroes.
32.1) Prove that $S \in \mathbf{P}$.
32.2) Prove that $S \in \mathbf{L}$.

Hint — *Again, both points can be proved by describing an algorithm and showing that it has the required property.*

Solution 32

Consider an algorithm that scans the input sequence x and uses a counter to keep track of the number of consecutive zeroes it finds (resetting it everytime it finds a one), halting as soon as it is sure of the answer. For instance:

```
function subsequence ( x, k)
  l ← 0
  for each x_i ∈ x
    if x_i = 0
      l ← l + 1
      if l = k
        accept and halt
    else
      l ← 0
  reject and halt
```

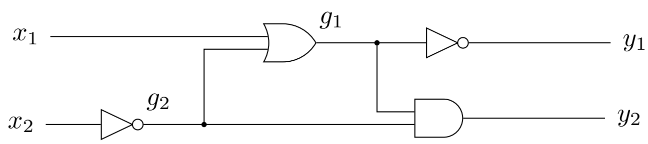
32.1) The algorithm clearly halts after a linear scan of the input, plus counter increments and comparisons, all of which can be carried out in polynomial time on a Turing machine. Therefore, $S \in \mathbf{P}$.
32.2) The algorithm only requires a constant number of counters (the position in the sequence, the counter l), each being of logarithmic size wrt the length of the input sequence x. Therefore, $S \in \mathbf{L}$.

Observations

- Although the algorithm only mentions one additional variable l to be used as a counter, the actual implementation might require more than one. For instance, if the algorithm were to be implemented on a TM, at least another counter to keep track of the current position in the input string might be necessary. What's important, is that a constant number of variables is used, and that each is logarithmic wrt input size.
- Counters cannot have *constant* size, otherwise they would not work for larger inputs.

Exercise 33

Define a CNF formula $f(x_1, x_2, y_1, y_2, g_1, g_2)$ that is satisfiable by precisely the truth values compatible with the following Boolean circuit:



Solution 33

The circuit can be represented by the following CNF

$$
\begin{aligned}
f(x_1, x_2, y_1, y_2, g_1, g_2) &= (g_2 = \neg x_2) \wedge (g_1 = x_1 \vee g_2) \wedge (y_2 = g_1 \wedge g_2) \wedge (y_1 = \neg g_1) && \text{(B.1)} \\
&= (\neg x_2 \vee \neg g_2) \wedge (x_2 \vee g_2) && \text{(B.2)} \\
&\quad \wedge(\neg g_1 \vee x_1 \vee g_2) \wedge (g_1 \vee \neg x_1) \wedge (g_1 \vee \neg g_2) \\
&\quad \wedge(\neg y_2 \vee g_1) \wedge (\neg y_2 \vee g_2) \wedge (y_2 \vee \neg g_1 \vee \neg g_2) \\
&\quad \wedge(\neg y_1 \vee \neg g_1) \wedge (y_1 \vee g_1),
\end{aligned}
$$

where every relationship in B.1 is "exploded" in CNF form in as described in Section 3.5, in particular in Fig. 3.2.