# Notes of Computability and Computational Complexity

Marco Calautti

marco[dot]calautti[at]unimi.it
Department of Computer Science
University of Milan

### Abstract

These lecture notes were prepared over the 3 years of teaching the Computability and Computational Complexiry course at the University of Trento, while serving as an Assistant Professor between 2020 and 2022. The goal of these notes is to offer both lecturers and students a comprehensive yet digestible overview of the foundational concepts in the field. Organized into concise chapters of 6-8 pages each, the material covers a wide spectrum, from basic computability theory, including Turing machines, their expressiveness, and the boundaries of decidability and undecidability, to fundamental concepts in computational complexity, such as NP-completeness, space complexity, and a brief introduction to optimization problems.

**Motivation.** The hope is that the lecture notes will support the students' learning process, providing a clear and accessible guide to both the theoretical foundations and the practical significance of the field. For what is worth, my students seemed to really appreciate the lecture notes, and in particular they found the narrative that ties the topics together very useful to keep having the big picture in mind. Each concept is motivated by the practical need to understand the limitations of computation, and the implications of the theoretical results are explored in a way that highlights, when appropriate, their relevance in the real-world.

**Organization.** The lecture notes are organized in 24 chapters, comprising both theory and exercises, where each chapter should support a 2 hours lecture. Hence, the notes were meant to support a 48 hours course, mainly aimed at first year Master students, or 3rd year undergraduate students. The lectures are organized in a way that students only need a basic background on set theory, and simple logical reasoning, such as knowing the difference between a sufficient and a necessary condition in a statement, what "iff" means, etc. Having some background in Algorithms and Data Structures would be useful but it is not mandatory.

**Distribution.** The lecture notes are licensed under the CC BY-NC-SA 4.0 license, which means you are free to redistribute and modify the lecture notes as you like for non-commercial use, as far as you give proper credit to the original author, and distribute your modifications under the same license. Thus, I would be grateful if those who use the notes for their courses will give appropriate credit by linking to my repository where the original notes are hosted, and if you find any mistakes or have suggestions for improvements, I would be very happy to hear from you!

Here are the books I mostly used as a reference for writing the notes:

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*
- Michael Sipser. *Introduction to the Theory of Computation*
- Sanjeev Arora, Boaz Barak. *Computational Complexity: A Modern Approach*

# Contents

# 1 Problems, Algorithms and a glimpse of undecidability

Until now, most of you have been focusing, during their studies, on algorithms. That is, an algorithm is a finite sequence of basic instructions (in some programming language), that in a *finite amount of time*, solves a certain *problem*, given some data as input. Usually, you were already given a problem, and the focus was to devise an algorithm solving it.

Examples of problems are: given an array of integers $v$ and an integer $x$, find the position in the array in which $x$ occurs, or given a directed graph $G = (V, E)$, with $V$ a set of nodes, that are connected by an edge occurring in $E$, a starting node $s$ and a ending node $t$, check whether there is a path from $s$ to $t$ in the graph $G$.

In this course, we will switch our focus on problems rather than algorithms. In particular, the course is made of two main parts.

The first one is *computability theory*, whose aim is to answer questions like

> "Can this problem be solved by an algorithm?
> (In other words, is there an algorithm that *for every input*, provides the right answer to the problem?)"

Problems of the above kind are called *computable*.

Another kind of question that computability theory aims at answering is:

> "If two problems are not computable, can we still conclude that one is harder than the other, and which is harder?".

The first question might seem strange, as we are so used to solve everything with a program, app, etc., that it is strange to believe that there exist problems that cannot be solved by an algorithm. As we are going to see informally later, such problems actually exist. In the next lectures we will tackle these questions more rigorously, by properly defining the notions of problem, algorithm, and we will identify some of their important properties.

The second part of the course is *complexity theory*: assume we know that a problem is actually computable. Clearly, the fact that an algorithm exists that *for every input*, provides the right answer to the problem does not say much on how *hard* the problem is, i.e., what are the resources in terms of time and memory required to solve the problem?

One of the main questions complexity theory asks is

> "What makes a problem harder than others in terms of resource usage?".

**Search and decision problems.** We distinguish between two main kinds of problems:

- Search problems: these are the problems that ask to search/construct some solution/output, for some given input. For example, search/construct the position (i.e., an integer) in which the number $x$ occurs in $v$.

- Decision problems: these are the problems whose answer is only of the form “yes” or “no”. For example, given a graph $G$ and nodes $s$ and $t$, the problem asking “Is there a path from $s$ to $t$ in $G$?” is a decision problem.

The input of a problem is also usually called *instance* of the problem.

Often, search problems have a corresponding decision problem, and vice versa. For example, given an array $v$ and number $x$, we might only ask “Does $x$ occur in some position in $v$?”. Clearly, a search problem is at least as hard as its decision version, because we could solve the decision version by finding a solution to the search one. For example, if we find in which position $x$ occurs in $v$, then the answer is “yes”, otherwise, if no position is found, the answer is “no”.

So, usually, to simplify the discussion, one focuses on decision problems, as if one is not able to solve a decision problem, there is no hope to solve its search version. In the rest, we will mostly focus on decision problems.

**Example of uncomputable problem.** We will now see an example of an uncomputable problem.

In particular, when focusing on a decision problem, we use the word *decidable* in place of computable, and *undecidable* in place of uncomputable, but they essentially mean the same thing.

Assume we are working with Python (or C++/Java, doesn’t matter), and so we write all our algorithms in Python. Consider the following simple function that counts the number of white spaces in a given string:

```
def countSpaces(myString):
  count = 0
  i = 0
  while i < len(myString):
    if myString[i]==’ ’:
      count = count + 1
      i = i + 1
  return count
```

Can you see the problem with this function? The author of the function has wrote the `i = i + 1` inside the if statement, rather than outside. This means that if the input string is, e.g., ”abc d”, the above function, rather than halting and returing the count of white spaces, it will not halt (i.e., it will run forever), because the first character is not a white space, and thus the index `i` will never be incremented.

It would be nice if we had an algorithm (again written, for example in Python) that is able to catch these kinds of bugs, even for very simple functions like the one above, that takes as input only one string.

So, let us call Python functions taking only one string *one-string functions*. Then, we would like to have some algorithm that is able to solve the following problem:

| | |
|---|---|
| PROBLEM : | HALTING |
| INPUT : | A string P representing the source code of a one-string function, e.g., P="`def someFunc(someString)...`", and another string $I$. |
| QUESTION : | Does the function in P halt when executed with $I$ as input? |

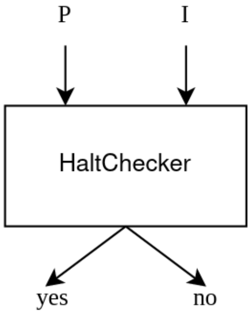
Clearly, you could start coming up with your own algorithm that tries to solve the above problem. Indeed, for very basic programs like countSpaces above, it is not difficult to write an algorithm that can understand whether it halts with a given input (e.g., you could check whether the variable i is always incremented, when there is a while of the form `while i < ...`).

However, what we are going to prove is that, not matter how hard you try, and how clever your algorithm is, there will always be some input $P$ and $I$ for which your algorithm will not be able to give the right answer to the HALTING problem.

So, essentially, we are going to prove the following:

HALTING is undecidable.
(In other words, there is no algorithm that *for every input* $P$ and $I$, gives the correct answer to the HALTING problem)

*Informal proof.* The proof we are going to provide is by contradiction. That is, assume, towards a contradiction, that indeed we have some algorithm, let us call it HaltChecker, that given *any* source code P of a one-string function, and an input string $I$, replies "yes" if the function halts with input $I$, and "no" otherwise.

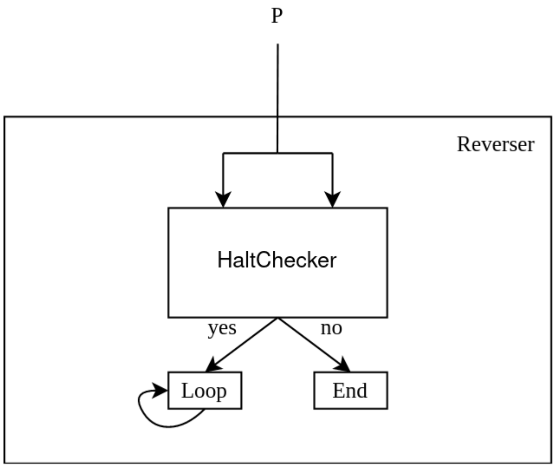


One can visualize HaltChecker as in the picture above, or if you want to think of its actual code, it will look like this:

```
def HaltChecker(P,I):
  ...
```

So, for example, if we give as input to HaltChecker P="`def countSpaces ...`" and I="abc", HaltChecker will return "no", since we have shown that countSpaces does not halt with input ”abc”. However, giving as input P=”`def countSpaces...`” and I=”     ”, HaltChecker will return ”yes”. Actually, according to our assumption, *no matter how P and I look like*, HaltChecker will always provide the right answer to the HALTING problem.

Since we assume HaltChecker exists, we can use it to costruct other algorithms that rely on it. In particular, let us try constructing the following algorithm:



What the above algorithm does, which we call Reverser, is to take as input the source code P of a one-string function and it asks to HaltChecker what that function will do if we give it as input its own source code. If HaltChecker replies that the function halts, then Reverser will go in an endless loop. Otherwise, if HaltChecker concludes that the function would not halt, then Reverser simply does nothing and stops.

One could implement Reverser as the following python code:

```python
def Reverser(P):
  halts = HaltChecker(P,P)
  if halts=="yes":
    while(True):
      pass
```

For example, if we give as input to Reverser the source code P=”`def countSpaces...`”, it will check if `countSpaces` halts with input the string of its source code. Since its source code starts with ”`def`”, `countSpaces` does not halt, and thus HaltChecker(P,P) returns ”no”. Hence, Reverser will simply finish execution. On the other hand, if we gave Reverser the source code P of a function that actually halts with input its source code, Reverser would loop forever.

That is why we call it Reverser: it does the opposite of what the function in P would do, if given its source code as input.

Everybody can agree on the fact that Reverser is a very strange, and probabily useless function. However, it nonetheless exists, assuming HaltChecker exists. So, let us play a bit with it, and let us see what it does with other inputs.

**Question:** What happens if we execute Reverser with input P="`def Reverser...`"? That is, we give it as input its own source code!

So, let us track down this execution. First, Reverser will take its own source code P as input, and will give (P,P) as input to HaltChecker. Let us now distinguish two cases.

1. Assume that HaltChecker answers "yes". Since HaltChecker always provides the right answer, by assumption, this means that Reverser would halt when given its own source code as input. Continuing with the execution, Reverser now checks this answer, and it then decides to loop forever. So Reverser actually does not halt with input its source code, while HaltChecker said otherwise. This contradicts our assumption that HaltChecker is always correct!

2. So, it must mean that HaltChecker answers "no". However, if this is the case, then Reverser would not halt when given as input its source code. Continuing with the execution, Reverser now checks this answer, and it then decides to finish. So, Reverser actually halts with input its source code, while HaltChecker said otherwise. This again contradicts the fact that HaltChecker is always correct!

So, we concluded that there are inputs for which HaltChecker is not able to provide the right answer (in this case, such an input is (P,P), where P is the source code of the function Reverser). This contradicts our initial assumption that HaltChecker is always correct, and thus, we conclude that an algorithm like HaltChecker that is able to solve the HALTING problem *for very input* cannot exist. Hence, HALTING is undecidable.

**Remark:** Let me emphasize again that what we have just proved is that there is no *general algorithm* that is able to solve the HALTING problem for all inputs. We can very well come up with very smart algorithms that give the right answer for many inputs (P,I), but they will never be perfect. There will always be some source code P of a function and its input I for which your smart algorithm will not be able to answer correctly.[1]

[1] There is a very nice video on YouTube that explains the undecidability of the HALTING problem at an even higher level. I highly suggest taking a look: `https://www.youtube.com/watch?v=92WHN-pAFCs`

# 2 Alphabets, strings, languages and Turing Machines

In order to study problems and their properties in a rigorous way, we first need to formally define the notion of problem. Intuitively, whenever we think of a problem, we think of some kind of input we are given, and some task we need to solve, using that input. The task is then to produce some output (search problem) or just provide a “yes”/”no” answer. In the most abstract sense, we can think of inputs to our problems as strings over a certain alphabet, encoding the input to the problem. For example, consider the problem, given an array $v$ of integer and an integer $x$, asking to find the position where $x$ occurs in $v$. The input to this problem can be represented with a string of the form

$$([1, 4, 7, 1, 2], 2),$$

where $[1, 4, 7, 1, 2]$ is the input array $v$ and 2 is the integer $x$. Similarly, the output of the problem (which is an integer), can be represented with a string over the alphabet $\{0, 1, \ldots, 9\}$. So the overall alphabet for the input and output strings of our problem is the set of symbols

$$\Sigma = \{'[', ']', '(', ')', ',', 0, 1, \ldots, 9\}.$$

Let's recall some notation that will be useful in the rest. An *alphabet* is a **finite** set of symbols, and it is usually denoted with $\Sigma$. We use $\Sigma^n$ to denote all the strings of length $n \geq 0$ using symbols from the alphabet $\Sigma$. For example, if $\Sigma = \{a, b, c\}$, then

$$\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}.$$

Moreover, we use $\Sigma^*$ to denote the set of all possible strings (of any *finite* length, including the empty string), using symbols from the alphabet $\Sigma$. That is:

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n.$$

Since the empty string is, well... empty, when we want to use it explicitly somewhere in our formulas, we use $\epsilon$ to denote it. So, for example, if $\Sigma = \{0, 1\}$; then:

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}.$$

We can now formally defined what a problem is for us.

**Definition 1.** *A* problem *over an alphabet $\Sigma$ is a function of the form*

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

So, a problem is described as a function that maps each input, encoded via some string in $\Sigma^*$, to the right corresponding output (again, encoded as a string in $\Sigma^*$).

Note that we allow functions to take any string from $\Sigma^*$ as input, but clearly, there are strings in $\Sigma^*$ that do not represent valid inputs to the problem the function represents, e.g. $][,(2,42]]$ does not encode an array of integers. To solve this issue, we could be even more precise, and require that the function defining a certain problem should not have $\Sigma^*$ as the domain, but a subset $D \subseteq \Sigma^*$ that contain only strings encoding valid inputs to the problem.

However, this would complicate our mathematical treatment, as we will no longer be able to treat problems uniformily. Indeed, having problems take as input arbitrary strings in $\Sigma^*$ is not a big deal. Say, for example, our problem takes as input strings encoding an array of integers, e.g, [3,5,12,20]. One can always slightly extend the problem to a new problem that takes as input any string from $\Sigma^*$ such that: if the string is a bad encoding of an array, then the output is some default, "don't care" value (e.g., the empty string $\epsilon$), otherwise, if the input properly encodes an array, the output is the one expected by the original problem.

The new problem is fundamentally the same as the original one, except for this minor check on the validity of the input, which does not affect at all the difficulty of solving it. Indeed, any algorithm that is able to solve the original problem can easily solve the new one by first verifying if the input is valid, and in case it is, perform the same computation as before.

For example, consider the problem of finding the minimum element in a given array $v$ of integers. Our alphabet is

$$\Sigma = \{'[',']','(',')',',',0,1,\ldots,9\}.$$

the function representing this problem is the function $f : \Sigma^* \rightarrow \Sigma^*$ such that, for each $w \in \Sigma^*$

$$f(w) = \begin{cases} \epsilon & \text{if } w \text{ is not a valid encoding of an array} \\ \min(v) & \text{if } w \text{ encodes a valid array } v. \end{cases}$$

Since, as discussed, it is very easy, for an algorithm, to verify whether the input string is valid (if using a reasonable encoding), when defining problems, we will often omit the part discussing how bad inputs are dealt with (as this is always the same: if input is bad, output a default value). So, to simplify the discussion, we will define all our problems be specifing what is the expected output, assuming the input is properly encoded.

The above definition of problem is about *search problems*, i.e., given some input, we specify what we should construct as output.

If we want to focus on decision problems, i.e., with only "yes"/"no" answers, we use the following definition:

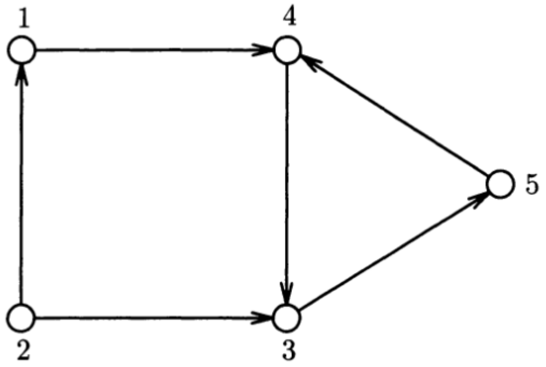
**Definition 2.** *A* decision problem *over an alphabet* $\Sigma$ *is a function of the form:*

$$f : \Sigma^* \rightarrow \{0,1\}.$$

So, we limit the output of the function to only be 0 (“no”) or 1 (“yes”).

Also here, we have the same ”issue” with badly encoded inputs. The question is: what would be this default value to output, when the input is badly encoded? For decision problems, the default is 0 (i.e., ”no”). Indeed, if a string does not encode a valid input to the decision problem, it is impossible for the string to satisfy any property requested by the problem, and so the answer is necessarily ”no”.

**Example decision problem.** Let's see an example of a decision problem over graphs. A graph can be depicted as a set of nodes (drawn as circles), where the nodes are connected by some edges.



If the edges are oriented, the graph is directed, otherwise it is undirected. So, we can represent a graph as a pair $G = (V, E)$, where $V$ is the set of nodes, represented with integers, and $E \subseteq V^2$ is a set of pairs $(u, v)$ denoting the fact that there is a (directed) edge from node $u$ to $v$. For example, the graph above is represented by the string $(\{1,2,3,4,5\},\{(1,4),(2,1),(2,3),(4,3),(5,4),(3,5)\})$. So, the alphabet is

$$\Sigma = \{'\{','\}','(',')',0,1,\ldots,9\}.$$

We want to consider the following *decision* problem:

| | |
|---|---|
| PROBLEM : | REACHABILITY |
| INPUT : | A directed graph $G = (V, E)$ and two nodes $s, t \in V$. |
| QUESTION : | Is there a path from $s$ to $t$? |

For example, with the graph above and $s = 1$ $t = 5$, the answer is “yes”. The problem REACHABILITY, is essentially the following function, for every string $w \in \Sigma^*$ encoding a graph $G = (V, E)$ and two nodes $s, t$:

$$\text{REACHABILITY}(w) = \begin{cases} 1 & \text{if there is a path from } s \text{ to } t \text{ in } G, \text{ and} \\ 0 & \text{otherwise} \end{cases}$$

Note that here we are saying what the problem requires as output, only for properly encoded inputs, but we should always have in mind that the problem is well-defined also when the input string $w$ is a bad encoding, for which the answer is implicitly 0.

**Decision problems as languages.** Note that since decision problems only require to output either 0 or 1, we can equivalently define a decision problem as simply the set of strings in $\Sigma^*$ for which the answer to the problem is "yes". For example, REACHABILITY is the set of all strings in $\Sigma^*$ the form $G, s, t$, where $G$ is a directed graph and $s, t$ are two nodes, such that there is a path from $s$ to $t$ in $G$.

In other words, decision problems can be defined as *languages*, i.e., subsets of the whole set of all possible strings over the alphabet. As another example, consider the decision problem of checking whether, given an array of integers $v$, the sum of all integers in $v$ is 0. Once we define the right alphabet $\Sigma$, we define the problem, which we can call ZEROSUM, as the language

$$\text{ZEROSUM} = \{w \in \Sigma^* \mid w \text{ encodes an array } v \text{ of integers, such that } \sum_{i=1}^{|v|} v[i] = 0\}.$$

So, we observe that

*given a string, solving a decision problem for that string is equivalent to check whether the string belongs or not to the language corresponding to the problem.*

As discussed in the previous lecture, for simplicity, we will focus only on decision problems, and hence on languages.

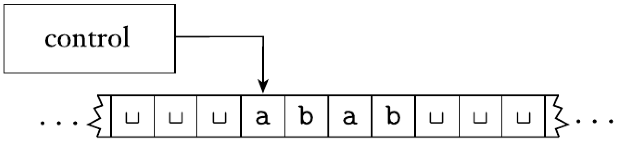
## 2.1 Introduction to Turing Machines

Now that we know what we mean by "problem", we need to formalize the notion of "algorithm". In particular, to understand which problems can be solved and which cannot, we need to understand the kind of computations we can perform, i.e., the kind of machines on which we execute our algorithms. We have many options, we could use standard computers, but they are quite complicated. Also, it is not difficult to understand that many parts of modern computers are not really needed to solve problems. For example, we could get rid of graphics cards, as it is just another kind of CPU in the machine, that the main CPU can simulate and so on. So, the question is,

"What is the simplest model of computation that is powerful enough to solve all problems that can be solved by any other model of computation?"

In the Theory of Computation literature, there has been a general consensus that this is the so called *Turing Machine.*

This is a very simple, theoretical machine devised by Alan Turing in the 1930s. The following picture shows how a TM looks like.



The data the machine works with is stored in an infinitely long tape (on both sides).

The tape is divided into cells, where each cell can contain some symbol. When the machine starts execution, the tape contains the input string, and on the left and right side, it is filled with a special symbol ⊔ called the blank symbol.

Then, there is a *control component*, that can read and write on the tape one cell at the time with a head.

The head is placed at the beginning of the input string, when it starts execution.

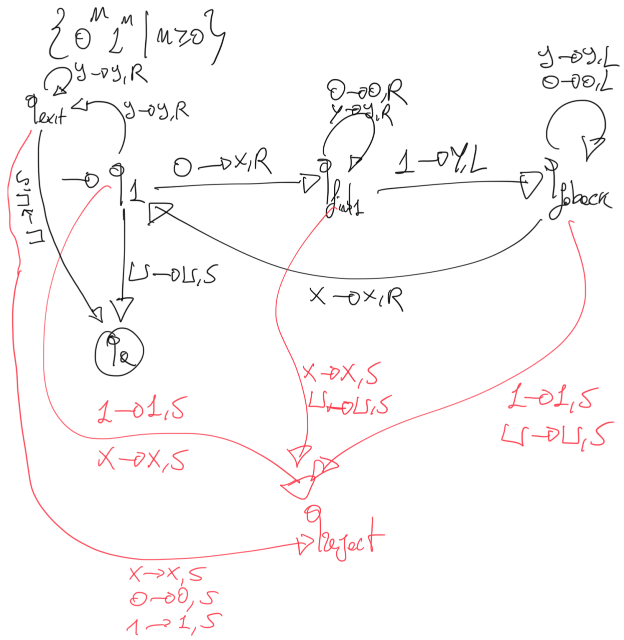
The control of the machine describes what the machine should do, according to the value the head is currently reading, and according to the internal state of the machine.

Let us consider as an example the following decision problem:

$$\{0^n 1^n \mid n \geq 0\}.$$

That is, given a string of 0s and 1, is the string of the form $0^n 1^n$?

This is a possible Turing machine that solves the problem.



**Syntax.** We now formally define a Turing Machine.

**Definition 3.** *A* Turing machine $M$ *is a tuple* $(Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$*,where*

1. *$Q$ is the set of states;*
2. *$\Sigma$ is the input alphabet, not containing the blank symbol $\sqcup$;*
3. *$\Gamma$ is the tape alphabet, such that $\Sigma \subseteq \Gamma$, and $\sqcup \in \Gamma$;*
4. *$q_1$ is the initial (or start) state;*
5. *$q_{accept} \in Q$ is the accepting state;*
6. *$q_{reject} \in Q$ is the rejecting state;*
7. *$\delta : Q \setminus \{q_{accept}, q_{reject}\} \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ is the transition function.*

Note that we distinguish between the input alphabet and the tape alphabet. This is because the input alphabet is the alphabet of the problem that the machine is trying to solve, but the machine might need some additional symbol to do its work.

Also note that the transition function takes as inputs a state that is not $q_{\texttt{accept}}, q_{\texttt{reject}}$, as when the machine reaches these states, it terminates its execution, and a tape symbol, and tells the machine what to do at the given state and tape symbol, i.e., in which state it should move, what is the new symbol the machine places in the tape at the current position, and where the head of the machine should move.

The Turing machine of the figure before is thus the tuple $(Q, \Sigma, \Gamma, \delta, q_1, q_{\texttt{accept}}, q_{\texttt{reject}})$, where

1. $Q = \{q_1, q_{\texttt{find1}}, q_{\texttt{goback}}, q_{\texttt{exit}}, q_{\texttt{accept}}, q_{\texttt{reject}}\}$;
2. $\Sigma = \{0, 1\}$;
3. $\Gamma = \{0, 1, X, Y, \sqcup\}$;

Moreover, $\delta$ is the following function, that we can represent with a table:

| $\delta$ | $0$ | $1$ | $X$ | $Y$ | $\sqcup$ |
|---|---|---|---|---|---|
| $q_1$ | $(X, q_{\texttt{find1}}, R)$ | $(1, q_{\texttt{reject}}, S)$ | $(X, q_{\texttt{reject}}, S)$ | $(Y, q_{\texttt{exit}}, R)$ | $(\sqcup, q_{\texttt{accept}}, S)$ |
| $q_{\texttt{find1}}$ | $(0, q_{\texttt{find1}}, R)$ | $(Y, q_{\texttt{goback}}, L)$ | $(X, q_{\texttt{reject}}, S)$ | $(Y, q_{\texttt{find1}}, R)$ | $(\sqcup, q_{\texttt{reject}}, S)$ |
| $q_{\texttt{goback}}$ | $(0, q_{\texttt{goback}}, L)$ | $(1, q_{\texttt{reject}}, S)$ | $(X, q_1, R)$ | $(Y, q_{\texttt{goback}}, L)$ | $(\sqcup, q_{\texttt{reject}}, S)$ |
| $q_{\texttt{exit}}$ | $(0, q_{\texttt{reject}}, S)$ | $(1, q_{\texttt{reject}}, S)$ | $(X, q_{\texttt{reject}}, S)$ | $(Y, q_{\texttt{exit}}, L)$ | $(\sqcup, q_{\texttt{accept}}, S)$ |

So, the TM $M$ solves the problem above (i.e., it is an algorithm solving it), because whenever it takes as input a string $w$ of 0s and 1s, it *always halts*, and it halts in the accepting state if the string is of the form $0^n1^n$, in which case we say that $M$ accepts $w$, otherwise, $M$ halts in the rejecting state, in which case we say that $M$ rejects $w$. Let us formalize these notions.

**Semantics.** If $M$ is a TM that is executed with input string $w$, note that at each step, the machine must have only visited a finite number of cells. Hence, at each step, the only relevant information for the machine is the *finite* part of the tape that contains all the cells where the input was placed, plus all the other cells the machine has visited. We call this part of the tape, the *relevant part* of the tape.

Hence, if one imagines that the machine "loses power" and "shuts down", the only information that we would need to remember before the shutdown, to perfectly restore the computation of $M$ with input $w$ at that point is

1. The relevant part of the tape;
2. The current state;
3. The position of the head.

We call these three informations together an *instantaneous description* (ID) of the machine $M$ with input $w$. If at some step, the relevant part of the tape is the string $\alpha_1, \alpha_2, \ldots, \alpha_n$, the current state is $q$ and the head is positioned on the cell with symbol $\alpha_i$, we can compactly represent the ID with the expression

$$\alpha_1, \alpha_2, \ldots, \alpha_{i-1} q \alpha_i, \ldots, \alpha_n.$$

That is, by placing the state $q$ on the left of the symbol pointed by the head.

If we consider the TM of the previous example, and the string $w = 0011$, an ID of $M$ with input $w$ is

$$q_1 0011.$$

That is, the ID of the machine when it starts execution. If we have a look at the transition function, the next ID should be:

$$X q_{\texttt{find1}} 011.$$

Then, we say that an ID

$$\alpha_1, \ldots, \alpha_{i-1} q \alpha_i, \ldots, \alpha_n$$

*yelds* the ID

$$\alpha_1, \ldots, \alpha_{i-1}, \beta, q', \alpha_{i+1}, \ldots, \alpha_n,$$

if $\delta(q, \alpha_i) = (q', \beta, R)$. The definition of "yelds" for left-moves and stay-moves is similar.

For example, $q_1 0011$ yelds $X q_{\texttt{find1}} 011$. The notion of "yelds" captures the behaviour of the machine, when it performs a single step.

We can now finally define what we mean that a machine accepts/rejects its input.

The *initial ID* of $M$ with input string $w = w_1 w_2 \cdots w_n$ is the ID

$$q_1 w_1 w_2 \cdots w_n.$$

For example, $q_1 0011$ is the initial configuration. An *accepting ID* is any ID whose state is $q_{\texttt{accept}}$, and a *rejecting ID* is any ID whose state is $q_{\texttt{reject}}$.

**Definition 4.** *Consider a TM $M$ with input alphabet $\Sigma$ and an input string $w = w_1 \cdots w_n \in \Sigma^*$. We say that $M$* accepts *(resp., rejects) $w$ if there is a sequence of IDs $C_1, \ldots, C_m$ such that:*

1. *$C_1$ is the initial ID of $M$ with input $w$, i.e., $C_1 = q_1 w_1 \cdots w_n$;*
2. *$C_i$ yelds $C_{i+1}$, for $1 \leq i < m$;*
3. *$C_m$ is an accepting (resp., rejecting) configuration.*

For example, considering the previous TM and the input $w = 01$, we have that

$$q_1 01 \text{ yelds } X q_{\texttt{find1}} 1 \text{ yelds } q_{\texttt{goback}} XY \text{ yelds } X q_1 Y \text{ yelds } XY q_{\texttt{exit}} \sqcup \text{ yelds } XY q_{\texttt{accept}} \sqcup .$$

Hence $M$ accepts $w$.

The *language of a TM*, or the *language accepted by a TM $M$*, denoted $L(M)$, is the set of all strings accepted by $M$. That is

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

**Remark.** Note that the fact that $M$ accepts a language $\mathcal{L}$ does not necessarily imply that $M$ is also able to reject all the other strings. In fact, $M$ could never halt with such strings as input. So, by just saying that $M$ accepts a language $\mathcal{L}$ we cannot conclude yet that $M$ is an algorithm solving the decision problem represented by $\mathcal{L}$. For this, we need that the machine should not only be able to determine whether a given string is in $\mathcal{L}$, i.e., when the answer to the problem is "yes" $M$ accepts, but $M$ should also be able to reject an input string not in $\mathcal{L}$, i.e., it should *halt* in the rejecting state. In this case, we say that $M$ *decides* the language $\mathcal{L}$. We will go into more details on this later on in the course.

For example, the TM $M$ of the previous example accepts the language

$$L(M) = \{0^n 1^n \mid n \geq 0\}.$$

Moreover, we also know that $M$ rejects every other string not in the language, so not only $M$ accepts the language, but it also *decides* the language (i.e., solves the decision problem described by the language).

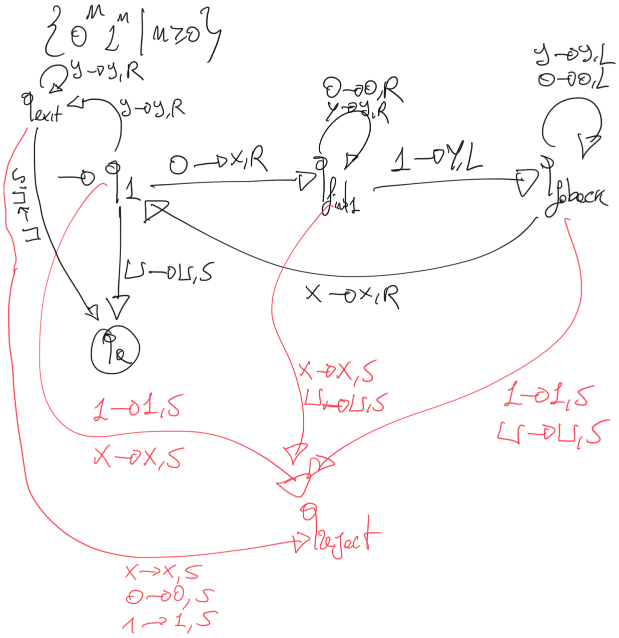
## 3 Input size, execution time and power of TMs

For a string $w \in \Sigma^*$, we use $|w|$ to denote the *length* (number of symbols) in the string $w$. For example, the string $0110 \in \{0,1\}^*$ is such that $|0110| = 4$.

We now want to measure the time needed by a TM $M$ to perform a computation when given a string of a certain length $n$. How many steps the machine $M$ requires to accept or reject a string of length $n$? It depends on the specific string, so we define the time as the maximum number of steps the machine requires, among all input strings of length $n$.

**Definition 5.** *The* time required by a TM $M$ with inputs of length $n$, *denoted $T_M(n)$, is the maximum number of steps that $M$ performs with an input string of length $n$ in order to accept or reject. If there is an input of length $n$ for which $M$ loops, we say that $T_M(n) = \infty$.*

For example, consider the TM we have seen in the previous lecture.



Let us consider inputs of length $n = 2$. If we consider the string $w = 01$, then the number of steps is 5, because the sequence of IDs that reaches an accepting configuration is:

$$q_1 01 \rightarrow X q_{\texttt{find1}} 1 \rightarrow q_{\texttt{goback}} XY \rightarrow x q_1 Y \rightarrow XY q_{\texttt{exit}} \sqcup \rightarrow XY q_{\texttt{accept}} \sqcup.$$

But with input string $w = 11$, we only have 1 step:

$$q_1 11 \rightarrow 1 q_{\texttt{reject}} 1.$$

It is not difficult to verify that the number of steps in the worst case is always achieved when the input string belongs to the language $\{0^n1^n \mid n \geq 0\}$. Now, counting the number of steps of a machine, precisely, can easily become unmanageable. So, rather than focusing on the *exact* number of steps, we are only interested in the *asymptotic growth* of the number of steps.

Most of you should be familiar with the notion of big-Oh. We recall it here.

**Definition 6.** *Consider two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$. We say that $f(n)$ is a big-Oh of $g(n)$, written $f(n) \in O(g(n))$, if there are constants $c \geq 0, N \geq 0$ such that $f(n) \leq c \cdot g(n)$, for each $n \geq N_0$.*

Intuitively, the rate of growth of $f(n)$ is at most the rate of growth of $g(n)$. For example, $f(n) = n^2 + n$ is a big-Oh of $g(n) = n^3$.

**Example 1.** *Let's analyse the time required by the example TM $M$ with a string of length $n$. The worst case is when the string belongs to the language. The machine, first scans $n/2$ cells, placing $X$ and $Y$ on the first 0 and 1 respectively and then goes back $n/2$ cells. So, one pass requires "roughly" $n$ steps (excluding some constant factor). The machine performs $n/2$ passes. So, roughly $n^2/2$ steps. Finally, the machine scans the remaining $Y$'s, that are $n/2$. So, overall, the machine requires roughly $n^2/2 + n/2$ steps. Thus, $T_M(n) \in O(n^2)$.*

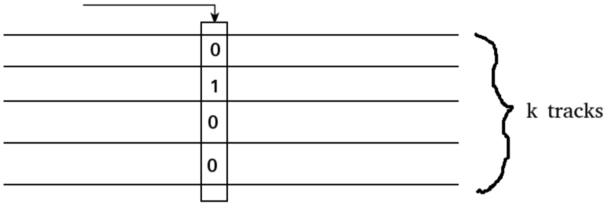
## 3.1 The Power of TMs

We said at the beginning of the course that we use TMs because they form a simple computational model, but at the same time they would allow us to solve all solvable problems. But why is that? Maybe we can make our TMs more powerful by adding some "optionals", which would allow these machines to solve more problems.

Let's see some possible extensions.

**Multi-track TMs.** One might say that storing only a symbol in a single cell might be limiting for a TM. Modern computers are able to store whole bytes in a single memory cell, so 8 symbols (bits). Let us try extending our computational model in this regard.

A multi-track TM, with $k$ tracks is a standard TM, where each cell is divided in $k$ tracks (i.e., contains $k$ symbols).



The machine still has one head that moves around, but the head can read all the $k$ symbols in a cell all at once, and writes all $k$ symbols in cell at the same time.

A transition in the transition function of a multi-track TM looks something like

$$q_i \overset{0100 \to 0000, L}{\longrightarrow} q_j.$$

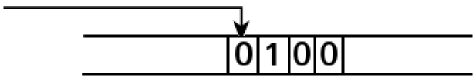
Now, the question is: are multi-track TMs “more powerful” than standard TMs? Moreover, what do we mean by “more powerful”? Intuitively, we compare two computational models by comparing the languages their machines can accept.

If they are able to accept the same languages, they are equally powerful. Clearly, every language a single-tape TM accepts can also be accepted by a TM with $k$ tracks (just use $k = 1$ tracks).

Surprisingly, we are going to prove also that every language a multi-track TM accepts can also be accepted by a single-tape TM.

**Theorem 1.** *Let $M$ be multi-track TM. Then, there exists a single-tape TM $M'$ such that $L(M) = L(M')$. Moreover, $T_{M'}(n) \in O(T_M(n))$.*

*Proof.* We show how a single-tape TM can simulate the behaviour of a multi-track TM. Assume $M$ is a TM with $k$ tracks. The idea is to store the $k$ symbols of each cell in the tape of $M$ into $k$ contiguous cells in the tape of $M'$. If we consider the example above, we will store the cell containing the symbols 0100 into 4 cells:

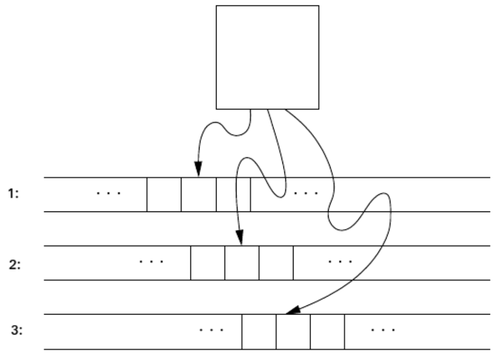


Now, whenever there is a transition in $M$ like the one in the example above, $M'$ needs $k$ steps to do what $M$ does in one step. That is, it first reads the 0, replaces it with 0 and moves to the right. Then reads 1, replaces it with 0 and moves to the right. And so on. Then, $M'$ needs to move its head on the left, and thus needs to move the head $2k$ positions on the left. So, clearly, if for an input string $w$, $M$ reaches the accepting state, so will $M'$, and vice versa. It will only take some more steps to do it. Thus, $L(M) = L(M')$.

How many more steps? In the worst case, one step by $M$ requires $3k$ steps by $M'$: $k$ for updating the symbols, and $2k$ to move on the left. The right and stay moves require less steps.

So, if $T_M(n)$ is the time required by $M$, $M'$ requires at most $3k \cdot T_M(n)$ steps. So, they only differ by a constant factor, hence $T_{M'}(n) \in O(T_M(n))$. □

**Multi-tape TMs.** One might say that using multiple tracks was a very simple addition. What if we give the machine access to more than one tape? Can it perform more computations than normal single-tape machines. Let us informally introduce multi-tape TMs. The structure is similar to single-tape machines, the only difference is that the control can read and/or write multiple tapes, and each tape has its own dedicated head.



The input string is placed in the first tape, all the others contain only blanks. An example transition in a multitape TM is of the form:

$$q_i \xrightarrow{\substack{1:a\to a,R\\ 2:b\to c,L\\ 3:c\to\sqcup,R}} q_j.$$

Consider the following decision problem

$$\mathcal{L} = \{w\#w \mid w \in \{a,b,c\}^*\}.$$

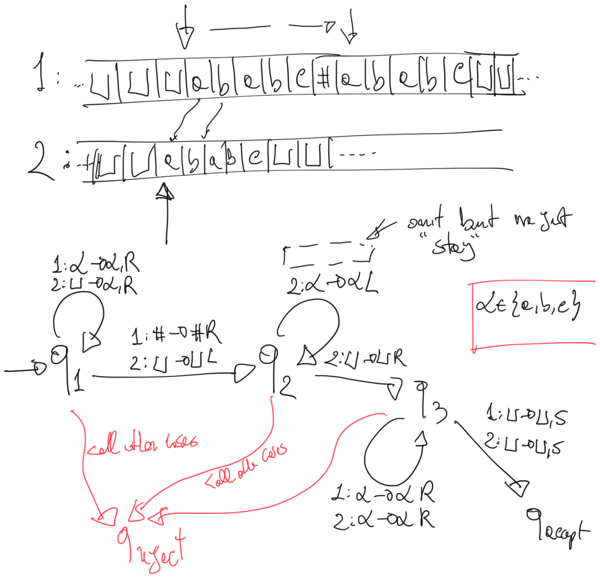
So, the decision problem that given a string over the alphabet $\{a,b,c,\#\}$, asks

"is the string made of two copies of the same string, separated by #?"

We can recognize this language with a TM with two tapes.

1. Copy the content of the first tape up to the symbol # to the second tape.
2. Compare the second string in the first tape with the string in the second tape.
3. If they match, accept.

For example, consider the string $w = ababc\#ababc$.



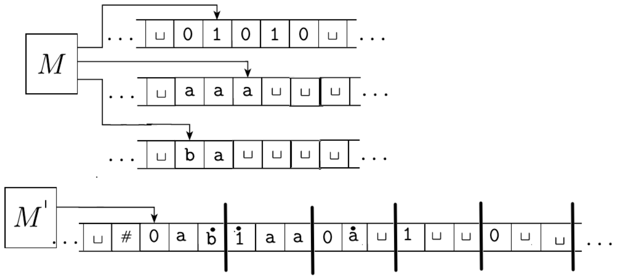
So, the language accepted by $M$ is $L(M) = \mathcal{L}$, as the set of strings it accepts coincides with $\mathcal{L}$. We also made the machine reject every other string, so $M$ also *decides* the language.

Clearly, every language accepted by a single-tape TM is also accepted by a multi-tape TM, as the former are a special case of the latter.

Surprisingly, we are going to prove also that every language accepted by a multi-tape TM can also be accepted by a single-tape TM.

**Theorem 2.** *Let $M$ be multi-tape TM. Then, there exists a single-tape TM $M'$ such that $L(M) = L(M')$. Moreover, $T_{M'}(n) \in O(T_M(n)^2)$.*

*Proof.* For the proof, we show how to convert a multi-tape TM with $k$ tapes to a single-tape TM $M'$. The idea is to store the content of all $k$ tapes in the single tape of $M'$. The idea is shown in the picture below.



So, all the symbols in the first cell of each tape are stored one after the other in the tape of $M'$, all the symbols in the second cell of each tape are stored one after the other, and so on. Moreover, we let $M'$ have some additional symbol. In particular, a special symbol # to denote the start of the tape, and for each symbol of the tape alphabet of $M$, $M'$ has also a "dotted version". This is used to keep track of where the head on a tape of $M$ is positioned.

Assume $M$ is a TM with $k$ tapes with input a string $w = w_1 w_2 \cdots w_n$. The single-tape TM $M'$ contains in its tape the string

$$\# \underbrace{\dot{w}_1 \dot{\sqcup}\dot{\sqcup}\cdots\dot{\sqcup}}_{k} \underbrace{w_2 \sqcup \sqcup \cdots \sqcup}_{k} \cdots \underbrace{w_n \sqcup \sqcup \cdots \sqcup}_{k}.$$

To simulate one step, $M'$ will first move $k$ cells at the time, searching for a dotted symbol, denoting that the head on the first tape is on that symbol. If $M'$ changed the first cell of $M$ and needs to go left, it writes $k$ blanks on the left (replacing #), and then writes # to the left of these blanks.

Once it finds it, it applies the step as dictated by $M$, by replacing the symbol, and moving right/left of $k$ cells, and placing a dot on the symbol. Then, it goes back to the beginning of the tape (it looks for #). Then, it does the same for the second tape, positioning its head on the second symbol, and then moving $k$ cells at the time. And so on, for all tapes.

Clearly, if $M$ accepts $w$, also $M'$ accepts, and vice versa. Thus, $L(M) = L(M')$.

Consider, for example, the transition:

$$q_i \xrightarrow{\substack{1:1\to 0,R\\ 2:a\to b,L\\ 3:b\to\sqcup,S}} q_j$$

Then, starting from the first symbol, $M'$ will look for the symbol $\dot{1}$, by moving $k$ cells at the time.

It finds it, and applies the change. Moves right by $k$ cells, places a dot, and goes back to #. Then, moves to the second symbol, and looks for $\dot{a}$, and so on. Once it is done, it moves to the state $q_j$.

Let's discuss now the time required by $M'$. Consider an input string $w$ and assume up to a certain point, $M$ has performed $m$ steps. Then, the dotted symbols in $M'$ tapes are not farther than $k \cdot m$ cells from the beginning of the tape. So, to perform one step of $M$, $M'$ needs to scan at most $k \cdot m$ cells to the right to find a dotted symbol, then $k$ cells to update the head of the "virtual tape" (in the worst case on the right). So overall, $k \cdot (m+1)$. Then, it needs to go back to the beginning scanning $k \cdot (m+1)$ cells once more. Overall $2k(m+1)$ steps for one tape of $M$. Thus, considering all $k$ tapes, overall $2k^2(m+1)$ steps are needed. Thus, if $M$ requires $m$ steps, $M'$ requires $2k^2(m+1) \cdot m$ steps. Thus, $T_{M'}(n) \in O(T_M(n)^2)$. □

**01-TMs.** So far, we always considered TMs that accept inputs strings over arbitrary alphabets. However, as computer scientists, we are used to the fact that we can actually encode data of any kind by using only two symbols, i.e., the binary alphabet $\{0, 1\}$. This intuition transfers naturally to TMs. That is, for every TM with some tape alphabet Γ, we can easily write a TM that encodes strings over that alphabet in binary and performs the same computations.

For example, if $\Gamma = \{a, b, c, d, \sqcup\}$, we can encode each symbol in Γ (besides ⊔) with 2 bits, e.g.

$$< a >= 00, < b >= 01, < c >= 10, < d >= 11.$$

Thus, a string abb would be encoded as 000101.

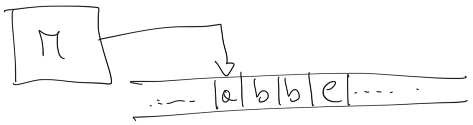
With this spirit, we show how we can simulate a single-tape TM with input tape Γ with a single-tape TM tape alphabet Γ′ containing only the symbols 0 and 1 (and the blank).

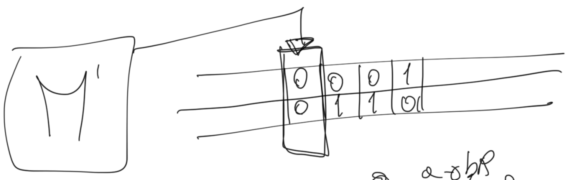
**Theorem 3.** *Consider a TM $M$ with tape alphabet $\Gamma$. Then, there exists a TM $M'$ with tape alphabet $\Gamma' = \{0, 1, \sqcup\}$ that simulates $M$. Moreover, $T_{M'} \in O(T_M(n))$.*

*Proof.* Assume there are $k$ symbols in Γ (excluding blank). We need $\lceil \log_2(k) \rceil$ bits to encode each such symbol. We can then use a multi-track TM with tape $\Gamma' = \{0, 1, \sqcup\}$, with $\lceil \log_2(k) \rceil$ tracks. The number of steps remain the same.

Moreover, since we can simulate a multi-track TM with a single-tape TM without changing the input and tape alphabet, we conclude that we can simulate $M$ with a single-tape TM with alphabet $\Gamma = \{0, 1, \sqcup\}$. The time required follows by the theorem on equivalence between multi-track and single-tape.

$\Gamma': \{a, b, c, d, \sqcup\}$





$q_i \xrightarrow{a \to b, R} q_j$

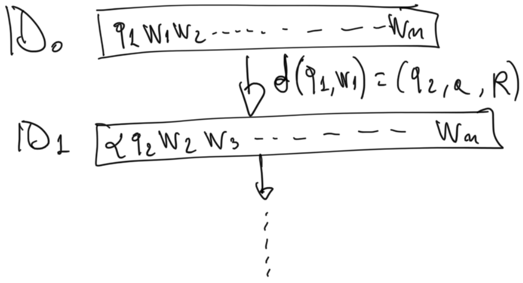
$q_i \xrightarrow{00 \to 01, R} q_j$

□

# 4 Non-deterministic TMs and Exercises on TMs

In this lecture we consider one last extension of TMs. Probably one of the most important, and the one that makes more evident how robust our computational model is. We now consider *non-deterministic* TMs. Until now, our machines performed "linear executions". That is, when the machine is in some state and reads a certain symbol from the tape, there is only a specific step that the machine can perform (e.g., change the symbol, and move to the right).

This results in a computation, where starting from the initial ID, the machine only moves to exactly one ID, and so on. So, once the input string is given, the complete computation of the machine is already determined.



$ID_0$
$q_1 w_1 w_2 \cdots w_n$
$\delta(q_1, w_1) = (q_2, \alpha, R)$
$ID_1$
$\alpha q_2 w_2 w_3 \cdots w_n$

What if we let our machines, at each step, to consider multiple choices? This way, if the machine is uncertain on what it has to do, to accept the language, it can consider different potential steps at the same time. So, as far as at least one of such choices leads to accepting the input string, the machine would have accomplished its goal.

We call these machines non-deterministic TMs, and they only differ from standard TMs in the way their transition function is defined.

**Definition 7.** *A* non-deterministic *TM (NTM) $M$ is a tuple*

$$(Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject}),$$

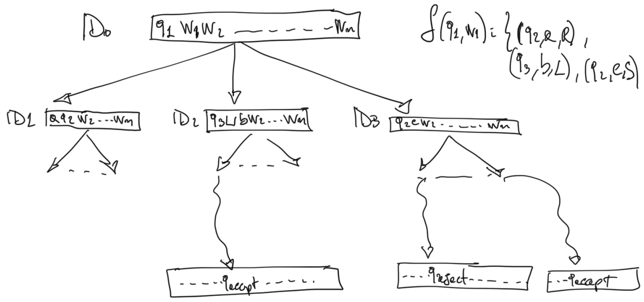
*where every element is as in a standard TM. The only difference is that $\delta$ is not a function from $Q \setminus \{q_{accept}, q_{reject}\} \times \Gamma$ to $Q \times \Gamma \times \{R, L, S\}$ anymore. Rather, a function of the form*

$$\delta : Q \setminus \{q_{accept}, q_{reject}\} \times \Gamma \to 2^{Q \times \Gamma \times \{R,L,S\}} \setminus \{\emptyset\}.$$

*That is, for a given state $q$ and symbol $\alpha$, $\delta(q, \alpha)$ is a non-empty* set *of triples like $(q', \beta, R)$.*

For example, $\delta(q_1, a) = \{(q_2, b, R), (q_3, c, L)\}$ means that when the machine is in state $q_1$ and reads the symbol $a$, it can either move to state $q_2$, change $a$ to $b$ and move right, or move to state $q_3$, change $a$ to $c$ and move to the left.

So, the computation of $M$ is no more a simple sequence of IDs, but from a certain ID, the machine can move to multiple IDs (like if it is considering all of them in parallel). So, the computation is a *tree*!

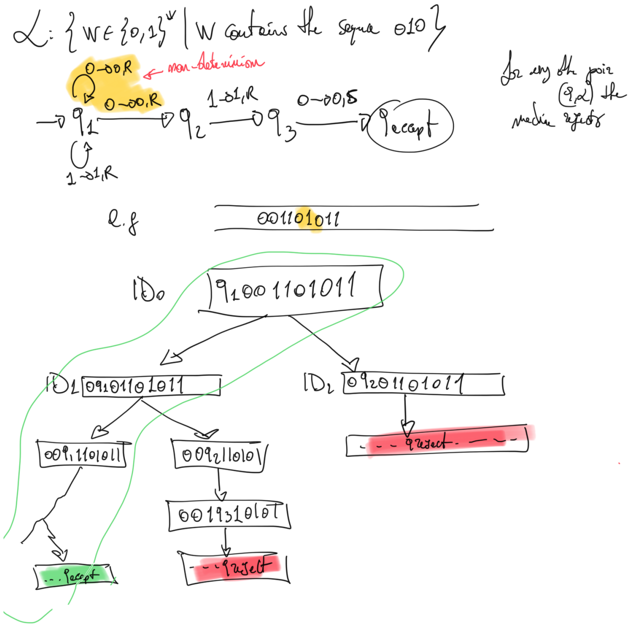


**Definition 8.** *A non-deterministic TM M* accepts *an input string w if there is a* finite *path in its computation tree, rooted on the initial ID, that ends in an accepting ID. M* rejects *w if all paths rooted on the initial ID are finite and end in a rejecting ID.*

Note that when $M$ accepts $w$, there might still be paths that lead to a rejecting ID, or paths that never end. However, since the machine is "somehow" considering all the paths in parallel, as far as it finds an accepting one, it does not matter if others reject or loop.

**Example 2.** *Let's see a very simple example of a non-deterministic TM. Consider the language*

$$\mathcal{L} = \{w \in \{0,1\}^* \mid w \textit{ contains the sequence 010}\}.$$



As we did for standard TM, we also need a notion of time required by a NTM.

**Definition 9.** *The* time required by a NTM $M$ with inputs of length $n$*, denoted* $T_M(n)$*, is the maximum number of steps that can occur in a path of any computation tree of* $M$*, when considering input strings of length* $n$*. If there is an input of length* $n$ *for which* $M$ *has an infinite path in its computation tree, we say that* $T_M(n) = \infty$*.*

So, intuitively, the time required by a NTM $M$ with inputs of length $n$ is the *worst case* number of steps it needs to perform, when considering all possible outcomes (paths) in "parallel".[2]

We can now define when a NTM accepts or decides a certain language, in a similar way as we did for standard TMs.

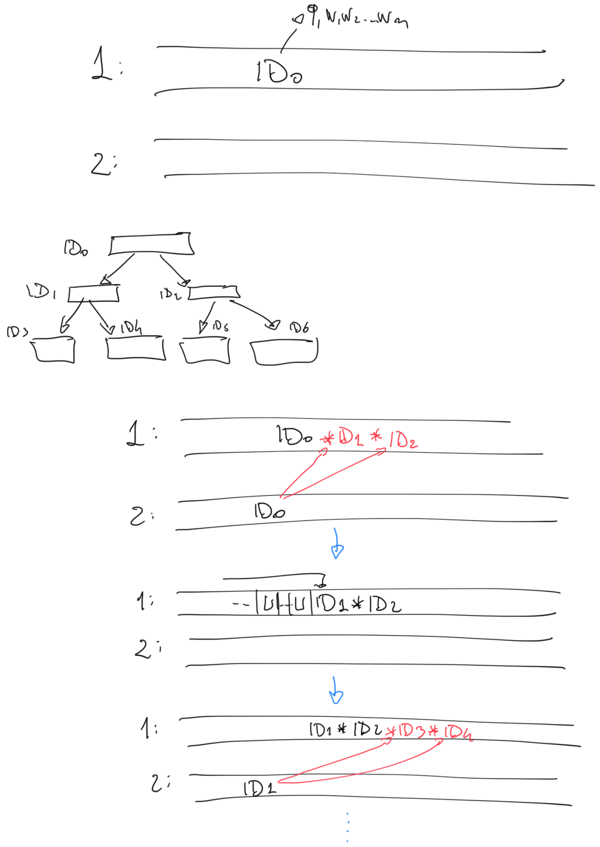
**Definition 10.** *Consider a NTM* $M$*. We use* $L(M)$ *to denote the set of all strings accepted by* $M$*. We say that* $M$ accepts *a language* $\mathcal{L}$ *if* $L(M) = \mathcal{L}$*. Moreover, we say that* $M$ decides *a language* $\mathcal{L}$ *if* $M$ *accepts* $\mathcal{L}$*, and* $T_M(n) \neq \infty$*, for all* $n \geq 0$*.*

So, with so much power added, do deterministic (i.e., standard) TMs and non-deterministic TMs still accept the same languages?

**Theorem 4.** *Let* $M$ *be a NTM. Then, there exists a 2-tape TM* $M'$ *such that* $L(M) = L(M')$*. Moreover, if* $T_M(n) \neq \infty$*,* $T_{M'}(n) \in O(c^{T_M(n)})$*, for some constant* $c > 0$ *that only depends on* $M$*.*

*Proof.* The idea is as follows. Assume you are given the computation tree of $M$ with input $w$. How would you go about finding a path that ends in the accepting/rejecting state? A depth first traversal is not enough, as remember that some paths can be even infinite, and if we enter such a path, there is no way to exit that path. We can use breadth-first: we visit the tree one level at the time. Our machine $M'$ does exactly this.

[2] Note that when for some input string $w$ of length $n$, the NTM accepts $w$, the presence of other infinite paths in the same computation tree makes the time required by $M$ with inputs of length $n$ infinite. Indeed, one should interpret the expression $T_M(n)$ as a way for us to understand, at a glance, on what the machine is doing with inputs of length $n$, and not as a concrete notion of time spent, since non-deterministic machines are not something we can actually build in the real world.



$M'$ has two tapes. In the first tape it first places the initial ID of $M$ with input $w$, $ID_0$. Then, it copies $ID_0$ to the second tape, and according to the transition function of $M$, it copies $ID_0$ at the end of the first tape (after $ID_0$), and then changes it to $ID_1$. The machine does the same with $ID_2$. After the machine has written $ID_1$ and $ID_2$, it checks if the state $q_{\texttt{accept}}$ appears in any of them, in which case, $M'$ accepts. Otherwise, $M'$ then removes $ID_0$ from the two tapes, and positions the first head on $ID_1$. It then copies $ID_1$ to the second tape, and writes $ID_3$ and $ID_4$ at the end of the first tape, and keeps doing this. If at some point $M'$ cannot construct new IDs anymore, and all the IDs in the first tape are rejecting, then $M'$ rejects.

Clearly, if there is a path in the computation tree of $M$ ending in an accepting state, $M'$ will find it and halt accepting. Thus, $L(M) = L(M')$.

We point out that the construction of $M'$ also shows that if there is no accepting path, and all paths of $M$ are rejecting, $M'$ will reject, as it will eventually not be able to construct any new IDs. Moreover, If there is no accepting path, but $M$ has at least one infinite path, then $M'$ loops, as it will keep constructing new IDs. So, $M'$ not only accepts the same language as $M$, but if $M$ decides a certain language, $M'$ decides it as well.

We now focus on the time required by $M'$. Assume $T_M(n) \neq \infty$. For this, let us focus on the computation tree of $M$ with input some string of length $n$. The depth of this tree must be at most $T_M(n)$, since by definition $T_M(n)$ is the maximum number of steps we can have on all paths of all computation trees of $M$ with inputs of length $n$.

Let us consider now the number of children that an ID of a computation tree of $M$ can have. Regardless of the length of the input, the number of children only depends on the transition function of $M$, i.e., how many triples $\delta$ outputs, when given a state and a symbol as input. Assume the maximum number of such triples is some number $c > 0$. Note that $c$ only depends on the control of $M$ and not on the input we give to $M$.

So, in the worst case, the computation tree of $M$ with inputs of length $n$ has depth equal to $T_M(n)$ and each node has $c$ children. So, the total number of IDs in the tree is the sum of all IDs that occur at each level of the tree. That is:

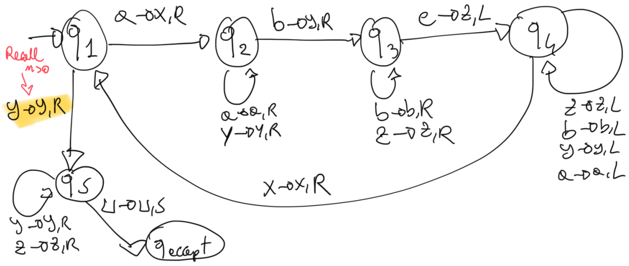
$$\sum_{i=0}^{T_M(n)} c^i \in O(c^{T_M(n)}).$$

Since, in the worst case, $M'$ needs to visit all IDs in the computation tree of $M$, it needs to roughly perform (modulo some constant factors) a number of steps equal to the number of IDs in the computation tree of $M$, and thus $T_{M'}(n) \in O(c^{T_M(n)})$, and the claim follows. □

## 4.1 Exercise 1

$L = \{a^n b^n c^n \mid n > 0\}$ e.g. aaabbbccc

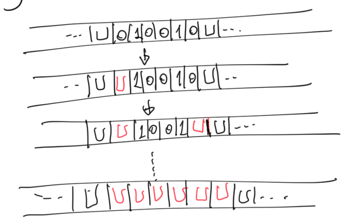
Note $\epsilon \notin L$

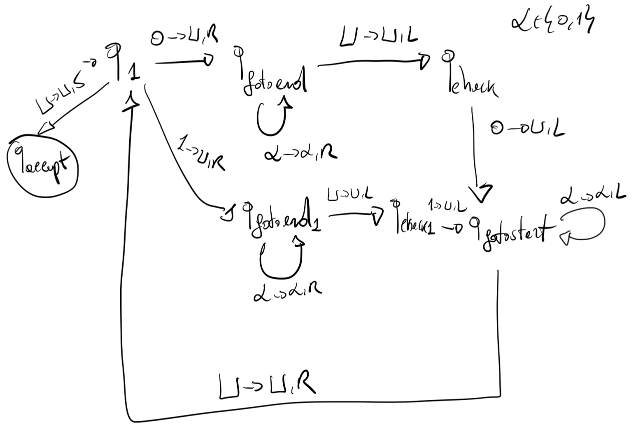


## 4.2 Exercise 2

In this exercise, for a string $w$, $w^R$ represents the reverse of $w$. For example, if $w = 0100$, then $w^R = 0010$.

$L = \{w w^R \mid w \in \{0,1\}^*\}$

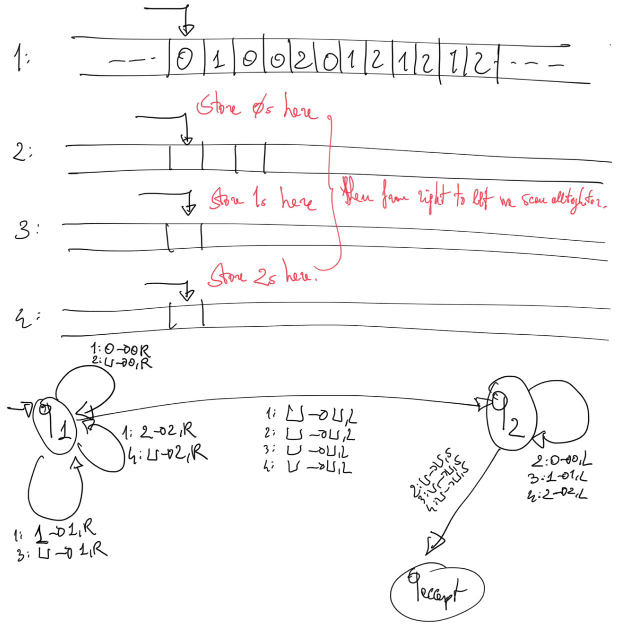




## 4.3 Exercise 3

$\mathcal{L} = \{ w \mid w \in \{0,1,2\}^*$ and the number of 0s, 1s and 2s is the same $\}$

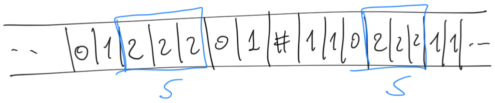
Let's see a multi-tape version... We use 4 tapes



## 4.4 Exercise 4

NTM!!!

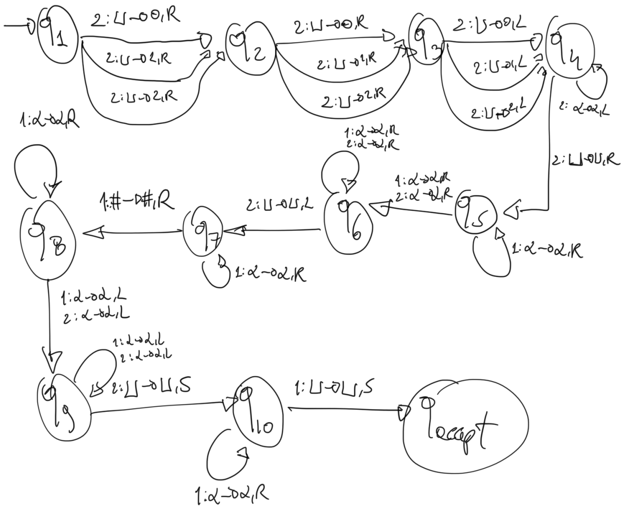
$\mathcal{L} = \{ A\#B \mid A,B \in \{0,1,2\}^+ \text{ and } \exists S \in \{0,1,2\}^+ \text{ s.t. } |S| = 3 \text{ and } S \text{ is a substring of } A \text{ and } B \}$



Idea: The machine has two tapes:

- First, it writes 3 symbols by guessing
- Then it verifies A and B contain the string by guessing where S starts in A and B (for B, it goes backwards)

$\alpha \in \{0,1,2\}$


$q_1$
$q_2$
$q_3$
$q_4$
$q_5$
$q_6$
$q_7$
$q_8$
$q_9$
$q_{10}$
$q_{accept}$
$2: \sqcup \to 0, R$
$2: \sqcup \to 1, R$
$2: \sqcup \to 2, R$
$2: \sqcup \to 0, R$
$2: \sqcup \to 1, R$
$2: \sqcup \to 2, R$
$2: \sqcup \to 0, L$
$2: \sqcup \to 1, L$
$2: \sqcup \to 2, L$
$2: \alpha \to \alpha, L$
$2: \sqcup \to \sqcup, R$
$1: \alpha \to \alpha, R$
$1: \alpha \to \alpha, R$
$2: \alpha \to \alpha, R$
$1: \alpha \to \alpha, R$
$2: \alpha \to \alpha, R$
$2: \sqcup \to \sqcup, L$
$1: \alpha \to \alpha, R$
$1: \# \to \#, R$
$1: \alpha \to \alpha, R$
$1: \alpha \to \alpha, L$
$2: \alpha \to \alpha, L$
$1: \alpha \to \alpha, L$
$2: \alpha \to \alpha, L$
$2: \sqcup \to \sqcup, S$
$1: \sqcup \to \sqcup, S$
$1: \alpha \to \alpha, R$

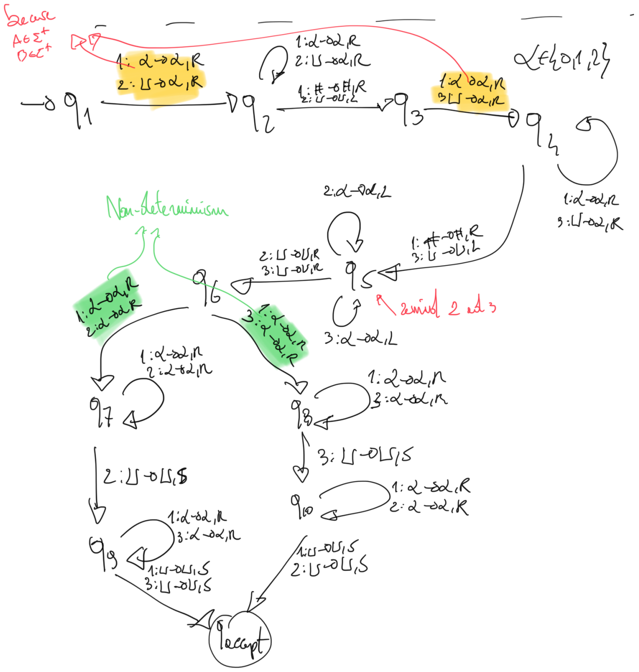
## 4.5 Exercise 5

$$\mathcal{L} = \{ A\#B\#W \mid A,B,W \in \{0,1,2\}^+ \text{ and } (W=AB \text{ or } W=BA) \}$$

e.g. 000#111#000111 or 000#111#111000

Idea: - Copy A and B in two tapes.

- Compare ~~W~~ with A (or B, must guess) and then with B (or A).



Because $A \in \Sigma^+$ $B \in \Sigma^+$
$\alpha \in \{0,1,2\}$
$q_1$
$1: \alpha \to \alpha, R$
$2: \sqcup \to \alpha, R$
$q_2$
$1: \alpha \to \alpha, R$
$2: \sqcup \to \alpha, R$
$1: \# \to \#, R$
$2: \sqcup \to \sqcup, L$
$q_3$
$1: \alpha \to \alpha, R$
$3: \sqcup \to \alpha, R$
$q_4$
$1: \alpha \to \alpha, R$
$3: \sqcup \to \alpha, R$
$1: \# \to \#, R$
$3: \sqcup \to \sqcup, L$
$q_5$
$2: \alpha \to \alpha, L$
$3: \alpha \to \alpha, L$
rewind 2 and 3
$2: \sqcup \to \sqcup, R$
$3: \sqcup \to \sqcup, R$
$q_6$
Non-determinism
$1: \alpha \to \alpha, R$
$2: \alpha \to \alpha, R$
$3: \alpha \to \alpha, R$
$1: \alpha \to \alpha, R$
$q_7$
$1: \alpha \to \alpha, R$
$2: \alpha \to \alpha, R$
$2: \sqcup \to \sqcup, S$
$q_9$
$1: \alpha \to \alpha, R$
$3: \alpha \to \alpha, R$
$1: \sqcup \to \sqcup, S$
$3: \sqcup \to \sqcup, S$
$q_8$
$1: \alpha \to \alpha, R$
$3: \alpha \to \alpha, R$
$3: \sqcup \to \sqcup, S$
$q_{10}$
$1: \alpha \to \alpha, R$
$2: \alpha \to \alpha, R$
$1: \sqcup \to \sqcup, S$
$2: \sqcup \to \sqcup, S$
$q_{accept}$

# 5 Universality, Limits of TMs and Computational Classes

We have seen in the previous lecture that TMs are actually quite powerful, in that seemingly strong extensions to the computational model do not increase the set of languages they accept.

Here we give one last evidence of the robustness of TMs as a computational model, which should (together with the previous lecture) convince ourselves that every problem that can be solved by any physically realizable model of computation is also solvable by a TM.

Up until now, our TMs were a bit "limited" in their *scope*. That is, each machine is dedicated to solve only one problem. But for TMs to be really placed at the same level of modern computers, they need something more: they should be *programmable*. That is, can we devise a *general purpose TM $U$*, that given as input (some encoding of) another TM $M$ and a string $w$, it is able to simulate the execution of $M$ with input $w$. So, we can use $U$ as a programmable machine, that can run the code of other machines.

We call the machine $U$ a *universal TM*.

**Encoding of a TM.** Since our machine $U$ must take another machine $M$ as input, we first need to clarify how the machine $M$ is encoded in the input tape of $U$. So, we need an *encoding of TMs*.

Consider a TM $M$ of the form

$$(Q, \Sigma, \Gamma, \delta, q_1, q_{\texttt{accept}}, q_{\texttt{reject}}).$$

We want to encode $M$ as a string $< M >$ in binary. Remember that every TM is equivalent to a TM whose tape alphabet is only $\Gamma = \{0, 1, \sqcup\}$. So, from now on we will always assume our machines are over the binary alphabet, as they can encode their input and other data in binary.

Assume $Q = \{q_1, \ldots, q_n, q_{\texttt{accept}}, q_{\texttt{reject}}\}$. We encode the state $q_i$ with a sequence of $i$ zeros. So $q_1$ is encoded as 0, $q_3$ as 000 and so on. The special states $q_{\texttt{accept}}$ and $q_{\texttt{reject}}$ with $n+1$ and $n+2$ zeroes.

Similarly, we need to represent the symbols of the alphabet $\Gamma = \{0, 1, \sqcup\}$ (which also contains $\Sigma$). We use a similar encoding.

We represent 0 with 0, 1 with 00 and the special blank symbol with 000.

Finally, we need to represent the transition function $\delta$. For example,

$$\delta(q_1, 1) = (q_3, \sqcup, R)$$

is represented as the string

$$0\ 1\ 00\ 1\ 000\ 1\ 000\ 1\ 00\ 11\ \cdots$$

Here, for example we encode $L$ with 0, $R$ with 00 and $S$ with 000. Each transition is separated from the next with two 1s.

**TMs as numbers.** So a TM can be represented by a binary string, which can be seen as a natural number encoded in binary. So, every machine is essentially a number! In principle however, not every number represents a machine. But, we can adopt the convention that every number not corresponding to a valid encoding of a TM, represents a dummy TM that rejects all strings, i.e., its language is the empty set. For example, such a machine is the one with transitions $q_1 \overset{\alpha\to\alpha,S}{\to} q_{\texttt{reject}}$, for each tape symbol $\alpha$.

## 5.1 The Universal TM

Now that we know how to encode a TM in a binary string, we can show how our universal TM $U$ works.

**Goal:** Devise a TM $U$ that with input $< M >$ (i.e., the encoding of a TM $M$) and a string $w \in \{0,1\}^*$, $U$ simulates the execution of $M$ with input $w$, i.e., $U$ accepts/rejects/loops with input $(< M >, w)$ whenever $M$ accepts/rejects/loops with input $w$.

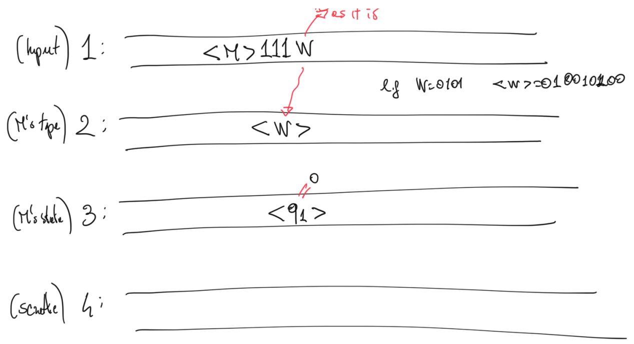
**Structure of $U$.** Our universal TM $U$ is a TM with 4 tapes.

1. Tape 1: contains the encoding of $M$ and the string $w$, e.g., stored like $< M >$ $111w$, where $< M >$ is the encoding of $M$ as described before, then 111 are used as separators, and $w$ is the input string to $w$. Note that $w$ is stored "as is".

2. Tape 2: This is used to simulate $M$'s tape. So, at the beginning, it must contain $w$. However, since $U$ must simulate $M$'s execution on input $w$, it needs first to convert the symbols $w$ to the encoding used for the symbols of $M$, so that $U$ can correctly use the transitions encoded in $< M >$. So, for example, if
$$w = 0101,$$
When $U$ starts, it writes the string $< w >= 0\ 1\ 00\ 1\ 0\ 1\ 00$ in the second tape., and positions the second tape's head to the beginning.

3. Tape 3: $U$ writes the current state of $M$. At the beginning it is its start state $q_1$. Also this must be stored properly encoded. Remember that the encoding of the start state is a plain 0, i.e., $< q_1 >= 0$.

4. Tape 4: This is used by $U$ to store temporary data, we will see in a moment.



**How $U$ works.** After $U$ initializes its tapes as discussed above, it does the following:

1. It first scans tape 1, and checks if $< M >$ actually encodes a valid TM. If this is only garbage, $U$ immediately moves to its rejecting state. Note this is correct, because we assumed that invalid encodings represent TMs that reject all strings. So regardless of the content of $w$, $M$ rejects $w$, and so does $U$. Otherwise,

2. $U$ scans tape 1, searching for a transition encoded like so:

$$0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m.$$

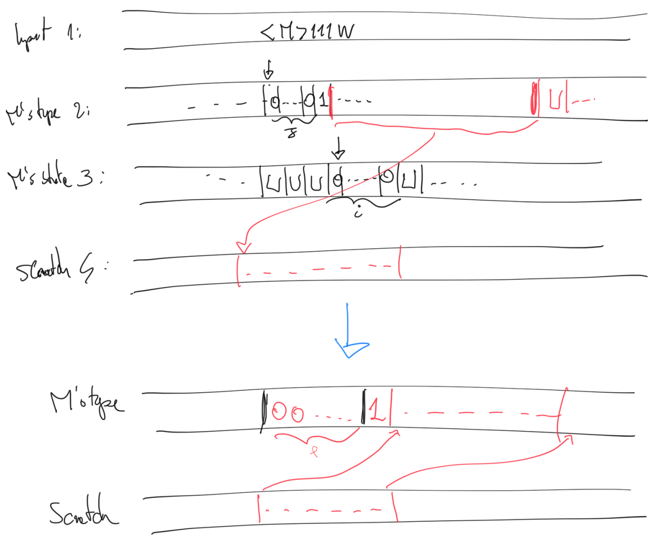
   where $0^i$ encodes the state stored on tape 3, $0^j$ encodes the symbol that is pointed by $U$'s tape 2 head. When it finds it, it must simulate the transition.

3. (a) $U$ clears the string $0^i$ on tape 3 with blanks, and writes the new state $0^k$.

   (b) $U$ replaces the current symbol $0^j$ with the new symbol $0^l$. If $0^l$ is shorter or longer than $0^j$, then before writing $0^l$, $U$ copies the content of tape 2 to the right of where $0^j$ resides to the scratch tape. Then, writes $0^l$, and the copies it back to tape 2.

   (c) $U$ moves the head in tape 2 to the position of the next 1 to the right/left, if $0^m$ encodes $R$ or $L$ respectively. If it encodes $S$, it leaves the head there. If moving U's second tape head to the right (resp., left), the head goes over $\sqcup$, then U writes 1 there and then writes 000 from left to right (resp., from right to left) in the second tape, to represent the fact that the head of M is on a blank.

4. If, after step 3, $U$ has written the encoding of $M$'s state $q_{\texttt{accept}}$ (resp., $q_{\texttt{reject}}$) on tape 3, it moves to *its* accepting (resp., rejecting) state.



It should be clear, by the construction, that $U$ accepts/rejects/loops its input $< M > 111w$ iff the TM $M$ accepts/rejects/loops $w$.

**Church-Turing Thesis.** With the above construction, we should be now fairly convinced that the TM is a very robust computational model, as not only all the extensions we tried did not increase its expressive power, but it is also powerful enough to behave like a general purpose computer. During the years, TMs have been shown to be powerful enough to simulate many other computational models, including modern computers. All such observations led two of the fathers of modern computer science, i.e., Alan Turing and Alonzo Church, to conjecture that there actually is no more powerful computational model than TMs. That is,

*Any problem that can be solved by some physically realizable model of computation can also be solved by a Turing Machine.*

The above claim is known as the **Church-Turing Thesis**.

## 5.2 Limits of TMs

Despite the great computational power that TMs enjoy, they also have their limits. In fact, we are going to prove that there exist a language for which no TM exists that accepts it.

This, together with the fact that we believe no more powerful computational model exist, implies that there are problems that are *inherently impossible to solve*, no matter which kind of computer/machine etc. you are willing to use.

**Diagonal language.** Consider all possible binary strings $w_1, w_2, w_3, \ldots$. Recall that each string $w_i$ represents a TM $M_i$ (dummy if bad encoding). We now consider the language

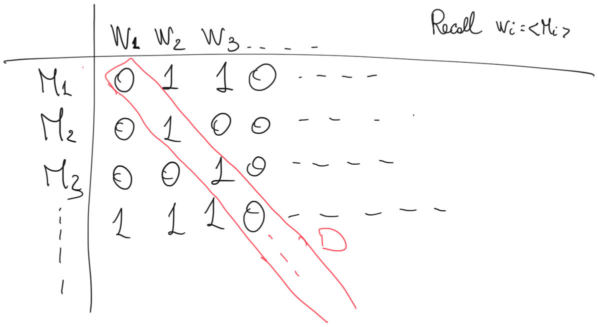
$$\mathcal{L}_d = \{w_i \mid M_i \text{ does not accept } w_i\}.$$

That is, $\mathcal{L}_d$ contains all binary strings that are *not accepted* by the TM they represent. By "not accepted", then we mean that either $M_i$ rejects $w_i$ or even that $M_i$ loops with input $w_i$.

We prove the following.

**Theorem 5.** *There is no TM $M$ such that $L(M) = \mathcal{L}_d$.*

*Proof.* Imagine we write down an infinite table, were each row is associated to a TM, and each column to a binary string. We write a 1 in the cell corresponding to $M_i$ and $w_j$ if $M_i$ accepts $w_j$, and a 0 otherwise.



| | $W_1$ | $W_2$ | $W_3$ | ... |
|---|---|---|---|---|
| $M_1$ | 0 | 1 | 1 | 0 ... |
| $M_2$ | 0 | 1 | 0 | 0 ... |
| $M_3$ | 0 | 0 | 1 | 0 ... |
| ⋮ | 1 | 1 | 1 | 0 ... |

So, for example $M_1$ accepts $w_2$, but does not accept $w_1$.

Note that the row of some TM $M_i$ essentially represents the language of $M_i$, e.g. for $M_1$, the row

$$[0110\cdots]$$

indicates which strings belong to the language of $M_1$, $L(M_1)$. The row of $M_i$ is called *characteristic vector of $M_i$*.

Consider now the diagonal of the matrix

$$D = [0110\cdots].$$

If we flip the bits in $D$, and obtain

$$\bar{D} = [1001\cdots],$$

we see that $\bar{D}$ is actually the characteristic vector of the TM that accepts $\mathcal{L}_d$, as the strings with a 1 in $\bar{D}$ are the strings not accepted by the machine they represent. Hence the name *diagonal language.*

However, there is no row in the table equal to $\bar{D}$. Indeed, assume, towards a contradiction, that some row of the matrix coincides with $\bar{D}$. For example, say this is the $i$-th row. Let $\alpha$ be the $i$-th symbol in this row. This is the same to say that the $i$-th element of the diagonal of the matrix contains $\alpha$. Then, this row cannot be $\bar{D}$, because the $i$-th element in $\bar{D}$ is the opposite of the $i$-th element of the diagonal, by construction of $\bar{D}$, and we obtain a contadiction. So, no row of the matrix coincides with $\bar{D}$.

Then, if there is no row (i.e., TM) in the matrix which coincides with $\bar{D}$, i.e., the characteristic vector of $\mathcal{L}_d$, then there is no TM whose language is $\mathcal{L}_d$ (recall that in the matrix we list *all* TMs). □

## 5.3 Computational Classes

So, the diagonal language $\mathcal{L}_d$ is an example of a language for which there is no TM that is even just able to accept it. But are there only languages that are not accepted and languages that can be decided or there is more?

In order to properly classify languages, we first need to define some important classes of languages. Here we focus only on languages over the binary alphabet $0, 1$, as we have already seen that the alphabet is not actually important, as we can always find an encoding for the strings in binary.

The first class, denoted RE, is the so-called class of *recursively enumerable* languages, also called Turing-recognizable, or semi-decidable.

$$\text{RE} = \{\mathcal{L} \mid \text{ there is a TM M that accepts } \mathcal{L}\}.$$

So, RE contains all that languages that are accepted by some TM. Recall, the fact that $\mathcal{L}$ is accepted by $M$ does not imply that $M$ rejects all other strings, it may very well not halt at all.

The other class, denoted R, is the class of *recursive* languages, also called *decidable*. R collects languages whose corresponding decision problem can be effectivelly solved.

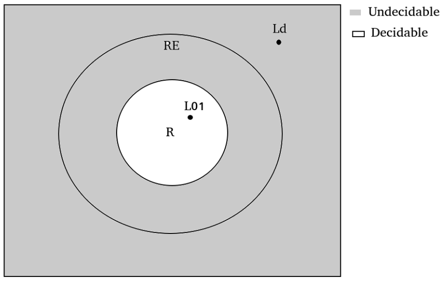
$$\text{R} = \{\mathcal{L} \mid \text{ there is a TM M that decides } \mathcal{L}\}.$$

So, a language $\mathcal{L}$ is in R if we can devise a TM $M$ that not only accepts every string in $\mathcal{L}$, but if given a string not in $\mathcal{L}$, $M$ always halts in the rejecting state.

Such kind of machines are much closer to the notion of algorithm we are used to, i.e., a procedure that is able to *completely solve a problem* because given an input, if the answer is "yes", the algorithm halts answering "yes", and if the answer is "no", the algorithm also halts answering "no".

Ideally, if we are trying to solve a problem, we would like the problem to be decidable, as this means we can provide an algorithm that always gives the right answer in a finite amount of time.

**Properties.** Clearly, by definition, if there is a TM $M$ that decides a language $\mathcal{L}$, then $M$ also accepts $\mathcal{L}$. Hence, $\text{R} \subseteq \text{RE}$.



We have already shown that the diagonal language has no TM that accepts it. So, we know that $L_d$ lies outside of the larger circle.

And we also know that there are definitely languages inside the inner circle. For example the language

$$\mathcal{L}_{01} = \{0^n 1^n \mid n \geq 0\}.$$

We have seen a TM that *decides* $\mathcal{L}_{01}$ in previous lectures.

Every problem that is not in R, i.e., it is not decidable, is called *undecidable*. These are all the languages $\mathcal{L}$ for which either there is no TM that accepts $\mathcal{L}$ at all, i.e., $\mathcal{L} \notin$ RE, or there is a TM that accepts $\mathcal{L}$ but it is not guaranteed to halt with all other strings not in $\mathcal{L}$, i.e. $\mathcal{L}$ lies in the ring ($\mathcal{L} \in \text{RE} \setminus \text{R}$).

**Remark.** Often, distinguishing whether a language is either in R or outside of R is much more important than distinguishing between being in RE or not in RE, since in the latter case, whatever is the outcome, we cannot conclude if the problem represented by such a language is completely solvable, as there might not be a machine that always halts with the right answer.

So, usually, given a language $\mathcal{L}$, the main question we want to answer is:

*Does $\mathcal{L}$ actually lie in R or outside of it?*

To answer these kind of questions, we are going to prove some useful results.

## 5.4 Properties of recursive and recursively enumerable languages

The first property we show is about complements of recursive languages.

**Definition 11.** *Consider a language $\mathcal{L}$. The* complement of $\mathcal{L}$ *is the language $\bar{\mathcal{L}}$ that contains all strings but the ones in $\mathcal{L}$.*

Essentially, if $\mathcal{L}$ describes a decision problem, where the answer is "yes" when the input is in $\mathcal{L}$, and "no" otherwise, $\bar{\mathcal{L}}$ is the problem that flips the answer, i.e., the answer is "yes" if the input is *not* in $\mathcal{L}$, and "no" otherwise.

For example, "Does a given vector of integers $v$ contain the number 0?". Its complement is "Is it the case that the given vector of integer $v$ *does not* contain the number 0?".

**Theorem 6.** *If $\mathcal{L} \in R$, then $\bar{\mathcal{L}} \in R$.*

*Proof.* The intuition is that if $\mathcal{L} \in$ R it means that the machine $M$ accepting $\mathcal{L}$ always halts, either in an accepting or a rejecting state. So, consider the TM $M'$ where we just swap $q_{\mathtt{accept}}$ with $q_{\mathtt{reject}}$ in $M$. If $w \in \mathcal{L}$, $M$ accepts $w$, and thus $M$ rejects $w$. If $w \notin \mathcal{L}$, $M$ rejects $w$ (because $M$ decides $\mathcal{L}$), and thus $M'$ accepts $w$. Thus, $M'$ *decides* $\bar{\mathcal{L}}$. □

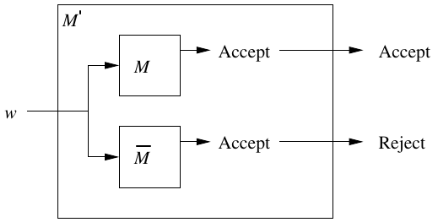
Note that the same does not necessarily hold for RE, i.e., $\mathcal{L} \in$ RE does not imply that $\bar{\mathcal{L}} \in$ RE.[3]

What about languages in RE?

**Theorem 7.** *If $\mathcal{L} \in RE$ and $\bar{\mathcal{L}} \in RE$, then $\mathcal{L} \in R$ (and thus $\bar{\mathcal{L}} \in R$).*

*Proof.* If $\mathcal{L} \in$ RE and $\bar{\mathcal{L}} \in$ RE it means that we have two TMs, $M$ and $\bar{M}$. $M$ halts in an accepting state if its input $w$ belongs to $\mathcal{L}$, and $\bar{M}$ halts in an accepting state if its input $w$ does not belong to $\mathcal{L}$.

So, given a string $w$, we can decide whether $w \in \mathcal{L}$ or not by running $M$ and $\bar{M}$ in parallel with input $w$.[4] While executing $M$ and $\bar{M}$, if $M$ accepts, then $w \in \mathcal{L}$, and $M'$ will accept. If $\bar{M}$ accepts, then $w \notin \mathcal{L}$, and $M'$ will reject.



□

[3] If this was the case, then RE = R, as a consequence of the fact that $\mathcal{L} \in$ RE and $\bar{\mathcal{L}} \in$ RE implies $\mathcal{L} \in$ R (see next theorem). But, we are going to see that there are languages in RE but not in R.

[4] This can be achieved by executing $M$ and $\bar{M}$ one step at the time, i.e. first execute one step for $M$, then one step for $\bar{M}$, and so on.

# 6 Universal Language and the Halting Problem

In the previous lecture we have seen an example of a language that is not in RE, i.e., the diagonal language $\mathcal{L}_d$. We also introduced some key results regarding languages and their membership in R and RE that will help us in our journey in classifying even more languages.

We now show an important problem (language) that is RE but not in R. Recall that we have seen that TMs can be encoded via binary strings.

**Definition 12.** *The* universal language *is the language*

$$\mathcal{L}_u = \{(M, w) \mid M \text{ accepts } w\}.$$

Formally, $(M, w)$ is not a binary string, so $\mathcal{L}_u$ is made of binary strings that encode $(M, w)$, e.g. $< M > 111w$ as we have seen in the previous lecture. From now on, whenever we define languages, we always assume its elements are properly encoded in binary. The language $\mathcal{L}_u$ essentially represents the decision problem asking the following

*Given the code of a TM $M$ and a string $w$, does $M$ accept $w$?*

We first show that $\mathcal{L}_u$ is in RE.

**Theorem 8.** $\mathcal{L}_u \in RE$.

*Proof.* To show the claim, we need to show that there is a TM $M'$ such that $L(M') = \mathcal{L}_u$. The latter means that if $(M, w) \in \mathcal{L}_u$, then $M'$ accepts $(M, w)$ and if $(M, w) \notin \mathcal{L}_u$, then $M'$ does not accept $(M, w)$ (which can mean either that $M'$ rejects, or loops).

We actually already know such a machine $M'$. We have seen that the computation of a TM $M$ with input $w$ can be simulated by the Universal TM $U$. In fact, with input an (encoded) pair $(M, w)$, $U$ simulates $M$, and if $M$ accepts $w$, then so does $U$. If $M$ does not accept $w$, $U$ does not accept. So, $L(U) = \mathcal{L}_u$.

**Remark.** One might wonder why $U$ is not also able to *decide* $\mathcal{L}_u$, i.e., given $(M, w)$, if $M$ accepts $w$, $U$ accepts $(M, w)$, and if $M$ does not accept $w$, $U$ rejects $(M, w)$. However, remember that $M$ not accepting $w$ can mean two things:

1. Either $M$ *halts* in a rejecting state, or

2. $M$ never halts.

In the first case, $U$, with input $(M, w)$, also halts and rejects, however in the second case, to actually decide $\mathcal{L}_u$, $U$ should also halt and reject, which is not the case, as it simulates $M$, and it does exactly what $M$ does, i.e., never halt. □

Note that the fact that the machine we have shown only accepts $\mathcal{L}_u$ but does not decide $\mathcal{L}_u$, does not mean that there is no more clever TM that does the job. We now show that indeed, such a machine does not exist.

**Theorem 9.** *$\mathcal{L}_u \notin R$.*

*Proof.* The proof is by contradiction. Assume, towards a contradiction that

$$\mathcal{L}_u \in \mathrm{R}.$$

Thus, by Theorem 6

$$\bar{\mathcal{L}}_u \in \mathrm{R},$$

where

$$\bar{\mathcal{L}}_u = \{(M, w) \mid M \textit{ does not } \text{accept } w\}.$$

The fact that $\bar{\mathcal{L}}_u \in \mathrm{R}$ means that there is a TM $\bar{M}$ that accepts all strings in $\bar{\mathcal{L}}_u$ and rejects all other strings. However, we can show that $\bar{M}$ can be used to build a TM $M'$ that accepts (actually even decide) the diagonal language $\mathcal{L}_d$.

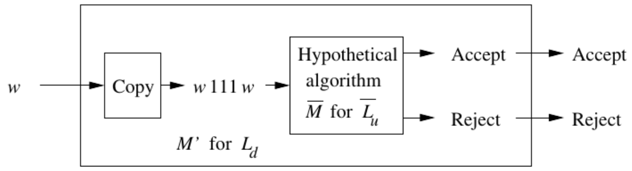
Recall that

$$\mathcal{L}_d = \{w \mid M_w \text{ does not accept } w\}.$$

The idea is that $\mathcal{L}_d$ is somehow a special case of $\bar{\mathcal{L}}_u$, where the input machine and the input string are not arbitrary, but they are the same.

1. $M'$ takes as input a string $w$ in its input tape. It then writes 111 after that and then copies $w$ after that, and moves the head back to the beginning.

2. $M'$ transition function contains a copy of $\bar{M}$ transition function, and after step 1 above, it moves to the start state of $\bar{M}$.



To prove that $M'$ decides $\mathcal{L}_d$, we need to prove two claims: if $w \in \mathcal{L}_d$, $M'$ accepts $w$, and if $w \notin \mathcal{L}_d$, then $M'$ *rejects* $w$.

1. Assume $w \in \mathcal{L}_d$, i.e., $w$ encodes a TM $M_w$ such that $M_w$ does not accept $w$. This means that $\bar{M}$, with input $w111w$ accepts, as it decides the complement of $\mathcal{L}_u$.

2. Assume $w \notin \mathcal{L}_d$, i.e., $w$ encodes a TM $M_w$ such that $M_w$ accepts $w$. This means that $\bar{M}$, with input $w111w$ rejects.

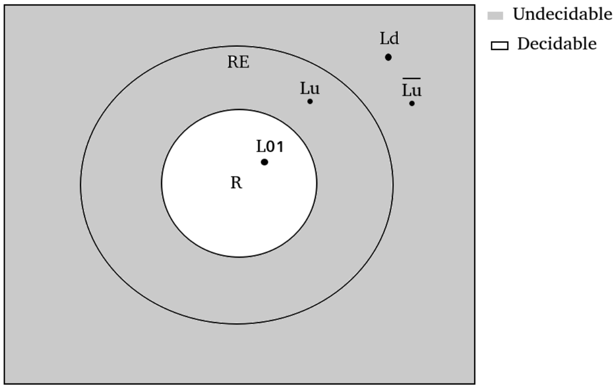
Thus, $M'$ *decides* $\mathcal{L}_d$, which means that $\mathcal{L}_d \in \mathrm{R}$. However, we know that $\mathcal{L}_d \notin \mathrm{RE}$, let alone R. This is a contradiction, and thus $\mathcal{L}_u \notin \mathrm{R}$. □

The fact that $\mathcal{L}_u \in \mathrm{RE}$ and $\mathcal{L}_u \notin \mathrm{R}$ also tells us something about the complement of $\mathcal{L}_u$, i.e.,

$$\bar{\mathcal{L}_u} = \{(M, w) \mid M \text{ does not accept } w\}.$$

We conclude that $\bar{\mathcal{L}_u}$ is not in RE, because if it was, $\mathcal{L}_u \in \mathrm{RE}$ together with $\bar{\mathcal{L}_u} \in \mathrm{RE}$ implies that $\mathcal{L}_u \in \mathrm{R}$ (thanks to Theorem 7). However, we have just shown that $\mathcal{L}_u \notin \mathrm{R}$. Thus, $\bar{\mathcal{L}_u} \notin \mathrm{RE}$.

So, the overall picture of the languages we have seen is:



## 6.1 The Halting problem

Let us see another example of a language that is in RE but not in R. The language, or better, the decision problem it represents, is known as the *Halting problem.*

$$\mathrm{HALT} = \{(M, w) \mid M \textit{ halts} \text{ with input } w\}.$$

That is, it represents the decision problem asking the following question

*Given the encoding of a TM $M$ and a string $w$, does $M$ ever halt its computation when given $w$ as input?*

**Remark.** Note that this is not the same as the Universal language $\mathcal{L}_u$, as here we are not asking whether $M$ accepts $w$, rather we are interested if $M$ ever halts its computation with input $w$, *regardless* if $M$ accepts or rejects $w$.

This is a very important decision problem, both for historical and for technical reasons.

This is one of the first problems that have been shown to be undecidable, and the first proved so by Alan Turing in 1936. Moreover, many relevant problems in computer science can be seen as some variant of the halting problem.

For example, an antivirus trying to understand if a program will ever execute a malicious instruction can be seen as a variant of HALT.

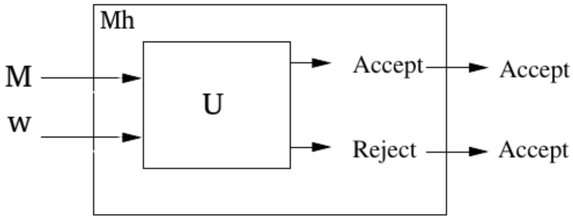
Another example is in Machine Learning: given a small[5] sample of a dataset

[5] Of size independent of the size of the whole dataset.

where each entry in the dataset has a label, decide whether we can actually train a model (i.e., build an algorithm) that given an entry (from the whole dataset), provides the right label, with high probability.

**Theorem 10.** *$HALT \in RE$ but $HALT \notin R$.*

*Proof.* Once again, to prove that HALT $\in$ RE, we need to prove that a TM exists accepting HALT. For this, we rely again on the Universal TM $U$, and build a TM $M_h$ that accepts HALT, by using $U$ as a subroutine. The construction is similar to the one given for the universal language. In fact, since $U$ accepts or rejects whenever the input $M$ accepts or rejects, it is enough to change $U$ in such a way that even if $M$ rejects $w$, $U$ moves anyway to an accepting state.



Thus, to prove that HALT $\in$ RE, we need to prove that $M_h$ accepts HALT, i.e.,

1. If $(M, w) \in$ HALT, then $M_h$ accepts $(M, w)$;
2. If $(M, w) \notin$ HALT, then $M_h$ does not accept $(M, w)$ (i.e., $M_h$ rejects or loops).

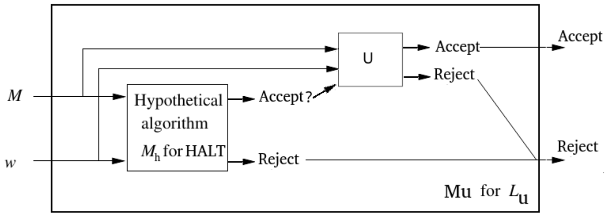
If $(M, w) \in$ HALT, then $U$ with input $(M, w)$ either accepts of rejects, thus, in any case, $M_h$ accepts. If $(M, w) \notin$ HALT, $U$ does not halt, and thus also $M_h$ does not halt, hence $M_h$ does not accept $(M, w)$.

We now prove that HALT $\notin$ R. Assume, towards a contradiction, that HALT $\in$ R. So, there is a TM $M_h$ that *decides* HALT, i.e., if $M$ halts with input $w$, $M_h$ accepts, but also if $M$ does not halt with $w$, $M_h$ rejects. We show that we can use $M_h$ to build another TM $M_u$ that decides the universal language, obtaining a contradiction.

The machine $M_u$ does the following, with input $(M, w)$.

1. It first executes $M_h$ with input $(M, w)$).
2. If $M_h$ rejects, then $M_u$ rejects.
3. If $M_h$ accepts, then $M_u$ executes the universal machine $U$ with input $(M, w)$.
4. If $U$ accepts, then $M_u$ accepts.

5. If $U$ rejects, then $M_u$, rejects.



To show that $M_u$ decides $\mathcal{L}_u$ we need to prove two things:

1. If $(M, w) \in \mathcal{L}_u$, then $M_u$ accepts $(M, w)$;
2. If $(M, w) \notin \mathcal{L}_u$, then $M_u$ *rejects* $(M, w)$.

If $(M, w) \in \mathcal{L}_u$, it means that $M$ accepts $w$. Thus, $M$ halts in the accepting state. This means that $M_h$ with input $(M, w)$ will conclude that $M$ halts with input $w$. Thus, $M_u$ will execute $U$ with input $(M, w)$, which will accept, and thus $M_u$ accepts as needed.

If $(M, w) \notin \mathcal{L}_u$, it means that $M$ does not accept $w$. This can mean two things: $M$ rejects $w$, or $M$ loops with input $w$. If $M$ loops, then $M_h$ rejects, and so does $M_u$ as needed. If instead $M$ rejects $w$, $M_h$ accepts, ans thus $U$ is executed with input $(M, w)$, which makes $U$ rejects, and thus $M_u$ rejects as needed.

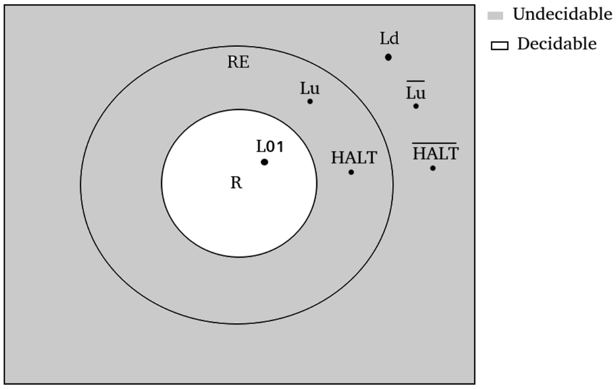
Thus, $M_u$ decides $\mathcal{L}_u$, and we obtain a contradiction.

□

As we did for the universal language, knowing that HALT $\in$ RE, but HALT $\notin$ R, implies that its complement

$$\overline{\text{HALT}} = \{(M, w) \mid M \text{ does not halt with input } w\}$$

is not in RE. Thus, this is the overall picture.



**Remark.** We remark, once again, that the difficulty of HALT (as well as $\mathcal{L}_u$ and $\mathcal{L}_d$), i.e., of not having a TM that decides them, lies in the fact that we cannot find a TM that *for every input* $(M, w)$, it provides the right answer, i.e., does $M$ halt on $w$ or not? But, of course there can be a TM that, for *some inputs* is able to give the right answer, but for some other it will necessarily loop.

**A variation of HALT.** We now focus on one last language, which can be seen as a special case of the Halting problem.

$$\text{HALT-}\epsilon = \{M \mid M \text{ halts with input the empty string } \epsilon\}.$$

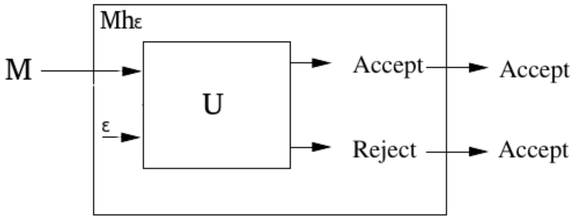
The above language describes the following decision problem:

*Given the encoding of a TM $M$ (and nothing else), does $M$ ever halt its computation when executed with an empty tape?*

We prove that HALT-$\epsilon$ is in RE but not in R.

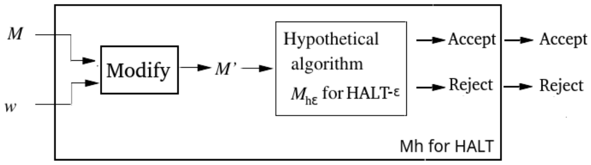
**Theorem 11.** *HALT-$\epsilon \in$ RE, but HALT-$\epsilon \notin$ R.*

*Proof.* As usual, to prove that a language is in RE, we need to exhibit a TM that accepts all strings in the language, and does not accept all others. Our such a TM, dubbed $M_{h\epsilon}$, is almost identical to the one used for HALT. The only difference is that we need to accept HALT-$\epsilon$ which takes as input only a TM, and needs to verify halting for the empty string as input:



Now, if $M \in$ HALT-$\epsilon$, then it means that $M$ halts with input the empty string. Thus, $U$ will either accept or reject, and in both cases, $M_{h\epsilon}$ accepts $M$. If $M \notin$ HALT-$\epsilon$, then it means that $M$ does not halt with input th empty string. Thus, $U$ will not halt as well, and thus $M_{h\epsilon}$ does not accept $M$.

We now prove that HALT-$\epsilon \notin$ R. Again assume, towards a contradiction, that HALT-$\epsilon \in$ R, and let $M_{h\epsilon}$ be a TM that *decides* HALT-$\epsilon$. We now show how to build a TM that uses $M_{h\epsilon}$ as a subroutine to *decide* HALT. We dub such a machine $M_h$, and report it below:



Here, we pick $M$ and $w$ and construct a TM $M'$ that contains some additional transition rules that, before starting $M'$ execution, first erase the content of its tape (so, $M'$ erases its input), and replace it with the string $w$. So, regardless of its input (even the empty string), $M'$ will execute like if its input is $w$.

To prove that $M_H$ above decides HALT, we need to prove two things, as usual:

1. If $(M, w) \in$ HALT, then $M_h$ accepts $(M, w)$;
2. If $(M, w) \notin$ HALT, then $M_h$ rejects $(M, w)$.

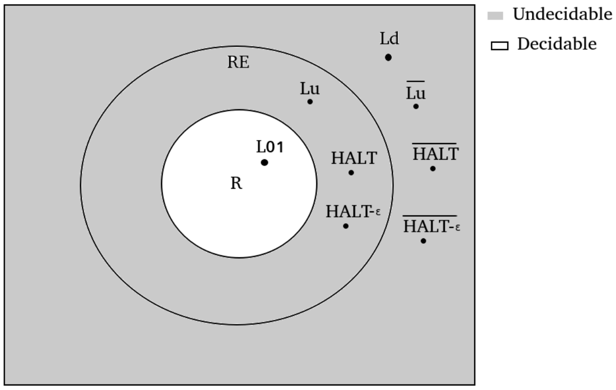
If $(M, w) \in$ HALT, it means that $M$ halts with input $w$, which means that $M'$, even when executed with the empty string, halts. So, $M_{h\epsilon}$ accepts, and $M_h$ accepts overall.

If $(M, w) \notin$ HALT, it means that $M$ does not halt with input $w$, which, by construction of $M'$, means that $M'$ does not halt (even with input the empty string, since it anyways replaces it with $w$). So, $M_{h\epsilon}$ rejects, and so does $M_h$. Thus, $M_h$ decides HALT, i.e., HALT $\in$ R, which is a contradiction. So, HALT-$\epsilon \notin$ R. □

Also here, we can conclude that the complement of HALT-$\epsilon$,

$$\overline{\text{HALT-}\epsilon} = \{M \mid M \text{ does not halt with input the empty string } \epsilon\},$$

is not in RE.



# 7 Reductions and the $\mathcal{L}_e$, $\mathcal{L}_{ne}$ languages

In the previous lectures, we have seen a number of undecidable languages. Moreover, most of our undecidability results have relied on dedicated proofs that we came up with from scratch. However, there is something in common between the above proofs: they somehow exploit the fact that some other language is known to be undecidable.

In particular, the story always goes like this:

1. We want to show that a language $\mathcal{L}$ is undecidable.

2. We assume, by contradiction, that a TM $M$ deciding $\mathcal{L}$ exists.

3. We show that we can construct a TM $M'$, which uses $M$, that can decide another language $\mathcal{L}'$ which we know is undecidable.

4. We obtain a contradiction.

What the above proof is actually doing is known as a *reduction* from the language $\mathcal{L}'$ to the language $\mathcal{L}$. That is, one can decide $\mathcal{L}'$ if she is able to decide $\mathcal{L}$.

**Example 3.** *As a more concrete example of a reduction between two languages (decision problems), consider the problem, where given the map of a city and two cities, we must decide whether two cities are connected in the map. You can reduce this problem to the REACHABILITY problem on directed graphs. Indeed, if you can decide the REACHABILITY problem, the city-map problem can be decided as well. How?*

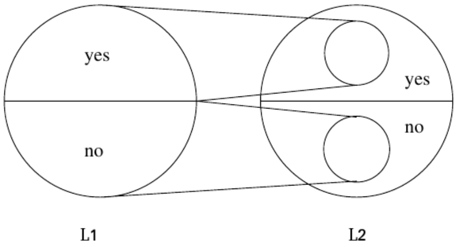
*First, convert the map of the city into a graph (e.g., cities become nodes, and roads become edges), and then convert the source and target city to the corresponding nodes in the graph. Finally, run the reachability algorithm on the obtained graph and the two nodes. The answer given by the algorithm is precisely the answer to the city-map problem.*

As you can see, we did not need to know precisely how an algorithm for the REACHABILITY problem works. It was enough to convert instances of the city-map problem to instances of the REACHABILITY problem in "the right way" (i.e., we must map yes-instances to yes-instances, and no-instances to no-instances).

There exist many kinds of reductions. The onde described in the city-map example is called a *many-one* reduction.

Intuitively, a many-one reduction from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$, is an algorithm that converts yes-instances of $\mathcal{L}_1$ to yes-instances of $\mathcal{L}_2$, and no-instances of $\mathcal{L}_1$ to no-instances of $\mathcal{L}_2$. So, to decide whether a given string $w$ belongs to $\mathcal{L}_1$ or not, one can first convert $w$ to another string $w'$ using the reduction, and then decide whether $w'$ belongs to $\mathcal{L}_2$ or not.

The picture below shows the behaviour of a many-one reduction.



We first formalize the kind of algorithms we use for converting instances from one language to another.

**Definition 13** (Transducer)**.** *A* transducer *is a 3-tapes TM $T$, enjoying the following properties:*

1. *Tape 1 is the* input *tape, which is read only (i.e., $T$ cannot write symbols to it);*
2. *Tape 2 is the* work *tape, which is read/write (i.e., $T$ can both read and write symbols from/to it);*
3. *Tape 3 is the* output *tape, which is write only (i.e., $T$ can only write symbols to it).*
4. *$T$ always accepts its input.*

For a string $w$ and a transducer $T$, we use $T(w)$ to denote the portion of the output tape on which $T$ has written to, after $T$ accepts $w$.

Note that since our reduction must just convert strings to strings, it does not make much sense for the machine to reject its input, and more importantly, we want the machine to always halt, as we want an effective way of converting *any instance* to another instance.

Moreover, we use different tapes, than a standard single tape TM, because we would like to clearly separate the input and output from the amount of data the machine actually uses to perform the conversion. This will come up useful later on in the course, when we will be concerned on the space resources required by our reductions.[6]

**Definition 14** (Reduction)**.** *A (many-one)* reduction *from a language $\mathcal{L}_1 \subseteq \{0,1\}^*$ to a language $\mathcal{L}_2 \subseteq \{0,1\}^*$ is a transducer $T$ such that, for each $w \in \{0,1\}^*$:*

1. *If $w \in \mathcal{L}_1$ then $T(w) \in \mathcal{L}_2$, and*
2. *if $w \notin \mathcal{L}_1$ then $T(w) \notin \mathcal{L}_2$.*

[6] When measuring the memory used by the TM (or any algorithm), it does not make sense to also count the space occupied by the input and the output.

*We write $\mathcal{L}_1 \leq \mathcal{L}_2$ to say that $\mathcal{L}_1$* reduces *to $\mathcal{L}_2$, i.e., there is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$.*

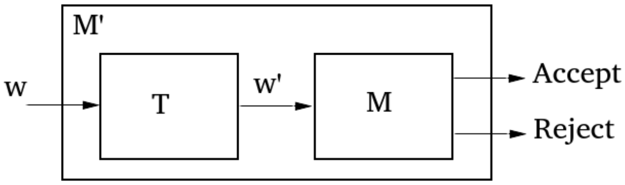
We can now formally prove that our definition of reduction allows us to transfer different properties between languages.

**Theorem 12.** *If $\mathcal{L}_1 \leq \mathcal{L}_2$, then the following hold:*

1. *If $\mathcal{L}_1 \notin R$, then $\mathcal{L}_2 \notin R$.*
2. *If $\mathcal{L}_1 \notin RE$, then $\mathcal{L}_2 \notin RE$.*

*Proof.* Assume that $\mathcal{L}_1 \leq \mathcal{L}_2$, i.e., there exists a reduction $T$ from $\mathcal{L}_1$ to $\mathcal{L}_2$.

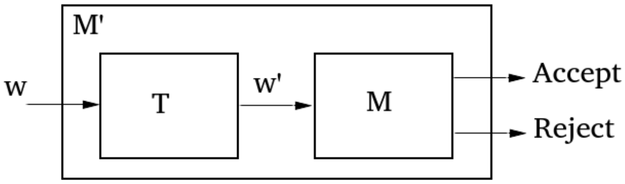
(Item 1) Assume $\mathcal{L}_1 \notin$ R, but, towards a contradiction, assume that $\mathcal{L}_2 \in$ R. This means that there exists a TM $M$ that *decides* $\mathcal{L}_2$, i.e., for each string $w \in \{0,1\}^*$, if $w \in \mathcal{L}_2$, then $M$ accepts $w$, *and* if $w \notin \mathcal{L}_2$, then $M$ rejects $w$. Then, consider the TM $M'$ below:



We show now that $M'$ decides $\mathcal{L}_1$, i.e., if $w \in \mathcal{L}_1$, then $M'$ accepts $w$, and if $w \notin \mathcal{L}_1$, then $M'$ rejects $w$. Assume that $w \in \mathcal{L}_1$. Since $T$ is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, $T(w)$, i.e., the output of $T$ with input $w$, is a new string $w'$ such that $w' \in \mathcal{L}_2$. Thus, since $M$ decides $\mathcal{L}_2$, it follows that when $M'$ executes $M$ with input $w'$, it will accept. Assume that $w \notin \mathcal{L}_1$. Since $T$ is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, $T(w)$, i.e., the output of $T$ with input $w$, is a new string $w'$ such that $w' \notin \mathcal{L}_2$. Thus, since $M$ decides $\mathcal{L}_2$, it follows that when $M'$ executes $M$ with input $w'$, it will reject. Hence, $M'$ decides $\mathcal{L}_1$.

However, we assumed that $\mathcal{L}_1 \notin$ R, i.e., $\mathcal{L}_1$ is not decidable, obtaining a contradiction. Thus, $M$ cannot exist, which implies that $\mathcal{L}_2 \notin$ R.

(Item 2) Assume $\mathcal{L}_1 \notin$ RE, but, towards a contradiction, assume that $\mathcal{L}_2 \in$ RE. This means that there exists a TM $M$ that *accepts* $\mathcal{L}_2$, i.e., for each string $w \in \{0,1\}^*$, if $w \in \mathcal{L}_2$, then $M$ accepts $w$, and if $w \notin \mathcal{L}_2$, then $M$ does not accept $w$. Then, consider the TM $M'$ below:



We show now that $M'$ accepts $\mathcal{L}_1$, i.e., if $w \in \mathcal{L}_1$, then $M'$ accepts $w$, and if $w \notin \mathcal{L}_1$, then $M$ does not accept $w$.

Assume that $w \in \mathcal{L}_1$. Since $T$ is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, $T(w)$, i.e., the output of $T$ with input $w$, is a new string $w'$ such that $w' \in \mathcal{L}_2$. Thus, since $M$ accepts $\mathcal{L}_2$, it follows that when $M'$ executes $M$ with input $w'$, it will accept.

Assume that $w \notin \mathcal{L}_1$. Since $T$ is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, $T(w)$, i.e., the output of $T$ with input $w$, is a new string $w'$ such that $w' \notin \mathcal{L}_2$. Thus, since $M$ accepts $\mathcal{L}_2$, it follows that when $M'$ executes $M$ with input $w'$, $M'$ will not accept $w$ (i.e., it either loops or rejects). Hence, $M'$ accepts $\mathcal{L}_1$.

However, we assumed that $\mathcal{L}_1 \notin$ RE, i.e., there is no TM that accepts $\mathcal{L}_1$, obtaining a contradiction. Thus, $M$ cannot exist and $\mathcal{L}_2 \notin$ RE. □

The above results allow us to transfer "negative" knowledge from the source language $\mathcal{L}_1$ to the target language $\mathcal{L}_2$. Nonetheless, the existence of a reduction also allows us to transfer "positive" knowledge from the target language to the source. In fact, the following is a corollary of the theorem above

**Corollary 1.** *If $\mathcal{L}_1 \leq \mathcal{L}_2$, then the following hold:*

1. *If $\mathcal{L}_2 \in R$, then $\mathcal{L}_1 \in R$.*

2. *If $\mathcal{L}_2 \in RE$, then $\mathcal{L}_1 \in RE$.*

*Proof.* One can see the two implications of Theorem 12 in the opposite way. That is, when we have an implication of the form $A$ implies $B$, this is really just the same thing as saying that "not $B$" implies "not $A$". Thus, saying that $\mathcal{L}_1 \notin$ R implies $\mathcal{L}_2 \notin$ R is equivalent to say that $\mathcal{L}_2 \in$ R implies $\mathcal{L}_1 \in$ R. The same applies to the second implication about RE. □

With Theorem 12 and Corollary 1 at hand, we now have a set of powerful formal tools for placing new problems in the right class. Let us see some examples.

**Remark.** Although we defined our notion of reduction very formally, via transducers, it is usually very difficult and convoluted to design reductions in this form. We should be confident enough at this point that TMs are as powerful as many other computational models, including modern computers. So, for this reason, when we find it useful, rather than describing our reductions as transducers, we will describe their behaviour by means of algorithms written in pseudo code. This is similar in spirit to what most of you did when studying algorithms in the Algorithms and Data Structures course.

This will keep our discussion light, but still allow us to provide formal claims about languages.

## 7.1 Emptiness of a TM's language

Assume you are given the code of a procedure. One important question that we might want to answer about this procedure is whether the procedure is "trivial", i.e., it does not accept any string. The above question can be seen like asking whether the procedure does not have any purpose at all.

**Definition 15.** *The language $\mathcal{L}_e \subseteq \{0,1\}^*$ is the set of all encodings of TMs $M$ that accept no string. That is,*

$$\mathcal{L}_e = \{< M >| \ L(M) = \emptyset\}.$$

Recall that we encode TMs as binary strings, so we focus on the alphabet $\{0,1\}$.

Another interesting decision problem is the complement of $\mathcal{L}_e$, i.e., deciding whether a given TM actually accepts something.

**Definition 16.** *The language $\mathcal{L}_{ne} \subseteq \{0,1\}^*$ is the set of all encodings of TMs $M$ that accept at least one string. That is,*

$$\mathcal{L}_{ne} = \bar{\mathcal{L}}_e = \{< M >| \ L(M) \neq \emptyset\}.$$

**The language $\mathcal{L}_{ne}$.** Let us start by studying $\mathcal{L}_{ne}$.

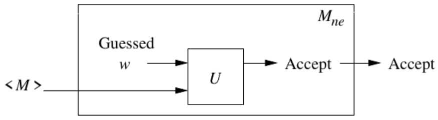
**Theorem 13.** *$\mathcal{L}_{ne} \in RE$.*

*Proof.* To show that $\mathcal{L}_{ne}$ is recursively-enumerable, we need to show that there exists a TM $M_{ne}$ that accepts $\mathcal{L}_{ne}$, i.e., $L(M_{ne}) = \mathcal{L}_{ne}$. As we know that all variations of TMs we have seen until now are equally powerful, we can devise any TM for this purpose, even a non-deterministic one. Since it is easier to devise a machine accepting $\mathcal{L}_{ne}$ by using non-determinism, we construct an *NTM* accepting $\mathcal{L}_{ne}$.

Our $M_{ne}$ must take as input the encoding of a TM $M$, and accept it, if $L(M) \neq \emptyset$. The idea would be that $M_{ne}$ iterates over all strings $w$, and for each such a string, simulates the computation of $M$ over $w$, searching for the one accepted by $M$ (if it exists). However, this approach would not work, since while searching for this string, we might simulate $M$ over a string that makes $M$ loop, and thus we would never be able to move on to the next string.

The trick is to let $M_{ne}$ be a NTM that simply *guesses* a string $w$, and verifies that $M$ accepts it by simulating the computation of $M$ with input $w$ (this can be done via the universal TM $U$). If such a string exists, there will be a path in the computation tree of $M_{ne}$ that ends in an accepting ID.

The idea is shown in the picture below:



□

So, we now can "partially solve" the decision problem encoded by $\mathcal{L}_{ne}$. But, can we actually solve it? Is $\mathcal{L}_{ne}$ decidable? We show this is not the case, via a reduction from a known, undecidable language.

**Theorem 14.** *$\mathcal{L}_{ne} \notin R$.*

*Proof.* We prove the claim by showing that the universal language $\mathcal{L}_u$ reduces to $\mathcal{L}_{ne}$, i.e., $\mathcal{L}_u \leq \mathcal{L}_{ne}$. Since we know that $\mathcal{L}_u \notin$ R, by Theorem 12 (Item 1), we conclude that $\mathcal{L}_{ne} \notin$ R.

**Before starting.** The goal of our reduction is to convert a string $w$ into another string $w'$ in such a way that, if $w$ belongs to $\mathcal{L}_u$ then $w'$ belongs to $\mathcal{L}_{ne}$, and if $w$ does not belong to $\mathcal{L}_u$, then $w'$ does not belong to $\mathcal{L}_{ne}$. When devising a reduction, we need first to understand how strings of the two languages we are considering look like. Regarding $\mathcal{L}_u$, strings in the language are binary-encoded pairs of the form $(M, w)$, where $M$ is a TM and $w$ a binary string such that $M$ accepts $w$. If the string is not in the language, it can be either a wrong encoding of a pair $(M, w)$ or a valid encoding but $M$ does not accept $w$. In principle, our reduction must be able to deal with all three cases above, and thus if the input string is an invalid encoding of a pair $(M, w)$ (which means it is a no-instance for $\mathcal{L}_u$), should return a no-instance for $\mathcal{L}_{ne}$.

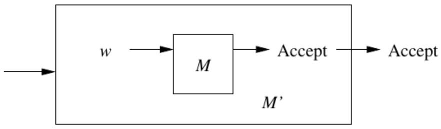
However, note that this part is the least interesting of the reduction, and usually quite trivial to accomplish: just check that the input string is invalid, and if this is the case, then output an (arbitrary) no-instance for the destination language.

So, to keep our reductions simple and readable, we assume we will not describe how our reductions deal with invalid encodings, as we assume by default that they will always first check whether the encoding is invalid, and in which case, return some default no-instance for the target language. In this way, we can focus only on the real "juice" of the reduction, i.e., mapping yes-instances to yes-instances and *valid* no-instances to *valid* no-instances.

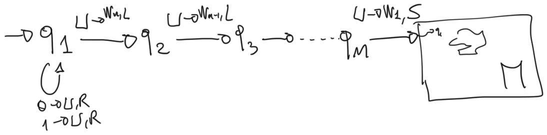
We now proceed with devising our reduction. We need to devise a procedure $T$ that given an encoding of a pair $(M, w)$ outputs the encoding of a TM $M'$.

Moreover, $T$ must be such that $M$ accepts $w$ implies that $L(M') \neq \emptyset$, and $M$ does not accept $w$ implies that $L(M') = \emptyset$.

Our reduction $T$, given $(M, w)$ constructs the following TM $M'$:



Essentially, the TM $M'$ that our reduction constructs completely ignores its input, by replacing it with $w$, and then executes the control of $M$. If $M'$ reaches an accepting state, it means that $M$ accepts $w$. Note that here we are not using the universal TM to simulate $M$ with input $w$. We could have done that, but it would make $M'$ look unnecessarily complex. Indeed, the control of $M'$ is quite simple; assume $w = w_1 \cdots w_n$.



$M'$ first erases its input, then fills its input tape with the string $w = w_1 \cdots w_n$, and then moves to the initial state of $M$ (i.e., $M'$ contains the control of $M$ in its transition function). Let us see why $T$ is a reduction. Consider a pair $(M, w)$ as input to $T$.

Assume $M$ accepts $w$, then the TM $M'$ that $T$ constructs will always reach an accepting state, no matter the input string, i.e., $M'$ accepts all strings, and thus $L(M') \neq \emptyset$.

Assume $M$ does not accept $w$, then the TM $M'$ that $T$ constructs will never accept, no matter its input string, i.e., $M'$ does not accept any input string, and thus $L(M') = \emptyset$.

So, $T$ is a reduction from $\mathcal{L}_u$ to $\mathcal{L}_{ne}$, and since $\mathcal{L}_u \notin$ R, we conclude that $\mathcal{L}_{ne} \notin$ R.

□

**The language $\mathcal{L}_e$.** Now that we have a precise understanding of $\mathcal{L}_{ne}$, we can place $\mathcal{L}_e$ in the right classes, very easily.

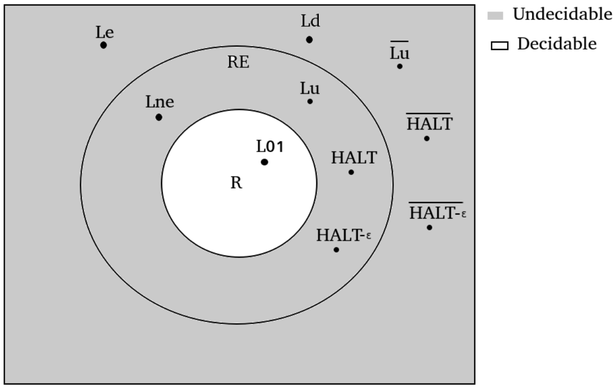
**Theorem 15.** *$\mathcal{L}_e \notin$ RE.*

*Proof.* Towards a contradiction, assume that $\mathcal{L}_e \in$ RE. By Theorem 13, we know that $\mathcal{L}_{ne} \in$ RE. So, both are in RE. Since $\mathcal{L}_e$ and $\mathcal{L}_{ne}$ are the complements of each other, Theorem 7 tells us that since they are both in RE they both also must be in R. However, we have seen in Theorem 14 that $\mathcal{L}_{ne} \notin$ R, obtaining a contradiction. □

**Remark.** It is important to note that languages and their complements do not necessarily behave in the same way. That is, if a language is in RE, there is no guarantee that its complement is in RE. The reason for this is that the fact that a language $\mathcal{L} \in$ RE actually means that we have a TM $M$ that accepts all strings in $\mathcal{L}$, but $M$ could either reject or loop with a string not in the language. Thus, we cannot simply take $M$, invert its accepting state with the rejecting state, and hope that it will accept all strings in $\bar{\mathcal{L}}$. This is because $M$ might loop for some given string not in $\mathcal{L}$. Thus, $M'$ might loop when given a string in $\bar{\mathcal{L}}$!

Only for decidable languages, there is a kind of symmetry with their complement, as flipping the accepting/rejecting states indeed has the expected effect.

Below we report the final picture of all the languages we have studied so far.



# 8 More on Reductions

In this lecture, we further explore the technique of reductions, by showing more problems are undecidable. Until now, our undecidable languages were all defined in terms of TMs, i.e., asking questions about TMs. We will see now that indeed there are more undecidable languages that do not talk about TMs at all! Indeed, we are now going to show that there are some puzzles that are impossible to solve in general. We first focus on the so called Post Correspondence Problem.

## 8.1 Post Correspondence Problem (PCP)

This is an undecidable problem that was introduced by Emil Post in 1946. Let us first state the problem in natural language, and then provide a formal definition as a language of strings.

You are given a table with two columns A and B, with a certain (finite) amount of rows. Each row stores two strings (of possible different lengths), over some arbitrary alphabet. For example,

| A | B |
|---|---|
| 1 | 111 |
| 10111 | 10 |
| 10 | 0 |

The problem is to decide whether we can arrange one or more rows (but not necessarily all) of the table, by also allowing to pick the same row more than once, in such a way, the string we obtain by reading the first column from top to bottom coincides with the string we read on the second column from top to bottom. For example, we can consider the following arrangement of (possibly repeated occurrences of) the rows of our table above:

| | |
|---|---|
| 10111 | 10 |
| 1 | 111 |
| 1 | 111 |
| 10 | 0 |

The string we obtain by reading the first column from top to bottom is 101111110. But this is also the case for the second column. So, the answer to the PCP with input the table shown before is "yes". If there is no way to arrange the rows of the table as to get the same string on both sides, then the answer is "no".

Let us now define the PCP as a language.

$$\mathcal{L}_{\text{PCP}} \quad = \quad \left\{ (A,B) \;\middle|\; \begin{array}{l} A \text{ and } B \text{ are two equally long lists of strings and,} \\ \exists i_1, i_2, \ldots, i_n \text{ such that } A[i_1] \cdots A[i_n] = B[i_1] \cdots B[i_n] \end{array} \right\}.$$

Note that in the above definition, $n$ must be strictly greater than 0, i.e., we need to pick at least one row, otherwise the problem becomes trivial.

A restriction of the PCP, called modified PCP (MPCP), requires that when we choose the rows of the table, the first row we choose *must always be* the first row of the table. So, we define $\mathcal{L}_{\text{MPCP}}$ in the same way as we did for $\mathcal{L}_{\text{PCP}}$ but we additionally require that $i_1 = 1$.

We are first going to show that $\mathcal{L}_{\text{MPCP}} \notin \text{R}$, via a reduction from the universal language $\mathcal{L}_u$. Then, we will show that $\mathcal{L}_{\text{MPCP}} \leq \mathcal{L}_{\text{PCP}}$.

**Theorem 16.** *$\mathcal{L}_u \leq \mathcal{L}_{MPCP}$.*

*Proof.* Recall that instances of $\mathcal{L}_u$ are pairs $(M, w)$ such that $M$ is a TM and $w$ a string; $(M, w)$ is a yes-instance, i.e., $(M, w) \in \mathcal{L}_u$ if $M$ accepts $w$, otherwise it is a no-instance. Our reduction must take as input pairs $(M, w)$ and output two lists $A$ and $B$ in such a way that $M$ accepts $w$ implies $(A, B) \in \mathcal{L}_{\text{MPCP}}$, and if $M$ does not accept $w$, $(A, B) \notin \mathcal{L}_{\text{MPCP}}$.

Consider a pair $(M, w)$, and let $\delta$ be the transition function of $M$, $\Gamma$ its tape alphabet, and let $w = w_1 w_2 \cdots w_n$. Given $(M, w)$, our reduction constructs the following "table" (actually constructs two lists).

| A | B | Comment |
|---|---|---|
| $\#$ | $\#q_1 w_1 w_2 \cdots w_n \#$ | Initial ID of $M$ with input $w$ |
| $\alpha$ | $\alpha$ | $\forall \alpha \in \Gamma$ |
| $\#$ | $\#$ | - |
| $q\alpha$ | $\beta q'$ | if $\delta(q, \alpha) = (q', \beta, R)$, for each $\alpha \in \Gamma$ |
| $q\#$ | $\beta q' \#$ | if $\delta(q, \sqcup) = (q', \beta, R)$ |
| $\gamma q \alpha$ | $q' \gamma \beta$ | if $\delta(q, \alpha) = (q', \beta, L)$, for each $\alpha \in \Gamma$ |
| $\# q \alpha$ | $\# q' \sqcup \beta$ | |
| $\gamma q \#$ | $q' \gamma \beta \#$ | if $\delta(q, \sqcup) = (q', \beta, L)$ |
| $\alpha q_{\texttt{accept}}$ | $q_{\texttt{accept}}$ | $\forall \alpha \in \Gamma$ |
| $q_{\texttt{accept}} \alpha$ | $q_{\texttt{accept}}$ | |
| $q_{\texttt{accept}} \# \$$ | $\$$ | - |

**Remark.** In the above table we did not specify how we map "stay" moves of the machine. Actually, these can be simulated with two moves, one right move that goes to a new special state, and a left move going back to the previous head position. So, we assume we map the stay moves in this way in the table.

We try to give an intuition of the construction. Since, to match the two strings, we necessarily need to use row 1 first, we have that our string on the left is just # and on the right we already have the complete initial ID of $M$ with input $w$.

$$\begin{array}{cc}
\# & \# \\
 & q_1 \\
 & w_1 \\
 & . \\
 & . \\
 & . \\
 & w_n \\
 & \#
\end{array}$$

Now, if the string on A's side wants to "catch up" with B, because it is shorter, it needs to match the content of the initial ID. So, for example, we need to pick the row where on the first column we have the string $q_1 w_1$. If $\delta(q_1, w_1) = (q_2, a, R)$, then, using this row will also increase the length of the string on B's side, by essentially appending to the string on the right the result of applying the transition of $M$, when it is on state $q_1$ and reads the symbol $w_1$.

$$\begin{array}{cc}
\# & \# \\
{\color{red} q_1} & q_1 \\
{\color{red} w_1} & w_1 \\
 & . \\
 & . \\
 & . \\
 & w_n \\
 & \# \\
 & {\color{red} a} \\
 & {\color{red} q_2}
\end{array}$$

Then, all the remaining symbols up to the second occurrence of # will be matched one by one using the second and third row of our table (this will also add these symbols at the bottom of the string on B's side). So essentially, as soon as the string on A's side catches up with the initial ID, the string on B's side will now contain the ID that $M$ produces after the initial one.

$$\begin{array}{cc}
\# & \# \\
q_1 & q_1 \\
w_1 & w_1 \\
. & . \\
. & . \\
. & . \\
w_n & w_n \\
\# & \# \\
 & a \\
 & q_2 \\
 & . \\
 & . \\
 & . \\
 & w_n \\
 & \#
\end{array}$$

So, the string on A's side needs to catch up again, but then we append an additional ID to the string on B's side, and so on. The only way for the string on A's side to eventually catch up with the one on B's side, is to match an ID containing the accepting state (i.e., $M$ accepts $w$), as in that case, although the string on B might still be longer than the one on A, it eventually loses some symbol (for example when we match $\alpha q_{\texttt{accept}}$, we only add $q_{\texttt{accept}}$ to the string on B's side, losing one symbol). This "consumption of symbols" will eventually allow the string on A's to just need to match the accepting state "alone" between two #'s, i.e., $\#q_{\texttt{accept}}\#$. In this case, the string on A's side will first match # using the third row of the tabl,e, and then conclude using the last row, matching $q_{\texttt{accept}}\#\$$ which adds \$, on the right side.

$$\begin{array}{cc}
. & . \\
. & . \\
. & . \\
\# & \# \\
q_{\texttt{accept}} & q_{\texttt{accept}} \\
\# & \# \\
\$ & \$
\end{array}$$

On the other hand, if $M$ does not accept $w$, either $M$ loops or rejects. If $M$ loops, then the string on A's side will never be able to catch up with the one on B's side, as the length of the string on B will always be greater than the one on A. If $M$ rejects, then, at some point the string on B's will contain the symbol $q_{\texttt{reject}}$, but there is no way for the string on A's side to match that symbol, as $q_{\texttt{reject}}$ does not occur anywhere on the left column of our table. □

Since $\mathcal{L}_u \notin \mathrm{R}$, with the above theorem in place, we conclude that the modified PCP is undecidable, i.e., $\mathcal{L}_{MPCP} \notin \mathrm{R}$. However, we are not done yet. Our goal was to prove that the PCP is undecidable. So, we complete our proof by showing that we can reduce MPCP to PCP.

**Theorem 17.** $\mathcal{L}_{MPCP} \leq \mathcal{L}_{PCP}$.

*Proof.* Our reduction should do the following. Given a table of strings (in the form of two lists $A$ and $B$), construct a new table of strings ($A'$ and $B'$) such that $(A, B) \in \mathcal{L}_{\text{MPCP}}$ iff $(A', B') \in \mathcal{L}_{\text{PCP}}$. Before showing our reduction, we need to introduce an auxiliary notation. Let $w = w_1 \cdots w_n$ be some string. We define the notation

$$\begin{array}{rcl} \star w & = & *w_1 * w_2 \cdots * w_n \\ w\star & = & w_1 * w_2 * \cdots w_n * \\ \star w \star & = & *w_1 * w_2 * \cdots * w_n * \end{array}$$

That is, $\star w$ places the symbol * on the left of each symbol in $w$; $w\star$ places the symbol * on the right of each symbol in $w$; $\star w\star$ places the symbol * on both left and right of each symbol in $w$.

Our reduction does the following. Consider an input table of strings:

| A | B |
|---|---|
| $S_1$ | $T_1$ |
| $S_2$ | $T_2$ |
| . | . |
| . | . |
| . | . |
| $S_k$ | $T_k$ |

Our reduction converts the above table into the following one:

| A | B |
|---|---|
| $\star S_1 \star$ | $\star T_1$ |
| $S_1 \star$ | $\star T_1$ |
| $S_2 \star$ | $\star T_2$ |
| . | . |
| . | . |
| . | . |
| $S_k \star$ | $\star T_k$ |
| \$ | $*\$$ |

Assume we have a match in the original table, i.e., there are indices $i_1, \ldots, i_n$, with $i_1 = 1$, such that

$$S_{i_1} \cdots S_{i_n} = T_{i_1} \cdots T_{i_n}.$$

Then, the following

$$[\star S_{i_1} \star][S_{i_2} \star] \cdots [S_{i_n} \star][\$] = [\star T_{i_1}][\star T_{i_2}] \cdots [\star T_{i_n}][*\$]$$

is a match in the new table.[7]

[7] We use the symbols [ and ] just to logically separate the different strings, but these symbols do not appear in the match.

On the other hand, if there is a match in the new table, this must necessarily start from the first row, as is the only one where both strings start with the same symbol. That is, it is of the form:

$$[\star S_{i_1}\star][S_{i_2}\star]\cdots[S_{i_n}\star][\$] = [\star T_{i_1}][\star T_{i_2}]\cdots[\star T_{i_n}][*\$]$$

where $i_1 = 1$.

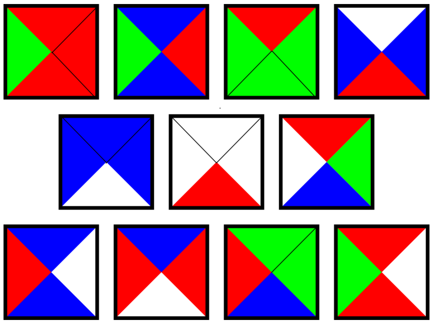
Moreover, note that any such a match, the symbols of the original table that appear on the left and the right strings are always separated by *exactly one* * symbol. So, the non-* symbols are kept aligned between the two strings. Thus if we remove the symbols * and \$, the match becomes a match of the original table.

**Remark.** Note that in the constructed table we write the first row of the original table twice, but in different ways. The first copy is to force any sequence to start from the first row, while the second one treats the first row as all other rows, and this is to allow sequences where the first row is used multiple times (not necessarily only as the first one). Moreover, we add a last row with the \$ symbol to let the right string catch up with the left one at the end of the match, since the left strings is always one symbol (the *) longer. □

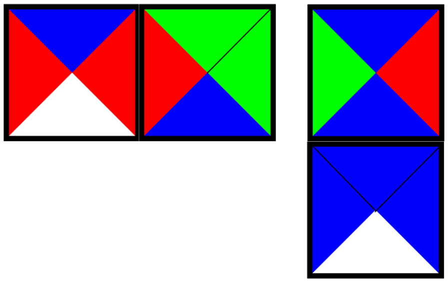
Hence, since $\mathcal{L}_{MPCP} \notin \text{R}$, and since we can reduce $\mathcal{L}_{MPCP}$ to $\mathcal{L}_{PCP}$, we conclude that $\mathcal{L}_{PCP} \notin \text{R}$.

## 8.2 The Tiling Problem

We now consider another puzzle problem that does not talk about TMs. We first introduce the problem informally. You are given a certain (finite) number of tile types, where each tile type is colored in some way on each of its four sides. For example, we might get the following 11 tile types.



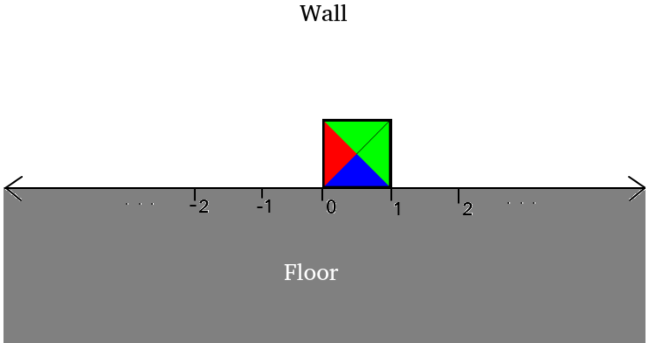
You have infinitely many tiles at your disposal, but only of the types given to you. Now, you can place two tiles one after the other (or one on top of the other), only if the border they have in common is of the same color. You are not allowed to rotate or mirror the tiles. For example, these are valid placements.



The goal is the following. Assume you are standing on the floor, and you are in front of a white wall, and this wall extends, starting from the floor, infinitely in all directions (left, right, and up). This is depicted in the picture below.

Assume also that some *initial* tile of one of the given types is already placed on the wall exactly in the middle of the wall, at floor level, i.e., the tile is at position (0,0). Then, we want to decide whether it is possible to cover the whole wall with tiles in such a way that

1. two adjacent tiles share the same color on their common border;
2. no "holes" are left on the wall;
3. You leave the initial tile where it is.



We now formally define the problem. First, we define what an instance is for such a problem.

**Definition 17.** *An instance of the Tiling problem is a pair of the following form*

$$(T, t_0),$$

*where $T$ is a finite set of quadruples (tile types) $t = (n, e, s, w)$ of colors, where $n$ is the color north of the tile type $t$, $e$ is the color east of it and so on. Finally, $t_0 \in T$ is the initial tile type.*

Now, we define what is a solution to the tiling problem; below we use $\mathbb{N}_0$ to denote the natural numbers including the 0, i.e., the set $\{0, 1, 2, 3, \ldots\}$.

**Definition 18.** *Given an instance $(T, t_0)$ of the Tiling problem, a* tiling *for $(T, t_0)$ is a function $f : \mathbb{Z} \times \mathbb{N}_0 \rightarrow T$ (assigning to each cell of the infinite wall a tile type), such that*

1. $f(0, 0) = t_0$*.*
2. *the east color of $f(i, j)$ coincides with the west color of $f(i + 1, j)$;*
3. *the north color of $f(i, j)$ coincides with the south color of $f(i, j + 1)$.*

Our language is then easily defined.

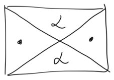
$$\mathcal{L}_{\text{Tiling}} = \{(T, t_0) \mid (T, t_0) \text{ is an instance of the tiling problem} \\ \text{for which a tiling exists}\}.$$

To prove the claim, we provide a reduction from $\overline{\text{HALT-}\epsilon}$. That is, the language of encodings of TMs $< M >$ such that $M$ *does not* halt when executed with no input (i.e., the tape is empty):

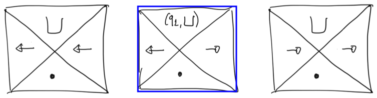
$$\overline{\text{HALT-}\epsilon} = \{M \mid M \text{ does not halt with the empty input}\}.$$

**Theorem 18.** $\overline{\mathit{HALT}\text{-}\epsilon} \leq \mathcal{L}_{\mathit{Tiling}}$.

*Proof.* Given a TM $M$, our reduction needs to construct a set of tile types $T$, and an initial tile type $t_0$ such that $M$ does not halt with the empty input iff $(T, t_0)$ has a tiling $f$. Let $q_1$ be the initial state of $M$, $\Gamma$ its tape alphabet and $\delta$ its transition function. From $M$, our reduction constructs the following tile types:

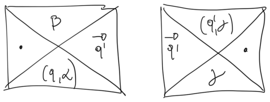


$\forall \alpha \in \Gamma$



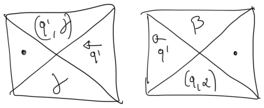
if $\delta(q,\alpha) = (q',\beta,R)$ , $\forall \gamma \in \Gamma$

$q\alpha\gamma \rightsquigarrow \beta q'\gamma$



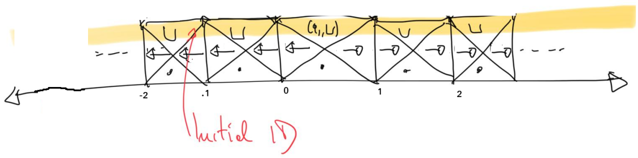
if $\delta(q,\alpha) = (q',\beta,L)$ $\forall \gamma \in \Gamma$

$\gamma q\alpha \rightsquigarrow q'\gamma\beta$



Also here, we assume we map stay moves with two moves which go right and then left. For each tape symbol, we have a tile type having that symbol both in the north and south as the color. On west and east it has the dummy color •. The tile highlighted in blue is the initial tile, and has a right and a left arrow on the east and west sides, respectively. We also have two other tile types having blank on the north and left (resp., right arrow) on both east and west; the south has the dummy color •.

These last two tile types are essentially needed to cover the first row of the wall with the initial ID of the TM $M$, as shown in the picture below.[8]

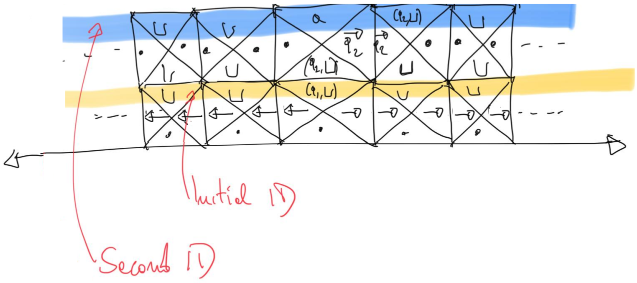


Then, the two tile types associated to transitions of the form $\delta(q,\alpha) = (q',\beta,R)$ need to allow us to cover the next row of the wall, for this, we need at the bottom of one of the two tiles, the state-symbol combination $(q,\alpha)$ to match it with the north face of a tile on the previous row. On the north of this tile we need to place the new symbol $\beta$, and on the east we specify that the head moves right to state $q'$. This is needed to match the other tile type (actually there are

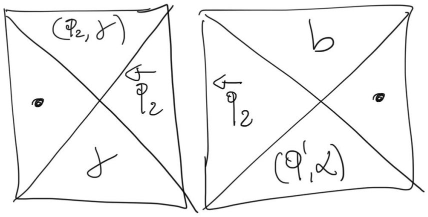
[8] Note that in the way we have chosen the west/east colors of these tiles, this is the only way to fill the first row of the wall.

many, one for each possible tape symbol $\gamma$). This second tile type matches the state $q'$ on its west. On the south it "asks" to the bottom tile what was the symbol in that position of the tape (say $\gamma$), and on its north specifies that in the new ID we have state $q'$ over the symbol $\gamma$.

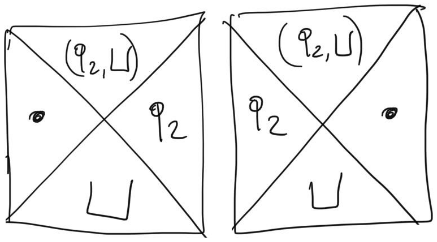
Similar tile types are constructed for left moves. So, if for example there is a transition $\delta(q_1, \sqcup) = (q_2, a, R)$, then, with the wall covered as shown in the previous picture, the next row of the wall will be covered as follows:



**Remark.** Note that having the color $\vec{q_2}$ on the two tiles above, rather than just $q_2$ is important. In fact, using the arrow, we guarantee that the tile applying the transition can only be connected on the right with its paired tile. In fact, assume we also have a transition of the form $\delta(q', \alpha) = (q_2, b, L)$, for some symbols $\alpha, b$. The corresponding tiles should be:

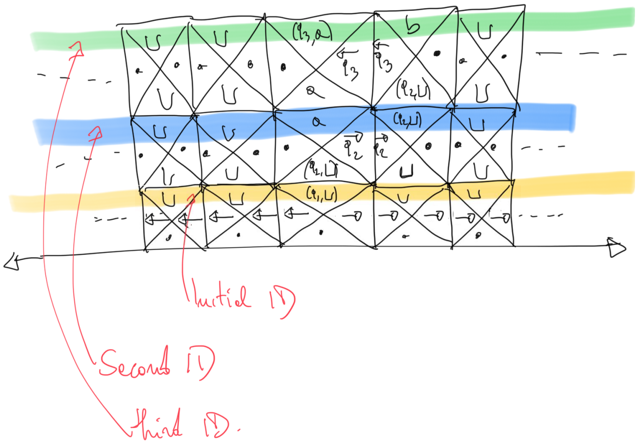


However, if we don't use the arrows, we allow the two tiles below (one coming from $\delta(q_1, \sqcup) = (q_2, a, R)$, and the other from $\delta(q', \alpha) = (q_2, b, L)$) to be placed one after the other:



This would allow us to place two states on the same row, which clearly does not represent a valid ID of our TM.

We conclude the discussion with an additional example showing how to fill one more row. If, for example, we have the transition $\delta(q_2, \sqcup) = (q_3, b, L)$, then we can use the tile types for left moves, and the first tile types, to cover the third row of the wall as follows:



Thus, if the TM $M$ does not halt with empty input, there are infinitely many IDs $M$ visits, and thus we can fully cover the wall. If $M$ halts with empty input, there will be only a finite number of IDs the machine can visit, and thus we can only cover a finite number of rows of the wall, and leave the rest uncovered. □

From the theorem above, and from the fact that $\overline{\text{HALT-}\epsilon} \notin \text{RE}$, we conclude that $\mathcal{L}_{\text{Tiling}} \notin \text{RE}$.

# 9 Rice's Theorem

During the previous lectures, quite a good amount of the languages we have seen that talk about TMs are undecidable, i.e., they are not in R. All such languages essentially encode decision problems that ask something about TMs, in particular, ask whether the *language* of a given TM enjoys some *property.* For example, the diagonal language asks "Given a TM $M$, is it the case that the language accepted by $M$ does not contain the string $w_M$ encoding $M$?" As another example, the language $\mathcal{L}_e$ asks "Given a TM $M$, is it the case that the language accepted by $M$ is empty?" Or still, $\mathcal{L}_{ne}$ asks "Given a TM $M$, is it the case that the language accepted by $M$ is non-empty?"

**Properties.** So, it seems that asking whether the language of a given TM has some property always turns out to be undecidable. But is this always the case? Or are there properties of the languages of TMs that we can actually decide?

This question is answered by a fundamental result in Computability Theory known as Rice's Theorem: it tells us precisely which properties about the language of TMs cannot be decided.

Let us first define formally the notion of property.

**Definition 19.** *A* property $\mathcal{P}$ *is a language of TMs (i.e., a set of encodings of TMs).*

So, intuitively, a property $\mathcal{P}$ collects all TMs that have something in common, and given a TM $M$, checking if $M \in \mathcal{P}$ means checking if $M$ has the property that all machines in $\mathcal{P}$ have in common.

For example, if we consider the property

$$\mathcal{P}_1 = \{< M >| \ L(M) \text{ contains only strings of even length}\}.$$

The property $\mathcal{P}_1$ essentially collects all TMs that only accept strings of even length. So, $\mathcal{P}_1$ encodes the decision problem asking:

"Given a TM $M$, is it the case that $M$ accepts only strings of even length?"

Another property could be

$$\mathcal{P}_2 = \{< M >| \ M \text{ has exactly 5 states}\},$$

which encodes the decision problem asking:

"Given a TM $M$, is it the case that $M$ has exactly 5 states?"

Among all properties that one might want to check about TMs, there are two properties that are *trivial.* That is, they are essentially not interesting to ask.

**Definition 20.** *A property $\mathcal{P}$ is* trivial *if it is either empty, or it is the set of the encodings of all possible TMs.*

Let us see why these properties are trivial. Consider the case that $\mathcal{P}$ is the set of all TMs. Then, the language $\mathcal{P}$ encodes the decision problem asking

"Given a TM $M$, is it the case that $M$ is a TM?"

The above question is trivial, because the answer to the question is always "yes", no matter the input TM, because obviously a TM $M$ is a TM.

Assume now that $\mathcal{P} = \emptyset$. Then, the language $\mathcal{P}$ encodes the decision problem that essentially asks

"Given a TM $M$, is it the case that $M$ is not a TM?"

Also the above question is trivial, as the answer is always "no", no matter the input TM, because a TM $M$ *is* a TM.

Another criterion with which we classify properties is on the basis whether they ask something about the the *language* of a TM or not. For example, the property $\mathcal{P}_1$ is only concerned on the language of TMs, i.e., *what* the TMs do, rather than *how* they do something. We call such properties *semantic*, and give a formal definition in a second. Other properties are not concerned about the language of a TM, but on its inner workings, like $\mathcal{P}_2$. These are not semantic properties.

**Definition 21.** *A property $\mathcal{P}$ is* semantic *if whenever two TMs $M_1$ and $M_2$ accept the same language, i.e., $L(M_1) = L(M_2)$, then, either both $M_1, M_2$ have the property, i.e., $<M_1>, <M_2> \in \mathcal{P}$, or none has, i.e., $<M_1>, <M_2> \notin \mathcal{P}$.*

So, the above definition says that a semantic property must be only concerned about the language of a TM. Thus, if two machines accept the same language, it is not possible that one has the property, and the other does not, as if this was the case, it means that *despite having the same language*, the property is "selecting" one of the two machines but not the other (i.e., it is not selecting a TM based off its language).

For example, the property $\mathcal{P}_1$ is semantic, because if we pick two TMs $M_1, M_2$ accepting the same language, two things can happen. The language $\mathcal{L} = L(M_1) = L(M_2)$ they both accept contains only even-length strings, in which case $< M_1 >, < M_2 > \in \mathcal{P}_1$. On the other hand, if $\mathcal{L}$ also contains odd-length strings, then $< M_1 >, < M_2 > \notin \mathcal{P}_1$. So, it is not possible that for two TMs accepting the same language, one has the property $\mathcal{P}_1$ and the other doesn't.

Considering property $\mathcal{P}_2$ instead, if we pick a TM $M_1$ having exactly 5 states, we can easily build a TM $M_2$ having more states that accepts the same language as $M_1$ (e.g., simply make $M_2$ the same as $M_1$ but with one more dummy, unreachable state $q'$ the loops on itself). So, although $M_1, M_2$ accept the same language, $< M_1 > \in \mathcal{P}_2$, but $< M_2 > \notin \mathcal{P}_2$. Hence, property $\mathcal{P}_2$ is not semantic.

**Rice's Theorem.** So, besides trivial properties, that are trivially decidable, what about the other properties? Are there properties $\mathcal{P}$ for which, given a TM $M$, we can *decide* whether $M$ has the property $\mathcal{P}$ (i.e., $M \in \mathcal{P}$)?

It turns out that checking whether a TM has *any* (non-trivial) semantic property is undecidable.

**Theorem 19** (Rice's Theorem)**.** *Every non-trivial, semantic property $\mathcal{P}$ is undecidable, i.e., $\mathcal{P} \notin R$.*

*Proof.* Let $\mathcal{P}$ be a non-trivial and semantic property. To prove the claim, we first further assume that there is no TM in $\mathcal{P}$ that accepts the empty language, i.e, for each $M \in \mathcal{P}$, $L(M) \neq \emptyset$.

Under this assumption, we provide a reduction from the universal language $\mathcal{L}_u$ to $\mathcal{P}$. Before starting, let us make an observation about $\mathcal{P}$. Since $\mathcal{P}$ is non-trivial (and thus not empty), and since we assumed all its TMs accept a non-empty language, we can pick some machine $M_{\mathcal{L}}$ from $\mathcal{P}$ accepting some non-empty language $\mathcal{L}$.

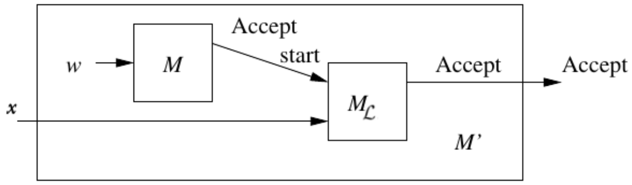
We now proceed with our reduction. To show that $\mathcal{L}_u \leq \mathcal{P}$, we need to provide an algorithm that converts yes-instances of $\mathcal{L}_u$ to yes-instances of $\mathcal{P}$, and no-instances to no-instances.

Recall that an instance of $\mathcal{L}_u$ is a pair $(M, w)$ of a TM $M$ and a string $w$, and an instance of $\mathcal{P}$ is just a TM. So, our reduction must be a mapping:

$$(\mathbf{M},\mathbf{w}) \overset{\mathbf{T}}{\rightsquigarrow} <\mathbf{M'}>$$

such that, if $(M, w) \in \mathcal{L}_u$, i.e. $M$ accepts $w$, then $<M'> \in \mathcal{P}$, and if $(M, w) \notin \mathcal{L}_u$, i.e. $M$ does not accept $w$, then $<M'> \notin \mathcal{P}$.

Our reduction $T$, given a pair $(M, w)$, constructs the following TM $M'$:



What $M'$ intuitively does is to first verify that $M$ accepts $w$ (note that $w$ is not the input to $M'$, rather $M'$ writes it down on a secondary tape as soon as it starts. In case $M$ accepts $w$, then $M'$ essentially "becomes" $M_{\mathcal{L}}$. That is, with a given string $x$ to its input, $M'$ simply executes the control of $M_{\mathcal{L}}$ with input $x$.

Let us see why $T$ is a reduction.

Assume $(M, w)$ is such that $M$ accepts $w$. Then, $M'$ will take its input string $x$ and accept it iff $x \in \mathcal{L}$, i.e., $L(M') = L(M_{\mathcal{L}}) = \mathcal{L}$. Since $\mathcal{P}$ is semantic,

$M_{\mathcal{L}}$ belongs to $\mathcal{P}$, and $M'$ accepts the same language as $M_{\mathcal{L}}$, we conclude that also $<M'>\in \mathcal{P}$.

Assume instead that $(M, w)$ is such that $M$ does not accept $w$. Then, $M'$ will never execute the control of $M_{\mathcal{L}}$, and thus, whatever string $x$ the TM $M'$ takes as input, $M'$ will not accept it, i.e., $M'$ accepts the empty language. Since we assumed that TMs accepting the empty language do not belong to $\mathcal{P}$, we conclude that $<M'>\notin \mathcal{P}$.

The proof is not done yet, as the proof above has only shown that $\mathcal{P}$ is undecidable only when $\mathcal{P}$ does not contain TMs accepting the empty language. What about non-trivial, semantic properties having such TMs?

Assume $\mathcal{P}$ is a non-trivial, semantic property having some TMs accepting the empty language i.e., $\exists <M>\in \mathcal{P}$ such that $L(M) = \emptyset$. Note that since $\mathcal{P}$ is semantic, if even one TM accepting the empty language belongs to $\mathcal{P}$, then *every* TM accepting the empty language is in $\mathcal{P}$. Now, let us consider the property $\bar{\mathcal{P}}$ i.e., the complement of $\mathcal{P}$.

$\bar{\mathcal{P}}$ is still non-trivial, because $\mathcal{P}$ is non-trivial, and thus some TMs that are not in $\mathcal{P}$ now belong to $\bar{\mathcal{P}}$, and some TMs that are in $\mathcal{P}$, cannot belong to $\bar{\mathcal{P}}$.

Also, $\bar{\mathcal{P}}$ is semantic. To see why, assume, towards a contradiction, that $\bar{\mathcal{P}}$ is not semantic, thus there must be two TMs $M_1, M_2$ such that $L(M_1) = L(M_1)$, but one is in $\bar{\mathcal{P}}$, e.g., $< M_1 >\in \bar{\mathcal{P}}$, and the other is not, e.g. $< M_2 >\notin \bar{\mathcal{P}}$. The latter means that $< M_2 >\in \mathcal{P}$ (since $\bar{\mathcal{P}}$ is the complement of $\mathcal{P}$). However, since $\mathcal{P}$ is semantic, if $< M_2 >\in \mathcal{P}$, then also $< M_1 >$ must be in $\mathcal{P}$, since they both accept the same language. However, we said $< M_1 >$ is in $\bar{\mathcal{P}}$, and obtain a contradiction.

Finally, *no TM accepting the empty language* belongs to $\bar{\mathcal{P}}$, because all such machines were in $\mathcal{P}$, as discussed before.

So, $\bar{\mathcal{P}}$ is a non-trivial, semantic property with no TMs accepting the empty language. By the first part of our proof, we know that $\bar{\mathcal{P}}$ is undecidable. But this implies that $\mathcal{P}$ cannot be in R, because this would imply that its complement $\bar{\mathcal{P}}$ is in R (Theorem 6). This completes our proof. □

**Examples.** The above theorem is a powerful tool to study the undecidability of many languages of TMs. As an example, consider again the problem asking

"Given a TM $M$, is it the case that $M$ accepts only strings of even length?"

As we discussed, we can reformulate the above decision problem as a property $\mathcal{P}$ (i.e., a language of TMs), where

$$\mathcal{P} = \{< M >\mid L(M) \text{ contains only strings of even length}\}.$$

To prove that $\mathcal{P}$ is undecidable, it is enough to prove that $\mathcal{P}$ is non-trivial, and semantic. For showing non-triviality, we need to show that there is at least a TM in $\mathcal{P}$ (i.e., $\mathcal{P}$ is not empty), but there is also a TM that is *not* in $\mathcal{P}$.

Clearly, at least one TM belongs to $\mathcal{P}$, e.g. the TM that accepts the simple language made of a single string of even length $\{00\}$, and there are TMs not in

$\mathcal{P}$, like the one accepting the language $\{0, 1, 00, 111\}$, which contains odd length strings. Thus, $\mathcal{P}$ is not trivial.

Now, we need to show that $\mathcal{P}$ is semantic. Thus, we must show that whatever pair of TMs $M_1, M_2$ we consider that accept the same language, i.e., $L(M_1) = L(M_2)$, they are either both in $\mathcal{P}$, or none of them is in $\mathcal{P}$.

**Remark.** Another equivalent way to state that a property is semantic is that for every two arbitraty TMs $M_1, M_2$ that accept the same language, i.e., $L(M_1) = L(M_2)$, if $< M_1 >$ is in the property $\mathcal{P}$, then also $< M_2 >$ is in $\mathcal{P}$.

Using the above definition of semantic property, consider two arbitrary TMs $M_1, M_2$ with $L(M_1) = L(M_2)$. Assume $< M_1 > \in \mathcal{P}$. Then, by definition of $\mathcal{P}$, $L(M_1)$ contains only strings of even length. Since $L(M_1) = L(M_2)$, also $L(M_2)$ contains only strings of even length, and thus, by definition of $\mathcal{P}$, $< M_2 >$ is in $\mathcal{P}$. Hence, $\mathcal{P}$ is semantic. Since $\mathcal{P}$ is non-trivial and semantic, by Rice's Theorem $\mathcal{P}$ is undecidable (i.e., $\mathcal{P} \notin \mathrm{R}$).

The latter means that the problem of checking whether a TM only accepts strings of even length is undecidable.

Let us consider other example decision problems about TMs.

1. Given a TM $M$, does $M$ accept a finite language?
2. Given a TM $M$, does $M$ accept an infinite language?
3. Given a TM $M$, does $M$ accept a language accepted only by TMs having 5 states?

Consider the first problem. Let us write it down as a property of TMs, first.

$$\mathcal{P}_1 = \{< M >| \ L(M) \text{ is finite}\}.$$

Now, to see if it is undecidable, we just need to prove that $\mathcal{P}$ is non-trivial and semantic.

$\mathcal{P}_1$ contains at least one TM, because any finite language can be easily accepted by a TM that compares the input string with each of the (finitely many) strings of the language. Moreover, there are TMs not in $\mathcal{P}$, i.e., the ones accepting an infinite language, like $\{0^n \mid n > 0\}$. So, $\mathcal{P}_1$ is non-trivial.

$\mathcal{P}_1$ is semantic. Indeed, consider two arbitrary TMs $M_1, M_2$ with $L(M_1) = L(M_2)$, and assume $< M_1 > \in \mathcal{P}_1$. By definition of $\mathcal{P}_1$, $L(M_1)$ is finite. Since $L(M_1) = L(M_2)$, also $L(M_2)$ is finite, and thus $< M_2 > \in \mathcal{P}_1$. Hence, $\mathcal{P}_1$ is semantic. Thus, by Rice's Theorem, $\mathcal{P}_1$ is undecidable.

Consider the second problem. So,

$$\mathcal{P}_2 = \{< M >| \ L(M) \text{ is infinite}\}.$$

Clearly, as discussed above, there are TMs that accept infinite languages, and thus $\mathcal{P}_2$ is not empty, and there are TMs accepting finite languages, and thus no all TMs are in $\mathcal{P}_2$. Thus, $\mathcal{P}_2$ is non-trivial. With an identical reasoning to the one used for $\mathcal{P}_2$, we also conclude that $\mathcal{P}_2$ is semantic (indeed, it is clear that $\mathcal{P}_2$ "selects" TMs only based off the language they accept). Thus, $\mathcal{P}_2$ is undecidable.

Consider the third problem. So,

$$\mathcal{P}_3 = \{< M >| \ L(M) \text{ is accepted only by TMs with 5 states}\}.$$

We now show that $\mathcal{P}_3$ is trivial. In fact, we show that $\mathcal{P}_3 = \emptyset$.

To prove it, it is enough to show that *every* language that is accepted by a TM with 5 states, is also accepted by a TM with more states. Indeed, if a language $\mathcal{L}$ is accepted by a TM with 5 states, we can just build another TM that has one more state, which we regard as the new accepting state, and we add a transition from the old accepting state to the new one. This machine still accepts $\mathcal{L}$ and has more than 5 states.

So, no TM $M$ exists accepting a language $L(M)$ accepted *only* by TMs with 5 states, as there always exists a TM with more states accepting it. Thus, $\mathcal{P}_3 = \emptyset$ is trivial. Since trivial properties are decidable, we conclude that $\mathcal{P}_3$ is decidable.

**Remark.** As a last example, to remark that what is *necessarily* undecidable about TMs are properties of their languages, whereas properties regarding something else might or might not be undecidable, consider the following property:

$$\mathcal{P}_4 = \{< M >| \ M \text{ has an even number of states}\}.$$

The above property is clearly non-trivial, as we can easily construct a TM with an even number states, and there definitely are TMs with an odd number of states. However, $\mathcal{P}_4$ is not semantic. In fact, if we consider two arbitrary TMs $M_1, M_2$ with $L(M_1) = L(M_2)$, assuming that $< M_1 >\in \mathcal{P}_4$ (i.e., $M_1$ has an even number of states), can we conclude that also $< M_2 >$ is in $\mathcal{P}_4$ (i.e., $M_2$ has an even number of states)? No, because even if $M_1$ and $M_2$ accept the same language, and $M_1$ has an even number of states, $M_2$ does not need to use an even number of states as well.

For example, if we consider a TM $M_1$ with an even number of states, we can build another TM $M_2$ which is almost identical to $M_1$: the only difference is that we add a dummy state $q'$ which is not reachable by any other state of the TM, and $q'$ loops on itself. Thus, both $M_1$ and $M_2$ accept the same language, but $M_1$ has an even number of states, while $M_2$ has an odd number of states. Thus $< M_1 >\in \mathcal{P}_4$, whereas $< M_2 >\notin \mathcal{P}_4$. Thus, Rice's Theorem is not applicable. Note that this *does not mean* that $\mathcal{P}_4$ is decidable. It only means that we need to analyse the language $\mathcal{P}_4$ in some other way.

The above language is decidable, because checking whether a TM $M$ has an even number of states simply requires scanning its encoding $< M >$ and count the number of states.

# 10 More exercises on TMs and undecidability

We start by considering the following language:

$$\begin{array}{rcll}\mathcal{L} & = & \{A\#B\#W \mid & A, B, W \in \{0,1\}^* \text{ and} \\ & & & |B| < 2 \cdot |A| \text{ and } |W| > 2 \cdot |A| - |B| \text{ and} \\ & & & B \text{ is a substring of } W \text{ and } A^R \text{ is a substring of } W\}.\end{array}$$

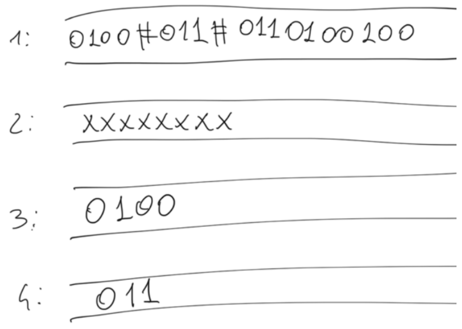
As an example, a string belonging to the above language is:

$$\underbrace{0100}_{A}\#\underbrace{011}_{B}\#\underbrace{0110100100}_{W}.$$

Indeed, $|B| = 3 < 2 \cdot |A| = 8$, and $|W| = 10 > 2 \cdot |A| - |B| = 5$. Moreover, $B$ is a substring of $W$, as it occurs at the beginning of $W$, and the reverse of $A$, i.e., 0010 occurs in $W$ as well, starting from the 6th symbol in $W$.

**Question.** Devise a TM that decides the above language.

We first give an high level description of how our TM looks like, and what are its main steps. We use 4 tapes, as shown below:



Besides the first tape, that contains the input string, the machine writes, in tape 2, two X's for each symbol in $A$, hence the number of X's will be exactly twice as $A$'s length. Then, in tape 3, our machine will write a copy of string $A$, and in tape 4, a copy of string $B$.

The high level steps of our machine can be summarized as follows:

1. Write two X's, for each symbol in $A$, on tape 2.
   - While doing this, also copy each symbol of $A$ in tape 3.
2. Check that $|B| < 2 \cdot |A|$, by erasing one X from tape 2, for each symbol read from $B$ on the first tape. If some X remains on tape 2, after $B$ has been completelly scanned, then move to the next step, otherwise reject.
   - While scanning B, also copy each symbol of $B$ in tape 4.

3. Note that now, tape 2 contains exactly $2 \cdot |A| - |B|$ X's. Check that $|W| > 2 \cdot |A| - |B|$ by erasing one X from tape 2, for each symbol read from $W$ on the first tape. If no X's remain on tape 2, while we can still read a 0 or 1 from $W$, then move to the next step, otherwise reject.

4. Check if the rest of $W$ contains only 0 and 1 (i.e., it does not contain the # symbol). If yes, move to the next step, otherwise reject.

5. Check if $B$ is a substring of $W$. In particular,

    (a) If $B$ is empty, $B$ is trivially a substring of $W$, so move to step 6.

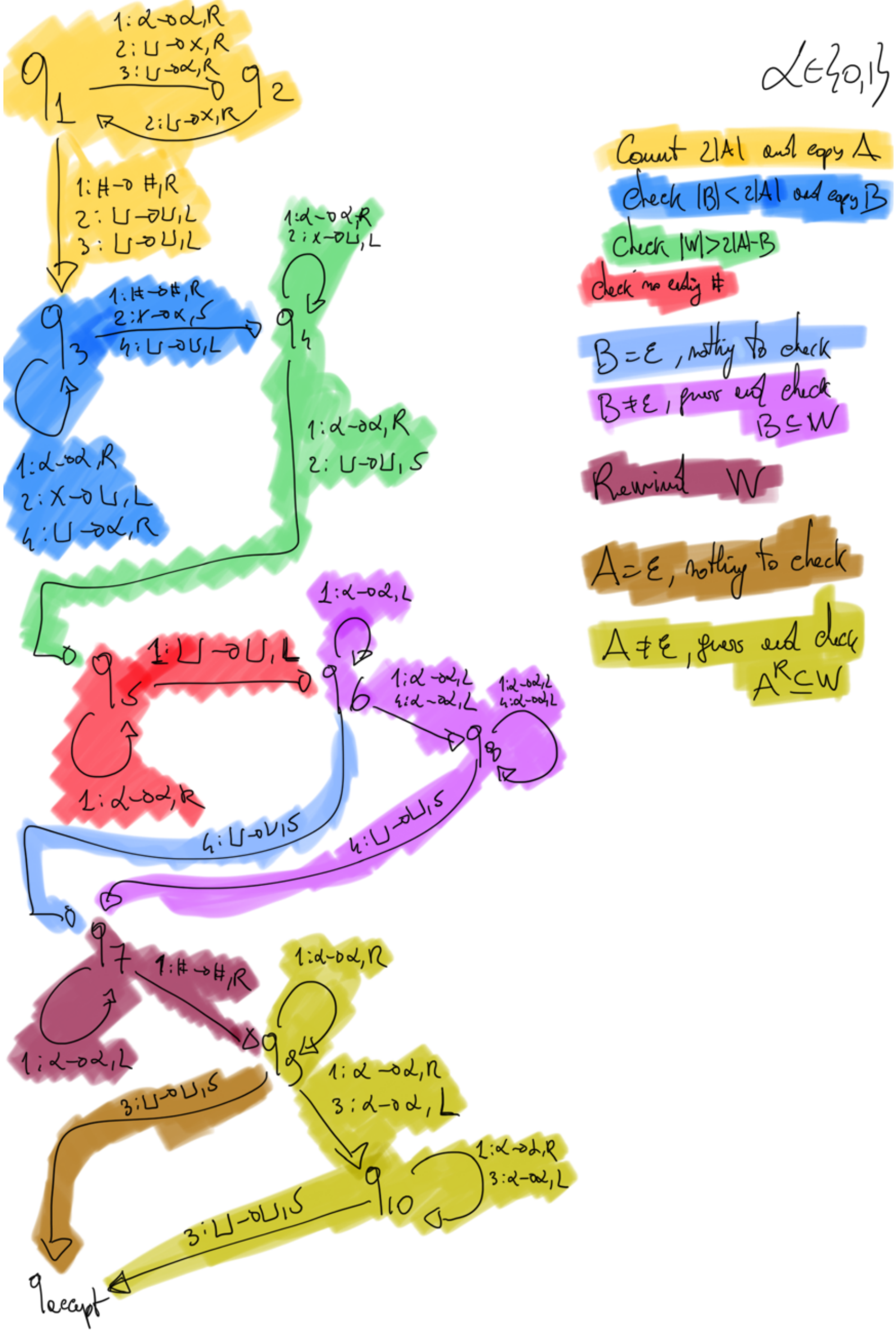
    (b) If $B$ is not empty, guess a position in $W$ and check that the substring of $W$ starting at that position coincides with $B$. If yes, move to step 6, otherwise reject.

6. Check if $A^R$ is a substring of $W$. In particular,

    (a) If $A$ is empty, $A^R$ is trivially a substring of $W$, so accept.

    (b) If $A$ is not empty, guess a position in $W$ and check that the substring of $W$ starting at that position coincides with $A^R$. If yes, accept, otherwise reject.

We now implement each of the above steps in our TM. In particular, each part of the TM implementing a certain step is highlighted with a different color. On the right of the TM you can see a small legend, describing which color corresponds to which step.


α∈{0,1}
Count 2|A| and copy A
check |B|<2|A| and copy B
Check |W|>2|A|-B
check no ending #
B=ε, nothing to check
B≠ε, guess and check B⊆W
Rewind W
A=ε, nothing to check
A≠ε, guess and check A^R⊆W
q1
q2
1: α → α, R
2: ⊔ → x, R
3: ⊔ → α, R
2: ⊔ → x, R
1: # → #, R
2: ⊔ → ⊔, L
3: ⊔ → ⊔, L
q3
1: α → α, R
2: x → ⊔, L
4: ⊔ → α, R
1: # → #, R
2: x → α, S
4: ⊔ → ⊔, L
q4
1: α → α, R
2: x → ⊔, L
1: α → α, R
2: ⊔ → ⊔, S
q5
1: α → α, R
1: ⊔ → ⊔, L
q6
1: α → α, L
1: α → α, L
4: α → α, L
q8
1: α → α, L
4: α → α, L
4: ⊔ → ⊔, S
4: ⊔ → ⊔, S
q7
1: α → α, L
1: # → #, R
q9
1: α → α, R
1: α → α, R
3: α → α, L
3: ⊔ → ⊔, S
q10
1: α → α, R
3: α → α, L
3: ⊔ → ⊔, S
qaccept

Consider now the language $\mathcal{L}$ we have discussed above. Discuss the decidability/undecidability status of the following properties.

$$\begin{array}{rcll}
\mathcal{P}_1 & = & \{< M > & \mid \ M \text{ accepts } \mathcal{L}\} \\
\mathcal{P}_2 & = & \{< M > & \mid \ M \text{ accepts } \mathcal{L} \text{ and} \\
 & & & \quad\ \text{each string in } \mathcal{L} \text{ is accepted in less than 100 steps}\} \\
\mathcal{P}_3 & = & \{< M > & \mid \ M \text{ does not accept } \mathcal{L} \cap \{0,1,\#\}^{100}\}
\end{array}$$

**$\mathcal{P}_1$.** This is a language over TMs, so it is a property of TMs. We can first try to see if this property is non-trivial and semantic, in which case Rice's Theorem will allow us to conclude that $\mathcal{P}_1$ is not in R. $\mathcal{P}_1$ is non-trivial because it is not empty, as there is at least a TM that accepts $\mathcal{L}$ (we just devised one that even decides $\mathcal{L}$), and there clearly are TMs not belonging to $\mathcal{P}_1$, e.g., pick any TM accepting a language different than $\mathcal{L}$.

For $\mathcal{P}_1$ to be semantic, we must show that if we consider two arbitrary TMs $M_1, M_2$ with $L(M_1) = L(M_2)$, if $< M_1 >$ is in $\mathcal{P}_1$, then $< M_2 >$ is in $\mathcal{P}_1$.

So, let $M_1, M_2$ be arbitrary TMs accepting the same language, i.e., $L(M_1) = L(M_2)$. If $< M_1 >$ is in $\mathcal{P}_1$, then it means that $M_1$ accepts $\mathcal{L}$, i.e., $L(M_1) = \mathcal{L}$. Since $L(M_1) = L(M_2)$, then also $L(M_2) = \mathcal{L}$. Hence, $M_2$ accepts $\mathcal{L}$, which means $< M_2 > \in \mathcal{P}_1$. So, $\mathcal{P}_1$ is semantic, and by Rice's Theorem, $\mathcal{P}_1$ is undecidable.

**$\mathcal{P}_2$.** Also $\mathcal{P}_2$ is a language over TMs, i.e., it is a property. Let's see if it is non-trivial and semantic. $\mathcal{P}_2$ is trivial. In particular, in this case we have that $\mathcal{P}_2 = \emptyset$. This is because no TM that accepts $\mathcal{L}$ can accept each string in $\mathcal{L}$ in less than 100 steps. For example, consider the string $0^{100}\#1^{100}\#0^{100}1^{100}$. You can easily verify the string belongs to $\mathcal{L}$. There is no way, however for a TM to recognize that this string is in $\mathcal{L}$ in only 100 steps, as the machine will not be able to scan the whole string in 100 steps, to realize, for example, that the $B$ part is shorter than twice the length of the $A$ part. Since $\mathcal{P}_2$ is trivial, $\mathcal{P}_2$ is decidable, as the TM that always rejects decides $\mathcal{P}_2$.

**$\mathcal{P}_3$.** This is also a language of TMs. Let's see if it is non-trivial and semantic. $\mathcal{P}_3$ collects all TMs that do not accept any string of exactly length 100, that appear in $\mathcal{L}$. The property is not trivial, there definitely are TMs in $\mathcal{P}_3$, as these are the ones that accept any language different than $\mathcal{L} \cap \{0,1,\#\}^{100}$ (pick any of the ones we have seen in our exercises), and there is at least one TM $M$ not in $\mathcal{P}_3$, i.e., $M$ accepts $\mathcal{L} \cap \{0,1,\#\}^{100}$. There are many ways to prove this. One way is observe that $\mathcal{L} \cap \{0,1,\#\}^{100}$ is a finite language, and every finite language is decidable, and thus can be accepted by some TM. Another way is to pick the TM $M$ we built for $\mathcal{L}$ at the beginning of these notes, and before starting, it first checks that its input is of length 100. If it is not, rejects, otherwise, it executes normally. Thus, $\mathcal{P}_3$ is non-trivial.

Let's see if $\mathcal{P}_3$ is semantic. Intuitively, this is the case because it refers to the language of the TMs therein, but let us prove it formally. Consider two arbitrary

TMs $M_1, M_2$ accepting the same language $L(M_1) = L(M_2)$. If $< M_1 >\in \mathcal{P}_3$, then $M_1$ does not accept $\mathcal{L} \cap \{0,1,\#\}^{100}$, i.e., $L(M_1) \neq \mathcal{L} \cap \{0,1,\#\}^{100}$. Since $L(M_1) = L(M_2)$, also $L(M_2) \neq \mathcal{L} \cap \{0,1,\#\}^{100}$, and thus $< M_2 >$ is in $\mathcal{P}_3$. Thus, $\mathcal{P}_3$ is semantic. Since $\mathcal{P}_3$ is both non-trivial and semantic, by Rice's Theorem, $\mathcal{P}_3$ is undecidable.

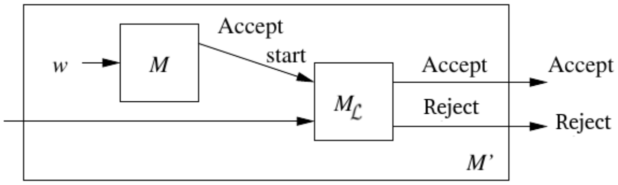
**Another example.** We consider one last example.

$$\mathcal{P}_4 = \{< M >\mid M \textit{ decides } \mathcal{L}\}.$$

This property might seem identical to $\mathcal{P}_1$ but they are not! $\mathcal{P}_1$ is asking whether a TM *accepts* $\mathcal{L}$, whereas $\mathcal{P}_4$ asks whether a TM *decides* $\mathcal{L}$. This can make a difference. $\mathcal{P}_4$ is still non-trivial, as we know that there is a TM deciding $\mathcal{L}$ (the one we devised at the beginning of these notes), and there of course are other TMs, that do not decide $\mathcal{L}$, e.g., they decide different languages than $\mathcal{L}$. However, as strange it might seem, the property is *not* semantic.

To show that $\mathcal{P}_4$ is not semantic, we need to show that there are two TMs $M_1, M_2$ accepting the same language, such that $<M_1>\in \mathcal{P}_4$, but $<M_2>\notin \mathcal{P}_4$. Let $M_1$ be the TM we devised before to decide $\mathcal{L}$. This, by definition, belongs to $\mathcal{P}_4$. Now, consider the TM $M_2$ that is indentical to $M_1$, with the only difference that every transition of $M_1$ moving to the rejecting state is replaced with a transition to a special state $q'$, on which $M_2$ then loops. Clearly, $M_2$ accepts $\mathcal{L}$, and thus $L(M_1) = L(M_2)$, but it *does not decide* $\mathcal{L}$, because, whenever the input string is not in $\mathcal{L}$, $M_2$ does not reject: it loops instead. Hence, $<M_2>\notin \mathcal{P}_4$. Thus, although $M_1$ and $M_2$ accept the same language, one of them is in $\mathcal{P}_4$, and the other is not, which implies that $\mathcal{P}_4$ is not semantic.

So, if $\mathcal{P}_4$ is undecidable, we need to prove it without Rice's Theorem, e.g., with a reduction. In the following, let us call $M_{\mathcal{L}}$ the TM that *decides* $\mathcal{L}$ (the one we implemented, for example). We reduce the universal language $\mathcal{L}_u$ to $\mathcal{P}_4$. Our reduction takes as input a pair $(M, w)$ and outputs a TM $M'$ defined as follows:



If $M$ accepts $w$, then $M'$ will execute the control of $M_{\mathcal{L}}$, and thus decide the language $\mathcal{L}$, and thus $<M'>\in \mathcal{P}_4$. If $M$ does not accept $w$, $M'$ will never execute the code of $M_{\mathcal{L}}$, and thus $M'$ rejects every string, i.e., $M'$ decides the empty language $\emptyset$. However, $\mathcal{L} \neq \emptyset$, and thus $<M'>\notin \mathcal{P}_4$.

We conclude that $\mathcal{P}_4$ is undecidable.

# 11 Introduction to Computational Complexity

By now, you should have a quite good understanding of the concepts of decidability and undecidability. We have shown that there exists a good number of languages that are undecidable, and this happens at different levels. Moreover, Rice's Theorem has shown us that actually almost all questions regarding the language of TMs is inevitably undecidable. But there also exists a plethora of problems that are indeed decidable, and thus can be effectively solvable by a TM.

However, everybody will agree that not all problems that are solvable are equally difficult to solve. It is reasonable to believe that some decidable problems require more resources (in time and memory) than other decidable problems. Our current classification of languages, however, does not take these parameters into account.

We then define some new classes of decidable languages, called "Complexity classes" that will help us better understand the resources that are needed by a TM in terms of time and memory in order to decide them.

For this, recall that for TM $M$, $T_M(n)$ denotes the time required by $M$ with inputs of length $n$.

## 11.1 The Complexity class P

Let us first setup some notation.[9]

**Definition 22.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a function. We define*

$$\text{DTIME}(f(n)) = \{\mathcal{L} \mid \exists \textit{ a TM } M \textit{ that decides } \mathcal{L} \textit{ and } T_M(n) \in O(f(n))\}.$$

So, essentially DTIME($f(n)$) collects all languages that can be decided by a *D*eterministic TM that always requires a number of steps that is at most of the order of $f(n)$, where $n$ is the size of its input.

With the above definition at hand, we can now define our first complexity class.

**Definition 23.** *The class of* polynomial-time languages *is defined as*

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

So, **P** collects all languages that can be decided by a TM in at most a polynomial number of steps (and the polynomial can be of any fixed degree).

**Remark.** Note that we could have considered multi-tape, or multi-track TMs as well, and the class **P** would remain the same, since they can all be simulated by a single-tape, single-track TM with only a polynomial-time overhead.

[9] Recall that we focus on languages over the binary alphabet.

We usually regard problems described by languages in **P** as *tractable* (or feasible) problems. That is, the time required to provide the right answer "only" increases polynomially, as the size of the input increases.

One might argue, however, that there is nothing feasible in a language that can be decided by a TM in time $O(n^{1000})$, whereas a language decidable in time $O(2^{n/100})$ is more likely to be efficiently decided in practice.

This is true, in general. However, experience has shown that most of the time, when we show a language is in **P**, we usually do so by finding algorithms of lower complexity, like $O(n^5)$ or $O(n^2)$, and rarely we find languages whose algorithms are exponential with such low exponents like $n/100$. Moreover, if a language is first shown to have an algorithm of high polynomial complexity, like $O(n^{20})$, this is usually just a first step, and better algorithms are going to show up later on, that improve such a polynomial. Let's see some example languages that are in **P**.

**Remark.** If we are willing to be very formal, in order to show that a language is in **P**, we are required to exhibit a TM $M$ that decides the language with time $T_M(n) \in O(n^c)$, for some constant $c \geq 1$. However, devising such TMs can quickly become a very tedious task, and we will spend most of our time in being concerned of low-level details, such as the encoding of the input, the head moving on specific cells, and so on.

So, to keep the discussion at the right level of details, we usually resort to algorithms written in some sort of pseudo code. This should provide the high level ideas, and you should not find very difficult to imagine how this pseudo code could be actually implemented in a TM (maybe with multiple tapes).[10]

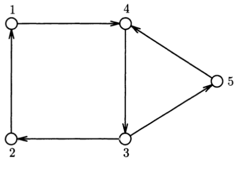
**REACHABILITY.** The first language we consider is the REACHABILITY language we have seen in our first lectures:

$$\text{REACHABILITY} = \{(G, s, t) \mid G \text{ is a directed graph such that there is a path from } s \text{ to } t \text{ in } G\}.$$

So, the above language describes the decision problem asking the question

*Given a directed graph $G$ and two nodes $s, t$, is there a path from $s$ to $t$ in $G$?*

As an example, given the graph $G$ below, and nodes $s = 1$ and $t = 5$, the answer is "yes", as there is a path 1,4,3,5 in $G$.



[10] Indeed, one can show that any algorithm written for a RAM machine, i.e., a modern computer, can be implemented in a TM with only a polynomial-time overhead. Since we are mostly concerned about languages in **P** and above, this is perfect for our purposes.

**Algorithm 1:** Algorithm for REACHABILITY

```
Input: A directed graph $G = (V, E)$ and two nodes $s, t$
Output: "Yes" if there is a path from $s$ to $t$ in $G$, and "no" otherwise

Init an empty queue $Q$;
Mark the node $s$ as visited;
Append $s$ at the end of $Q$;
while $Q$ is not empty do
    $v := Q.dequeue()$; //extracts the first element of the queue in $v$
    if $v = t$ then
        Answer "yes" and Halt;
    end
    foreach $(v, u) \in E$ such that $u$ has not been visited do
        Mark $u$ as visited;
        Append $u$ in $Q$;
    end
end
Answer "no";
```

We can easily show that REACHABILITY is in **P**. Indeed, we can use any of the many graph traversal algorithms that can be found in standard Algorithms books. For example, the breadth-first search, as shown in Algorithm 1 above.

The first three steps require constant time, while the bulk of the work is done inside the while loop. Indeed, in the worst case, the algorithm will need to visit all nodes, so $Q$ will become empty in at most $n = |V|$ iterations of the while loop. At each iteration, the queue is filled with all the neighbours of a node $v$, which in the worst case are all nodes in $V$, hence $n$. So, overall the algorithm requires time $O(n^2)$.

Note that $n$ is just the number of nodes of the graph, and not the size of the encoding of the whole input $(G, s, t)$. To perform a more precise analysis we should clarify how we encode a graph, e.g., do we use an adjacency matrix, or an adjacency list? In any case, this would not make the algorithm become exponential, and thus we do not go into these details. So, the analysis above is enough to conclude that the algorithm runs in time polynomial in the size of its input.

**PRIME.** Another interesting language in **P** is PRIME. For a natural number $N \in \mathbb{N}$, we use $< N >$ to denote its encoding in binary.

$$\text{PRIME} = \{< N >|\ N \text{ is a prime number}\}.$$

A very simple algorithm deciding PRIME is the one iterating over each possible divisor of $N$, excluding 1 and $N$ itself, i.e., $k = 2, \dots, N-1$, and checking if $k$ divides $N$. This procedure requires $O(N)$ iterations. However, we must be careful with our analysis here, as we have stated the running time of the algorithm in terms of a different parameter than the actual *size* of the input (i.e., the number of bits used to encode $N$). So, let us try to rewrite $N$ in terms of the the number of bits, say $n$, used to encode $N$.

How may bits $n$ do we need at most to encode $N$? In the worst case, this is $n = \lceil \log_2(N) \rceil$. Thus, by writing $N$ in terms of $n$, we conclude that $N \in O(2^n)$. Hence, the algorithm requires an exponential number of steps w.r.t. the size of its input.

Another way to see it is that if you increment the input size by one (i.e., you add only one bit to the input), then in the worst case you are doubling the number $N$, and thus double the running time of the algorithm.

This means that our algorithm is not able to deal with very large numbers efficiently, i.e., assume we give as input the number $N = 2^{1024}$ (this is a number with 308 digits in decimal!). This number only requires 1024 *bits* to be encoded (i.e., 128 bytes, which is very small), but our naive algorithm would require $2^{1024}$ steps, which is huge.

So, does it mean that PRIME is not in **P**? There has been no known algorithm that solves PRIME in polynomial time for quite a long time, until it has been shown, in the early 2000s, that indeed PRIME is in **P**. The algorithm, however, is far from being simple!

## 11.2 The Complexity class NP

We have seen the class **P** of all languages that are deemed tractable, i.e., they can be decided by a TM in polynomial time. However, there are also languages that do not seem to be in **P**. More precisely, despite the efforts of many researchers, during the last 50 years, we are still not aware of any algorithm that decides these languages in polynomial time.

**SAT.** A Boolean variable $x$ is a variable that can only take values in $\{0, 1\}$ (or equivalently $\{\text{false}, \text{true}\}$). A *literal* is either a variable $x$ or its negation $\neg x$. A Boolean formula in *conjunctive normal form* (CNF) is an expression over Boolean variables of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n,$$

where each $C_i$, called *clause*, is a disjunction of literals, i.e., it is of the form:

$$(\ell_1 \vee \cdots \vee \ell_k),$$

where each $\ell_i$ is a literal.

**Example 4.** *This is an example of a CNF Boolean formula:*

$$\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_5).$$

**Definition 24.** *Consider a Boolean formula $\varphi$ in CNF using variables $x_1, \ldots, x_n$. A* truth assignment *for $\varphi$ is a function $\tau : \{x_1, \ldots, x_n\} \to \{0, 1\}$, mapping each variable to a truth value.*

**Definition 25.** *A CNF Boolean formula $\varphi$ is* satisfiable *if there is a truth assignment $\tau$ that makes $\varphi$ true.*

**Example 5.** *For the formula $\varphi$ of the previous example,*

$$x_1 \to 1 \quad x_2, x_3, x_4, x_5 \to 0$$

*is a truth assignment that satisfies $\varphi$.*

The decision problem we are interested in is

*Given a Boolean formula in CNF $\varphi$, is $\varphi$ satisfiable?*

Thus, the SAT language is the following set of (encodings of) Boolean formulas:

$$\text{SAT} = \{\varphi \mid \varphi \text{ is a Boolean formula in CNF that is satisfiable}\}.$$

A naive procedure that checks if a given formula $\varphi$ is satisfiable simply iterates over every possible truth assignment, and checks if one satisfies the formula. Note that given a Boolean CNF formula $\varphi$ and a truth assignment, *checking* whether the assignment satisfies $\varphi$ is rather "easy". In fact, it is enough to iterate over each clause $C_i$ of $\varphi$, and for each $C_i$, verify that at least one literal of $C_i$ becomes true under the assignment.

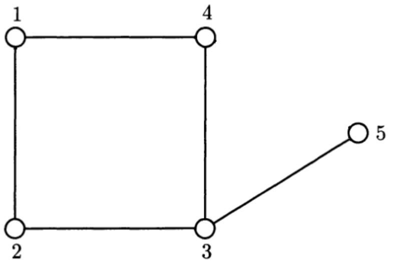
So, checking that a truth assignment satisfies $\varphi$ requires time $O(m \cdot c)$, if $m$ is the number of clauses and $c$ the maximum number of literals in a clause. However, if $n$ is the total number of Boolean variables in $\varphi$, there are $2^n$ possible truth assignments to try. Hence, the overall procedure requires $O(2^n \cdot m \cdot c)$ steps, in the worst case. Recalling that we want a time bound in terms of the overal size of the input, we can employ the usual trick, by observing that all such parameters $n, m, c$ are all necessarily smaller than the number of bits, say $b$, used to encode the input, and so, if $n \leq b$, $m \leq b$, and $c \leq b$, we have $O(2^n \cdot m \cdot c) = O(2^b \cdot b^2) = O(2^b \cdot 2^b) = O(4^b)$.[11]

Can we do better than the naive algorithm? The answer to this question is simply "We do not know". No known algorithm is able to decide SAT in polynomial time.

**Independent Sets.** Another language of which we do not know any polynomial time algorithm yet is about graphs. Consider an *undirected graph* $G = (V, E)$. That is, a graph where an edge $e \in E$ is a *set* $e = \{u, v\}$ of two *distinct* nodes $u, v$ of $V$, rather than an ordered pair $(u, v)$.[12]

[11] From now on, we will give time bounds using different parameters of the input, to make the analysis more clear, but we should always remember that these bounds must always be converted in terms of the size of the whole input.

[12] Note that since we assume an edge contains two *distinct* nodes, we do not allow self-loops in our undirected graphs. We will assume this throught the course for simplicity.



An *independent set* of $G$ is a set $S$ of nodes of $G$, such that no two nodes in $S$ are connected by an edge.

**Example 6.** *In the example graph above, $S_1 = \{2, 4, 5\}$ is an independent set, as no two nodes in $S_1$ are connected by an edge. Also the set $S_2 = \{2\}$ is an independent set, as well as every other singleton node set. Furthermore, also the empty set of nodes $S_3 = \emptyset$ is an independent set. The set $\{1, 3, 5\}$ is not an independent set, since* 3 *and* 5 *are connected by an edge of the graph.*

The decision problem we are interested in is

*Given an undirected graph $G$ and an integer $k$, does $G$ have an independent set with at least $k$ nodes?*

The corresponding language is the following set of (encodings of) pairs of an undirected graph $G$ and integer $k$:

$$\begin{aligned} \text{IS} \quad = \quad \{(G,k) \mid \quad & G \text{ is an undirected graph} \\ & \text{with an independent set } S \text{ such that } |S| \geq k\}. \end{aligned}$$

**Remark.** Note that we are not asking whether a graph $G$ has an independent set (of any size), as this problem is not interesting: the answer to this question is always "yes". Every graph has an independent set (e.g., the empty set). Thus, also asking for the existence of an independent set of size *at most* $k$ is trivial, as the cardinality of the empty set is smaller than any $k$.

Given a pair $(G, k)$, a naive procedure for deciding IS simply tries all possible sets of nodes of $G$ of cardinality at least $k$, and checks if one of these sets is independent. Again, given a graph $G$, and a set of nodes $S$, checking if $S$ is independent is easy: check that every pair of nodes in $S$ is disconnected. This requires time $O(n^2)$ ($S$ contains at most all $n$ nodes of $G$, and thus $n(n-1)/2$ is the maximum number of pairs). However, there are

$$\sum_{i=k}^{n} \binom{n}{i}$$

sets of nodes to try. Thus, the naive procedure does not decide IS in polynomial time.[13] Also for IS, no polynomial time algorithm is known to date.

[13] We recall that $\binom{n}{i} = \frac{n!}{i! \cdot (n-i)!}$ is the number of all possible sets of cardinality $i$ that can be constructed by using elements from a set of cardinality $n$.

There exist many other languages that have similar properties to SAT and IS and for which no polynomial time algorithm is known. Let us see what all these languages have in common.

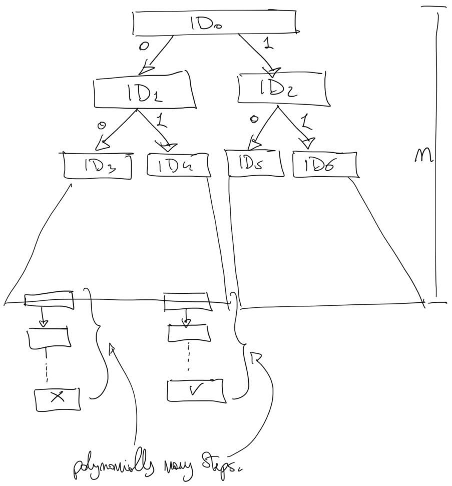
In all such languages, to decide whether a string belongs to the language, we need to consider many candidate solutions. Although there might be exponentially many candidates, we have that

- each candidate solution is "small", i.e., of polynomial size w.r.t. the size of the input (e.g., an independent set contains at most $|V|$ nodes), and
- verifying that a candidate solution is indeed a solution only requires polynomial time.

We already know a kind of TMs that is able to implement the above pattern efficiently: non-deterministic TMs.

Indeed, one could decide SAT in polynomial-time, if we are allowed to use non-determinism. Given a CNF Boolean formula $\varphi$ over some variables $x_1, \ldots, x_n$, our NTM $M$ needs to do the following:

1. Guess a truth assignment for $\varphi$, i.e., write $n$ bits on the tape, by non-deterministically choosing the value of each bit. So, the $i$-th bit is the guessed truth value for the variable $x_i$.
2. Verify that the guessed assignment satisfies $\varphi$, in which case accept, otherwise reject.



Thus, what we are doing here is to make use of non-determinism to try each candidate solution in a different path of the computation tree of $M$, and in each path, $M$ accepts if the candidate solution is indeed a solution, and rejects otherwise.

Moreover, the time required by the above NTM $M$ is polynomial in the size of $\varphi$, since each solution is of length polynomial (in this case, $n$), and thus $M$ non-deterministically writes polynomially many symbols in its tape. Then, $M$ verifies that what it has written is a truth assignment that satisfies $\varphi$, which can be done in polynomially many steps, as discussed before.

Similarly, we can decide IS with a NTM with the same pattern. Given an undirected graph $G$ with $n$ nodes and a number $k$, our NTM $M$ does the following:

1. For each node $v$ of $G$, non-deterministically decide if $v$ must be written in the tape or not;

2. If at least $k$ nodes have been written, and no two of them are connected in $G$, then accept, otherwise reject.

Again, we use non-determinism to guess a candidate solution, and then, verify that the guessed candidate solution is indeed a solution. Moreover, the total number of steps our machine performs in a path of its computation tree is at most polynomial w.r.t. size of the input.

**Definition 26.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a function. We define*

$$\mathrm{NTIME}(f(n)) = \{\mathcal{L} \mid \exists \textit{ a NTM } M \textit{ that decides } \mathcal{L} \textit{ and } T_M(n) \in O(f(n))\}.$$

So, NTIME$(f(n))$ collects all languages that can be decided by a *non-deterministic* TM in time $O(f(n))$.[14] We are now ready to define our class of languages.

**Definition 27.** *The class of* non-deterministic polynomial-time languages *is defined as*

$$\mathbf{NP} = \bigcup_{c \geq 1} \mathrm{NTIME}(n^c).$$

So, a language is in **NP** if it can be decided by a NTM in polynomial time.

**Remark 1.** Note that in the definition of **NP**, we are not requiring that the NTMs that decide the languages in **NP** follow the guess and check approach. In general, a NTM can use non-determinism at any point of its computation, and not only at the beginning. This might indicate that our definition of **NP** not only collects languages decidable with the guess and check approach, but also others. However, as we are going to see, later on in the course, any polynomial-time NTM can be converted to a polynomial-time NTM that uses the guess and check approach.

**Remark 2.** Note that **NP** does not mean "not polynomial", but it means "non-deterministic polynomial time". That is, if we are allowed to get some help from non-determinism, we can still decide the language in polynomial time. This is,

[14] Recall that for a NTM $M$, the time required $T_M(n)$ is the length of the longest path in its computation tree, when considering all inputs of length $n$.

in some sense, our way to identify a class of problems that are not that hard to solve, when given some help.

We conclude observing that our discussion above implies that $\text{SAT}, \text{IS} \in \mathbf{NP}$.

# 12 P vs NP and NP-completeness

In the previous lecture we introduced two key complexity classes: **P** and **NP**. We also briefly discussed some languages in both classes, and identified languages in **NP** that seemingly do not admit any (deterministic) polynomial time procedure for deciding them, but admit a non-deterministic, polynomial-time one.

Indeed, by their very definition, we clearly see that $\mathbf{P} \subseteq \mathbf{NP}$, since if a language is decided by a TM in polynomial time, this TM is itself also a NTM with precisely one choice at each step. However, it is not at all clear whether some languages in **NP** are really harder to decide, or it is just the case that we were not clever enough to find better algorithms.[15]

In other words, is it the case that $\mathbf{P} = \mathbf{NP}$? That is, can every language decided by a NTM in polynomial time be also decided by a "more clever" TM, that without non-determinism still requires polynomial time?

The above question has been asked for the first time in 1971 by computer scientist and mathematician Stephen Cook, and to date, the answer to this question is still unknown.

Different efforts have been put in answering the $\mathbf{P} = \mathbf{NP}$ question, and different formal tools have been developed to get closer to an answer.

One way of tackling the $\mathbf{P} = \mathbf{NP}$ question is to identify some very hard languages in **NP**, that we call $\mathbf{NP}-$complete. The property that these languages enjoy is the following:

There can be a polynomial-time algorithm for deciding an $\mathbf{NP}-$complete language iff *every* language in **NP** can be decided in polynomial time (i.e., $\mathbf{P} = \mathbf{NP}$).

So, intuitively, $\mathbf{NP}-$complete languages are considered to be the hardest languages to decide, among all languages in **NP**, and deciding one in polynomial time is equivalent to given an answer to the $\mathbf{P} = \mathbf{NP}$ question. We now formalize the notion of $\mathbf{NP}-$completeness.

## 12.1 NP−complete languages

We first need to introduce the notion of polynomial-time reductions.

**Definition 28.** *A* polynomial-time reduction *$M$ from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$ such that $T_M(n) \in O(n^c)$, with $c > 0$.*

*We write $\mathcal{L}_1 \leq_p \mathcal{L}_2$ to say that $\mathcal{L}_1$ reduces to $\mathcal{L}_2$ in polynomial time, i.e., there is a polynomial-time reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$.*

So, a polynomial-time reduction is just a reduction that requires a polynomial number of steps for converting a string into another string. We now define the languages that are at least as hard as all the languages in **NP**.

[15] Note that we are not saying that *every* language in **NP** is not in **P**, as REACHABILITY, for example, is in **P** and thus also in **NP**. Rather, there are some special **NP** languages that seem to not be in **P**.

**Definition 29.** *A language $\mathcal{L}$ is* $\mathbf{NP}-$hard *if for every language $\mathcal{L}' \in \mathbf{NP}$,*

$$\mathcal{L}' \leq_p \mathcal{L}.$$

So, intuitively, it means that if we were able to decide an $\mathbf{NP}-$hard language $\mathcal{L}$ in polynomial time, then, we could decide every $\mathbf{NP}$ language in polynomial time, by first converting the input string to a string for $\mathcal{L}$ in polynomial time, and then decide if the string is in $\mathcal{L}$ in polynomial time.

Hence, $\mathbf{NP}-$hard languages are at least as hard to decide as *all* the languages in $\mathbf{NP}$. However, we wanted to discuss about the hardest languages *among the ones in* $\mathbf{NP}$. The fact that a language is $\mathbf{NP}-$hard does not mean this is also in $\mathbf{NP}$. To guarantee this, we need that our language is $\mathbf{NP}-$complete.

**Definition 30.** *A language $\mathcal{L}$ is* $\mathbf{NP}-$complete *if $\mathcal{L} \in \mathbf{NP}$ and $\mathcal{L}$ is* $\mathbf{NP}-$hard.

We can now show the connection between $\mathbf{NP}-$completeness and the $\mathbf{P} = \mathbf{NP}$ question.

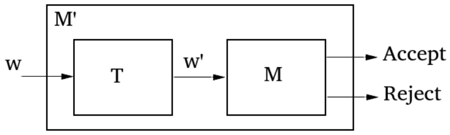
**Theorem 20.** *Let $\mathcal{L}$ be an* $\mathbf{NP}-$complete *language. Then,*

$$\mathcal{L} \in \mathbf{P} \textit{ if and only if } \mathbf{P} = \mathbf{NP}.$$

*Proof.* To prove the claim we need to prove two parts: that $\mathcal{L} \in \mathbf{P}$ implies $\mathbf{P} = \mathbf{NP}$, and that $\mathbf{P} = \mathbf{NP}$ implies $\mathcal{L} \in \mathbf{P}$. We first focus on the second.

Assume $\mathbf{P} = \mathbf{NP}$. Then, for every language in $\mathbf{NP}$, there exists a deterministic TM that decides the language in polynomial time. Since $\mathcal{L}$ is also in $\mathbf{NP}$, as it is $\mathbf{NP}-$complete, and not just $\mathbf{NP}-$hard, $\mathcal{L}$ must also have a deterministic TM that decides it in polynomial time, and thus $\mathcal{L} \in \mathbf{P}$.

Assume now that $\mathcal{L} \in \mathbf{P}$, i.e., there exists a deterministic TM $M$ that decides $\mathcal{L}$ in polynomial time. Let $\mathcal{L}'$ be an arbitrary language in $\mathbf{NP}$. Since $\mathcal{L}$ is $\mathbf{NP}-$complete, and thus $\mathbf{NP}-$hard, there is a polynomial-time reduction $T$ from $\mathcal{L}'$ to $\mathcal{L}$. As usual, the following deterministic TM $M'$ decides $\mathcal{L}'$, by first converting the input string $w$ to a string $w'$ and then using $M$ to decide whether $w \in \mathcal{L}'$ or not.



Moreover, if $|w| = n$, $M'$ performs in two phases. It first executes $T$ in a number of steps that is $O(n^c)$, for some $c > 0$. Thus, since in the worst case, $T$ writes one symbol in its output tape at each step, we have that $m = |w'| \in O(n^c)$. Then, in a second phase, $M'$ executes the control of $M$ with input $w'$. Since we assumed that $M$ requires a polynomial number of steps, $M$ requires $O(m^d) = O(n^{c \cdot d})$ steps, for some $d > 0$. Thus, $T_{M'}(n) \in O(n^c + n^{c \cdot d})$, and thus $M'$ decides $\mathcal{L}'$ in a polynomial number of steps w.r.t. the size of its input, and we conclude that $\mathcal{L}' \in \mathbf{P}$.

The above holds for *any* language $\mathcal{L}' \in \mathbf{NP}$, and thus, $\mathbf{NP} \subseteq \mathbf{P}$, that together with the fact that $\mathbf{P} \subseteq \mathbf{NP}$, implies $\mathbf{P} = \mathbf{NP}$. □

**Remark.** Note that for proving the above claim, the use of polynomial-time reductions, rather than any kind of reductions, is crucial.

Indeed, if the reduction from a language $\mathcal{L}_1$ to $\mathcal{L}_2$ is, e.g., exponential, and we assumed $\mathcal{L}_2$ is in $\mathbf{P}$, we would not be able to combine the reduction with the polynomial-time TM deciding $\mathcal{L}_2$, and obtain a polynomial-time TM for deciding $\mathcal{L}_1$.

Another way to see it is that the goal of a reduction, in complexity theory, is to guarantee that if we have an efficient algorithm for solving the target problem, we can use the reduction to solve the source problem by first converting the instance with the reduction and then use the efficient algorithm for the target. But if converting the instances is more demanding than solving the source problem itself, then there is no point in using the reduction, and we could very well solve the source problem directly.

So, it seems $\mathbf{NP}-$completeness is a useful notion for tackling the $\mathbf{P} = \mathbf{NP}$ question. However, for this tool to be really effective, it is important that at least an $\mathbf{NP}-$complete language exist. So, how would we go about showing that an $\mathbf{NP}-$complete language $\mathcal{L}$ exists?

Generally, we should apply the definition: prove that every language $\mathcal{L}'$ in $\mathbf{NP}$ reduces to $\mathcal{L}$ in polynomial time. This seems quite a challenging task (and it actually is).

The very first language that has been shown to be $\mathbf{NP}-$complete is SAT, which we already mentioned in the previous lecture:

$$\text{SAT} = \{\varphi \mid \varphi \text{ is a Boolean formula in CNF that is satisfiable}\}.$$

Recall that a formula is satisfiable if there is an assignment $\tau : \{x_1, \ldots, x_n\} \to \{\text{true}, \text{false}\}$ of a truth value to each variable of $\varphi$ such that $\tau$ satisfies $\varphi$.

**Theorem 21.** *SAT is* $\mathbf{NP}-$complete*.*

The proof for the above result is quite involved, and we will see it much later on in the course. For the moment, let us just take the above theorem for granted.

In the meanwhile, one question that we might ask is the following: is explicitly reducing every language in $\mathbf{NP}$ to our language $\mathcal{L}$ the only way to prove that $\mathcal{L}$ is $\mathbf{NP}-$hard?

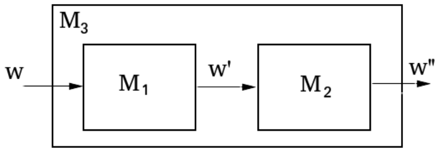
We are now going to show that once we know that at least one $\mathbf{NP}-$hard language exists (e.g., SAT), showing that other languages are $\mathbf{NP}-$hard becomes much easier (not easy, but easier!).

First, we show that polynomial-time reductions can be composed.

**Theorem 22.** *If $\mathcal{L}_1 \leq_p \mathcal{L}_2$ and $\mathcal{L}_2 \leq_p \mathcal{L}_3$, then*

$$\mathcal{L}_1 \leq_p \mathcal{L}_3.$$

*Proof.* Let $M_1$ and $M_2$ be the polynomial-time reductions from $\mathcal{L}_1$ to $\mathcal{L}_2$, and from $\mathcal{L}_2$ to $\mathcal{L}_3$ respectively. Since they are polynomial-time reductions, we have $T_{M_1}(n) \in O(n^c)$, and $T_{M_2}(n) \in O(n^d)$. Thus, the following TM $M_3$ is such that $T_{M_3}(n) \in O(n^c + (n^c)^d) = O(n^{c \cdot d})$.



So, $M_3$ is still a polynomial-time TM. $M_3$ is also a reduction from $\mathcal{L}_1$ to $\mathcal{L}_3$.

Assume $w \in \mathcal{L}_1$. Then $w' \in \mathcal{L}_2$ (since $w'$ is obtained with the reduction $M_1$), and since $w' \in \mathcal{L}_2$, then $w'' \in \mathcal{L}_3$ (since $w''$ is obtained with the reduction $M_2$). Similarly, if $w \notin \mathcal{L}_1$, we conclude that $w'' \notin \mathcal{L}_3$. □

We now show how we can transfer the knowledge we have about an **NP**−hard language to another language.

**Theorem 23.** *If a language $\mathcal{L}_1$ is* **NP**−hard *and $\mathcal{L}_1 \leq_p \mathcal{L}_2$, then*

$$\mathcal{L}_2 \textit{ is } \mathbf{NP}\text{−hard}.$$

*Proof.* Assume $\mathcal{L}_1$ is an **NP**−hard language and assume $\mathcal{L}_1 \leq_p \mathcal{L}_2$. We need to prove that for every language $\mathcal{L} \in$ **NP**, $\mathcal{L}$ reduces to $\mathcal{L}_2$ in polynomial time.

Let $\mathcal{L}$ be an arbitrary language in **NP**. Since $\mathcal{L}_1$ is **NP**−hard, then $\mathcal{L} \leq_p \mathcal{L}_1$. Since $\mathcal{L} \leq_p \mathcal{L}_1$ and $\mathcal{L}_1 \leq_p \mathcal{L}_2$, by Theorem 22, we conclude that $\mathcal{L} \leq_p \mathcal{L}_2$. Thus, every language in **NP** reduces to $\mathcal{L}_2$ in polynomial time and then $\mathcal{L}_2$ is **NP**−hard. □

With Theorem 23 above in place, we now have a very useful tool for showing that a language $\mathcal{L}$ is **NP**−complete. It is enough to prove two things: $\mathcal{L}$ is in **NP**, and that we can reduce a known **NP**−hard language (e.g., SAT) to $\mathcal{L}$ in polynomial time.

## 12.2 Another NP−complete language

We now show an application of Theorem 23. We consider a "simpler" version of SAT, called 3-SAT, and we show that even this simpler version is **NP**−complete.

A Boolean formula $\varphi$ is in 3-CNF if each of its clauses contains *at most* 3 literals.

**Example 7.** *The Boolean formula we have seen in the previous lecture is in 3-CNF:*

$$\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_5).$$

Thus, the language we are interested in is

$$\text{3-SAT} = \{\varphi \mid \varphi \text{ is a Boolean formula in 3-CNF that is satisfiable}\}.$$

That is,

*Given a Boolean formula $\varphi$ in 3-CNF, is $\varphi$ satisfiable?*

**Theorem 24.** *SAT $\leq_p$ 3-SAT.*

*Proof.* We need to show how to convert (in polynomial time) a general Boolean CNF formula $\varphi$ to a formula in 3-CNF $\varphi'$, and at the same time be such that $\varphi$ is satisfiable iff $\varphi'$ is satisfiable.

Consider a Boolean formula in CNF

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m.$$

Let us focus on a generic clause $C_i$. A clause is the disjunction of some literals, i.e., $C_i$ is of the form

$$(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k).$$

Our goal is to lower the number of literals in each clause $C_i$. To do so, we construct a new formula $\varphi'$ starting from $\varphi$ as follows. We check if $C_i$ has more than 3 literals. If not, we copy the clause in $\varphi'$ without changing it.

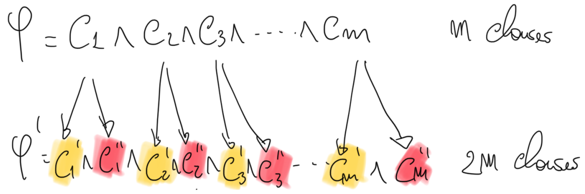
Otherwise, we convert $C_i$ into the conjunction of two clauses $C_i'$ and $C_i''$. In clause $C_i'$, we leave the first two literals $\ell_1, \ell_2$ and replace the remaining $k-2$ literals with a fresh new Boolean variable, say $h_i$. So, we obtain the clause

$$C_i' = (\ell_1 \vee \ell_2 \vee h_i).$$

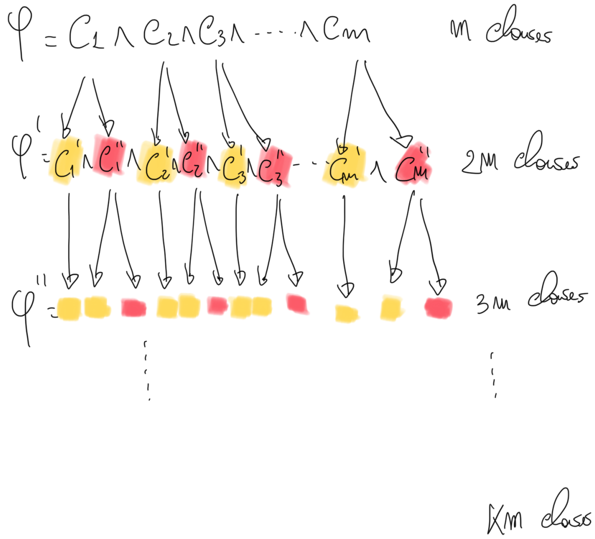
Moreover, the clause $C_i''$ contains the remaining literals, plus $\neg h_i$:

$$C_i'' = (\ell_3 \vee \cdots \vee \ell_k \vee \neg h_i).$$

We do the above for all clauses of $\varphi$.


φ = C1 ∧ C2 ∧ C3 ∧ ···· ∧ Cm
m clauses
φ' = C1' ∧ C1'' ∧ C2' ∧ C2'' ∧ C3' ∧ C3'' ··· Cm' ∧ Cm''
2m clauses

Note that the new formula $\varphi'$ has at most $2m$ clauses, where $m$ of them have 3 literals (the yellow ones), and the other $m$ have one literal less than the clause from which they come from (the red ones). By applying this process again on $\varphi'$, we obtain a formula $\varphi''$ with $3m$ clauses, where $2m$ clauses have 3 literals (the yellow ones), and $m$ clauses have one literal less (the red ones), and so on, as shown below.


φ = C1 ∧ C2 ∧ C3 ∧ ···· ∧ Cm
m clauses
φ' = C1' ∧ C1'' ∧ C2' ∧ C2'' ∧ C3' ∧ C3'' ··· Cm' ∧ Cm''
2m clauses
φ'' =
3m clauses
Km clauses

So, if $k$ is the maximum number of literals in a clause of $\varphi$, we can apply this process at most $k$ times, and obtain a formula in 3-CNF with at most $k \cdot m$ clauses. Thus, the overall procedure converts $\varphi$ to a 3-CNF formula in polynomial time.

What is left to prove is that the above transformation preserves satisfiability. If we show it for one of the $n$ steps of the above process, as a consequence, the whole procedure preserves satisfiability.

Assume $\varphi$ is satisfiable, i.e., there is a truth assignment $\tau$ over the Boolean variables of $\varphi$ that makes every clause of $\varphi$ true. We need to prove that there is also a truth assignment $\tau'$ of the variables of $\varphi'$ that makes each clause of $\varphi'$ true.

Let $\tau : \{x_1, \ldots, x_n\} \to \{\text{true}, \text{false}\}$ be such a truth assignment of the variables of $\varphi$ that makes each clause $C_i$ of $\varphi$ true. This means that at least one literal $\ell_j$ in $C_i$ becomes true thanks to $\tau$. Thus, either the clause

$$C_i' = (\ell_1 \vee \ell_2 \vee h_i)$$

or the clause

$$C_i'' = (\ell_3 \vee \cdots \vee \ell_k \vee \neg h_i)$$

in $\varphi'$ becomes true under $\tau$, because contains the literal $\ell_j$. To see that $\varphi'$ is satisfiable, observe that the Boolean variables of $\varphi'$ are precisely the ones of $\varphi$, i.e., $x_1, \ldots, x_n$, plus the variables $h_1, \ldots, h_m$ (one for each clause). Then, the truth assignment $\tau'$ that satisfies $\varphi'$ is the one that assigns to $x_1, \ldots, x_n$ the same values as $\tau$, and if the true literal $\ell_j$ of a clause $C_i$ appears, e.g., in the clause $C_i'$ of $\varphi'$ (and thus $\tau'$ makes $C_i'$ true), then $\tau'(h_i) = \text{false}$ (making $C_i''$ true as well). Otherwise, if the true literal $\ell_j$ of a clause $C_i$ appears in $C_i''$ (and thus $\tau'$ makes $C''_i$ true), then $\tau'(h_i) = \text{true}$ (making $C'_i$ true as well). Hence, $\tau'$ satisfies $\varphi'$.

We now need to prove the converse, i.e., if $\varphi'$ is satisfiable, then $\varphi$ is satisfiable. Assume $\varphi'$ is satisfiable, i.e., there is a truth assignment $\tau'$ over the Boolean variables of $\varphi'$ that makes every clause of $\varphi'$ true. Remember that $\varphi'$ is of the form:

$$C'_1 \wedge C''_1 \wedge C'_2 \wedge C''_2 \wedge \cdots \wedge C'_m \wedge C''_m,$$

where $C'_i = (\ell_1 \vee \ell_2 \vee h_i)$ and $C''_i = (\ell_3 \vee \cdots \vee \ell_k \vee \neg h_i)$ come from the same clause $C_i = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$ of $\varphi$. Let $\tau$ be truth the assignment over the variables $x_1, \ldots, x_n$ of $\varphi$ such that $\tau(x_i) = \tau'(x_i)$ for each $i \in \{1, \ldots, n\}$ (i.e., $\tau$ is precisely as $\tau'$, but does not assign any value to $h_1, \ldots, h_m$). Since $\tau'$ makes both $C'_i$ and $C''_i$ true, and since $\tau'$ must assign one specific truth value to $h_i$, it cannot make make both $C'_i$ and $C''_i$ true only thanks to $h_i$, but it must make some literal $\ell_j$ of $C_i$ true as well. This implies that $\tau(\ell_j) = \text{true}$, and thus $\tau$ satisfies $C_i$. This holds for all clauses of $\varphi$, and thus $\varphi$ is satisfiable. □

**Theorem 25.** *3-SAT is* **NP**−complete.

*Proof.* We need to prove that 3-SAT is in **NP** and it is **NP**−hard. Since SAT is in NP, also its simplified version 3-SAT must be in **NP**. **NP**−hardness follows from Theorem 24 (SAT $\leq_p$ 3-SAT), Theorem 21 (SAT is **NP**−complete), and Theorem 23. □

**Remark.** Note that the approach used to reduce SAT to 3-SAT cannot be used to convert a Boolean CNF formula in 2-CNF. In fact, to reduce the number of literals in a clause

$$C_i = (\ell_1 \vee \ell_2 \vee \ell_3),$$

we might try to construct the clauses

$$C'_i = (\ell_1 \vee h)$$

and

$$C''_i = (\ell_2 \vee \ell_3 \vee \neg h).$$

However, $C''_i$ still contains three literals.

Indeed, there is no known reduction from SAT to 2-SAT, and a strong evidence for the non existence of such a reduction is the fact that 2-SAT is known to be in **P** (we do not show any proof for this).

**Theorem 26.** *2-SAT is in* **P**.

So, if we were able to reduce SAT to 2-SAT in polynomial time, we would show that 2-SAT is **NP**−hard but also in **P**, and thus conclude that **P** = **NP**.

**Exercise.** We conclude this lecture with an exercise. We have defined 3-SAT as the language of Boolean formulas in CNF that are satisfiable and have *at most* 3 literals in each clause. Consider instead the language

$$\text{EXACT-3-SAT} = \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula in CNF with } \textit{exactly } 3 \text{ literals per clause}\}.$$

How would you adapt the reduction we used in Theorem 24 to show that EXACT-3-SAT is **NP**−complete?

Alternatively, you could try reducing 3-SAT to EXACT-3-SAT, and only be concerned about “enlarging” clauses with less than 3 literals.

# 13 Some NP−complete languages

In this lecture we will prove more languages being **NP**−complete. As we are going to see in the following lectures, the notion of **NP**−completeness is almost ubiquitous in many different fields. That is, natural and important problems in mathematics, computer science, optimization, graph theory, etc. turned out to be **NP**−complete and thus believed to be intractable.

## 13.1 Independent set

The first language we consider is one we mentioned already some lectures ago, i.e., the independent set problem.

Recall that given an undirected graph $G = (V, E)$, an independent set of $G$ is a set of nodes $S \subseteq V$ such that no two nodes in $S$ are connected. The language we want to study is thus:

$$\begin{array}{rcll} \text{IS} & = & \{(G,k) \mid & G \text{ is an undirected graph} \\ & & & \text{with an independent set } S \text{ such that } |S| \geq k\}. \end{array}$$

**Theorem 27.** *IS is* **NP**−complete*.*

*Proof.* To prove the claim, we need to show that IS is in **NP**, and that it is **NP**−hard. We have already shown that IS in **NP** in a previous lecture. To prove that IS is **NP**−hard we provide a polynomial-time reduction from EXACT-3-SAT to IS.

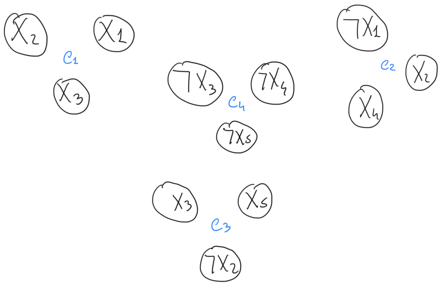
Our reduction must convert (in polynomial time) a Boolean formula $\varphi$ with exactly 3 literals per clause to a pair $(G, k)$ of a graph and an integer in such a way that: if $\varphi$ is satisfiable, then $G$ has an independent set with at least $k$ nodes, and if $\varphi$ is unsatisfiable, $G$ has no independent set with at least $k$ nodes.

We describe our reduction by using an example Boolean formula. The general construction should become immediately clear, after the example is given. Consider the formula

$$\varphi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\neg x_2 \vee x_3 \vee x_5)}_{C_3} \wedge \underbrace{(\neg x_3 \vee \neg x_4 \vee \neg x_5)}_{C_4}.$$

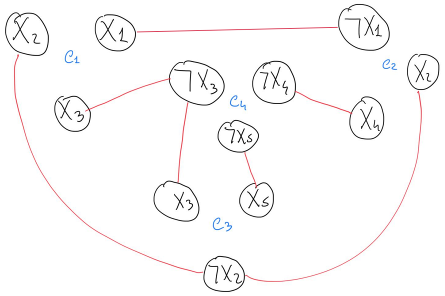
We observe that there is some kind of correspondence between satisfying a Boolean formula, and finding an independent set in a graph. In fact, to satisfy a formula, we need to choose which literals become true and which false. Similarly, to find an independent set, we need to choose which nodes are part of the independent set, and which not. Moreover, we cannot make true or false one literal and its negated at the same time.

So, in our reduction we represent this by assigning each occurrence of a literal in $\varphi$ to a node in the graph $G$. In particular, if a literal appears in different clauses, we consider multiple nodes labelled with the same literal. With the formula above, the nodes we have are:



For convenience, we place the literals appearing together in a clause, close to each other in the graph, and highlight these nodes with the name of the clause where they belong.

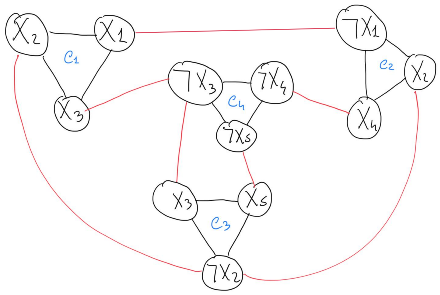
To model the fact that we cannot give the same truth value to a literal and its negation, we want in our graph to not be able to place in an independent set two nodes labelled with complementary literals. Thus, we add an edge between each pair of nodes labelled with complementary literals. The new edges are drawn in red.



The next observation is that once one has chosen a truth assignment for the literals in $\varphi$, this assignment satisfies $\varphi$ if and only if *each* clause in $\varphi$ becomes true, which means, if $m$ is the number of clauses of $\varphi$ (in this case 4), our assignment makes at least $m$ literals true, one per each clause.

In the corresponding graph, then we must be sure that this satisfying assignment corresponds to an independent set of at least $m$ nodes. So, our reduction sets $k = m$. However, although we now require an independent set of at least $m = 4$ nodes, we can still pick as an independent set the three nodes of clause $C_1$ plus the node $x_2$ of the second clause. However, this choice does not correspond to a satisfying assignment for $\varphi$, as the other clauses are left unsatisfied.

To force our independent sets to contain one literal per clause, we connect each group of three literals in a triangle.



In this way, if we want to construct an independent set with at least 4 nodes, we must necessarily pick one node for each clause, and no two such nodes are labeled with complementary literals. Actually, in the graph we can pick no more than 4 nodes (one node per clause), so our independent sets contain exactly 4 nodes.

The above construction obviously performs in polynomial time w.r.t. the size of $\varphi$ since we simply copied the clauses. Moreover, the reduction adds 3 edges for each clause, plus one edge for each pair of a variable and its negation, thus overall it adds at most $3m + n$ edges, where $m$ is the number of clauses and $n$ are all the literals appearing in $\varphi$. Finally, setting $k = m$ is trivial.

We now prove that the above procedure is a reduction.

Assume $\varphi$ is satisfiable, i.e., there is a truth assignment $\tau : \{x_1, \ldots, x_n\} \rightarrow \{\text{true}, \text{false}\}$ of its variables that makes $\varphi$ true. Thus, for each clause $C$ of $\varphi$, at least one literal of $C$ is made true by $\tau$. That is, there is at least one "reason" (i.e., a true literal) for which the clause $C$ is true. Let, for each clause $C$, $\ell^C$ be one of such literals, and let $v(\ell^C)$ be the node in the graph corresponding to $\ell^C$. Then, the set of nodes

$$S = \{v(\ell^C) \mid C \text{ is a clause of } \varphi\}$$

is an independent set of the graph with $k = m$ nodes.[16] In fact, by construction, $|S| \geq k$, since we have picked one literal per clause. Now, note that for each node $v(\ell^C)$ in $S$, the only nodes connected to it in the graph are the ones in its own triangle, and the ones labelled with its complementary literal. However, since we picked only one literal per clause, only one node of each triangle is in $S$, and since $\tau$ is a truth assignment, there is no way we could have placed in $S$ two nodes labelled with complementary literals. Thus, $S$ is an independent set.

Assume now that $G$ has an independent set $S$, such that $|S| \geq k$. Note that $S$ is suggesting how to build a truth assignment that satisfies $\varphi$. That is, consider an assignment $\tau : \{x_1, \ldots, x_n\} \rightarrow \{\text{true}, \text{false}\}$ where, for each node in $S$ with label $x_i$, $\tau(x_i) = \text{true}$, and for each node in $S$ with label $\neg x_i$, $\tau(x_i) = \text{false}$; we do not care of how we assign the remaining variables. We have that $\tau$ is

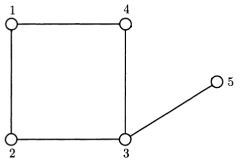
[16] Intuitively, for each clause $C$ of $\varphi$, the set $S$ contains one of the "reasons" why the clause $C$ is true.

well-defined (i.e., it assigns only one truth value to each variable), because there are no two nodes in $S$ that are labeled with complementary literals, and thus $\tau$ does not assign two different values to the same variable. Moreover, since $S$ has at least $k = m$ nodes, it must contain one node for each triangle. By construction of $\tau$, this means that each clause of $\varphi$ has a literal that is made true by $\tau$. Hence, $\varphi$ is satisfiable. □

## 13.2 Vertex Cover

We now consider another problem over graphs. Consider an undirected graph $G = (V, E)$. A *vertex cover* of $G$ is a set of nodes $VC \subseteq V$ that "touches" all edges of $G$, i.e., for each edge $\{u, v\} \in E$, $VC$ contains either $u$ or $v$, or even both.

Consider the following undirected graph:



The set of nodes VC $= \{1, 3\}$ is a vertex cover, as every edge of the graph is "touched" by at least one node in VC. Trivially, also the complete set of nodes $\{1, 2, 3, 4, 5\}$ is a vertex cover.

Note that for a graph $G$, a vertex cover always exists, e.g. the one containing all nodes of $G$. Thus, asking if a vertex cover exists is not a very interesting question. The question is whether there is vertex cover with at most a certain number of nodes.

This problem often appears in optimization contexts. For example, if the edges of $G$ represent corridors in a house, and nodes are where two or more corridors join, a vertex cover represents a set of points in the house where we can place a security camera in order to "see" all corridors of the house. Of course, one wants to minimize the number of installed cameras, and thus we want to know if a vertex cover of at most a certain size exists or not.

$$\begin{aligned}\text{VCOVER} \;=\; \{(G, k) \mid\;& G \text{ is an undirected graph}\\ & \text{with a vertex cover } VC \text{ such that } |VC| \leq k\}.\end{aligned}$$

We show that VCOVER is **NP**−complete. For this, we make an observation. Consider the previous example graph and the vertex cover VC $= \{1, 3\}$. If we consider the "complement" of VC, i.e., the set of nodes $S = V \setminus \text{VC} = \{2, 4, 5\}$, we have that $S$ is an independent set of the graph. We prove that this is not a coincidence.

**Lemma 1.** *Let $G = (V, E)$ be an undirected graph, and $S \subseteq V$ a set of nodes. Then, $S$ is an independent set of $G$ iff $VC = V \setminus S$ is a vertex cover of $G$.*

*Proof.* Let $S$ be some set of nodes of $G$, and let $VC = V \setminus S$.

Assume that $S$ is an independent set of $G$, but, towards a contradiction, assume $VC$ is not a vertex cover of $G$. Thus, there must be at least one edge $\{u, v\} \in E$ that is not covered by $VC$, i.e., neither $u$ nor $u$ appear in $VC$. However, since $S$ contains all nodes that are not in $VC$, means that both $u$ and $v$ are in $S$. This, however implies that $S$ is not independent, obtaining a contradiction. Hence, $VC$ is a vertex cover of $G$.

Assume now that $VC$ is a vertex cover of $G$, but, towards a contradiction, assume that $S$ is not an independent set of $G$. If $S$ is not independent, then, there are at least two nodes $u, v \in S$ connected by an edge, i.e., $\{u, v\} \in E$. However, since $VC$ contains all nodes that are not in $S$, neither $u$ nor $v$ appear in $VC$. This, however implies that the edge $\{u, v\}$ is not "covered" by $VC$, and thus $VC$ is not a vertex cover, obtaining a contradiction. Hence, $S$ is an independent set. □

So, the above lemma says that the "complement" of an independent set is a vertex cover, and vice versa. With this lemma, proving the following **NP**−completeness is straightforward.

**Theorem 28.** *VCOVER is* **NP**−complete.

*Proof.* VCOVER is in **NP**, as we can easily devise a NTM that, given a graph $G$ and an integer $k$, first guesses a subset $VC$ of $V$ and then verifies that $|VC| \geq k$ and that each edge in $E$ is covered by $VC$.

To show that VCOVER is **NP**−hard we reduce IS to VCOVER. The reduction must convert a pair $(G, k)$ to a pair $(G', k')$ such that if $G$ has an independent set with at least $k$ nodes, $G'$ has a vertex cover with at most $k'$ nodes, and vice versa.

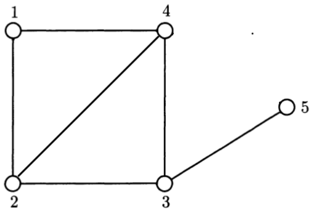
The construction is very simple. We let $G' := G$, and $k' = |V| - k$. This is clearly feasible in polynomial time.

Assume $G$ has an independent set $S$ with $|S| \geq k$. Then, by Lemma 1, $VC = V \setminus S$ is a vertex cover of $G$ (and thus of $G'$). Moreover, since $|S| \geq k$, $|VC| = |V \setminus S| \leq |V| - k = k'$.

Similarly, if $G'$ has a vertex cover $VC$ with $|VC| \leq k'$, then, by Lemma 1, $S = V \setminus VC$ is an independent set of $G'$ (and thus of $G$). Moreover, since $|VC| \leq k'$, $|S| = |V \setminus VC| \geq |V| - k' = k$. □

## 13.3 Clique

We now consider one last language. This is again a language over graphs. Consider an undirected graph $G = (V, E)$. A *clique* of $G$ is a set of nodes $C$ that are all connected to each other with an edge, i.e., for each pair $u, v$ of *distinct* nodes of $C$, $\{u, v\} \in E$.



Consider the undirected graph above. The set of nodes $C_1 = \{1, 2, 4\}$ is a clique. Also the set $C_2 = \{2, 3, 4\}$ is a clique. However, the set $\{1, 2, 3, 4\}$ is not a clique, since the two distinct nodes 1 and 3 are not connected by an edge.

Note that an undirected graph $G$ always has a clique: the empty set is trivially a clique. Hence, asking whether $G$ has a clique is not an interesting question. Rather, we are interested in large cliques, i.e., with at least a certain number of nodes.

Cliques are a very useful tool to model groups of friends in social networks. Indeed, if the nodes of a graph represent people and an edge between two people represent the fact that these two people are friends, a clique of a certain size is a group of people that all know each other, i.e., a group of very close friends.

$$\begin{aligned} \text{CLIQUE} \quad = \quad \{(G,k) \mid \quad & G \text{ is an undirected graph} \\ & \text{with a clique } C \text{ such that } |C| \geq k\}. \end{aligned}$$

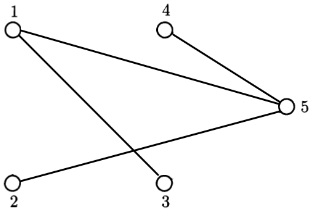
To prove that CLIQUE is **NP**−hard we exploit the following observation: an independent set in a graph $G$ corresponds to a clique in the graph where we "complement" the edges, and vice versa.

**Definition 31.** *The* complement *of an undirected graph* $G = (V, E)$ *is the graph* $\bar{G} = (\bar{V}, \bar{E})$*, where* $\bar{V} = V$*, and*

$$\bar{E} = \{\{u, v\} \mid u, v \text{ are distinct nodes of } G \textit{ such that } \{u, v\} \notin E\}.$$

The above definition essentially says that the complement of an undirected graph $G$ is a graph $\bar{G}$ over the same nodes of $G$, such that two distinct nodes $u, v$ are adjacent in $\bar{G}$ iff they are not adjacent in $G$.

For example, the following graph is the complement of the graph we have seen above.



You can see that the set of nodes $\{1, 2, 4\}$ is a clique in the original graph, but it is an independent set in the complement. Vice versa, the set of nodes $\{1, 3\}$ is an independent set in the original graph, but it is a clique in the complement graph.

**Theorem 29.** *CLIQUE is* **NP**−complete*.*

*Proof.* CLIQUE is clearly in **NP**, as given $G$ and $k$, it is enough to guess at most $|V|$ nodes, and then verify that we guessed at least $k$ nodes, and that all pairs are connected.

The reduction is from IS. Given a pair $(G, k)$, we construct $(G', k')$, where $k' := k$ and $G' := \bar{G}$ is the complement of $G$. Constructing $G'$ requires copying $|V|$ nodes and adding at most $|V|^2$ edges. Setting $k' = k$ is trivial. Hence, the reduction is in polynomial time. We now show that the above procedure is a reduction.

If $G$ has an independent set $S$ with at least $k$ nodes, i.e., no two nodes in $S$ are connected, then in $G'$, necessarily all distinct pairs of nodes in $S$ are connected, and thus $S$ is a clique with at least $k = k'$ nodes in $G'$.

Conversely, if $G'$ has a clique $C$ with at least $k'$ nodes, i.e., all pairs of distinct nodes in $C$ are connected, then no pair of distinct nodes of $C$ are connected in $G$. Thus, $C$ is an independent set with at least $k' = k$ nodes in $G$. □

# 14 More NP−complete languages

We keep exploring the intricate net of **NP**−complete languages. In this lecture we will discuss a problem related to optimization, and another problem over graphs.

## 14.1 Binary Integer Programming

The first language we consider regards the Binary Integer Programming problem. Assume you are given $m \times n$ integers $a_{ij} \in \mathbb{Z}$ and $m$ integers $b_i \in \mathbb{Z}$. The question we want to answer is: does the following system of linear inequalities

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

have a solution with $x_1, \ldots, x_n \in \{0, 1\}$?

The above is the same as saying, given a matrix $A \in \mathbb{Z}^{m \times n}$ and a vector $\bar{b} \in \mathbb{Z}^m$, is there a vector $\bar{x} \in \{0,1\}^n$ such that $A \cdot \bar{x} \leq \bar{b}$?

The language we are going to focus on, then is:

$$\text{BIP} = \{(A, \bar{b}) \mid A \in \mathbb{Z}^{m \times n}, \bar{b} \in \mathbb{Z}^m, \text{ and there is } \bar{x} \in \{0,1\}^n \text{ such that } A \cdot \bar{x} \leq \bar{b}\}.$$

**Theorem 30.** *BIP is* **NP**−complete.

*Proof.* To see that BIP is in **NP**, given $A \in \mathbb{Z}^{m \times n}$ and $\bar{b} \in \mathbb{Z}^m$, we can use a NTM to guess a 0/1 value for each variable in $\bar{x} = (x_1, x_2, \ldots, x_n)$, and check that each inequality of the system $A \cdot \bar{x} \leq \bar{b}$ is satisfied. The above operations require $n$ steps to guess $\bar{x}$ and then $m$ checks: one for each inequality. Checking one inequality requires simple addition and multiplication of polynomially many terms.

We now show that BIP is **NP**−hard by reducing EXACT-3-SAT to BIP in polynomial time. For this, we need to convert a Boolean formula $\varphi$ where each clause has exactly 3 literals to a system of inequalities. The conversion must be carried out in polynomial time, and $\varphi$ is satisfiable iff the system admits a solution in $\{0, 1\}$.

Again, to simplify the discussion, we present the reduction with an example, and the general procedure will become clear, afterwards.

Consider the following Boolean formula:

$$\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4).$$

Choosing a truth value for the Boolean variables in $\varphi$ can be seen as choosing a 0/1 value to the arithmetic variables of our system. So, for each Boolean variable $x_i$ of $\varphi$, our system of inequalities will have a corresponding *arithmetic* variable, that we call $y_i$, rather than $x_i$, just to avoid confusion. In our example, our system will have the arithmetic variables $y_1, y_2, y_3, y_4$. When a Boolean variable $x_i$ is true, then we model this by setting the arithmetic variable $y_i = 1$, and when false, we use $y_i = 0$.

To model the fact that a clause of $\varphi$ must be true, i.e., at least one of its literals is true, we use an inequality stating that the sum of its literals must be at least 1. If a literal is a simple Boolean variable $x_i$, we use the arithmetic variable $y_i$, if the literal is a negated variable $\neg x_i$, then we use $(1 - y_i)$ to invert the value in $y_i$. So, for example, the clause $(x_1 \vee x_2 \vee \neg x_3)$ corresponds to the inequality

$$y_1 + y_2 + (1 - y_3) \geq 1.$$

We do the above for all clauses of $\varphi$, obtaining the system:

$$\begin{cases} y_1 + y_2 + (1 - y_3) & \geq 1 \\ (1 - y_2) + y_3 + (1 - y_4) & \geq 1 \end{cases}$$

Note that the above system is using the $\geq$ comparison and some constants are still on the left-hand side, while instances of BIP are about inequalities with $\leq$ and only one constant on the right. This is easy to obtain. We move all constants from left to right, changing their sign, and then multiply by -1 both sides, obtaining an inequality using $\leq$. In our example, the final system is:

$$\begin{cases} -y_1 - y_2 + y_3 & \leq 0 \\ y_2 - y_3 + y_4 & \leq 1 \end{cases}$$

The reduction is clearly polynomial, as it constructs $m$ inequalities, one for each clause of $\varphi$ and each inequality contains at most $n + 1$ coefficients.

We now prove the above procedure is a reduction. For this, we use the first form of inequalities using $\geq$ as a reference, as it is easier to argue about.

Assume $\varphi$ is satisfiable, i.e., there is a truth assignment $\tau : \{x_1, \ldots, x_n\} \rightarrow \{\text{true}, \text{false}\}$ assigning each Boolean variable of $\varphi$ either true or false, such that $\tau$ makes $\varphi$ true. Then, consider the solution to our system where we set $y_i = 1$ if $\tau(x_i) = \text{true}$, and set $y_i = 0$ when $\tau(x_i) = \text{false}$.

Since $\tau$ satisfies $\varphi$, every clause has at least one true literal, thus at least one of the three expressions summed on the left-hand side of the inequality corresponding to the clause is at least 1. Since whatever arithmetic variable $y_i$ we consider, $y_i$ or $(1 - y_i)$ are always non-negative, the overall left-hand sum is at least 1, and thus the inequality is satisfied. This holds for all clauses, and thus all inequalities are satisfied.

Assume our system has a solution, i.e., there is an assignment to each arithmetic variable $y_i$ to 0 or 1 that satisfies all inequalities. Consider the truth assignment $\tau$ such that $\tau(x_i) = \text{true}$ if the arithmetic variable $y_i$ is 1, and false, otherwise. For an inequality to be satisfied, at least one of the 3 expressions that are summed on its left-hand side must be 1. Thus, at least one literal of the corresponding clause must be true, and thus the clause is true. This holds for all inequalities, and thus all clauses are satisfied by $\tau$. □

**Remark.** The more general integer programming (IP) problem is defined in the same way as BIP, but the variables $x_1, \ldots, x_n$ can take values from $\{0, 1, 2, 3, \ldots\}$, and not just $\{0, 1\}$. BIP is a special case of IP, because the fact that each variable $x_i$ must be in $\{0, 1\}$ is just a different way of stating that the inequalities $x_i \geq 0$ and $x_i \leq 1$ hold. Thus, we immediately conclude that IP is also **NP**−hard (i.e., reduce BIP to IP by adding the inequalities $x_i \geq 0$ and $x_i \leq 1$, for each variable $x_i$).

However, proving that IP is in **NP** is not as easy. The main problem is that it is not clear how large is the number our NTM should guess for each variable $x_i$. In BIP, $x_i$ can only be 0 or 1, but in IP, variables in a solution could, in principle, take arbitrarily large numbers that would even require exponentially many bits (or double exponential, or even more, as there is no clear bound on how many bits are needed). Thus, our NTM has no clear time bound, let alone polynomial.

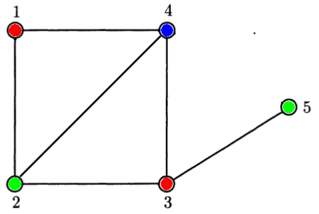
However, one can show that if a system of inequalities admits a solution over the non-negative integers, then it admits a solution over the non-negative integers where each $x_i$ is a number encoded with only polynomially many bits (see "On the Complexity of Integer Programming", 1981, Christos H. Papadimitriou). A consequence of this result is that our NTM can only focus on numbers using polynomially many bits, and thus IP is in **NP**. We will not see the proof, but just state the result.

**Theorem 31.** *IP is* **NP**−complete.

## 14.2 Vertex Coloring

The next language we consider is again over graphs. Given an undirected graph $G = (V, E)$ and an integer $k$, the question is whether we can color, using at most $k$ colors, each node of $G$ in such a way that no two adjacent nodes end up colored with the same color.

For example, if we have $k = 3$ colors, say red, green and blue, we can color the graph below in the following way



where no two adjacent nodes have the same color. However, if we are given only $k = 2$ colors, there is no way to color the above graph without coloring two adjacent nodes with the same color.

We now formalize the notion of coloring.

**Definition 32.** *Consider an undirected graph $G = (V, E)$ and an integer $k$. A* k-coloring *of $G$ is a function $f : V \to \{1, 2, \ldots, k\}$, assigning each node to a color in $\{1, 2, \ldots, k\}$, such that $f(u) \neq f(v)$, for each edge $\{u, v\} \in E$.*

Our language is thus:

$$\text{VCOL} = \{(G, k) \mid G \text{ is an undirected graph that admits a k-coloring}\}.$$

As an example application of VCOL, the nodes of $G$ can be seen as Wireless turrets in an area, and an edge between two turrets means that the turrets are close enough to interfere with each other, if they broadcast with the same frequency. The integer $k$ represents the number of available frequencies a turret can broadcast with. Deciding if a k-coloring exists means understanding whether we can assign a frequency to each turret without causing interferences.

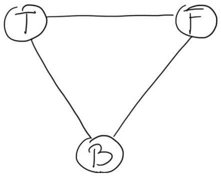
**Theorem 32.** *VCOL is* **NP**−complete.

*Proof.* To place VCOL in **NP**, we can guess a color in $\{1, 2, \ldots, k\}$, for each node, and then verify that no two nodes have the same color. Each color is a number $1 \leq i \leq k$, thus requires at most the same number of bits of $k$, and we need to choose one color per node. So, the machine guesses a color for each node in $O(||k|| \cdot n)$ steps, where $||k||$ is the size in bits of $k$ and $n$ is the number of nodes. Checking that chosen colors describe a $k$-coloring requires trying all possible edges, i.e. in time $O(|E|)$.

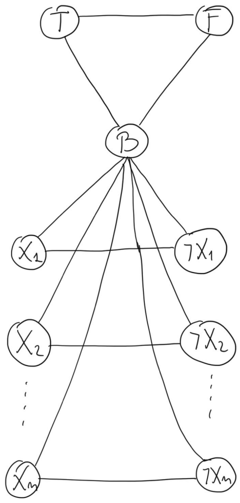
We now show that VCOL is **NP**−hard by reducing EXACT-3-SAT to VCOL. We must convert a Boolean formula $\varphi$ in CNF, where each clause has exactly 3 literals to a pair $(G, k)$. Moreover, the construction must be polynomial, and $\varphi$ is satisfiable iff $G$ admits a $k$-coloring.

We construct a pair $(G, k)$ with $k = 3$ colors.

We first introduce in $G$ three nodes, connected in a triangle, labeled $T$, $F$ and $B$, as shown below. In any 3-coloring of our graph, these three nodes will have different colors. We will call the color that $T$ will have in a 3-coloring as the color "true". Similarly, we call the color of node $F$ "false", and the color of $B$ "base".

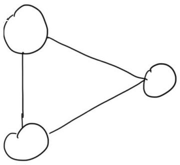


We represent each Boolean variable $x_i$ of $\varphi$ with two nodes labeled with $x_i$ and its negation $\neg x_i$. We want that these two nodes only get assigned the "true" or the "false" color, thus we connect them to the node $B$, forcing their color to be either the one of node $T$ or the one of node $F$. Moreover, since it is not possible for a variable $x_i$ and its negation $\neg x_i$ to have the same truth value, we also connect the nodes $x_i$ and $\neg x_i$ with an edge.

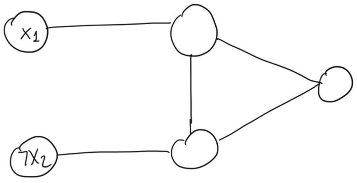


So, with the current state of the graph, any 3-coloring will assign different colors to each $x_i$ and its negation, and the only colors are the "true" or the "false" color. So, a 3-coloring in the above graph essentially describes a truth assignment of the Boolean variables of the formula $\varphi$.

Now, we need to model the fact that some literals make certain clauses true. For this, we show how to model the OR of two literals. The OR operation is simply a circuit that has two inputs and one output. We model this circuit with the following triangle:



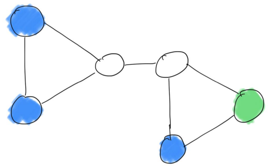
The right-most node represents the output of the OR. The left-most nodes are the two inputs. If we want to compute the OR of, e.g., the literals $x_1$, $\neg x_2$, we simply connect these two nodes to the input nodes of the circuit:



Note that if all two input literals are colored with "false", then the right-most node **must** be colored with "false", since the two intermediate nodes must have different colors, and both must be different than "false". So, one intermediate node will be colored with “true” and the other with “base”. Hence, the only available color remaining for the right-most node is “false”.

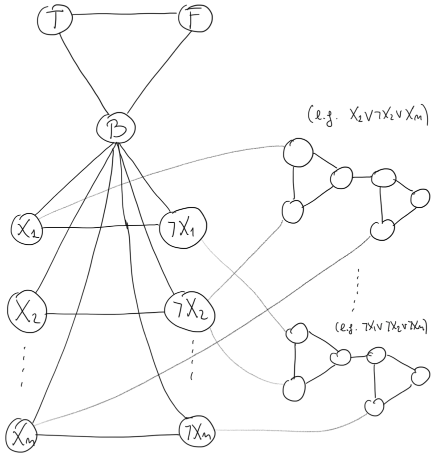
Moreover, if at least one of the two input literals is colored with “true”, then **there exists** a way to color the output node with “true”. That is, for *just one* of the “true” input nodes, color the intermediate node connected to it with ”false” and the other intermediate node with the “base” color. Thus, the right-most node must be colored with “true”.

Now that we have a way to simulate an OR between two literals, we can simulate an OR between three literals (a clause) by combining two ORs in a “Clause-gadget”:

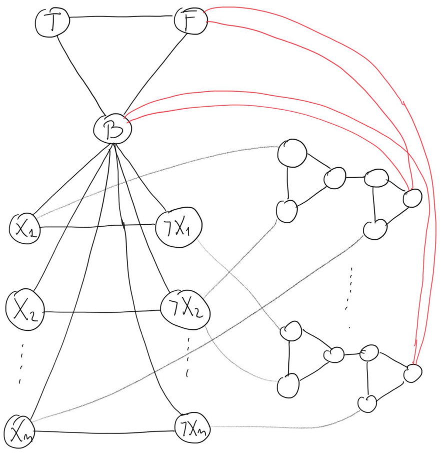


where the nodes highlighted in blue are the input nodes, and the one highlighted in green is the output node. The first triangle computes the OR of the first two literals, and the second triangle computes the OR of the result of the first OR with the third literal.

We can now complete our graph. For each clause of the Boolean formula $\varphi$, we add one Clause-gadget, and each node corresponding to a literal of the clause is connected to the input nodes of the Clause-gadget.



The only part remaining is to force each clause to be true. For this, we connect the output node of each Clause-gadget to both the $F$ and the $B$ nodes. The final graph is:



Let $n$ be the number of variables of $\varphi$ and $m$ the number of its clauses. The above construction adds 3 nodes in the top triangle, $2n$ nodes (2 for each variable), and $m$ Clause-gadgets, each with 6 nodes. So, overall $O(n+m)$ nodes. Also the number of edges is polynomial.

We now prove the above procedure is a reduction.

Assume $\varphi$ is satisfiable, i.e., there is a truth assignment $\tau : \{x_1, \dots, x_n\} \rightarrow \{\text{true}, \text{false}\}$ that makes every clause of $\varphi$ true. Then, consider a coloring where we color node $T$ with "true", $F$ with "false" and $B$ with "base". Moreover, if $\tau(x_i) = \text{true}$, we color node with label $x_i$ with "true" and $\neg x_i$ with "false", and vice versa. This is a valid 3-coloring, according to the left-part of the graph $G$. Finally, since each clause in $\varphi$ has at least one true literal because of $\tau$, from the previous discussion on the Clause-gadgets, the output node of each Clause-gadget can be 3-colored in such a way that the output node is colored with "true". Thus, overall, $G$ has a 3-coloring.

Assume that $G$ has a 3-coloring. By construction of $G$, nodes $T$, $F$ and $B$ have three different colors. Let "true" be $T$'s color and "false" $F$'s color. Thus, two nodes $x_i$ and $\neg x_i$ have different colors, and they can only be either "true" or "false". Moreover, if we consider one Clause-gadget, its output node must be colored with "true" since this node is connected to $F$ and $B$. This means that at least one of the nodes connected to an input node of the Clause-gadget must be colored with "true". If this was not the case (i.e., they are all colored with "false"), we said that the output node must be colored with "false", which is not the case. Thus, the colors given to each node labelled with some $x_i$ define a truth assignment of the variables of $\varphi$ that satisfy $\varphi$. □

**Remark.** Note that our reduction always uses 3 colors, regardless of the shape of $\varphi$. Thus, Vertex Coloring is **NP**−hard even if the number of colors $k$ is not part of the input, but it is implicitly assumed to be 3. That is, the language

$$\text{3-VCOL} = \{G \mid G \text{ is an undirected graph admitting a 3-coloring}\}$$

is **NP**−complete.

# 15 Alternative definition of NP

By now, you should have a good understanding of the complexity class **NP**, and what are some of the hardest problems in the class. You should also have noted that every time we had to place a language in **NP**, the story is always the same. Given a string $w$, to verify whether $w$ is in our language:

1. **Guess** some data on the tape, within **polynomially** many steps, and
2. **check**, within **polynomially** many steps, that what we guessed witnesses the fact that $w$ is in the language.

That is, what our NTM guesses is a *certificate* of the fact that $w$ is indeed in the language. For example, if we consider SAT, the certificate is a truth assignment $\tau$, and after guessing one, we simply need to check that it satisfies the input Boolean formula $\varphi$. If such an assignment exists, the NTM will find it.

Another example is IS. Given as input a pair $(G, k)$ our machine guesses a set of nodes $S$ of $G$, and then verifies that $|S| \geq k$ and $S$ is an independent set. The set $S$ is our certificate.

In this lecture, we are going to show that the occurrence of this pattern, in the context of **NP** languages, is not a coincidence, but rather, it is an alternative way to characterize the complexity class **NP**. That is, although our current definition of **NP** does not require our NTMs to follow this pattern, we are going to see that any language in **NP** is characterized by certain kinds of certificates that can be guessed in polynomial time, and verified in polynomial time.

First, we need to formalize what we mean that a language $\mathcal{L}$ can be characterized by polynomially guessable and verifiable certificates. For this, we define a "new" class that collects all such languages. We call this class **PC** (from *P*olynomial and *C*ertificates); in what follows, we use $\{0,1\}^{\leq n}$ to denote the set of all binary strings of length *at most* $n$, i.e., $\{0,1\}^{\leq n} = \bigcup_{i=0}^{n} \{0,1\}^i$.

**Definition 33.** *A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class* **PC** *if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$, and a polynomial-time (deterministic) TM $M$ such that, for every string $w \in \{0,1\}^*$,*

$$w \in \mathcal{L} \quad \textit{iff} \quad \textit{there is a string (a certificate) } u \in \{0,1\}^{\leq p(|w|)} \textit{ such that } M \textit{ accepts } (w, u).$$

So, intuitively, when a language $\mathcal{L}$ is in **PC** it means that the presence of a string $w$ in $\mathcal{L}$ is characterized by the existence of some "small" certificate $u$ (where, by small, we mean polynomial in $|w|$), that once given, allows a deterministic TM to confirm that $w$ is in the language, in polynomial time. However, if $w$ is not in the language, no such a certificate exists.

The above definition is very reminiscent of how the NTMs we devised for our **NP** languages look like.

**Example 8.** *Consider, for example, the independent set language:*

$$IS = \{(G, k) \mid G \textit{ is an undirected graph with an independent set } S \textit{ with } |S| \geq k\}.$$

*Our certificates are sets $S$ of nodes of $G$. In fact, $S$ is of size polynomial w.r.t. the length of the encoding of $(G, k)$, as $S$ can contain at most $n$ nodes, where $n$ is the number of nodes in $G$.*

*Our deterministic TM $M$ confirming that a pair $(G, k)$ is in IS takes as input $(G, k)$ and a certificate $S$, and simply verifies that $S$ is an independent set of $G$ with at least $k$ nodes. The latter can be carried out with polynomially many steps.*

*Thus, if $(G, k) \in$ IS, then a set $S$ as defined above exists such that $M$ accepts $(G, k, S)$. If, however, $(G, k) \notin$ IS, no such a certificate exists, i.e., for every set $S$, the TM $M$ will not accept $(G, k, S)$.*

So, in a sense, the above definition explicitly splits the two phases of our Guess and check NTMs. The "guess" phase is modelled via certificates of polynomial length, and the "check" is modelled by the (deterministic) TM $M$. Let us consider another example.

**Example 9.** *Consider the SAT language:*

$$SAT = \{\varphi \mid \varphi \text{ is a Boolean formula in CNF that is satisfiable}\}.$$

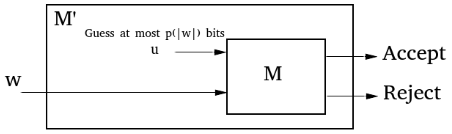
*Let $X = \{x_1, \dots, x_n\}$ be the Boolean variables of $\varphi$. Our certificates are truth assignments $\tau : X \to \{true, false\}$. A truth assignment $\tau$ is clearly of polynomial size w.r.t. the length of the encoding of $\varphi$, as we can encode $\tau$ as a string of bits, where the bit in position $i$ denotes the truth value given to variable $x_i$. So, its length is $n$.*

*Our deterministic TM $M$ confirming that a formula $\varphi$ is in SAT takes $\varphi$ and a truth assignment $\tau$ as input, and simply verifies that $\tau$ satisfies $\varphi$. The latter can be obviously carried out with polynomially many steps.*

As you might have now realized, the class **PC** is nothing else than **NP**. That is, **PC** = **NP**. We now proceed to prove this result.

**Theorem 33.** **PC** = **NP**.

*Proof.* To prove that two sets are the same, it is enough to show that one is included in the other, and vice versa. We start by showing that **PC** $\subseteq$ **NP**. For this, we must show that any language $\mathcal{L} \in$ **PC** is also in **NP**. Let $\mathcal{L}$ be some language in **PC**, and let $p : \mathbb{N} \to \mathbb{N}$ be its polynomial and $M$ its deterministic TM. We now show that there exists an NTM $M'$ that decides $\mathcal{L}$ in polynomial time. Our machine has the following shape:



1. $M'$ first guesses $p(|w|)$ bits in a secondary tape, essentially constructing a certificate $u$. Since $p(|w|) \in O(|w|^c)$, this phase requires polynomially many steps.

2. $M'$ executes the control of $M$ with input $w$ and $u$. Since $M$ is a polynomial-time TM, $M'$ requires polynomially many steps, overall.

We now show that $M'$ decides $\mathcal{L}$.

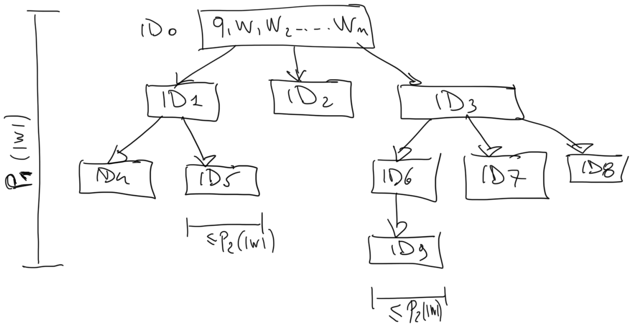
Assume that $w \in \mathcal{L}$. Thus, since $\mathcal{L} \in \mathbf{PC}$, there exists $u \in \{0,1\}^{p(|w|)}$ such that $M$ accepts $(w,u)$. Thus, among all strings $u$ that $M'$ guesses, there is such a string such that $M$ accepts $(w,u)$, and thus $M'$ accepts $w$.

If, instead, $w \notin \mathcal{L}$, since $\mathcal{L} \in \mathbf{PC}$, for every string $u \in \{0,1\}^{p(|w|)}$, $M$ does not accept $(w,u)$. Since $M$ is a polynomial-time TM, it actually means that $M$ *rejects* $(w,u)$. Thus, whatever string $u$ the NTM $M'$ guesses, $M$ will always reject $(w,u)$, and thus $M'$ rejects $w$.

We now show that $\mathbf{NP} \subseteq \mathbf{PC}$. For this, we must show that any language $\mathcal{L} \in \mathbf{NP}$ is also in $\mathbf{PC}$, i.e., any language $\mathcal{L} \in \mathbf{NP}$ has "small"certificates that can be verified in polynomial time.

Let $\mathcal{L}$ be a language in $\mathbf{NP}$, and thus let $M_{\mathcal{L}}$ be the NTM that decides $\mathcal{L}$ in polynomial time. Note that by definition of $\mathbf{NP}$, $M_{\mathcal{L}}$ does not necessarily follow the guess and check pattern, but can do anything, as far as it requires polynomially many steps, and decides $\mathcal{L}$. So, what are our certificates?

To answer this question, let us consider a string $w = w_1 \cdots w_n$, and let us have a look at the computation tree of $M_{\mathcal{L}}$ with input $w$.



Since $M_{\mathcal{L}}$ requires polynomially many steps w.r.t. $|w|$ in *any* of its computation paths, there is a polynomial $p_1 : \mathbb{N} \to \mathbb{N}$ such that $p_1(|w|)$ is the maximum number of IDs in a computation path. Moreover, when moving from one ID to the other, $M_{\mathcal{L}}$ can only move the head of one cell. Thus, since $M_{\mathcal{L}}$ performs at most $p_1(|w|)$ steps in a path, the size of each ID is also polynomial w.r.t. $|w|$. Hence, there is a polynomial $p_2 : \mathbb{N} \to \mathbb{N}$ such that $p_2(|w|)$ is the maximum size (in bits) of an ID.[17]

We conclude that a path

$$\mathrm{ID}_0\mathrm{ID}_1 \cdots \mathrm{ID}_k$$

[17] The polynomial $p_2$ is different, in principle, from $p_1$, since to encode an ID, we also need to store the state, besides the content of the tape.

of the computation tree of $M_{\mathcal{L}}$ with input $w$ can be stored in $p(|w|) = p_1(|w|) \cdot p_2(|w|)$ bits, where $p$ is a polynomial.

To show that $\mathcal{L} \in \mathbf{PC}$, we use, for a string $w$, the paths of the computation tree of $M_{\mathcal{L}}$ with input $w$ as certificates.

What is left to discuss is the shape of the (deterministic) TM $M$ that is in charge of confirming, given $w$ and a certificate $u$, whether $w$ is in $\mathcal{L}$. The TM $M$ is quite simple, as it only performs the following steps. Assume $(w, u)$, where $w = w_1 \cdots w_n$, and $u = ID_0, ID_1, \ldots, ID_m$, is the input to $M$:

1. Check that $ID_0 = q_1 w_1 \cdots w_n$. If this is not the case, reject, otherwise continue.
2. for each $i := 0$ to $m - 1$, do
   (a) If $ID_i$ does not yeld $ID_{i+1}$, according to the transition function of $M_{\mathcal{L}}$, reject, otherwise continue.
3. If $ID_m$ contains the accepting state of $M_{\mathcal{L}}$, then accept, otherwise reject.

So, essentially $M$ simply verifies that $u$ encodes an accepting path of the computation tree of $M_{\mathcal{L}}$ with input $w$.

Step 1 clearly requires polynomially many steps, as it is enough to scan $ID_0$, which contains polynomially many bits, as discussed before.

The for loop requires $m$ iterations, where $m$ is the length of the path, which is polynomial.

The "if" inside the for loop simply requires *i)* searching $ID_i$ for a state $q$ with a symbol $\alpha$ on the right (requiring at most $|ID_i|$ steps); *ii)* verify that there is $(q', \beta, *) \in \delta(q, \alpha)$ such that $ID_{i+1}$ is obtained from $ID_i$ by the rule $(q, \alpha) \to (q', \beta, *)$, where $* \in \{L, R, S\}$. The latter requires scanning $\delta$ which is of constant size (i.e., it is independent of the size of $w$), and requires scanning $ID_{i+1}$ in at most $|ID_{i+1}|$ steps. Hence, everything is polynomial.

The last step is clearly polynomial as well.

It remains to show that $\mathcal{L} \in \mathbf{PC}$.

Assume $w \in \mathcal{L}$. Then, the computation tree of $M_{\mathcal{L}}$ with input $w$ has a path ending in an accepting state. Thus, this path is a certificate $u$ such that the TM $M$ discussed above accepts $(w, u)$.

If, instead, $w \notin \mathcal{L}$, every path of the computation tree of $M_{\mathcal{L}}$ with input $w$ is rejecting. Thus, every certificate (i.e., path) $u$ is such that the TM $M$ rejects $(w, u)$. Thus, there is no certificate $u$ such that $M$ accepts $(w, u)$. □

**Remark.** With the above theorem we have shown that the complexity class **NP** has an alternative definition, in terms of certificates. This is not only interesting on its own right, as it provides an alternative tool for placing a language in **NP** (i.e., just explain what your certificates are, and devise a TM that confirms the inputs string $w$ together with the certificate), but also gives some more insights on why most researchers believe that $\mathbf{P} \neq \mathbf{NP}$ (besides the fact that after half a century, no polynomial-time algorithm is known yet, for any of the known $\mathbf{NP}-$complete languages).

Citing the “Philosophical Argument” by Scott Aaronson.[18]

*“If* $\mathbf{P} = \mathbf{NP}$*, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in ’creative leaps’,* ***no fundamental gap between solving a problem and recognizing the solution once its found****. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett. Its possible to put the point in Darwinian terms: if this is the sort of universe we inhabited, why wouldnt we already have evolved to take advantage of it? [...]”*

[18] `https://www.scottaaronson.com/blog/?p=122`

# 16 Cook's Theorem

Some lectures ago, I promised that at some point, we would finally prove that SAT is **NP**−complete. Indeed, all **NP**−hardness results we proved up to this point heavily rely on the assumption that SAT is **NP**−complete. We have seen that knowing that at least one language (e.g., SAT) is **NP**−complete can greatly help in the process of proving that other languages are **NP**−complete.

However, it is now time to add this missing part to our discussion. The main challenge of proving that SAT is **NP**−complete clearly lies in the proof of its **NP**−hardness, as we cannot now rely on any known **NP**−hard language, because there are no such languages yet, if we do not prove that SAT is **NP**−hard!

So, the only way we have for proving that SAT is **NP**−hard is by using the definition. That is, we must prove that *every language* in **NP** reduces to SAT in polynomial time.

Let us recall, once again, the definition of SAT, and make some observation.

$$\text{SAT} = \{\varphi \mid \varphi \text{ is a Boolean formula in CNF that is satisfiable}\}.$$

Recall that a Boolean formula $\varphi$ is in CNF (conjunctive normal form) if it is of the form

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each $C_i$ is a disjunction of literals (i.e., variables and/or their negations). Recall also that in SAT, we do not assume any bound on the number of literals in a clause, like we do for 3-SAT, or EXACT-3-SAT. So, clauses can have an arbitrary number of literals.

Finally, we recall that an implication $A \rightarrow B$ is just a shorthand for the formula $\neg A \vee B$. We will use this fact in the proof.

**Theorem 34** (Cook's Theorem)**.** *SAT is* **NP**−complete.

## Proof

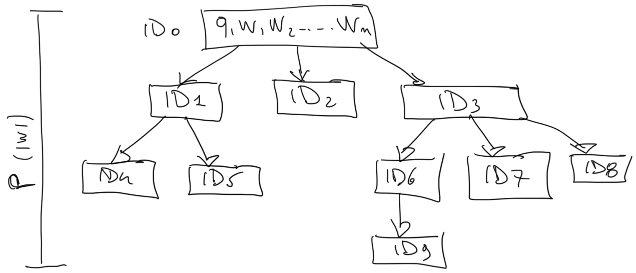
We have already shown that SAT is in **NP**. Let us focus on the hardness. We have to show that every language $\mathcal{L} \in \mathbf{NP}$ reduces to SAT in polynomial time. So, for each language $\mathcal{L} \in \mathbf{NP}$, we must show that there exists a reduction $T_{\mathcal{L}}$ that converts strings $w$ to Boolean formulas $\varphi$ in such a way that $w \in \mathcal{L}$ iff $\varphi$ is satisfiable. Moreover, the procedure must require no more than polynomially many steps. Consider some arbitrary language $\mathcal{L} \in \mathbf{NP}$, and let

$$M = (Q, \Sigma, \Gamma, \delta, q_1, q_{\texttt{accept}}, q_{\texttt{reject}})$$

be the NTM that decides $\mathcal{L}$ in polynomial time.

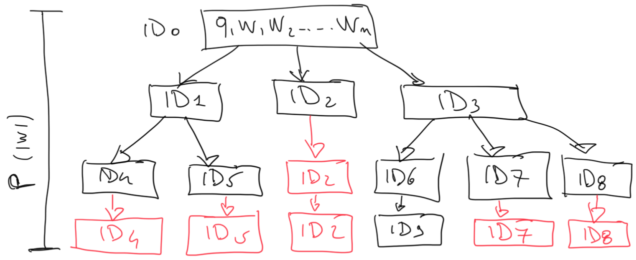
### Observations

To convert a string $w$ to a Boolean formula $\varphi$, we first make some observations on the paths and the number of cells used in the computation tree of $M$ with input $w = w_1 \cdots w_n$.



**Length of a path.** The length of a path in the tree is *at most* $p(n)$, where $p : \mathbb{N} \to \mathbb{N}$ is some polynomial.

Since different paths can have different lengths (but all bounded by $p(n)$), to simplify the discussion, we focus on a modified version of the tree, where each path that has length less than $p(n)$ is extended by connecting the leaf ID of the path to a copy of itself, until the path is of length $p(n)$.



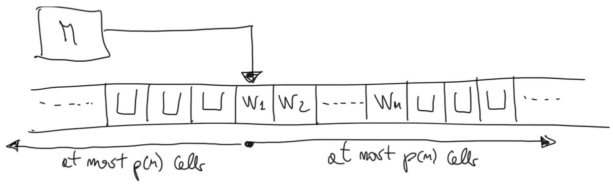
*Remark.* Note that this is not the computation tree of $M$ with input $w$, but we can treat it as such. In fact, there is a strong connection between the two: if a path ending in $q_{\texttt{accept}}$ exists in the computation tree of $M$, then a path ending in $q_{\texttt{accept}}$ also exists in this new tree. Moreover, if all paths end in $q_{\texttt{reject}}$, also all paths of the new tree end in $q_{\texttt{reject}}$.

So, you can see the new tree as the one representing the computation of $M$ with input $w$ with the additional condition that once $M$ reaches $q_{\texttt{accept}}$ (resp., $q_{\texttt{reject}}$), if the length of the path where this happens is less than $p(n)$, then $M$ "waits" in $q_{\texttt{accept}}$ (resp., $q_{\texttt{reject}}$) for the remaining time.

**Number of cells.** Regarding the number of cells that the TM $M$ can visit, this is at most $p(n)$, because from one ID to the next, the head can only move to one cell at the time.

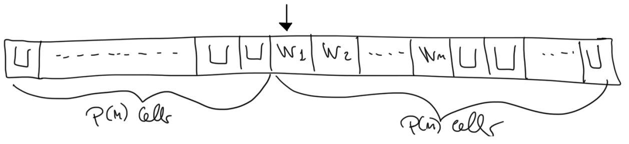
The size of an ID is larger than $p(n)$, because we also store the state, pointing to the current cell. Nonetheless, if we just focus on the cells visited, they are indeed at most $p(n)$.

So, to perform its computation, $M$ does not really need an infinite tape, but only a finite tape of adequate size. The question is, what is the adequate size?



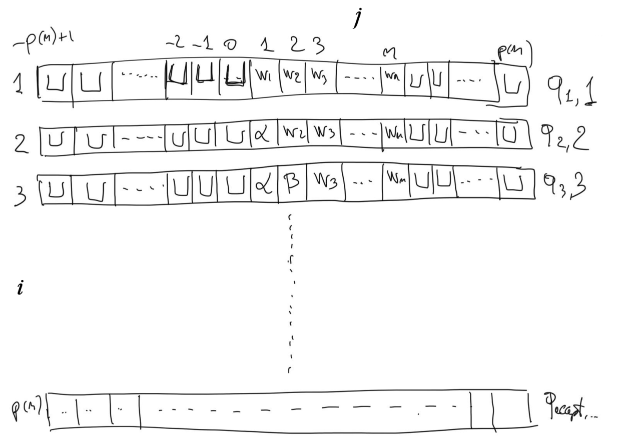
The adequate size would be $p(n)$ cells. However, since $M$'s head can move both to the right and to the left, part of the $p(n)$ visited cells could be, for example, on the right of the initial position, and the rest on the left. So, if we allocate $p(n)$ cells, in order for $M$ to have the correct amount of space on the right and the left, we should also know *where* to exactly place the input string, so to let $M$ have enough space in both directions.

To avoid this issue, we can simply allocate double the space, i.e., $2 \cdot p(n)$ cells, and let the input string start from the middle cell of the tape, as shown in the picture below. In this way, in the worst case, if $M$ always moves to the right, or always moves to the left, there will be enough cells.



So, we can assume $M$ works on a finite tape with $2 \cdot p(n)$ cells, where the input string starts from the middle cell and the head points to the middle cell.

**Representing an ID.** Our last observation is that an ID is essentially the combination of three elements: a tape with $2 \cdot p(n)$ cells, the current state, and the position of the head on the tape. Although we usually encode these three informations all together in the ID string, for the proof, we find more convenient to keep these three elements separated. So, every path in our modified tree can be visualized as in the picture below, where each step is represented by three elements: the finite tape, the current state, and the head position.



We number the cells of the finite tape from $-p(n) + 1$ to $p(n)$, where cell 1 is the middle cell. We also number the different steps in a path from 1 to $p(n)$ (recall that in the modified tree, every path is of length $p(n)$). We use the letter $i$ for the step number, and the letter $j$ for the cell number. We now proceed with our proof.

### The formula $\varphi$

The main goal is to construct a Boolean formula $\varphi$ in such a way that every truth assignment that satisfies $\varphi$ describes a path in our modified tree that ends in an accepting state. So, if such a path exists, a truth assignment satisfying $\varphi$ exists, and if no such a path exists, every truth assignment makes the formula $\varphi$ false.

**Boolean variables.** We want, with the Boolean variables of $\varphi$, to describe the computation of $M$ with input $w$ on a generic path of the modified tree. So, we need to keep track of which is the state at a certain step $i$, also what is the position of the head at step $i$, and what is the content of the finite tape at step $i$. For this, we use the following sets of variables.

1. **Current state.** For each step $1 \leq i \leq p(n)$, and for each state $q \in Q$, we introduce the variable
$$q^i.$$
Intuitively, if the variable $q^i$ is true, this represents the fact that at step $i$, the current state is $q$.

2. **Head position.** For each step $1 \leq i \leq p(n)$, and each cell position $-p(n) < j \leq p(n)$, we introduce the variable
$$h^{i,j}.$$
Intuitively, if $h^{i,j}$ is true, it means that at step $i$, the head of $M$ is positioned on cell number $j$.

3. **Content of each cell.** For each step $1 \leq i \leq p(n)$, each cell position $-p(n) < j \leq p(n)$, and each symbol of the alphabet $\alpha \in \Gamma$, we use the variable
$$\alpha^{i,j}.$$
Intuitively, if $\alpha^{i,j}$ is true, it means that at step $i$, the cell number $j$ contains the symbol $\alpha$.

4. **Unchanged cells.** We introduce one last set of variables. These variables will be useful when representing the transition of $M$ from one step to the other. In particular, for each step $1 \leq i \leq p(n) - 1$, and each cell number $-p(n) < j \leq p(n)$, we introduce the variable
$$u^{i,j}.$$
Intuitively, if $u^{i,j}$ is true, it means that when $M$ transitions from step $i$ to $i + 1$, the content of cell $j$ remains unchanged. This happens when the head is *not* positioned on cell $j$ and thus, its value is kept as is in the next step.

Note that we are using $|Q| \cdot p(n)$ variables for the states, $p(n) \cdot 2p(n)$ variables for the head position, $|\Gamma| \cdot p(n) \cdot 2p(n)$ variables for the cells content, and finally $(p(n) - 1) \cdot 2p(n)$ variables for the unchanged cells. Hence, polynomial.

Now, it is time to use the above variables to describe the computation of $M$ on input $w$. For this, we introduce different sets of Boolean expressions that, when ANDed ($\wedge$) together, will form the formula $\varphi$. Each set of expressions is dedicated to model a specific property of the computation of $M$ with input $w$.

**Consistency.** First of all, note that, for example, nothing prevents that two variables $\alpha^{i,j}$ and $\beta^{i,j}$ are both true in a satisfying truth assignment, with $\alpha, \beta \in \Gamma$. This, however, should not be allowed, as this would represent the fact that at step $i$, cell $j$ contains both $\alpha$ and $\beta$. Similarly, for example, we might have that all variables of the form $q^i$ are false, implying that at step $i$ there is no state. So, our formula $\varphi$ must prevent such things, and force every truth assignment satisfying $\varphi$ to be consistent.

1. ***M* is in exactly one state, in each step.** First, we introduce $p(n)$ expressions, one for each step $i$, to assure that at least one state is the current state at step $i$:
$$\bigvee_{q \in Q} q^i.$$
Then, we introduce $p(n) \cdot |Q|$ expressions, one for each combination of step $i$ and state $q$, to assure that if the current state in step $i$ is $q$, no other state can be the current state:[19]
$$q^i \rightarrow \bigwedge_{\hat{q} \in Q \setminus \{q\}} \neg \hat{q}^i.$$

[19] Recall that an implication $A \rightarrow B$ is just the formula $\neg A \vee B$.

The above are polynomially many expressions, and each contain polynomially many variables.

2. **$M$'s head is on exactly one cell, in each step.** First, we introduce $p(n)$ expressions, one for each step $i$, to assure that the head of $M$ must point to some cell, when in step $i$:
$$\bigvee_{-p(n)<j\leq p(n)} h^{i,j}.$$
Then, we introduce $p(n)\cdot 2p(n)$ expressions, one for each combination of step $i$ and cell number $j$, to assure that if the head of $M$ is on cell $j$, when in step $i$, then, no other cell has the head on it, in the same step.
$$h^{i,j}\rightarrow \bigwedge_{k\neq j} \neg h^{i,k}.$$
Also here, the number of expressions is polynomial, and they contain polynomially many variables.

3. **Each cell in $M$'s tape contains exactly one symbol.** First, we introduce $p(n)\cdot 2p(n)$ expressions, one for each combination of step $i$ and cell number $j$, to assure that the cell $j$ contains at least a symbol, when in step $i$:
$$\bigvee_{\alpha\in\Gamma} \alpha^{i,j}.$$
Then, we introduce $p(n)\cdot 2p(n)\cdot|\Gamma|$ expressions, one for each combination of step $i$, cell number $j$, and tape symbol $\alpha\in\Gamma$ to assure that if cell $j$ contains the symbol $\alpha$, when in step $i$, then no other symbol can occur in cell $j$, in the same step:
$$\alpha^{i,j}\rightarrow \bigwedge_{\beta\in\Gamma\setminus\{\alpha\}} \neg\beta^{i,j}.$$
Also here, the number of expressions and their size is polynomial.

4. **Cells that are unchanged from one step to the next must keep the same value.** We introduce $(p(n)-1)\cdot 2p(n)\cdot|\Gamma|$ expressions, one for each combination of step $1\leq i\leq p(n)-1$, cell number $j$, and tape symbol $\alpha\in\Gamma$, to assure that if cell $j$ contains the symbol $\alpha$, when in step $i$, and this cell is unchanged in the next step, then cell $j$ contains the symbol $\alpha$, also in the next step:
$$\alpha^{i,j}\wedge u^{i,j}\rightarrow \alpha^{i+1,j}.$$
Also here, the variables are polynomial.

If we put in AND ($\wedge$) all the expressions described in Items 1,2,3,4 above, we obtain the expression
$$\mathsf{Cons}.$$
The presence of $\mathsf{Cons}$ in $\varphi$ guarantees that every truth assignment that satisfies $\varphi$ must assign truth values to our variables in a consistent way.

**Initial ID.** We now need to force a truth assignment satisfying $\varphi$ to state that at step 1, the current state is $q_1$, the head is on cell 1, and the tape contains the string $w = w_1 \cdots w_n$ surrounded by blanks. For this, we need one expression:

$$\mathsf{Init} =$$

$$q_1^1 \wedge h^{1,1} \wedge \underbrace{\sqcup^{1,-p(n)+1} \wedge \cdots \wedge \sqcup^{1,0}}_{p(n) \text{ blanks before } w} \wedge \underbrace{w_1^{1,1} \wedge w_2^{1,2} \cdots \wedge w_n^{1,n}}_{\text{The string } w} \wedge \underbrace{\sqcup^{1,n+1} \wedge \cdots \wedge \sqcup^{1,p(n)}}_{p(n)-n \text{ blanks after } w}.$$

The expression contains $2 \cdot p(n)$ variables plus two variables (state and head position). Hence, a polynomial number.

**Transitions.** We now model the fact that $M$ transitions from one step to the other. Remember that $M$ is non-deterministic, and thus, it might decide to transition to one of many possible new IDs. The choice of one next ID or the other will depend on the truth assignment of our variables.

For each step $1 \leq i \leq p(n) - 1$, each cell number $j$ that is not the far-left or the far-right cell, and each pair $(q, \alpha)$ of a state $q$ and symbol $\alpha \in \Gamma$, assuming that, for example, $\delta(q, \alpha) = \{(\hat{q}_1, \beta, L), (\hat{q}_2, \gamma, R)\}$, we introduce the expression:

$$q^i \wedge h^{i,j} \wedge \alpha^{i,j} \rightarrow \bigwedge_{\ell \neq j} u^{i,\ell} \wedge$$
$$\left( \underbrace{(\hat{q}_1^{i+1} \wedge h^{i+1,j-1} \wedge \beta^{i+1,j})}_{\text{Choice 1}} \vee \underbrace{(\hat{q}_2^{i+1} \wedge h^{i+1,j+1} \wedge \gamma^{i+1,j})}_{\text{Choice 2}} \right).$$

Intuitively, the expression says that if at step $i$, the current state is $q$, the head is on cell $j$, and the symbol on cell $j$ is $\alpha$, then

- Every cell that is not $j$ remains unchanged in the next step $i + 1$, and
- at the next step $i + 1$, either $\hat{q}_1$ is the new state, the head is on cell $j - 1$, and the cell $j$ contains $\beta$, or $\hat{q}_2$ is the new state, the head is on cell $j + 1$, and the cell $j$ contains $\gamma$.

If the transition $\delta(q, \alpha)$ contains more choices, it should be now clear how we map them in an expression like the one above.

There is one last kind of transition to be modeled, which is the fake transition that "waits" in the accepting/rejecting state, until the very last step. For this, it is like we have, for each symbol $\alpha \in \Gamma$, the (not properly valid) transitions $\delta(q_{\texttt{accept}}, \alpha) = (q_{\texttt{accept}}, \alpha, S)$ and $\delta(q_{\texttt{reject}}, \alpha) = (q_{\texttt{reject}}, \alpha, S)$. Then, we model these transitions in the way we described above. If we put in AND ($\wedge$) all the expressions we discussed, we obtain the expression

$$\mathsf{Trans}.$$

The total number of expressions in $\mathsf{Trans}$ is polynomial.

*Remark.* Note that we only consider the "internal" cells of the tape (i.e., we exclude the first and the last), since once $M$ reaches those cells, it will not move anymore, as it exhausted all steps.

Moreover, note that the non-deterministic choice of a transition in $\delta(q, \alpha)$ is modelled with the OR ($\vee$) on the right-hand side of the expression above. That is, if a truth assignment chooses to make $\hat{q}_1^{i+1}$, $h^{i+1,j-1}$, and $\beta^{i+1,j}$ true, this models the fact that $M$ chooses the first transition.

**Acceptance.** There is one last part we need to model: we want that the last step (i.e., step $p(n)$) has the accepting state, i.e., $M$ accepts $w$. This can be done with the simple expression:

$$\mathsf{Accept} = q_{\mathtt{accept}}^{p(n)}.$$

Thus, our final formula will be

$$\varphi = \mathsf{Cons} \wedge \mathsf{Init} \wedge \mathsf{Trans} \wedge \mathsf{Accept}.$$

It should be hopefully clear that if $w \in \mathcal{L}$ (i.e., $M$ accepts $w$), then $\varphi$ is satisfiable, and if $w \notin \mathcal{L}$ (i.e., $M$ rejects $w$), then $\varphi$ is not satisfiable. □

## 16.1 Some remarks on the proof

We now clarify why it is so important to focus on the modified computation tree of $M$, where each path is of the same length $p(n)$.

Assume $M$ rejects $w$, i.e., all paths in its *original* computation tree end in $q_{\mathtt{reject}}$. Now, all such paths can be of different lengths. We would expect that every truth assignment $\tau$ does not satisfy our formula $\varphi$.

However, if $\tau$ represents the choices made by $M$ that follow a path of length $k < p(n)$, then, since $M$ actually does not transition from $q_{\mathtt{reject}}$ to another state in its original transition function, the value of the Boolean variables $q^{k+1}, q^{k+2}, \ldots, q^{p(n)}$, for each $q \in Q$, regarding the current state at steps after $k$, do not depend on the values that $\tau$ assigns to the variables regarding steps up to $k$.

Thus, $\tau$ can, for example assign false to

$$q^{k+1}, q^{k+2}, \ldots, q^{p(n)},$$

for each state $q \in Q \setminus \{q_{\mathtt{accept}}\}$, and true to

$$q_{\mathtt{accept}}^{k+1}, q_{\mathtt{accept}}^{k+2}, \ldots, q_{\mathtt{accept}}^{p(n)}.$$

Thus, we make the expressions in $\mathsf{Trans}$ regarding steps $k+1, k+2, \ldots, p(n)-1$ trivially true, as their left-hand side (which only contains states different from $q_{\mathtt{accept}}$ and $q_{\mathtt{reject}}$) becomes false. Hence, $\tau$ satisfies $\mathsf{Cons}$, $\mathsf{Init}$, $\mathsf{Trans}$, and also $\mathsf{Accept}$, i.e., $\tau$ satisfies $\varphi$, but $M$ rejects $w$!

As a final comment, we note that in our proof, the Boolean formula $\varphi$ we constructed is not in CNF. In fact, $\varphi$ is the conjunction of different expressions. The expression $\mathsf{Init}$ is a conjunction of clauses, each with only one literal, and thus it is fine as it is. The same for $\mathsf{Accept}$, which is a simple clause with one literal. However, $\mathsf{Cons}$ and $\mathsf{Trans}$ are not conjunctions of clauses.

To complete the proof, we should prove that $\mathsf{Cons}$ and $\mathsf{Trans}$ can be rewritten as conjunctions of clauses, in time polynomial w.r.t. $|w| = n$. We leave the latter task as an exercise.

As a hint on how to do it, note that each expression in $\mathsf{Cons}$ and $\mathsf{Trans}$ is either already a disjunction of literals, like

$$\bigvee_{q \in Q} q^i,$$

and thus we do not need to change it, or it is an implication. For example, like

$$\begin{aligned} q^i \wedge h^{i,j} \wedge \alpha^{i,j} \rightarrow & \bigwedge_{\ell \neq j} u^{i,\ell} \wedge \\ & \big( (\hat{q}_1^{i+1} \wedge h^{i+1,j-1} \wedge \beta^{i+1,j}) \vee (\hat{q}_2^{i+1} \wedge h^{i+1,j+1} \wedge \gamma^{i+1,j}) \big) . \end{aligned}$$

To convert an implication to a conjunction of clauses, you can use the following properties of Boolean formulas.

1. $(A_1 \wedge \cdots \wedge A_n) \vee (B_1 \wedge \cdots \wedge B_m)$ is equivalent to

   $$(A_1 \vee B_1) \wedge (A_1 \vee B_2) \wedge \cdots \wedge (A_1 \vee B_m) \wedge (A_2 \vee B_1) \wedge \cdots \wedge (A_n \vee B_m),$$

   by distributivity;

2. $\neg(A_1 \wedge \cdots \wedge A_n)$ is equivalent to $\neg A_1 \vee \cdots \vee \neg A_n$ (De Morgan's law);

3. $\phi \rightarrow \psi$ is equivalent to $\neg\phi \vee \psi$;

4. $\phi \rightarrow \psi_1 \wedge \psi_2$ is equivalent to $(\phi \rightarrow \psi_1) \wedge (\phi \rightarrow \psi_2)$;

For example, the implication

$$A \wedge B \rightarrow (C \wedge D) \vee (E \wedge F),$$

by rule 1, becomes

$$A \wedge B \rightarrow (C \vee E) \wedge (C \vee F) \wedge (D \vee E) \wedge (D \vee F),$$

which, by rule 4, becomes

$$(A \wedge B \rightarrow C \vee E) \wedge (A \wedge B \rightarrow C \vee F) \wedge (A \wedge B \rightarrow D \vee E) \wedge (A \wedge B \rightarrow D \vee F),$$

which, by rules 2 and 3, becomes

$$(\neg A \vee \neg B \vee C \vee E) \wedge (\neg A \vee \neg B \vee C \vee F) \wedge (\neg A \vee \neg B \vee D \vee E) \wedge (\neg A \vee \neg B \vee D \vee F).$$

# 17 Complements of NP languages and other time classes

In this lecture, we discuss about the asymmetry between languages in **NP** and their complements, which is similar in spirit to the asymmetry we experienced for RE languages and their complements. We discuss what we know about the relationship between **NP** languages and their complements, and then briefly discuss higher time complexity classes.

## 17.1 coNP

We have seen in previous lectures that the class **NP** can be defined in different, equivalent ways. One of them is that a language $\mathcal{L}$ is in **NP** if membership of a string $w$ in $\mathcal{L}$ is witnessed by the existence of a "small" certificate that can be verified in polynomial time. In terms of NTMs, this means that there is a NTM $M$, requiring polynomially many steps, such that with input $w$, $w \in \mathcal{L}$ iff there *exists* a path in the computation tree $M$ that ends in the accepting state.

Consider now the following language, which is the complement of SAT:

$$\text{UNSAT} = \{\varphi \mid \varphi \text{ is a Boolean formula in CNF that is not satisfiable}\}.$$

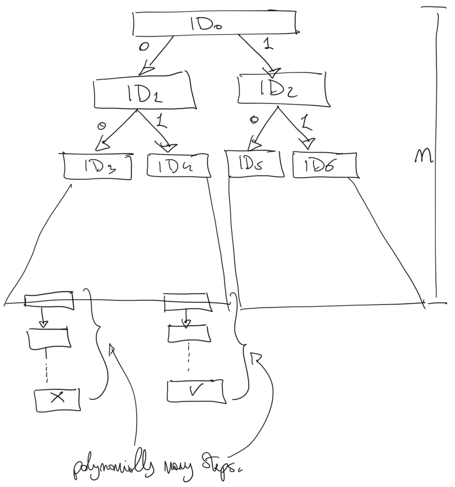
Could you prove that UNSAT is in **NP**? What should we guess first, to verify that an input formula $\varphi$ is in UNSAT?

The common mistake is to devise the following NTM:

1. Guess a truth assignment $\tau$ for the variables in $\varphi$;

2. Check that $\tau$ does not satisfy $\varphi$, in which case accept, otherwise reject.

However, the NTM above does not decide UNSAT. To understand why, let us have a look at the computation tree of the machine, with input a formula $\varphi$.



Assume $\varphi$ is not satisfiable (i.e., every truth assignment makes $\varphi$ false), then, clearly, any path of the computation tree would lead to the accepting state, and thus our machine accepts $\varphi$. However, if $\varphi$ is satisfiable, this does not mean that every truth assignment satisfies $\varphi$, it can very well be the case that *some* truth assignments satisfy $\varphi$, but others do not. Then, some paths correspond to a truth assignment that does not satisfy $\varphi$, (i.e., the end state is the accepting state) and others correspond to truth assignments that satisfy $\varphi$ (i.e., the end state is the rejecting state). Hence, our NTM is still accepting $\varphi$, although it should reject.

**Remark.** The reason why the above procedure does not work, is because what our NTM is guessing is not a certificate for answering "yes", rather, a certificate for answering "No". Another way to see it is that the (wrong) procedure we discussed above is essentially the procedure we use to decide SAT, where we swap $q_{\texttt{accept}}$ with $q_{\texttt{reject}}$.

However, in the same way as this does not work for languages in RE and their complements, this does not work for languages in **NP** and their complements, as acceptance and rejection in a NTM are defined in an asymmetric way: there must exist *at least one* accepting path to accept, but *all paths* must be rejecting to reject.

So, in which class does UNSAT belong to? To answer this question, we define the following complexity class

**Definition 34.**

$$\textbf{coNP} = \{\mathcal{L} \mid \bar{\mathcal{L}} \in \textbf{NP}\}.$$

So, **coNP** collects all the languages whose *complement is in* **NP**. For example, the complement of UNSAT is SAT. Since SAT is in **NP**, we conclude that UNSAT is in **coNP**.

**Remark.** Note that **coNP** *is not* the complement of **NP**, i.e., **coNP** does not contain every language that is not in **NP**. Rather, **coNP** contains the *complements of* **NP** *languages*. This makes a huge difference. In fact, if a language is in the complement of **NP**, it necessarily cannot be in **NP**, by definition. However, there can be languages that are both in **NP** and **coNP**, as we will see later in this lecture.

One question that arises is whether this asymmetry between **NP** and **coNP** is real, or languages in **NP** and their complements have actually the exact same complexity. In other words, do we know whether

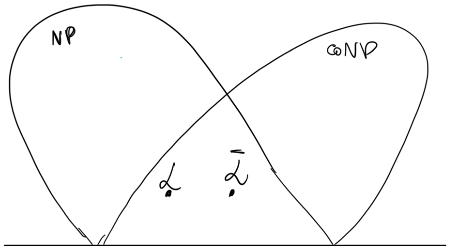
$$\textbf{NP} = \textbf{coNP}?$$

It turns out that this is yet another question for which we do not have an answer yet. However, **NP**−completeness can help us, once again, to identify some ways of tackling the above question.

**Theorem 35.** **NP** = **coNP** *iff there is an* **NP**−complete *language that is also in* **coNP**.

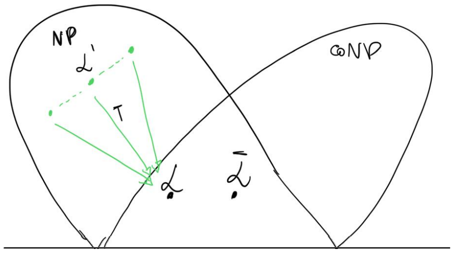
*Proof.* We first prove the left to right direction of the equivalence. Assume **NP** = **coNP**. Since **NP**−complete languages are just special **NP** languages, they clearly must belong to **coNP**, under the assumption that **NP** = **coNP**.

We now prove the other direction. Assume that $\mathcal{L}$ is **NP**−complete and also that $\mathcal{L} \in$ **coNP**. The latter means that $\bar{\mathcal{L}} \in$ **NP**. This fact is highlighted in the following picture:

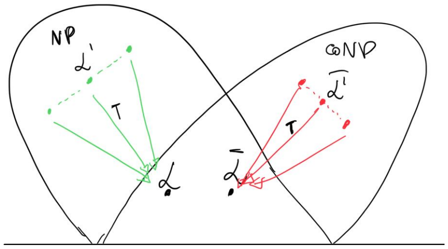


We prove that under the above assumption, **NP** = **coNP**. We do this by showing that **NP** ⊆ **coNP** and **coNP** ⊆ **NP**.

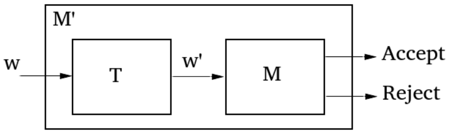
To prove that **NP** ⊆ **coNP**, consider some arbitrary language $\mathcal{L}'$ in **NP**. We show that $\mathcal{L}'$ is also in **coNP**. Since $\mathcal{L}$ is **NP**−complete, and thus **NP**−hard, there is a polynomial time reduction $T$ from $\mathcal{L}'$ to $\mathcal{L}$. This fact is shown in the picture below:



Note that the complement of $\mathcal{L}'$, denoted $\bar{\mathcal{L}}'$, is in **coNP**, because $\mathcal{L}'$ is in **NP**. Then, since for both $\bar{\mathcal{L}}'$ and $\bar{\mathcal{L}}$, "yes" and "no" instances are inverted, the reduction $T$ is also a reduction from $\bar{\mathcal{L}}'$ to $\bar{\mathcal{L}}$. This is shown in the picture below:



Now, let $M$ be the NTM that decides $\bar{\mathcal{L}}$ in polynomial time (it exists, because we assumed $\mathcal{L}$ is in **coNP**, and thus its complement is in **NP**). Then, if we combine the reduction $T$ from $\bar{\mathcal{L}}'$ to $\bar{\mathcal{L}}$, and the machine $M$:



we obtain a NTM $M'$ that decides $\bar{\mathcal{L}}'$ in polynomial time. Thus, $\bar{\mathcal{L}}'$ is in **NP**, which implies its complement $\mathcal{L}'$ is in **coNP**. Hence, the language $\mathcal{L}'$ not only is in **NP** but it is also in **coNP**.

To prove that **coNP** $\subseteq$ **NP**, consider some arbitrary language $\bar{\mathcal{L}}'$ in **coNP**, as shown in the previous picture. We show that $\bar{\mathcal{L}}'$ is also in **NP**. As already discussed, there is a reduction $T$ from $\bar{\mathcal{L}}'$ to $\bar{\mathcal{L}}$, as shown in the previous picture. Then, if we combine $T$ with the NTM $M$ deciding $\bar{\mathcal{L}}$ in polynomial time, as we did before, we obtain a NTM $M'$ deciding $\bar{\mathcal{L}}'$ in polynomial time. Hence, $\bar{\mathcal{L}}'$ not only is in **coNP**, but also in **NP**. □

So, the above theorem tells us that if **NP** = **coNP**, then we could prove it by showing that an **NP**−complete language is in **coNP**. To date, no one has been able to show one single **NP**−complete language to also be in **coNP**. This is one of the main reasons why most believe that **NP** $\neq$ **coNP**.

However, there can be **NP** languages that are also in **coNP** (and thus, unlikely to be **NP**−complete). As an example, every language that is decidable in polynomial time is in **NP** $\cap$ **coNP**.

**Theorem 36.** **P** $\subseteq$ **NP** $\cap$ **coNP**.

*Proof.* Let $\mathcal{L}$ be a language in **P**. Since **P** $\subseteq$ **NP**, $\mathcal{L} \in$ **NP**. Consider now the complement of $\mathcal{L}$, i.e., the language $\bar{\mathcal{L}}$. The complement $\bar{\mathcal{L}}$ of every language $\mathcal{L} \in$ **P** is also in **P**. Indeed, if $M$ is the DTM that decides $\mathcal{L}$ in polynomial time, the machine $M'$ where $q_{\texttt{accept}}$ and $q_{\texttt{reject}}$ are swapped, decides $\bar{\mathcal{L}}$. But, if $\bar{\mathcal{L}} \in$ **P**, then $\bar{\mathcal{L}} \in$ **NP**. But, if the complement of $\mathcal{L}$ is in **NP**, by definition of **coNP**, it means that $\mathcal{L} \in$ **coNP**. □

**Remark.** Note that if we combine Theorem 35 and Theorem 36 above, we immediately conclude that if **P** = **NP**, then **NP** = **coNP**. In fact, we know that if **P** = **NP**, then there is an **NP**−complete language $\mathcal{L}$ that is in **P**. By Theorem 36, $\mathcal{L}$ is also in **coNP**. Thus, by Theorem 35, since $\mathcal{L}$ is both **NP**−complete and in **coNP**, we conclude **NP** = **coNP**.

However, the opposite does not hold, i.e., **NP** = **coNP** does not imply that **P** = **NP**.

We continue by observing that there are also languages that are in **NP** $\cap$ **coNP**, which we have not been able to prove are also in **P**. They are somehow in a “limbo” between **P** languages and **NP**−complete languages. One such an example is the language

$$\text{FACTOR} = \{(N, k) \mid N \text{ is a natural number that has a prime factor } M \leq k\}.$$

The above is essentially the decision version of the problem asking which are the prime factors of $N$.

For example, the string $(175, 6)$ is in FACTOR, because 175 can be factored to $5 \cdot 5 \cdot 7$, where 5 and 7 are its prime factors, and 5 is less then 6. On the other hand, $(175, 4)$ is not in FACTOR, since none of the prime factors of 175 is less or equal than 4.

**Theorem 37.** *FACTOR* $\in$ **NP** $\cap$ **coNP**

*Proof.* FACTOR is in **NP**, since to verify that a pair $(N, k)$ *is* in FACTOR, we can guess a number $M$ encoded in binary, and then verify that:

1. $M \leq k$;
2. $M$ is prime;
3. $M$ divides $N$.

Note that $M$ must be less or equal than $k$, thus, to guess $M$ we can guess at most the same number of bits used to encode $k$. Hence. $||M||$ is linear in the size of the input $(N, k)$.

Verifying $M \leq k$ is clearly in polynomial time, and also checking if $M$ is prime can be performed in polynomial time, since PRIME is in **P**. Item 3 is also easy.

The above procedure is correct, because if a prime factor $M \leq k$ of $N$ exists, the above non-deterministic procedure will find it, and accept. If it does not exist, all computation paths in the computation tree will reject.

However, FACTOR is also in **coNP**. To prove this, we need to show that its complement is in **NP**. The complement of FACTOR is:

$$\overline{\text{FACTOR}} = \{(N, k) \mid N \text{ is a natural number and } \\ \textit{every} \text{ prime factor } M \text{ of N is such that } M > k\}.$$

How do we verify that $(N, k)$ is in $\overline{\text{FACTOR}}$ (or, equivalently, that it is *not* in FACTOR)?

We cannot guess a number $M$, and check that $M > k$ and $M$ is prime. The reason for this is that, even if $N$ has a prime factor $M > k$, this does not mean that $N$ has no other prime factors $M' \leq k$. So, $M$ is not a certificate of the fact that $(N, k) \in \overline{\text{FACTOR}}$.

What we should do is to guess *a factorization* of $N$, i.e., a sequence of (not necessarily distinct) prime numbers that when multiplied, give $N$. To do this, we can guess a list of numbers $2 \leq M_1, \ldots, M_n \leq N$, and then verify the following:

1. $M_1, \ldots, M_n$ are all primes;
2. $\prod_{i=1}^{n} M_i = N$;
3. each $M_i$ is strictly greater than $k$.

Since the factorization of a natural number $N$ is unique, if we ignore the order in which we enumerate the numbers therein, once Items 1 and 2 have been verified, we are certain that if each $M_i > k$, this necessarily means that $N$ has no prime factor less or equal than $k$.

Assume $m = ||N||$ is the number of bits used to encode $N$. Then, note that each number $M_i$ requires at most $m$ bits. Moreover, if we consider how many factors we need to guess, the worst case is when $N$ is the largest number using $m$ bits, i.e., $N = 2^m - 1$, and each $M_i$ is the smallest prime (i.e., 2). Thus, we need to guess at most $n = m$ prime factors $M_1, \dots, M_n$. Thus, our certificate is of polynomial size w.r.t. the input string $(N, k)$. Finally, checking Items 1,2, and 3 can be clearly performed in polynomial time. Thus, $\overline{\text{FACTOR}} \in \textbf{coNP}$. □

Most researchers believe FACTOR is not in **P**, and since it is in both **NP** and **coNP** it is also unlikely to be **NP**−complete, by Theorem 35. The belief that FACTOR is intractable is strong to the point that most of modern cryptography is designed in such a way that finding the cryptographic keys of a communication channel is equivalent to factorize a number $N$. Thus, finding the keys is believed to be unfeasible, for very large numbers, as far as FACTOR is not found to be **P** (which we find unlikely).

## 17.2 EXP and NEXP

Other complexity classes that are analogous to **P** and **NP** exist, that deal with higher time resource usage. We will not go into the details of these classes, rather just define them here.

**Definition 35.** *The class of* exponential time *and* non-deterministic exponential time *languages are respectively*

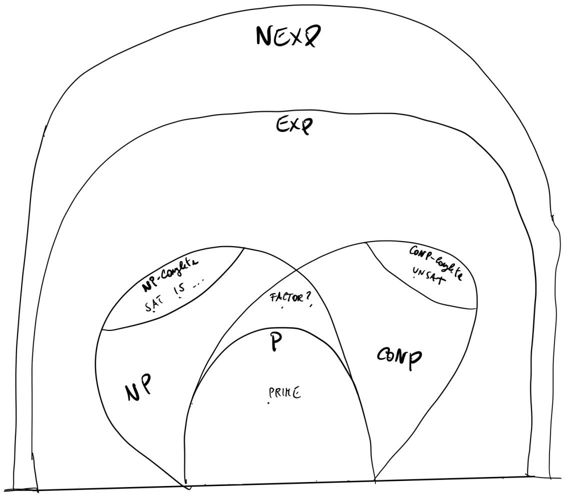
$$\mathbf{EXP} = \bigcup_{c \geq 1} \text{DTIME}(2^{n^c}),$$

$$\mathbf{NEXP} = \bigcup_{c \geq 1} \text{NTIME}(2^{n^c}).$$

So, the above are the analogous of **P** and **NP**, where exponential time is required to decide the languages. Clearly, $\mathbf{EXP} \subseteq \mathbf{NEXP}$, by definition. It is also not difficult to verify that $\mathbf{NP} \subseteq \mathbf{EXP}$, since we have already shown that a non-deterministic TM can be rewritten to a deterministic TM with an exponential overhead. To see why $\mathbf{coNP} \subseteq \mathbf{EXP}$, consider a language $\mathcal{L} \in \mathbf{coNP}$. It means that $\bar{\mathcal{L}} \in \mathbf{NP}$, and hence $\bar{\mathcal{L}} \in \mathbf{EXP}$. If $M$ is the *deterministic* TM that decides $\bar{\mathcal{L}}$ in exponential time, then the machine $M'$ where we invert $q_{\texttt{accept}}$ and $q_{\texttt{reject}}$ accepts $\mathcal{L}$.

Note that the above trick of inverting the ending states works because the machine $M$ is deterministic.

We can finally draw the picture of the complexity classes we discussed, in the way we believe are related (e.g., if a class strictly includes another in the picture below, this means we believe the super class *stricly* includes the lower class).



In the picture above we have also included **coNP**−complete languages. We do not deal with such languages, but their definition is similar to the one of **NP**−complete languages: a language is **coNP**−complete if it is in **coNP**, and every language in **coNP** reduces to it in polynomial time. UNSAT is an example of a **coNP**−complete language.

**Home work.** Prove that for every **NP**−complete language $\mathcal{L}$, its complement $\bar{\mathcal{L}}$ is **coNP**−complete.

# 18 Space Complexity, and the classes LOGSPACE and NL

Up to this point, we were mainly concerned with time constrained languages, i.e., languages that can be decided within a certain number of steps by a TM. Another dimension to consider, however, is the *space* required by the TM.

In this part of our lectures, we will move our focus on languages that can be decided by TMs that require no more than a given amount of space.

In order to formally discuss these languages, we first need to clarify what we mean by the "space required" by a TM, in a way similar to what we did when we defined the notion of "time required" by a TM. To even do this, we need to describe the *kind* of TMs we consider.

We already anticipated, when defining reductions, that when we want to precisely analyse the space employed by a TM during its computation, it is a good idea to separate this space from the space needed to store the input string, and (in the case of reductions outputting some string) from the space needed to store the output.

Thus, to properly understand the space usage of a TM deciding a language, we will focus on TMs using two tapes.

**TMs with two tapes.** The first tape is the *input tape*, and it is read-only, while the second tape is the *working tape*, and it is read/write. Differently from reductions, we do not have a third tape, as our TMs do not need to output anything. Rather, they only need to halt in the accepting or rejecting state, depending on the input string.

So, whenever we discuss space bounds, we always assume our TM is of the form above (it does not matter if it is deterministic or not). Now, we are ready to define the space required by a TM.

**Definition 36.** *The* space required by a deterministic TM $M$ *with inputs of length* $n$*, denoted* $S_M(n)$*, is the maximum number of cells that* $M$ *visits on its working tape, when executed with inputs of length* $n$*. If there is an input of length* $n$ *for which no bound on the visited cells exists, we say that* $S_M(n) = \infty$*.*

The space required by a NTM is analogous.

**Definition 37.** *The* space required by a non-deterministic TM $M$ *is the maximum number of cells that* $M$ *visits on its working tape, when considering* all computation paths *of its computation tree, when executed with inputs of length* $n$*. If there is an input of length* $n$ *and a path in the computation tree of* $M$ *with that input where no bound on the visited cells exists, then* $S_M(n) = \infty$*.*

Now that we formally specified what is the space required by a TM, we can start identifying some important classes of languages in terms of space requirements. As we did for time classes, we first introduce the following general notation.

**Definition 38.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a function. We define*

$$\text{DSPACE}(f(n)) = \{\mathcal{L} \mid \exists \text{ a DTM } M \text{ that decides } \mathcal{L} \text{ and } S_M(n) \in O(f(n))\},$$

$$\text{NSPACE}(f(n)) = \{\mathcal{L} \mid \exists \text{ a NTM } M \text{ that decides } \mathcal{L} \text{ and } S_M(n) \in O(f(n))\}.$$

So, DSPACE($f(n)$) (resp., NSPACE($f(n)$)) simply collects all languages that can be decided by a deterministic (resp., non-deterministic) TM that uses a number of cells in its working tape that is at most of the order of $f(n)$.

## 18.1 The space classes LOGSPACE and NL

With the above notation at hand, we can start defining our first classes of languages decidable with a certain amount of space. The first classes we consider collect languages that can be decided with a “very small” amount of space, i.e., logarithmic w.r.t. the input string.

**Definition 39.** *The class of* logspace languages *is defined as*

$$\textbf{LOGSPACE} = \text{DSPACE}(\log_2 n),$$

*while the class of* non-deterministic logspace languages *is defined as*

$$\textbf{NL} = \text{NSPACE}(\log_2 n).$$

Note that in the above definition, we arbitrarily chose logarithms in base 2. However, the actual choice does not matter. In fact, from basic logarithms properties, we can always convert a logarithm in one base to another by a constant factor division:

$$\log_c n = \frac{\log_2 n}{\log_2 c} \in O(\log_2 n).$$

Clearly, **LOGSPACE** $\subseteq$ **NL**.

Let us see an example of a language in **LOGSPACE**, i.e., a language we can decide with a DTM using only logarithmically many cells in the working tape. One such a language is our usual language

$$\mathcal{L}_{01} = \{0^n 1^n \mid n \geq 0\}.$$

The TM we devised at the beginning of our course required, in principle, more than logarithmic space (actually linear) w.r.t. the input string. This is because the machine had to “cross” symbols in the input string. But, to do this with a read-only input tape, it means the machine first needs to *copy* the whole input in the working tape, and thus requiring linear space.

We can use a new TM that only requires logarithmic space. Consider an input string $w$.

1. If $w$ is empty, accept, otherwise

2. Scan the input string left-to-right, and increment a counter (stored in the working tape), for each 0 encountered, until a 1 is found
3. Scan the remaining input left-to-right, and increment a second counter (stored after the first one), for each 1 encountered, until a blank is found.
4. Compare the counters, if they are the same, accept, otherwise reject.

Note that in the working tape, we are only storing the two counters. In the worst case, each counter needs to represent the number $n = |w|$, and, if we encode the counters in binary, only $O(\log_2 n)$ cells are required by each counter. Since we use a constant number of counters (2 in this case), the above TM $M$ is such that $S_M(n) \in O(\log_2 n)$, and thus $\mathcal{L}_{01} \in$ **LOGSPACE**.[20]

We now consider an example of a language in **NL**. Recall the language

$$\text{REACHABILITY} = \{(G, s, t) \mid G \text{ is a directed graph such that} \\ \text{there is a path from } s \text{ to } t \text{ in } G\}.$$

We have already seen an algorithm that decides the above language. This algorithm was based on a breadth search approach, and required polynomial time. However, this algorithm used a *queue* of nodes, that in the worst case can contain all nodes of the graph. Thus, the space required by that algorithm is linear w.r.t. the input length.

We show that we can decide REACHABILITY via a non-deterministic TM $M$ such that $S_M(n) \in O(\log_2 n)$.

**Observation.** When we are limited to use logarithmic space but we would also like to store in the working tape some data of the input string, the main idea is to store a "pointer" to the desired data. That is, if our TM $M$ wants to store a node of $G$ in the working tape, it instead stores a number in binary that denotes the position in the input string where the node occurs. By encoding this pointer in binary, we only need $O(\log_2 |w|)$ cells to refer to any data of the input string $w$.

**Theorem 38.** *REACHABILITY* $\in$ **NL**.

*Proof.* The main idea of our NTM is to initial set $s$ as the current visited node, and at each iteration, guess a candidate "next" node $v$. If the current node is connected to this new node $v$ by an edge, then $v$ becomes new current node. The machine keeps doing this until it reaches $t$. If it cannot reach $t$, then the machine rejects. The algorithm is depicted in Algorithm 2.

Note that Algorithm 2 does not need to store the full path that leads from $s$ to $t$. It only needs to store the current node, the next node, and the counter, and can forget about the complete path it has followed. One might think that this can lead the algorithm to loop. This, however, cannot happen, because the

[20] Note that if the number of counters was depending on the input, e.g. we had $n$ counters, the space would be $O(n \cdot \log_2 n)$, which is not logarithmic.

**Algorithm 2:** NL algorithm for REACHABILITY

```
   Input: A directed graph $G = (V, E)$ and two nodes $s, t$
1  Let $p$ be a pointer to node $s$, and store it in the working tape;
2  Let $cnt := 1$ be a counter stored in the working tape;
3  if $p$ points to a node equal to $t$ then
4      Accept;
5  end
6  Guess a pointer $p'$ to some node $v$ of $G$;
7  if $p$ points to a node that has no edge to the node pointed by $p'$ then
8      Reject;
9  end
10 Let $p := p'$;
11 Let $cnt := cnt + 1$;
12 if $cnt \leq |V|$ then
13     goto step 3;
14 else
15     Reject;
16 end
```

algorithm keeps track of how many nodes are visited with the counter. If it visits more than $|V|$ nodes, it just means it followed a wrong path (i.e., if $s$ can reach $t$, this must also be possible by visiting each node in the path *once*, and thus our machine does not need to consider paths longer than $|V|$).

The space (number of bits) needed to store a pointer to a node is $O(\log_2 n)$, where $n$ is the length of the encoding of $(G, s, t)$, because a pointer is just a number from 1 to $n$. The counter requires $O(\log_2 |V|)$ bits, because it needs to count from 1 up to $|V| + 1$.[21] So, overall, the algorithm stores three variables, each of logarithmic size, and thus the overall space is $O(\log_2 n)$.

Regarding correctness of the algorithm, assume there is a path $s, v_1, \ldots, v_k, t$ in $G$. Then, among all possible choices of our procedure, there is a sequence of choices where the algorithm guesses the nodes $v_1, \ldots, v_k, t$, one for each iteration, and thus will realize (in one of its computation paths) that a path exists from $s$ to $t$. If there is no path at all from $s$ to $t$, then whatever the algorithm guesses at each iteration, it will never reach $t$, and thus either will reach a dead end (it rejects on line 8), or it exhausted all possible nodes to visit (it rejects on line 15). □

**LOGSPACE vs NL.** A question that one might ask is whether we can do better, and decide REACHABILITY in logarithmic space, without relying on

[21] We need the counter to be able to reach at least $|V| + 1$ for the algorithm to realize it is looping.

non-determinism, i.e., whether REACHABILITY $\in$ **LOGSPACE**.

Actually, no logspace algorithm is known that decides REACHABILITY. This is essentially similar to what happens for SAT w.r.t. **NP**: no one has yet been able to find a polynomial-time algorithm deciding SAT. This thus raises the question:

$$\textbf{LOGSPACE} = \textbf{NL}?$$

We do not know the answer to the above question. But, as we did for the **P** = **NP** question, we have some tools to attack the question: the notion of complete languages.

The notion of **NL**−completeness is similar to the one of **NP**−completeness. The main difference is the kind of reductions we use.

Recall that if we can show that an **NP**−complete language is in **P**, then every language in **NP** is also in **P**, and thus conclude **P** = **NP**. For this to work, it was crucial that we use polynomial-reductions, i.e., reductions that are "less powerful" than the algorithms needed to decide **NP** languages.

With a similar spirit, we need to define **NL**−complete languages by using "less powerful" reductions than the algorithms needed to decide **NL** languages. Thus, we use logspace reductions. We use $\mathcal{L}_1 \leq_L \mathcal{L}_2$ to denote the fact that there is a logspace reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$.

**Definition 40.** *A language $\mathcal{L}$ is* **NL**−complete *if:*

1. *$\mathcal{L} \in$ **NL**, and*
2. *for each language $\mathcal{L}' \in$ **NL**, $\mathcal{L}' \leq_L \mathcal{L}$.*

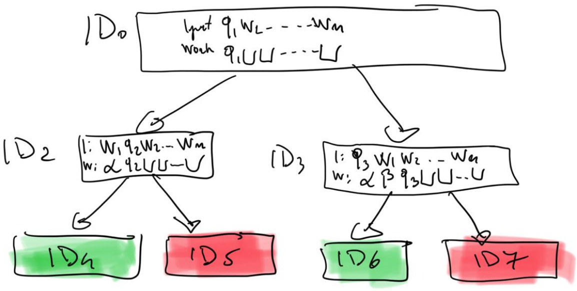
Although we do not prove it, the above definition fullfills its purpose.

**Theorem 39.** **LOGSPACE** = **NL** *iff there exists an* **NL**−complete *language that is also in* **LOGSPACE**.

It turns out that REACHABILITY is **NL**−complete.

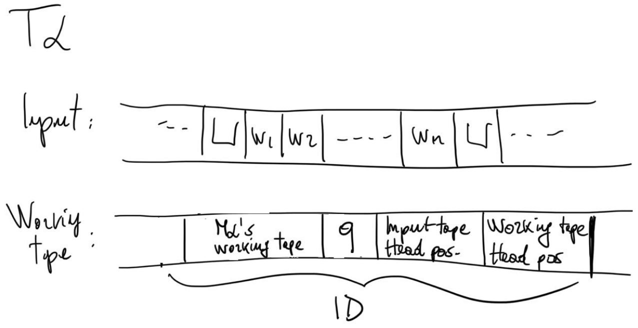
**Theorem 40.** *REACHABILITY is* **NL**−complete.

*Proof.* (*Sketch*) We have already shown that REACHABILITY is in **NL**. We now have to prove that *every* language $\mathcal{L} \in$ **NL** reduces to REACHABILITY in logspace. Let $\mathcal{L}$ be some arbitrary language in **NL**. Thus, there is a NTM $M_{\mathcal{L}}$ that decides $\mathcal{L}$ in logarithmic space. As usual, let us have a look at the computation tree of $M_{\mathcal{L}}$, when executed with some input $w = w_1, \ldots, w_n$.



Note that $M_{\mathcal{L}}$ has two tapes, and thus, we should store both tapes content in an ID, and for each tape, remember where each tape head was positioned. The computation tree of $M_{\mathcal{L}}$ is very similar to a directed graph. We have a starting node, which is $ID_0$, and we have some ending nodes. Some correspond to accepting IDs, and others to rejecting IDs. The idea of the reduction is to construct, starting from $w$, a directed graph $G$ that *contains* the computation tree of $M_{\mathcal{L}}$, and sets the starting node $s$ to $ID_0$, and the end node $t$ to some special node to which we connect all accepting IDs.

Our reduction, which we call $T_{\mathcal{L}}$, takes as input a string $w$, and constructs $(G, s, t)$, where $G$ is a graph and $s, t$ are nodes of $G$. Let $S_{M_{\mathcal{L}}}(n) \in O(\log_2 n)$ be the number of cells used by $M_{\mathcal{L}}$ in its working tape. Since our reduction $T_{\mathcal{L}}$ must deal with the IDs of $M_{\mathcal{L}}$, it needs a way to store each of them in logarithmic space. Since $T_{\mathcal{L}}$ takes $w$ as input, it can store a complete ID of $M_{\mathcal{L}}$ by storing $S_{M_{\mathcal{L}}}(n)$ bits representing the working tape of $M_{\mathcal{L}}$, then the current state of $M_{\mathcal{L}}$, and finally two binary numbers containing the input and working tape head positions of $M_{\mathcal{L}}$[22]:



A state requires constant space, because its size is independent of the input string $w$, while all other data requires logarithmically many cells. Thus, overall, $T_{\mathcal{L}}$ can store an ID of $M_{\mathcal{L}}$ in logarithmic space. Finally, as a last observation, such an ID stored in this way is nothing else than a binary number using $g(n)$ bits, for some function $g(n) \in O(\log_2 n)$. Thus, each ID is a number from 0 to $2^{g(n)} - 1$. With the above observations in mind, our reduction is quite simple.

1. Allocate two numbers $ID, ID'$ in the working tape, each using $g(n)$ bits.
2. Allocate an additional number $ID^*$ in the working tape, using $g(n) + 1$ bits, and set it to all ones.
3. for each $ID \in \{0, \ldots, 2^{g(n)} - 1\}$, do
   (a) for each $ID' \in \{0, \ldots, 2^{g(n)} - 1\}$, do

[22] It does not need to store also the content of the input tape, since it never changes, since it is read only, and if $T_{\mathcal{L}}$ will ever need its content, this is nothing than $w$ which is part of the input to $T_{\mathcal{L}}$.

i. If $ID$ and $ID'$ represent valid IDs, and $ID$ yelds $ID'$, then write the edge $(ID, ID')$ in the output tape.

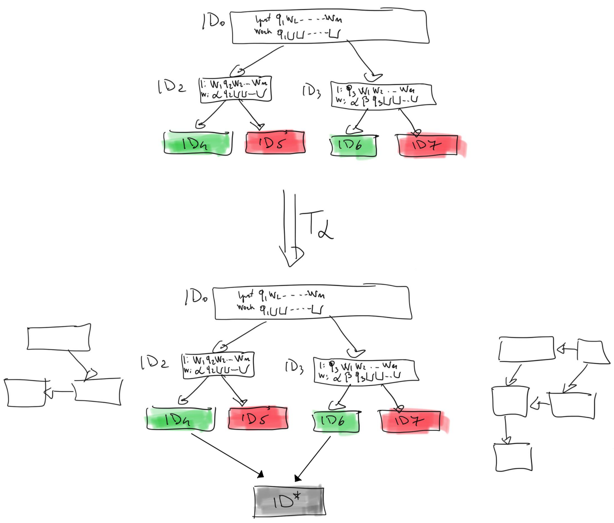
ii. Else, if $ID$ is a valid ID and it contains the accepting state, then write the edge $(ID, ID^*)$ in the output tape.

4. Write the node $s = ID_0$ in the output tape.

5. Write the node $t = ID^*$ in the output tape.

So, the main idea is that our reduction $T_{\mathcal{L}}$ is considering all possible pairs of IDs $ID$ and $ID'$. Some of these IDs might very well not be part of the computation tree of $M_{\mathcal{L}}$, and thus, the constructed graph $G$, will both contain the computation tree of $M_{\mathcal{L}}$, but also other "wrong" nodes and edges that do not appear in the computation tree. However, all these wrong nodes cannot be reached by node $ID_0$, because they are not part of the computation tree of $M_{\mathcal{L}}$.

Finally, $ID^*$ uses one more bit than the other nodes, and thus is different from all the other IDs used in the for loops, hence, $ID^*$ has no outgoing edges, but only incoming edges from accepting IDs. The above observations are highlighted in the picture below.



Thus, if $w \in \mathcal{L}$, and thus, there is a path from $ID_0$ to an accepting ID in the computation tree of $M_{\mathcal{L}}$, then $G$ has a path from $s = ID_0$ to $t = ID^*$. If $w \notin \mathcal{L}$, and thus, all paths in the computation tree of $M_{\mathcal{L}}$ are rejecting, then there is no way in $G$, starting from $s = ID_0$, to reach a node that is connected to $t = ID^*$. □

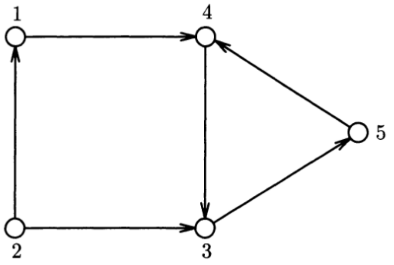
# 19 Savitch's Theorem

In the previous lecture, we discussed the space complexity classes **LOGSPACE** and **NL**, i.e., the classes of languages that can be decided by a deterministic (resp., non-deterministic) TM using $O(\log_2 n)$ cells in their working tape. We have also seen that REACHABILITY is in **NL**, by means of a NTM that guesses a path from the source node $s$ to the target node $t$, by guessing one "next node" at the time.

Finally, we have shown that REACHABILITY is actually **NL**−complete. No one has been able to prove whether **LOGSPACE** = **NL** or not, and most believe **LOGSPACE** ≠ **NL**. Thus, since REACHABILITY is **NL**−complete, we believe there is no deterministic TM that can decide it using $O(\log_2 n)$ space (otherwise **LOGSPACE** = **NL**).

However, we know a *deterministic* algorithm that can decide REACHABILITY using space $O((\log_2 n)^2)$, i.e., it requires *polylogarithmic space* (a logarithm to the power of some fixed constant). This result, besides being interesting on its own right, will be the key that allows us to prove the main theorem of this lecture: Savitch's Theorem.

We first describe the main idea behind the $O((\log_2 n)^2)$ algorithm for REACHABILITY. Consider our usual directed graph $G$:



and assume the starting node $s = 2$ and target node $t = 5$. Clearly, there is a path from $s$ to $t$ in $G$. For example, $s, 1, 4, 3, t$. Note that a path from $s$ to $t$ exists iff there is a node $u$, roughly in the middle of the path (for example, node 4), such that $s$ can reach $u$ in half the steps (i.e., half the edges) a $u$ can reach $t$ in half the steps.

We exploit the above observation to devise our $O((\log_2 n)^2)$ space algorithm for REACHABILITY.

**Theorem 41.** *REACHABILITY* $\in$ DSPACE$((\log_2 n)^2)$.

*Proof.* The algorithm we are going to devise is a recursive algorithm, that finds a path from $s$ to $t$ by finding two smaller paths that are joined by a middle node. We call the algorithm $\mathsf{exists} - \mathsf{path}$, and takes as input a directed graph $G$, two nodes $s, t$ and an additional integer parameter $k$. The algorithm accepts if there is a path from $s$ to $t$ in $G$ requiring at most $k$ steps, and rejects otherwise.

**Algorithm 3:** exists − path

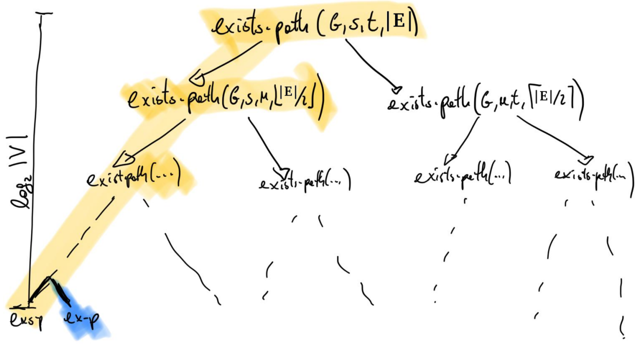
```
   Input: A directed graph $G = (V, E)$, two nodes $s, t$ and an integer $k$
   // $s$ must reach $t$ in zero steps
1  if $k = 0$ then
2      Accept if $s = t$, otherwise Reject;
3  end
   // $s$ must reach $t$ in at most one step
4  if $k = 1$ then
5      Accept if either $s = t$ or $(s, t) \in E$, otherwise Reject;
6  end
   // Find the middle node $u$
7  foreach $u \in V$ do
8      if exists − path$(G, s, u, \lfloor k/2 \rfloor)$ and exists − path$(G, u, t, \lceil k/2 \rceil)$ then
9          Accept;
10     end
11 end
12 Reject;
```

Clearly, when running exists − path setting $k$ to the total number of edges of $G$, exists − path becomes an algorithm deciding REACHABILITY. Let us now analyse the algorithm.

You should be fairly convinced that the algorithm is correct. Let us focus on the space it requires.[23] Excluding for a second the recursive calls in line 8, what exists − path actually needs to perform its operations are some pointers to a constant number of data of the input. That is, two pointers to the nodes $s$ and $t$ for comparing them, a pointer to the number $k$ to check if it is 0 or 1, and one pointer for the node $u$, which at each iteration of the for loop at line 7, can be erased and reused to point to a new node. Hence, overall, $O(\log_2 n)$ space is required, where $n$ is the length of the encoding of the input $(G, s, t, k)$.

We should not forget, however, that our algorithm is recursive, and thus, the space it uses must be added to the space used by the called sub-functions. To understand how much space this requires, let us depict the call tree of exists − path, when given $k = |E|$.

[23] Note that in terms of time, the algorithm is very inefficient (it is not even polynomial time), but we are not concerned with time at the moment.



When $\mathsf{exists-path}(G, s, t, |E|)$ is executed, it then executes itself twice, for each node $u \in V$. Note that when we finish with a node $u$, and move to the next, we can free the space used by the previous two calls, and reuse it for the next node. Thus, the space required by the for loop is only the space required to perform the two sub-calls.

When executing the sub-calls, the second call can occur only after the first call concludes. Similarly, when entering the first sub-call, for example $\mathsf{exists-path}(G, s, t, \lfloor |E|/2 \rfloor)$, again two sub-calls are performed, but only the first is executed, before executing the second.

Hence, when running our algorithm, at first, the chain of calls we have is the one highlighted in yellow. This chain cannot be longer than $\log_2 |E| \in O(\log_2 n)$, because at each call, we divide $k$ by half. When the call at the bottom of the chain concludes, the space it was using can be freed, and reused to execute the second call (highlighted in blue), and so on.

So, no more than $\log_2 |E|$ calls need to be stored in memory at the same time, and thus, the overall space needed by the algorithm is $\log_2 |E|$ times the space needed by one single call, i.e., overall $O((\log_2 n)^2)$. □

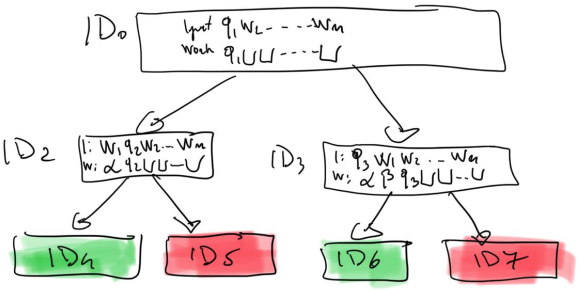
The above theorem essentially tells us that although REACHABILITY is one of the hardest languages in **NL** (i.e., it is **NL**−complete), there is actually a very mild price to pay, in terms of space, to solve it with a *deterministic* TM. This was not the case for **NP**−complete languages, for which the best deterministic TM we can get is an exponential-time one.

The above observation can be generalized to work not only for REACHABILITY, but for any language, and this leads us to the following theorem, proved by Walter Savitch in 1970.

**Theorem 42** (Savitch's Theorem)**.** *For every function $f : \mathbb{N} \to \mathbb{N}$, where $f(n) \in \Omega(\log_2 n)$ (i.e., $\log_2 n \in O(f(n))$),*

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

*Proof.* We provide a proof sketch. We need to show that every language $\mathcal{L} \in \mathrm{NSPACE}(f(n))$ can be decided by a deterministic TM using $O(f^2(n))$ space. Consider an arbitrary language $\mathcal{L} \in \mathrm{NSPACE}(f(n))$. Then, let $M_{\mathcal{L}}$ be the NTM that decides $\mathcal{L}$ using $O(f(n))$ space in its worktape. As usual, let us depict the computation tree of $M_{\mathcal{L}}$ with input some string $w = w_1, \ldots, w_n$.



You can see that deciding whether $w \in \mathcal{L}$ boils down to verify if there is a path from $ID_0$ to some accepting $ID$ of the computation tree.

Actually, we have already seen such a correspondence in the proof that REACHABILITY is **NL**−complete. In fact, the idea is the same. We build a graph $G$ corresponding to the computation tree of $M_{\mathcal{L}}$ with input $w$, and give this graph to $\mathsf{exists-path}$, where $s$ is the initial ID $ID_0$, $t$ is the special ID $ID^*$, and $k$ is the total number of edges.

However, we need to be careful in analysing the size of this graph, when given to $\mathsf{exists-path}$. Each node is a potential ID of the computation tree. We have seen in the proof of **NL**−completeness of REACHABILITY that we can store an ID by storing the content of the working tape, the current state and the two heads' positions. The working tape requires $O(f(n))$, since $\mathcal{L} \in \text{NSPACE}(f(n))$, the state requires constant space, and the two heads positions require $O(\log_2 n)$ for the input position, and $O(\log_2 f(n))$ for the work tape position.

So, all in all, since we assumed $f(n) \in \Omega(\log_2 n)$, the space occupied by one ID is $O(f(n))$. Recall that an ID encoded in this way is essentially a number in binary, using $O(f(n))$ bits. Thus, the maximum number of possible IDs (i.e., nodes in the graph) we can have is $O(2^{f(n)})$. Hence, the size of the graph $G$, representing the computation tree of $M_{\mathcal{L}}$, we give to algorithm $\mathsf{exists-path}$ is $O(2^{f(n)})$. Since $\mathsf{exists-path}$ requires $O((\log_2 m)^2)$ space, where $m$ is the size of *its input*, the overall required space is $O((\log_2 2^{f(n)})^2) = O(f(n)^2)$.

**Remark.** The reason why the above proof does not work if $f(n) \in \Omega(\log_2 n)$ does not hold is because whatever is the space required by the NTM deciding a language $\mathcal{L}$ (even constant space!), the nodes of the graph $G$ corresponding to the computation tree of the NTM still require at least $\log_2 n$ space. This is because each node contains the input head position (which requires logarithmic space w.r.t. input tape string). Hence, the size of $G$ is at least $2^{\log_2 n} = n$, regardless of the space required to decide $\mathcal{L}$. Hence, the space required by $\mathsf{exists-path}$ is at least $(\log_2 n)^2$. So, the space required to decide $\mathcal{L}$ by $\mathsf{exists-path}$ would *not be* $O(f^2(n))$, but more. □

Savitch's Theorem is quite a powerful result that highlights even more how little we understand about problems and their complexity. In fact, in contrast to what we know about time complexity classes, where we can solve an **NP** language with a deterministic TM requiring exponentially more time, there is not much a difference between languages decidable by NTMs or TMs, when considering space. The reason for this is that space can be *reused*, while time cannot.

In fact, the difference even disappears when considering languages for which at least polynomial space is required.

**Definition 41.** *The class of* deterministic polynomial space *languages is defined as*

$$\mathbf{PSPACE} = \bigcup_{c \geq 1} \mathrm{DSPACE}(n^c),$$

*while the class of* non-deterministic polynomial space *languages is defined as*

$$\mathbf{NPSPACE} = \bigcup_{c \geq 1} \mathrm{NSPACE}(n^c).$$

By Savitch's theorem, we immediately get

**Corollary 2.** **PSPACE** = **NPSPACE**.

*Proof.* It follows by Savitch's Theorem and from the fact that any polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$, when squared, i.e., $p(n)^2$, is still a polynomial. □

So, when going to polynomial space TMs, there is actually no difference in power between using non-determinism or not.

We conclude this lecture, by presenting the overall picture regarding all the complexity classes we have seen up to this point. To draw this picture, we need first to know how our space and time classes are related to each other.

**LOGSPACE** $\subseteq$ **NL**, by definition.

**PSPACE** = **NPSPACE**, as we have just shown.

**NL** $\subseteq$ **NPSPACE** by definition, and thus **NL** $\subseteq$ **PSPACE**, since we have shown **PSPACE** = **NPSPACE**.

Let us now compare our space classes with the time classes.

**LOGSPACE** $\subseteq$ **P**. Assume $M$ is a DTM that *decides* a language $\mathcal{L}$ using $O(\log_2 n)$ space. The fact that $M$ *decides* the language $\mathcal{L}$ implies that $M$ always halts. Since $M$ always halts and can use at most $O(\log_2 n)$ space, this means that $M$ can visit at most $O(2^{\log_2 n}) = O(n)$ possible IDs (if it visits more than that, it means it is visiting an already visited ID and thus the machine loops). Then, the number of steps is polynomial. Hence, $M$ is a polynomial-time DTM deciding the language $\mathcal{L}$, i.e., $\mathcal{L}$ is in **P**.

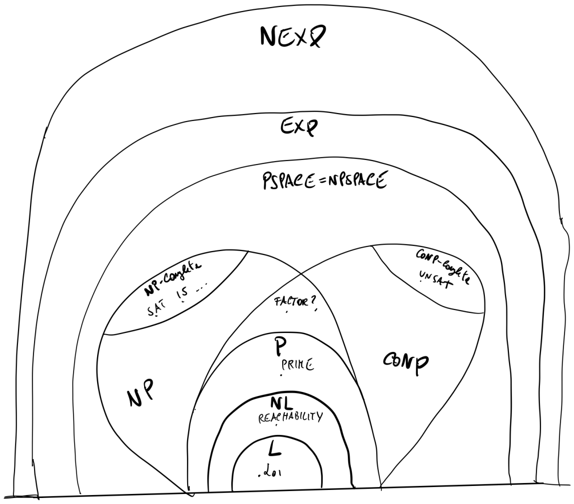
**NL** $\subseteq$ **P**. To prove this, note that we have a polynomial-time algorithm solving REACHABILITY (the breadth first algorithm). Since REACHABILITY is **NL**$-$complete, we can solve every language $\mathcal{L}$ in **NL** by first converting the input string $w$ of $\mathcal{L}$ to a graph $G$ and nodes $s, t$, using the reduction from

$\mathcal{L}$ to REACHABILITY. Since the reduction is a logspace reduction, from what we said above, it can only perform polynomially many steps. Then, we run the breadth first algorithm over this graph, and overall decide $\mathcal{L}$ in polynomial time.

**NP** $\subseteq$ **PSPACE**. If $\mathcal{L}$ is a language in **NP**, and thus has a NTM $M$ deciding $\mathcal{L}$ in polynomial time, it means $M$ performs at most a polynomial number of steps, and thus cannot use more than polynomially many cells. Hence $M$ is a NTM using polynomial space deciding $\mathcal{L}$, i.e., $\mathcal{L} \in$ **NPSPACE** $=$ **PSPACE**.

**coNP** $\subseteq$ **PSPACE**. For this, let $\mathcal{L}$ be a language in **coNP**. So, $\bar{\mathcal{L}} \in$ **NP**. Since we have shown that **NP** $\subseteq$ **PSPACE**, we conclude $\bar{\mathcal{L}} \in$ **PSPACE**. Thus, there a *deterministic* TM $M$ that decides $\bar{\mathcal{L}}$ in polynomial space. Since $M$ is deterministic, we can invert $q_{\texttt{accept}}$ with $q_{\texttt{reject}}$ in $M$, and obtain a TM that decides the complement of $\bar{\mathcal{L}}$, i.e., $\mathcal{L}$, with polynomial space. Hence, $\mathcal{L} \in$ **PSPACE**, and thus **coNP** $\subseteq$ **PSPACE**.

**PSPACE** $\subseteq$ **EXP**. For this, we use the same reasoning we used to prove **LOGSPACE** $\subseteq$ **P**: every DTM $M$ deciding some language $\mathcal{L}$ using $O(n^c)$ cells can only visit $O(2^{n^c})$ IDs. Hence, $M$ requires exponentially many steps to decide $\mathcal{L}$.



# 20 Turing Machines with Oracles, the Polynomial Hierarchy, and Search Problems

We have seen a pletora of decision problems, belonging to a good variety of complexity classes, spanning from very low space usage like **LOGSPACE** up to exponential-time classes like **EXP** and **NEXP**.

All such classes were defined as the sets of all languages that can be decided by a certain kind of TM (i.e., a polynomial-time TM, or a polynomial space TM, and so on), and in most cases, the TMs were required to perform their computation "on their own", i.e., without relying on any help by external "subroutines".

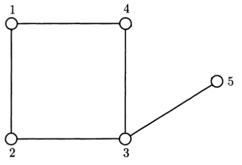
However, we are quite used to the fact that when solving a problem via an algorithm, one can decompose the problem into sub-problems, and then solving the main problem boils down to solve the sub-problems first, and then combine the solutions. This requires our main algorithm to be able to make calls to other algorithms, that are in charge of solving the sub-problems.

Consider, for example, the following language:

$$\begin{aligned}\text{MIN-VCOVER} = \{(G,k) \mid\ & G \text{ is an undirected graph and}\\ & k \text{ is the size of the the smallest vertex cover of } G\}.\end{aligned}$$

Recall that a vertex cover of $G = (V, E)$ is a set of nodes $VC \subseteq V$ such that every edge of $G$ is "touched" by some node in $VC$, i.e., $\forall \{u, v\} \in E$, either $u \in VC$, or $v \in VC$ or both $u, v \in VC$.

Note that the above language is different from VCOVER. VCOVER asks if $G$ has a vertex cover of size *at most* $k$. The above asks, given $(G, k)$, whether $k$ is *optimal*, i.e., the smallest vertex cover of $G$ is of size $k$. Consider the following graph:



A possible vertex cover for the above graph is $VC_1 = \{2, 4, 5\}$. However, this vertex cover is not of minimum size. In particular, we can find a smaller vertex cover $VC_2 = \{1, 3\}$. It is not difficult to verify that there are no smaller vertex covers than $VC_2$. Thus, $(G, 2) \in$ MIN-VCOVER, if $G$ denotes the graph in the above example.

So, how difficult is to decide MIN-VCOVER? What is its complexity?

Definitely, MIN-VCOVER can be easily placed in **PSPACE**, since a naive algorithm deciding MIN-VCOVER, given $(G, k)$, simply needs to try all possible sets of nodes $VC \subseteq V$. At each iteration, if $VC$ is a vertex cover of a size smaller than the previous ones, the algorithm updates the current minimum with the new size. The algorithm only needs to store, at each iteration, the set $VC$ (which requires $O(|V|)$ space), and the current minimum, which is a binary number $m$ that is at most $|V|$, and thus requires $O(\log_2 |V|)$ space.

Can we do better? Is MIN-VCOVER in **NP**? or is MIN-VCOVER in **coNP**? Towards showing MIN-VCOVER is in **NP**, let us analyse first the non-deterministic procedure deciding VCOVER. Let $(G, k)$ be the input, with $G = (V, E)$.

1. Guess a set of nodes $VC \subseteq V$.

2. Verify that $|VC| \leq k$ and that $VC$ is a vertex cover of $G$.

Note that the above procedure, alone, is not able to conclude if $k$ is the size of the smallest vertex cover, since finding one of size at most $k$ does not mean that there is no other vertex cover of smaller size. Indeed, to decide MIN-VCOVER we should verify *two* things: that $G$ has a vertex cover of size at most $k$ (which is what the above procedure does), but also verify that $G$ has **no** vertex cover of smaller size, i.e., of size at most $k - 1$.

So, it seems that to be able to decide MIN-VCOVER, we are required to solve both the problem VCOVER (since we ask if $G$ has a vertex cover of size at most $k$), and the problem $\overline{\text{VCOVER}} \in$ **coNP** (since we ask if **no** vertex cover of $G$ is of size at most $k - 1$). Since we believe **NP** $\neq$ **coNP**, it is unlikely that the above process can be carried out by a single non-deterministic procedure, in polynomial time (i.e., in **NP**), as this would imply an NTM can decide a **coNP**$-$complete language ($\overline{\text{VCOVER}}$) in polynomial time, and thus **NP** $=$ **coNP**.

Is there a better procedure that avoids the above issue? We are not aware of any, and if you try to place MIN-VCOVER in **coNP** by solving its complement, you will end up in a similar situation. So, we do not know if MIN-VCOVER is in **NP** or in **coNP**, and most find it unlikely, for the reasons discussed above.

So, the best we can do, at the moment, to analyse the complexity of MIN-VCOVER, is to exploit the fact, as we observed above, that MIN-VCOVER can be rephrased/redefined in terms of the language VCOVER. That is, MIN-VCOVER can be rewritten as:

$$\text{MIN-VCOVER} = \{(G, k) \mid (G, k) \in \text{VCOVER } \textbf{and } (G, k{-}1) \notin \text{VCOVER}\}.$$

So, we have rephrased our language as a language where we need to ask different questions. In this case, we are asking **two** questions regarding an **NP** language. One is whether $G$ has a vertex cover of size at most $k$, i.e., $(G, k) \in$ VCOVER, and the other is whether $G$ has no vertex cover of size at most $k - 1$, i.e., $(G, k - 1) \notin$ VCOVER. Hence, an algorithm that solves MIN-VCOVER is the one that uses some "sub-routine" that is able to decide VCOVER. Our algorithm only needs to call the sub-routine twice and make a decision on the basis of the results obtained from the above two calls. Assume $\mathsf{check} - \mathsf{vcover}$ is an algorithm that decides VCOVER.

1. Let $\mathsf{result}_1 := \mathsf{check} - \mathsf{vcover}(G, k)$;
2. Let $k' := k - 1$;
3. Let $\mathsf{result}_2 := \mathsf{check} - \mathsf{vcover}(G, k')$;
4. If $\mathsf{result}_1 = \text{true}$ and $\mathsf{result}_2 = \text{false}$, then **Accept**, otherwise **Reject**.

The above procedure shows that, assuming we have a sub-routine deciding VCOVER, then we only require a polynomial number of steps, to decide MIN-VCOVER.[24]

The above algorithm is what we call a TM with an *oracle*, i.e., a TM that can ask to an oracle whether a string is in a certain language o not.

**Remark.** Note that we are *not* claiming that MIN-VCOVER can be decided in polynomial time. What we are saying is that the main difficulty in deciding MIN-VCOVER lies in asking questions to the oracle, but excluding that, the rest is “easy”, i.e., in polynomial time. This gives us an indication of the complexity of MIN-VCOVER: it is “easy” to solve, for the most part, excluding the difficult part of asking questions to the oracle.

We now formally define this notion.

**Definition 42.** *Consider a language $\mathcal{L}$. A TM $M$ with an oracle for $\mathcal{L}$ is a standard TM, with the following additional elements:*

1. *An additional read/write tape, called the* oracle *tape;*
2. *Three additional states $q_?, q_{yes}, q_{no}$.*

The way a TM $M$ with an oracle for $\mathcal{L}$ performs its computation is similar to the one of any TM. The main difference is that when $M$ transitions to state $q_?$, then $M$ will automatically move, in one step, to the state $q_{\mathsf{yes}}$, if the string written in the oracle tape belongs to $\mathcal{L}$, otherwise, it moves to $q_{\mathsf{no}}$.

This is our way of saying that $M$ can ask questions to a sub-routine that is able to decide some language $\mathcal{L}$. Indeed, if $M$ needs to know if a certain string $w$ is in $\mathcal{L}$, before continuing its computation, it just needs to write $w$ in the oracle tape, and then move to the state $q_?$. If after this, $M$ is in state $q_{\mathsf{yes}}$, then $M$ concludes that the answer to the question $w \in \mathcal{L}$ is yes.

**Remark.** Note that we assume $M$ moves from $q_?$ to $q_{\mathsf{yes}}$ or $q_{\mathsf{no}}$ in *one step*. That is, we do not want to consider the time required by the oracle to answer the question. We do this, because usually we already know what is the complexity of the oracle $\mathcal{L}$, and we instead want to understand the actual amount of work that the main algorithm described by $M$ performs.

We can now define different complexity classes, based on the notion of TMs with an oracle.

[24] It might seem we require a constant number of steps, i.e., 4, but we must compute $k'$, which requires some time that depends on the number of bits $k$.

**Definition 43.** *If $\mathcal{C}$ is some complexity class (e.g., **P**, **NP**, etc.). We define*

$$\mathbf{P}^{\mathcal{C}} = \{\mathcal{L} \mid \mathcal{L} \textit{ can be decided by a polynomial-time DTM with an oracle for some language } \mathcal{L}' \in \mathcal{C}\}.$$

$$\mathbf{NP}^{\mathcal{C}} = \{\mathcal{L} \mid \mathcal{L} \textit{ can be decided by a polynomial-time NTM with an oracle for some language } \mathcal{L}' \in \mathcal{C}\}.$$

For example, $\mathbf{P}^{\mathbf{NP}}$ collects all the languages that can be decided by a polynomial-time procedure that can ask questions to an oracle for some **NP** language. MIN-VCOVER is an example of such a language. In fact, the algorithm we have shown before performs a polynomial number of steps, and makes some calls (two) to an oracle for the **NP** language VCOVER. Hence, MIN-VCOVER $\in \mathbf{P}^{\mathbf{NP}}$.

Note that every language $\mathcal{L} \in \mathbf{NP}$ necessarily is in $\mathbf{P}^{\mathbf{NP}}$, i.e., $\mathbf{NP} \subseteq \mathbf{P}^{\mathbf{NP}}$, for the simple reason that a trivial procedure deciding $\mathcal{L}$ is the one that takes as input a string $w$, asks an oracle for $\mathcal{L}$ if $w \in \mathcal{L}$, and then accepts/rejects accordingly. That is, this procedure is essentially a one-line algorithm that simply directly passes control to a sub-routine for $\mathcal{L}$. More interestingly, for each $\mathcal{L} \in \mathbf{coNP}$, $\mathcal{L}$ is also in $\mathbf{P}^{\mathbf{NP}}$. This is because TMs with oracles can retrieve an answer from the oracle, and then flip it, if they want. Thus, we can decide $\mathcal{L}$ as follows. Let $\bar{\mathcal{L}}$ be the complement of $\mathcal{L}$, which is in **NP**, by definition. Given a string $w$, we decide if $w \in \mathcal{L}$ by simply asking an oracle for $\bar{\mathcal{L}}$ if $w \in \bar{\mathcal{L}}$. If this is the case, reject, otherwise, accept. Thus, $\mathbf{coNP} \subseteq \mathbf{P}^{\mathbf{NP}}$.

**The Polynomial Hierarchy.** There exist many other languages that require even more complex TMs with oracles. For example, languages that are in $\mathbf{NP}^{\mathbf{NP}}$, or even languages that are in $\mathbf{NP}^{\mathbf{NP}^{\mathbf{NP}}}$. The latter means they can be solved in polynomial time by a NTM with an oracle for a language in $\mathbf{NP}^{\mathbf{NP}}$. One can keep defining larger and larger classes, in this way, and, indeed, this is what it has been done. We actually define an infinite hierarchy of complexity classes, called the *polynomial time hierarchy.*

Here we just quickly define it, and place it w.r.t. the other complexity classes we have seen so far.

**Definition 44.** *First, we define $\Sigma^{\mathrm{p}}_1 = \mathbf{NP}$. Then, for each $i \geq 1$, we define*

$$\Sigma^{\mathrm{p}}_{\mathrm{i}+1} = \mathbf{NP}^{\Sigma^{\mathrm{p}}_{\mathrm{i}}},$$

$$\Pi^{\mathrm{p}}_{\mathrm{i}} = \mathbf{co}\Sigma^{\mathrm{p}}_{\mathrm{i}} = \{\mathcal{L} \mid \bar{\mathcal{L}} \in \Sigma^{\mathrm{p}}_{\mathrm{i}}\},$$

$$\Delta^{\mathrm{p}}_{\mathrm{i}+1} = \mathbf{P}^{\Sigma^{\mathrm{p}}_{\mathrm{i}}},$$

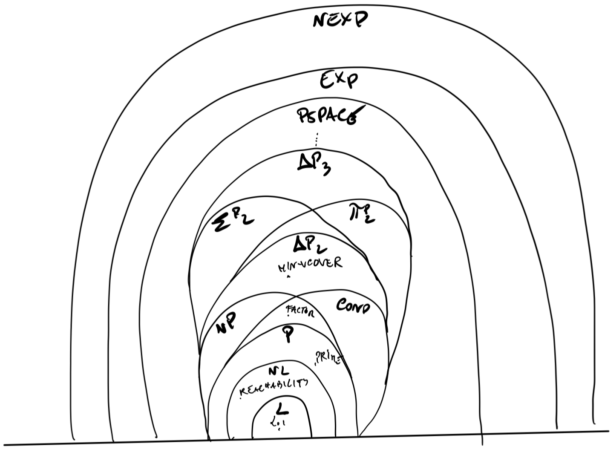
So, for example, $\Pi^{p}_1 = \mathbf{coNP}$. As another example, $\Sigma^{\mathrm{p}}_3 = \mathbf{NP}^{\Sigma^{\mathrm{p}}_2} = \mathbf{NP}^{\mathbf{NP}^{\mathbf{NP}}}$. Moreover, $\Pi^{\mathrm{p}}_3$ is the set of all languages whose complement is in $\Sigma^{\mathrm{p}}_3$. Finally, $\Delta^{\mathrm{p}}_3 = \mathbf{P}^{\Sigma^{\mathrm{p}}_2} = \mathbf{P}^{\mathbf{NP}^{\mathbf{NP}}}$.

Note that $\Delta_2^{\mathrm{p}} = \mathbf{P}^{\mathbf{NP}}$, and thus MIN-VCOVER $\in \Delta_2^{\mathrm{p}}$. We now discuss the relative inclusion of the above complexity classes.

We first show that $\Sigma_{\mathrm{i}}^{\mathrm{p}} \subseteq \Delta_{\mathrm{i+1}}^{\mathrm{p}}$, and $\Pi_{\mathrm{i}}^{\mathrm{p}} \subseteq \Delta_{\mathrm{i+1}}^{\mathrm{p}}$. By definition, $\Delta_{\mathrm{i+1}}^{\mathrm{p}} = \mathbf{P}^{\Sigma_{\mathrm{i}}^{\mathrm{p}}}$, and thus, similarly to what we said for the inclusion $\mathbf{NP} \subseteq \mathbf{P}^{\mathbf{NP}}$, trivially $\Sigma_{\mathrm{i}}^{\mathrm{p}} \subseteq \mathbf{P}^{\Sigma_{\mathrm{i}}^{\mathrm{p}}} = \Delta_{\mathrm{i+1}}^{\mathrm{p}}$. Similarly to what we discussed for the inclusion $\mathbf{coNP} \subseteq \mathbf{P}^{\mathbf{NP}}$, since TMs with an oracle can flip the answer they get from the oracle, we conclude that $\Pi_{\mathrm{i}}^{\mathrm{p}} \subseteq \mathbf{P}^{\Sigma_{\mathrm{i}}^{\mathrm{p}}} = \Delta_{\mathrm{i+1}}^{\mathrm{p}}$.

What is left to prove is that $\Delta_{\mathrm{i}}^{\mathrm{p}} \subseteq \Sigma_{\mathrm{i}}^{\mathrm{p}}$ and $\Delta_{\mathrm{i}}^{\mathrm{p}} \subseteq \Pi_{\mathrm{i}}^{\mathrm{p}}$. By definition, $\Delta_{\mathrm{i}}^{\mathrm{p}} = \mathbf{P}^{\Sigma_{\mathrm{i-1}}^{\mathrm{p}}}$, and $\Sigma_{\mathrm{i}}^{\mathrm{p}} = \mathbf{NP}^{\Sigma_{\mathrm{i-1}}^{\mathrm{p}}}$. Thus, any language that can be decided by a TM with an oracle in $\Sigma_{\mathrm{i-1}}^{\mathrm{p}}$, in polynomial time, can surely be decided by a NTM with the same oracle, in polynomial time (the NTM simply does not use any non-determinism at all). Thus, $\Delta_{\mathrm{i}}^{\mathrm{p}} \subseteq \Sigma_{\mathrm{i}}^{\mathrm{p}}$. To show $\Delta_{\mathrm{i}}^{\mathrm{p}} \subseteq \Pi_{\mathrm{i}}^{\mathrm{p}}$, consider a language $\mathcal{L} \in \Delta_{\mathrm{i}}^{\mathrm{p}} = \mathbf{P}^{\Sigma_{\mathrm{i-1}}^{\mathrm{p}}}$. This means there is a *deterministic* TM $M$ that decides $\mathcal{L}$, using an oracle for some language in $\Sigma_{\mathrm{i-1}}^{\mathrm{p}}$. Since $M$ is deterministic, the machine $M'$, where we invert $q_{\texttt{accept}}$ with $q_{\texttt{reject}}$ in $M$ decides the complement $\bar{\mathcal{L}}$ of $\mathcal{L}$. That is, $\bar{\mathcal{L}} \in \Delta_{\mathrm{i}}^{\mathrm{p}}$, and thus $\Delta_{\mathrm{i}}^{\mathrm{p}} = \mathbf{co}\Delta_{\mathrm{i}}^{\mathrm{p}}$. Since $\bar{\mathcal{L}} \in \Delta_{\mathrm{i}}^{\mathrm{p}}$, and $\Delta_{\mathrm{i}}^{\mathrm{p}} \subseteq \Sigma_{\mathrm{i}}^{\mathrm{p}}$, as we have shown above, we conclude that $\bar{\mathcal{L}} \in \Sigma_{\mathrm{i}}^{\mathrm{p}}$. Hence, its complement $\mathcal{L} \in \Pi_{\mathrm{i}}^{\mathrm{p}}$. Thus, $\Delta_{\mathrm{i}}^{\mathrm{p}} \subseteq \Pi_{\mathrm{i}}^{\mathrm{p}}$.

So, by using the above two kinds of inclusions, we conclude that our complexity classes are related as follows.



The only inclusion we did not prove (and we are not going to do it), is that every class of the polynomial hierarchy is contained in **PSPACE**.

Just to give some example of languages that belong to the various levels of the polynomial hierarchy, we mention some generalizations of SAT. Recall that SAT is the decision problem asking if some Boolean formula in CNF $\varphi$ is satisfiable. This can be rephrased as asking if a formula of the form

$$\exists x_1, \ldots, x_n : \varphi$$

is true, where $x_1, \ldots, x_n$ are the Boolean variables occurring in $\varphi$. That is, we want to know if there *exist* truth values, for each Boolean variable, such that $\varphi$ becomes true. One can generalize the above problem by considering formulas of the form:

$$\exists x_1, \ldots, x_n : \forall y_1, \ldots, y_m : \varphi.$$

That is, do *exist* truth values for $x_1, \ldots, x_n$, such that, *for all* truth values we assign to $y_1, \ldots, y_m$, $\varphi$ is true? We call formulas of the form above $\exists\forall$-Boolean formulas. $\exists\forall$-SAT is the language of $\exists\forall$-Boolean formulas in CNF that are true. $\exists\forall$-SAT is in $\Sigma_2^{\mathrm{p}}$, and it is actually $\Sigma_2^{\mathrm{p}}-$complete, where completeness is defined as usual, by considering polynomial-time reductions.

One can go one step further, and consider $\underbrace{\exists\forall\exists\forall\cdots}_{k}$-SAT, where we have $k$ quantifiers, in which case, $\underbrace{\exists\forall\exists\forall\cdots}_{k}$-SAT is $\Sigma_{\mathrm{k}}^{\mathrm{p}}-$complete. The corresponding complements are $\Pi_{\mathrm{k}}^{\mathrm{p}}-$complete.

**Solving search problems using their decision version.** At the beginning of this course, we anticipated that decision problems will be also useful to understand the complexity of their search version. Indeed, TMs with oracles are exactly the tool we need to transfer knowledge from a decision problem, to its search version.

For example, what if we want to solve the *search problem* corresponding to VCOVER? That is

Given an undirected graph $G$, *compute* the size $k$ of the smallest vertex cover of $G$.

Recall that we represent search problems as functions. So, the above problem can be defined as the function FMIN-VCOVER (that stands for functional minimum vertex cover) such that, for every undirected graph $G$,

$$\text{FMIN-VCOVER}(G) = \min\{|VC| \mid VC \text{ is a vertex cover of } G\}.$$

To solve the above problem, we should equip our TMs with an output tape, as we did for our reductions, and let the TM write the result in the output tape. As we did with our reductions, in this case our TM always accepts.

**Remark.** Note that only deterministic TMs have a clear notion of "output", since when they are executed, they only follow one computation path, and thus, given an input string $w$, the output is only determined by the content of $w$. On the other hand, for non-deterministic TMs, the output is not determined by $w$ alone, but also by the non-deterministic choices made by the machine. Hence, what would be **the** output of an non-deterministic TM? To avoid these issues, we only consider *deterministic* TMs with output tapes.

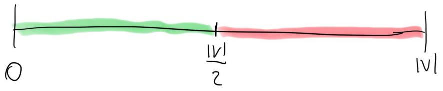
We show we can solve FMIN-VCOVER with a TM with an oracle for VCOVER.

```
1. Let $k := |V| - 1$;
2. while $(G, k) \in$ VCOVER, do
3.    (a) $k := k - 1$;
4. Write $k + 1$ in the output tape.
```

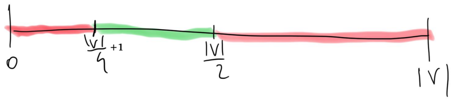
The main idea is to start asking the oracle if $G$ has a vertex cover of size at most $|V| - 1$ (there is no point in asking if it is of size at most $|V|$, because the answer is trivially yes). If this is the case, then we check if $G$ as a smaller vertex cover, i.e., of size at most $|V| - 2$. If this is the case, we check smaller vertex covers, and so on. The algorithm keeps going, until it finds some $k$ that is so small that no vertex cover of at most that size exists in $G$. This means that the previous $k$ is the minimum size of a vertex cover of $G$, and thus it outputs $k + 1$. The procedure requires $O(|V|)$ steps, and it asks questions to an oracle for the **NP** language VCOVER.

In which class does FMIN-VCOVER belong to? Since FMIN-VCOVER is a search problem, and not a decision problem, formally, we cannot place it in $\mathbf{P}^{\mathbf{NP}}$. However, we can define **FP** as the class of search problems (i.e., functions) that can be solved by a TM with output tape, in polynomial time. Hence, FMIN-VCOVER is in $\mathbf{FP}^{\mathbf{NP}}$.

Can we do better than this? Actually, we can devise a better algorithm solving FMIN-VCOVER. In fact, we can speed up the algorithm by employing binary search. The idea is to start asking the oracle if $G$ has a vertex cover of size at most $k = |V|/2$. If for example this is the case, then the smallest vertex covers must have size between 0 and $|V|/2$, and thus there is no point in considering a value of $k$ greater than $|V|/2$. This is shown in the figure below.



Now, we ask the oracle if $G$ has a vertex cover of size at most $k = |V|/4$. If, for example, the answer is no, it means that we have to look for vertex covers of size between $|V|/4 + 1$ and $|V|/2$, as shown in the picture below, and so on.



The algorithm continues like this until the green interval contains only one number $k$. This number is the size of the smallest vertex covers in $G$.

The procedure is still a polynomial time procedure, since it needs to initialize $k$ to $|V| - 1$, and thus must scan the whole input, and must also copy the data for the oracle in the oracle tape. However, this time, how many calls to the oracle does the algorithm make? It makes $O(\log_2 |V|)$ calls, in the worst case, rather than $O(|V|)$, as in the previous procedure.

To specify this, we define another complexity class, $\mathbf{FP}^{\mathbf{NP}[\log_2 n]}$, which collects the search problems that can be solved in polynomial time, by making $O(\log_2 n)$ calls to an oracle for a language in **NP**, where $n$ is the length of the input. We have that FMIN-VCOVER $\in \mathbf{FP}^{\mathbf{NP}[\log_2 n]}$.

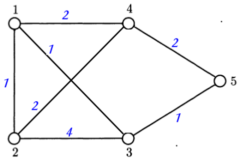
**Remark.** Note that knowing that the search problem FMIN-VCOVER uses its corresponding decision problem VCOVER as an oracle tells us something very important about the complexity of FMIN-COVER. Indeed, if we were able to solve FMIN-VCOVER in polynomial time, i.e., FMIN-VCOVER $\in \mathbf{FP}$, then, we could solve VCOVER in polynomial time. In fact, if FMIN-VCOVER $\in \mathbf{FP}$, we could check, given $(G, k)$, if there is a vertex cover of size at most $k$, by first computing the minimum size $k'$ of a vertex cover of $G$ in polynomial time, and then just verify that $k' \leq k$. Thus, we conclude that FMIN-VCOVER cannot be solved in polynomial time, unless VCOVER is in **P**, and thus, unless $\mathbf{P} = \mathbf{NP}$. Thus, as we anticipated in the very first lecture of this course, decision problems are not only interesting on their own, but also allow us to draw useful conclusions on their more general search version.

# 21 The Travelling Salesman Problem

In this lecture, we see another example of search problem, whose complexity can be understood by analysing the complexity of its decision version. To define this search problem, we first need to introduce a couple of notions.

**Definition 45.** *A* weighted *directed (resp., undirected) graph is a triple $G = (V, E, \lambda)$, where $(V, E)$ is a directed (resp., undirected graph) and $\lambda : E \to \mathbb{N}$ is a function assigning a weight to each edge in $E$.*

The following is an example of a weighted, undirected graph, where the weights are highlighted in blue.



The *cost* of a path (or a cycle) in a weighted graph like the one above is the sum of the weights of all edges in the path. So, for example, the cost of the cycle $1, 4, 5, 3, 2, 1$ is $2 + 2 + 1 + 4 + 1 = 10$.

The next important notion we need, in order to introduce our search problem, is the one of Hamiltonian cycle.

**Definition 46.** *An* Hamiltonian cycle *of a (possibly weighted) directed or undirected graph $G$ is a cycle of $G$ that visits every node of $G$ exactly once (excluding of course the fact that the starting and ending nodes must coincide).*

For example, the cycle $1, 4, 5, 3, 2, 1$ is an Hamiltonian cycle, since it visits every node of the graph (i.e., no node is left out), and each node is visited only once. The cycle $1, 2, 4, 1$ is not Hamiltonian, since although it does not visit the same nodes more than once, it leaves other nodes of the graph out.

This notion is at the basis of the Travelling Salesman Problem.

**Definition 47.** *The (Functional) Travelling Salesman Problem (FTSP) is the problem of computing, given a weighted undirected graph $G = (V, E, \lambda)$, the minimum cost of an Hamiltonian cycle of $G$, if $G$ has any, otherwise some default symbol/value (e.g., $\bot$) must be returned.*

For example, the Hamiltonian cycle $1, 4, 5, 3, 2, 1$ of the graph above has cost 10, while the Hamiltonian cycle $1, 3, 5, 4, 2, 1$ has cost 7. There are no other Hamiltonian cycles, and thus the minimum cost is 7.

The above search problem is a very important problem in logistics, as it can be restated as

> *"Given a set of cities, connected by some roads, where each road has a certain length, e.g., in kilometres, what is the shortest distance that a courier must travel, to be able to perform its delivery to every city exactly once, before going back to the deposit?"*

How can we solve the above search problem? We can do it, by using an oracle to the decision version of FTSP, which we simply call TSP, defined as follows.

$$\begin{aligned}\text{TSP} = \{(G,k) \mid\ & G \text{ is a weighted, undirected graph, and}\\ & G \text{ has an hamiltonian cycle of cost at most } k\}.\end{aligned}$$

We can very easily prove that TSP is in **NP**. Indeed, it is enough, given $(G,k)$, with $G=(V,E)$, to guess a sequence of $|V|$ nodes $v_1,\ldots,v_n$ (which is of polynomial size), and then verify that all nodes are different, $v_i$ is connected to $v_{i+1}$ by an edge, $v_n$ is connected to $v_1$ by an edge, and the cost of this cycle is smaller than $k$.

How can we solve FTSP using an algorithm having an oracle for TSP? We can use the same approach we used for FMIN-VCOVER. We try all costs $k$, starting from the largest, and ask the oracle for TSP if the graph $G$ has an Hamiltonian cycle of cost at most $k$. Once the answer is no, the previous $k$ is the minimum cost. Let $G=(V,E,\lambda)$ be the input weighted, undirected graph.

```
1. Let $k$ be the sum of all costs of all edges in $G$;
2. If $(G,k) \notin$ TSP, write $\perp$ in the output tape, and halt.
3. while $(G,k) \in$ TSP, do
4.    (a) $k := k-1$;
5. Write $k+1$ in the output tape;
```

The main difference with the algorithm for FMIN-VCOVER is that $G$ might not have an Hamiltonian cycle at all, and then we first check if it has any, by using the maximum cost possible. If at least an Hamiltonian cycle exists, we can proceed to search the minimum cost, otherwise a special symbol $\perp$ is outputted, to specify no minimum cost could be found.

At first sight, it might appear that the above algorithm requires a polynomial number of steps. However, we must be careful in our analysis, because the number of steps the algorithm performs is linear in the maximum cost. However, each cost associated to each edge is a binary number. If $m$ is the number of bits used to encode each cost, summing up $|E|$ such costs can lead to a number with $m+|E|$ bits at most. Thus, the value to which $k$ is initialized in line 1, is a $O(2^{m+|E|})$, and thus, the while loop performs exponentially many iterations. To solve the problem, we can use binary search, as we did for FMIN-VCOVER. We first define an interval $[a,b]$ of plausible costs of the minimum Hamiltonian cycle, where $a=0$ and $b$ is the maximum possible cost. Then, we iteratively discard half of the current interval, depending on the answer we obtain from the oracle. Moreover, while doing this, if we realize the middle point of the interval we are considering is the optimum, the procedure writes the result and halts.

```
1. Let $a = 0$ and let $b$ be the sum of all costs of all edges in $G$;
```

```
2. while $a \leq b$, do
3.   (a) Let $k = \lfloor (a+b)/2 \rfloor$;
     (b) If $(G, k-1) \in$ TSP, then let $b = k-1$;
     (c) Else, if $(G, k) \in$ TSP, write $k$ in the output tape, and halt;
     (d) Else, let $a = k+1$;
4. Write the symbol $\perp$ in the output tape.
```

After constructing the initial large interval $[a, b]$, the algorithm must verify one the following three things. If the cost of a Hamiltonian cycle is strictly smaller than the middle point $k$ (i.e., we ask $(G, k-1) \in$ TSP) then we move the left interval, excluding also $k$. If this is not the case, but the an Hamiltonian cycle has cost at most $k$, then $k$ is necessarily the cost we are looking for. Otherwise, it means the cost is strictly larger than $k$, and then we set $a = k+1$. If the graph has no Hamiltonian cycle at all, the condition on the while loop will eventually become false, and then, the algorithm simply writes a special symbol denoting that no minimum cost could be found.

The above algorithm performs $O(\log_2 C)$ iterations, where $C$ is the maximum cost computed in $b$ at line 1. Since $C \in O(2^{m+|E|})$, the overall number of iterations is $O(\log_2 2^{m+|E|}) = O(m + |E|)$, so it is polynomial in the size of the input. So, binary search not always allows us to conclude that the problem is in $\mathbf{FP}^{\mathbf{NP}[\log_2 n]}$, because it depends on the interval on which the algorithm is working. In this case, we have shown that FTSP is in $\mathbf{FP}^{\mathbf{NP}}$.

The final question is, can we do better? Can we solve FTSP in polynomial time? That is, is FTSP $\in \mathbf{FP}$? To answer this question, again, observe that FTSP is more general than TSP, and if we were able to to solve FTSP in polynomial time, we could solve TSP in polynomial time, by computing the minimum cost, and verify it is smaller than the given $k$. So, if TSP is $\mathbf{NP}-$complete, there is no hope to solve FTSP in polynomial time, unless $\mathbf{P} = \mathbf{NP}$.

The rest of this lecture is devoted to prove that TSP is $\mathbf{NP}-$complete, and thus we can rule out efficient algorithms for solving FTSP (unless $\mathbf{P} = \mathbf{NP}$).

## 21.1 Directed Hamiltonian cycle

To prove TSP is $\mathbf{NP}-$complete, we proceed in different steps. First, we focus on a language that is closely related to TSP, for which it will be easier to prove $\mathbf{NP}-$completeness. Then, we show how to reduce this language to TSP. The language is

$$\begin{aligned}\text{DIRECTED-HAMCYLE} = \{G \mid G \text{ is a (not weighted) directed graph, and}\\ G \text{ has an hamiltonian cycle}\}.\end{aligned}$$

**Theorem 43.** *DIRECTED-HAMCYCLE is* $\mathbf{NP}-$complete

*Proof.* Membership in **NP** is easy, and can be proved in the same way we did for TSP: guess a sequence of nodes, and verify it is an Hamiltonian cycle. We focus on the **NP**−hardness. For this, we show a polynomial-time reduction from 3-SAT to DIRECTED-HAMCYCLE. Recall that

$$\text{3-SAT} = \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula in CNF} \\ \text{with at most 3 literals per clause }\}.$$

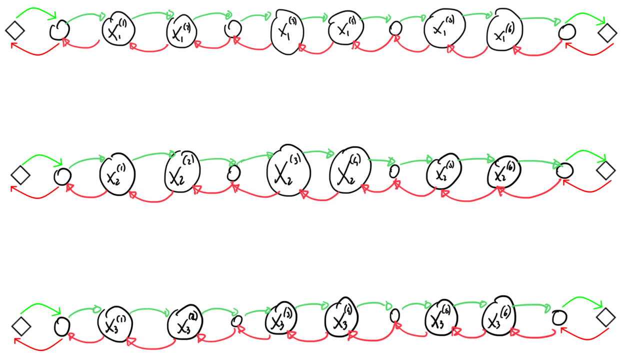
First, note that if a clause of a CNF formula $\varphi$ contains both a variable $x_i$ and its negation $\neg x_i$, this clause is always satisfiable, and thus can be removed from $\varphi$, without changing the satisfiability of $\varphi$. Thus, our reduction, before doing anything else, first removes these clauses. This is to simplify the reduction later. Assume $\varphi$ is the formula after removing the above clauses.

So, the goal is, given $\varphi$, to construct a directed graph $G$ such that $\varphi$ is satisfiable iff $G$ has an Hamiltonian cycle. Moreover, the procedure must require polynomially many steps. We describe the reduction using the following example Boolean formula:

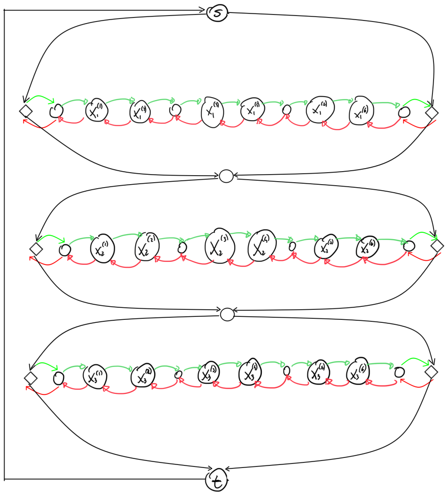
$$\varphi = \underbrace{(x_1 \vee x_2)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(\neg x_2 \vee \neg x_3)}_{C_3}.$$

The general reduction should be clear after we discuss it for the above formula.

Let $m$ be the number of clauses in $\varphi$, in this case $m = 3$. For each variable $x_i$ of $\varphi$, we introduce $2 \cdot m$ nodes, named $x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(2m)}$. Then, we connect all the nodes of a variable $x_i$ in a row. The row can be traversed in both directions, by surrounding every two of such nodes with a "filler" node on both sides. Moreover, the row starts and ends with a special node, which for illustration purposes, we represent with a diamond $\diamond$. The name of the filler and diamond nodes is unimportant. For example, for the variables $x_1, x_2, x_3$ of our example formula, we obtain:



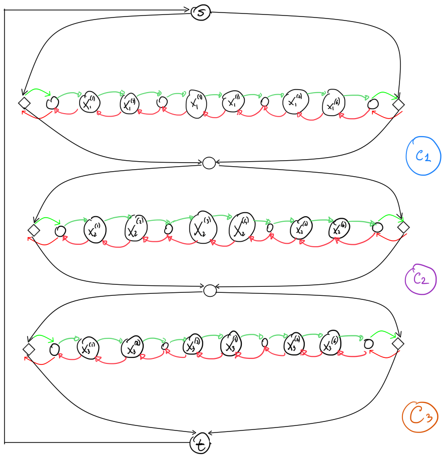
Ignoring for a second why we chose this amount of nodes for each variable, the intuition behind the above construction is that an Hamiltonian cycle will simulate choosing the value "true" for $x_i$ by visiting the row of $x_i$ by following the green path from left to right. The choice of the value “false” for $x_i$ is instead simulated by visiting the row of $x_i$ by following the red path from right to left. To allow an Hamiltonian cycle to make this “choice”, we add the following edges to the graph:



where $s$ and $t$ are two new nodes. Thus, if we want to simulate a truth assignment where $x_1$ is true, $x_2$ is false, and $x_3$ is false, then we have a corresponding Hamiltonian cycle doing the following. It starts from $s$, follows the left edge from $s$ to the left-most node of $x_1$'s row, and then follows the green path up to the right-most node. Then, since $x_2$ must be false, it follows the edge to the central node connecting $x_1$'s row with $x_2$'s row, and then follow the edge going on the right-most node of $x_2$'s row. This way, the cycle traverses $x_2$'s row following the red path from right to left. Since also $x_3$ must be false, it then follows the edge to the central node, and then the edge leading to the right-most node of $x_3$'s row, and finally traverses it using the red path. Then, it can finally reach $t$, and then goes back to $s$.

Note that any Hamiltonian cycle, once it enters a green or red path, cannot “escape” the path before it completely visits it, because to escape such a path, it is forced to go back to an already visited node.

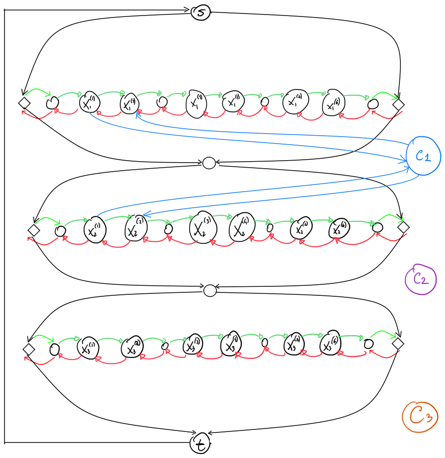
Thus, there is a one to one correspondence between all the truth assignments $\tau$ of $\varphi$ (even the ones not satisfying $\varphi$) and the Hamiltonian cycles of the graph starting from $s$. What is left to do, is to add some edges and nodes to the graph in order to only keep the Hamiltonian cycles representing truth assignments that satisfy $\varphi$. For this, we add one node for each clause of $\varphi$. In this case, 3 nodes $C_1, C_2, C_3$:



Now, our Hamiltonian cycles *must* also visit nodes $C_1, C_2, C_3$. However, we allow them to visit all three nodes $C_1, C_2, C_3$ only if they represent assignments satisfying $\varphi$. This means that the node of a clause $C_i$ can only be reached by Hamiltonian cycles that make at least one of $C_i$'s literals true.

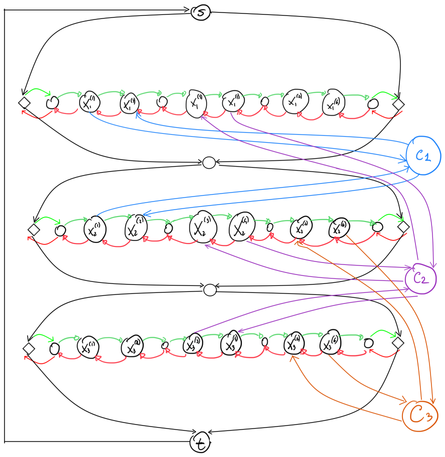
Let us focus on clause $C_1 = (x_1 \vee x_2)$. Our Hamiltonian cycle must be able to visit node $C_1$ only if it has either chosen the green path for $x_1$ or the green path for $x_2$.

To do this, we dedicate the first two "named" nodes $x_1^{(1)}$ and $x_1^{(2)}$ of $x_1$'s row and the first two named nodes $x_2^{(1)}$ and $x_2^{(2)}$ of $x_2$'s row to allow their green path to detour to $C_1$:



Thus, if, for example, $x_1$ is assigned to true (i.e., the cycle chooses the green path for $x_1$, then the cycle can detour from $x_1^{(1)}$ to $C_1$, and then go back to $x_1^{(2)}$, and continue. Similarly, for $x_2$. If $C_1$ contained a negated variable, then the detour must start from $x_1^{(2)}$ and return to $x_1^{(1)}$. We do this, because if the clause contains a negated variable $\neg x_i$, the detour must be only allowed when following the right-to-left red path of $x_i$'s row, and not when following the green path, and vice versa.

Let us now consider the second clause $C_2 = (\neg x_1 \vee \neg x_2 \vee x_3)$. Here we must allow the Hamiltonian cycle to visit node $C_2$ only when it is visiting the red path for $x_1$, or the red path for $x_2$ or the green path for $x_3$. So, we allow the detour to happen via the third and fourth named nodes of each $x_1$, $x_2$, and $x_3$. Depending on whether the variables appear negated or not, the detour will start either from the left or from the right. We do the same for clause $C_3$, and obtain the graph:



So, as an example, the truth assignment assigning all variables to false has no corresponding Hamiltonian cycle, because if we choose to traverse the graph, starting from $s$, by visiting the nodes only via the red paths, there is no way to reach all clause nodes without visiting a node twice. In particular, node $C_1$ becomes unreachable, because to reach it, we must go back to an already visited node.

Assume $\tau$ is a truth assignment satisfying $\varphi$. The corresponding Hamiltonian cycle in $G$ should be easy to imagine: start from $s$, follow the path of $x_1$ according to the truth value $\tau(x_1)$, and detour to all clauses to which we can detour. Keep going like this, and if a previous variable already detoured to a clause, skip it, and continue without detouring.

Assume instead that $G$ has an Hamiltonian cycle $C$. Since $C$ is a cycle, it does not matter from which node it starts. So, assume $C$ starts from $s$.

Note that in principle, $C$ might not traverse the nodes in the "good" way we described, i.e., when it follows a green or red path, it only detours to clauses in the direction corresponding to the colour, and "goes back" immediately to that path. However, we prove that $C$ indeed must necessarily following this scheme.

There are two cases in which $C$ does not follow the above scheme.

*1)* Assume, towards a contradiction, that $C$ starts from $s$, and follows a green path and visits $x_i^j$ and after that $x_i^{j+1}$, but now decides to detour to a clause from $x_i^{j+1}$ (this clause should not be visited in this way, but when coming from the right). If this is the case, now $C$ has only one way to visit the filler node on the right of $x_i^{j+1}$, i.e., coming from the right. But when $C$ visits the filler node from the right, $C$ will get "stuck" on the filler node (as it has no outgoing edges, besides the one going back to $x_i^j$), and thus cannot reach $s$ and close the cycle. This is also true if $x_i^j$ and $x_i^{j+1}$ are the last to "named" nodes of the row, i.e., $x_i^{2m-1}$ and $x_i^{2m}$, as on their right we still have a such a filler node with no outgoing edges. Similarly, if $C$ was traversing a red path, we have no detour from $x_i^j$.

*2)* Assume, towards a contradiction, that $C$ follows a green path and visits $x_i^j$, and correctly decides to detour to a clause, but it does not come back to $x_i^{j+1}$ immediately after. The only chance the cycle $C$ has to visit $x_i^{j+1}$ is then to enter it from the right filler node. But once $x_i^{j+1}$ is reached, there is no way to escape the path anymore, because $x_i^{j+1}$ has no outgoing edges (remember that $x_i$ and $\neg x_i$ cannot appear together in the clause where $C$ detoured). Thus, $C$ cannot go back to $s$. A similar argument can be used if $C$ follows a red path.

So, any Hamiltonian cycle of $G$ traverses the nodes in the expected way, and thus if $\varphi$ is not satisfiable, there is no way to visit all nodes exactly once by following the "expected way", as some clause will necessarily be left out.

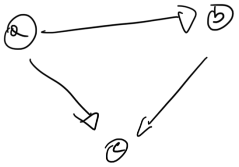
You can easily verify the reduction performs in polynomial time. □

The above was the more difficult result to prove. Note however, that TSP, the original decision problem we are interested in is defined on *undirected* graphs, but DIRECTED-HAMCYCLE is on directed graphs. To get closer to proving that TSP is **NP**−complete, we need another proof about the following language:

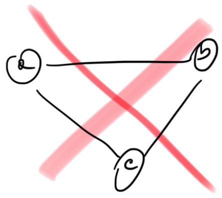
$$\text{UNDIRECTED-HAMCYLE} = \{G \mid G \text{ is a (not weighted) } \textit{undirected} \text{ graph, and } G \text{ has an hamiltonian cycle}\}.$$

**Theorem 44.** *UNDIRECTED-HAMCYCLE is* **NP**−complete

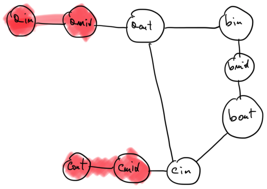
*Proof.* Membership in **NP** is as usual. To prove UNDIRECTED-HAMCYCLE is **NP**−hard we reduce DIRECTED-HAMCYCLE to UNDIRECTED-HAMCYLE. In particular, we simply need to show how to convert a directed graph $G$ to an undirected graph $G'$, such that $G$ has an Hamiltonian cycle iff $G'$ has one. Let us first discuss a simple reduction idea that *does not work.* Consider the simple directed graph:



The above graph has no Hamiltonian cycle (actually, has no cycles at all). We definitely cannot simply make all edges undirected:



because we would introduce cycles that do not exist in the original graph (these cycles traverse edges without following all edges in the same direction). The trick is instead to "split" each node $v$ of the original graph in 3 nodes called $v^{(in)}$, $v^{(mid)}$, and $v^{(out)}$. Each node $v$ is then represented by its 3 nodes, connected in a "line". For example, for node $a$, we have $a^{(in)} - a^{(mid)} - a^{(out)}$. If $v$ has an edge that goes *from* $v$ to some other node $u$, then we use an undirected edge between the "out" node of $v$ and the "in" node of $u$. With the graph of the example above, we obtain the following undirected graph:



Now, even if there is a cycle in the new graph, this cycle cannot traverse all nodes (the red ones are excluded), because once the cycle follows an edge in a certain direction (i.e., following the arrow, or following it in reverse), it now must be coherent and follow the next edges in the same direction.

Let us now prove formally the above claim. Let $G'$ be the graph obtained from $G$, after we split each node $v$ to $v^{(in)}$, $v^{(mid)}$, and $v^{(out)}$, and connect the edges as discussed.

Assume $v_1, v_2, \ldots, v_n, v_1$ is an Hamiltonian cycle in the (directed graph) $G$. Then, surely

$$v_1^{(in)}, v_1^{(mid)}, v_1^{(out)}, v_2^{(in)}, v_2^{(mid)}, v_2^{(out)}, \ldots, v_n^{(in)}, v_n^{(mid)}, v_n^{(out)}, v_1^{(in)}$$

is a cycle in $G'$. Moreover, since $v_1, \ldots, v_n$ are *all the nodes* in $G$, the nodes in the above cycle are *all* the nodes in $G'$, by construction of $G'$.

Assume now that $G'$ has an Hamiltonian cycle, and let $C$ be such a cycle.

**Observation 1.** A node $v^{(mid)}$ in $G'$ has only two neighbours, i.e., $v^{(in)}$ and $v^{(out)}$. Thus, since $C$ must visit *all* nodes of $G'$ exactly once, for each node $v$ of $G$, if $C$ wants to visit $v^{(mid)}$, and does not want to get "stuck", it must visit $v^{(in)}, v^{(mid)}, v^{(out)}$ one after the other, or $v^{(out)}, v^{(mid)}, v^{(in)}$, one after the other.

**Observation 2.** Note that two "out" nodes, or two "in" nodes are never connected together in $G'$, but only "in" nodes with "out" nodes can be connected.

By combining the two observations above, we conclude that $C$ must be of one of the follwing forms:

$$v_1^{(in)}, v_1^{(mid)}, v_1^{(out)}, v_2^{(in)}, v_2^{(mid)}, v_2^{(out)}, \ldots$$

$$v_1^{(out)}, v_1^{(mid)}, v_1^{(in)}, v_2^{(out)}, v_2^{(mid)}, v_2^{(in)}, \ldots,$$

The above two patterns correspond to traversing a cycle in $G$ by either following *all* the arrows of the cycle in the right direction, or by following *all* the arrows of the cycle in reverse direction, respectively. In any case, $C$ never "mixes" normal and inverse directions of the arrows in it. Hence, in both cases, $C$ corresponds to an Hamiltonian cycle of $G$.

Note that, if we did not use the middle nodes, $C$ would be allowed to be, for example, of the form $v_1^{(in)}, v_1^{(out)}, v_2^{(in)}, v_3^{(out)}$. That is, $C$ enters $v_2^{(in)}$, from $v_1^{(out)}$, and thus following the directed edge $(v_1, v_2)$, but then it "skips" $v_2^{(out)}$ by going to $v_3^{(out)}$, and thus following the directed edge $(v_3, v_2)$ in opposite direction w.r.t $(v_1, v_2)$. With the middle nodes, we are guaranteed that once $C$ enters $v^{(in)}$, it must necessarily exit from $v^{(out)}$, and vice versa, because $C$ must visit $v^{(mid)}$, and that is the only chance to do it, if $C$ does not want to get "stuck". □

We are finally ready to prove that TSP is **NP**−complete. Indeed, this is almost straightforward, since UNDIRECTED-HAMCYCLE is more or less a special case of TSP, where each edge has weight 1.

**Theorem 45.** *TSP is* **NP**−complete

*Proof.* We have already shown TSP $\in$ **NP**. The reduction from UNDIRECTED-HAMCYCLE is straightforward. From an undirected graph $G = (V, E)$, we construct a pair $(G', k)$, where $G'$ is the weighted undirected graph obtained from $G$, where each edge has weight 1, and the cost $k = |V|$. Clearly, if $G$ has an Hamiltonian cycle (which necessarily visits $|V|$ edges), then $G'$ has an Hamiltonian cycle (actually the same), of cost at most (actually equal to) $k = |V|$. If $G$ has no Hamiltonian cycle, clearly, $G'$ has no Hamiltonian cycle of any cost, let alone $k$. □

Hence, from what we discussed at the beginning of this lecture, the search version of TSP, i.e., computing the minimum cost of an Hamiltonian cycle on a weighted undirected graph cannot be computed in polynomial time, unless **P** = **NP**, because TSP is **NP**−complete.

# 22 Exercises on TMs and undecidability

Consider the following language

$$\mathcal{L} = \{A\#B\#W_1W_1^R \cdots W_nW_n^R \mid A, B, W_1, \ldots, W_n \in \{a,b,c,d\}^+ \text{ and}$$
$$n = |A| - |B| \text{ and } |W_i| \geq |B| \text{ for each } i = 1, \ldots, n\}.$$

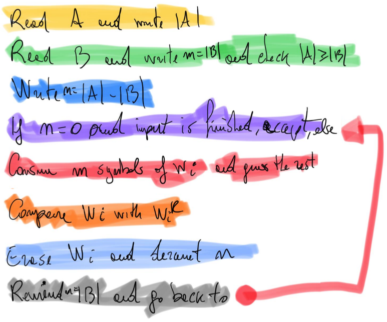
For example, the string

$$\underbrace{aabcda}_{A}\#\underbrace{bda}_{B}\#\underbrace{bba}_{W_1}\underbrace{abb}_{W_1^R}\underbrace{cdab}_{W_2}\underbrace{badc}_{W_2^R}\underbrace{aabdc}_{W_3}\underbrace{cdbaa}_{W_3^R}$$

is in $\mathcal{L}$, because we have $n = |A| - |B| = 6 - 3 = 3$ strings $W_i$, each string $W_i$ is of length at least $|B| = 3$, and each $W_i$ is followed by its reverse.

Devise a TM, possibly non-deterministic and with multiple tapes, that decides the above language $\mathcal{L}$.

The main idea of the TM is described below. The full control of the TM is presented in the next page, instead. In the pictures, $\alpha$ is a symbol in $\{a,b,c,d\}$.



Read A and write |A|

Read B and write m=|B| and check |A|≥|B|

Write m=|A|-|B|

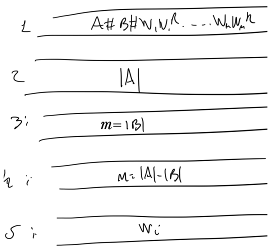
If m=0 and input is finished, accept, else

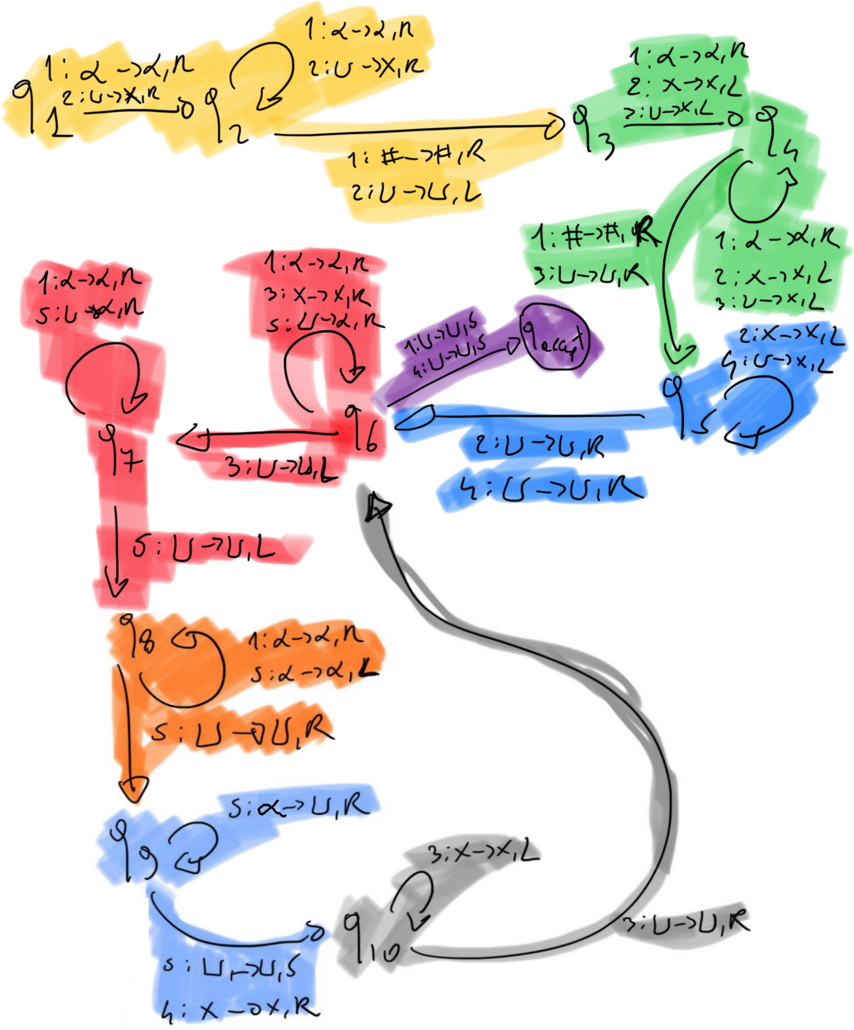
Consume m symbols of $W_i$ and guess the rest

Compare $W_i$ with $W_i^R$

Erase $W_i$ and decrement n

Rewind m=|B| and go back to





## 22.1 Undecidability

Consider the following properties of TMs:

$$\begin{aligned}
\mathcal{P}_1 &= \{< M >\mid M \text{ changes state at least once when executed with } \epsilon\}.\\
\mathcal{P}_2 &= \{< M >\mid M \text{ never remains on the same state for two consecutive steps}\}.\\
\mathcal{P}_3 &= \{< M >\mid M \text{ accepts all inputs}\}.
\end{aligned}$$

Study the decidability of the above properties. Use Rice's Theorem whenever possible. Recall that $< M >$ denotes the encoding of a TM $M$ that only uses the tape alphabet $\Gamma = \{0, 1, \sqcup\}$.

$\mathcal{P}_1$. Is $\mathcal{P}_1$ trivial? That is, is it empty or the set of all TMs? No, because the TM $M_1$ that moves directed from $q_1$ to $q_{\texttt{accept}}$, whatever is the read symbol is in $\mathcal{P}_1$, but $M_2$ that stays in $q_1$ when reads $\sqcup$ and goes to $q_{\text{accept}}$ otherwise, is not in $\mathcal{P}_1$. So, there are some TMs in $\mathcal{P}_1$, and some are not in $\mathcal{P}_1$.

Is $\mathcal{P}_1$ semantic? That is, is it the case that for every two TMs $M_1, M_2$, with $L(M_1) = L(M_2)$, when $< M_1 > \in \mathcal{P}_1$, then also $< M_2 > \in \mathcal{P}_1$? Consider the TM $M_2$ we discussed for proving $\mathcal{P}_1$ is not trivial. It never changes state when the input is $\epsilon$, and its language is $L(M_2) = \{0,1\}^* \setminus \{\epsilon\}$. Then, consider the TM $M'$ that, from $q_1$ goes to $q_{\texttt{accept}}$, when reading any symbol different from $\sqcup$, and when reading $\sqcup$, it first moves from $q_1$ to $q_1'$, without moving the head. Then, from $q_1'$ moves to $q_{\texttt{accept}}$, by reading any symbol different from $\sqcup$, otherwise it stays in $q_1'$, without moving the head. $M'$ accepts the same language as $M_2$, but $M_2$ is in $\mathcal{P}_1$, and $M'$ is not. Thus, we cannot use Rice's Theorem to prove undecidability of $\mathcal{P}_1$.

For sure, $\mathcal{P}_1$ is in RE, because, given a TM encoding $< M >$, we can simulate execution of $M$ with input $\epsilon$, and if it changes state, we accept. If $< M >$ changes some state with input $\epsilon$, the procedure we described accepts. If $M$ does not change state (i.e., it loops), our procedure does not accept.

But we can do better. Actually, $\mathcal{P}_1$ is in R, i.e., it is decidable. Let $M$ be a TM given as input to our procedure. Observe that $M$ changes state at least once iff it moves from the initial state $q_1$ to some other state. The rest of $M$'s code can be even ignored. Thus, we only need to focus on the part of the transition $\delta$ of $M$ that takes $q_1$ as input. Since we encode TMs $M$ with alphabet $\Gamma = \{0,1,\sqcup\}$, we have only three possible useful entries in $\delta$:

$$\begin{gathered}\delta(q_1, 0),\\ \delta(q_1, 1),\\ \delta(q_1, \sqcup).\end{gathered}$$

Moreover, each $\delta(q_1, \alpha)$ returns a triple of the form $(q', \beta, \star)$, with $q' \in Q$, $\beta \in \{0,1,\sqcup\}$ and $\star \in \{0,1,\sqcup\}$. So, what really determines whether $M$ changes state is the above part of $\delta$, plus of course the input string. But, the input string is fixed, it is always $\epsilon$, and in fact it is not part of our input (only $M$ is). The only part that actually depends on the input, which is the TM $M$, is the state $q'$, the values all the other elements like $\beta$, $\star$, $\alpha$, and even $q_1$ can take are independent of the specific TM we are given as input.

So, each of the three entries above can output one of $|Q| \cdot |\{0,1,\sqcup\}| \cdot |\{L,R,S\}| = 9 \cdot |Q|$ possible triples. Note, however, that the actual value of the state $q'$ is irrelevant to understand if the TM moves from $q_1$ to some state *different from* $q_1$, because once it does, it is irrelevant to which exact state $q'$ the machine moved. So, actually, for an entry $\delta(q_1, \alpha)$, what is only important to know about its output is *1)* whether the new state $q'$ differs or not from $q_1$, *2)* which symbol $\beta$ is written, and *3)* the head direction $\star$. Hence, we can very well replace $q'$ with a simple bit that tells us if the state is different from $q_1$ or not, and nothing would change. Hence, each entry $\delta(q_1, \alpha)$ can output one among

$$|\{0,1\}| \cdot |\{0,1,\sqcup\}| \cdot |\{L,R,S\}| = 18$$

possible triples. So, even if we consider infinitely many TMs $M$ as input, their three entries of the form $\delta(q_1, \alpha)$ must necessarily fall in one of the above 18 combinations.

Thus, for the purpose of deciding $\mathcal{P}_1$, a TM, no matter how large and complex, can be seen as a string that specifies how each of the three entries $\delta(q_1, \alpha)$ of its transition function looks like, according to the kind of information we discussed above. For example, we can use strings of the following form to represent a TM:

$$(q_1, 0) \mapsto (b_1, \beta_1, \star_1), (q_1, 1) \mapsto (b_2, \beta_2, \star_2), (q_1, \sqcup) \mapsto (b_3, \beta_3, \star_3).$$

If, for example, $b_1 = 1, \beta_1 = 0, \star_1 = L$, the above string says that the corresponding TM, when in state $q_1$ and when reads the symbol 0, it moves to a state different than $q_1$ (because $b_1 = 1$), writes the symbol 0, and moves the head to the left.

How many strings of the above form can we have? Since there are 18 possible triples $(b_i, \beta_i, \star_i)$, for each entry $(q_1, 0), (q_1, 1), (q_1, \sqcup)$, there are $18 \cdot 18 \cdot 18 = 18^3 = 5832$ possible strings.

Some of these 5832 strings represent TMs that change their state, and others represent TMs that do not change their state. We could now start checking each of them, to understand which correspond to TMs changing state, but we can simplify our life as follows. Say that $\mathcal{L}'$ is the language made of only the strings of the above form that represent the cases where the corresponding TM $M$ changes state. The language $\mathcal{L}'$ is finite, and thus decidable. Hence, a TM that decides $\mathcal{P}_1$ is the one that constructs the right string for the input TM, and then executes the TM that decides $\mathcal{L}'$.

**Remark.** Note that the above discussion is subtle, because even if *we* don't necessarily know which cases make the TM change state, and which not, since there are finitely many, *there exists* a TM (which we don't even known how it looks like), that has the right correspondence hard-coded in its transition function (this is possible, because there are only a constant number (5832) of cases to consider). This discussion highlights the fact that our Theory of Computation is not constructive. We do not require to be able to actually *construct* an algorithm, to show a language is in a certain class. We only require to prove that it *exists*.

**$\mathcal{P}_2$.** Is $\mathcal{P}_2$ trivial? No, because the TM that moves immediately from $q_1$ to $q_{\texttt{reject}}$, whatever is the input string is in $\mathcal{P}_2$, but the TM $M_2$ that loops in $q_1$, for every input symbol, without moving the tape head, is not in $\mathcal{P}_2$. Is $\mathcal{P}_2$ semantic? No, because the above TM $M_1$ does not accept any string, i.e., it accepts the empty language, and also $M_2$ does not accept any string (because it loops on $q_1$, for every input symbol, without moving the head). Thus, even if they accept the same language, one is in $\mathcal{P}_2$ and the other is not.

We cannot apply Rice's Theorem. We study the complement of $\mathcal{P}_2$. That is

$$\bar{\mathcal{P}}_2 = \{< M >| \text{ there is some input string } w \text{ such that when } M \\ \text{runs on } w, \text{ it remains on some state for two or more consecutive steps}\}.$$

We show $\bar{\mathcal{P}}_2$ is in RE with the following procedure. Given $< M >$, guess a string $w$, and simulate $M$ with input $w$. If, during the simulation, there is a state in which $M$ stays for two or more consecutive steps, then accept. The idea is that, if a string $w$ with this property exists, then the procedure will find it, and halt once $M$ stays in the same state for two or more consecutive steps. However, if such a string does not exist, our procedure does not necessarily reject, but might loop, "waiting" for $M$ to stay in the same state.

Can we do better? No, we prove $\bar{\mathcal{P}}_2$ is undecidable. The observation is that, given a TM $M$, if its transition function $\delta$ has some loop, i.e., $\delta(q, \alpha) = (q, \beta, \star)$, we can translate that loop in a two states loop, where we move from $q$ to a new state $q'$, i.e., $\delta(q, \alpha) = (q', \beta, \star)$, and then go back with the same transition, i.e., $\delta(q', \alpha) = (q, \beta, \star)$. Of course, every other transition that was exiting $q$ now must also exist $q'$, and every transition entering $q$, must also enter $q'$.

In this way, the new machine, call it $M^2$, will never stay in the same state for more than one step, because it has no self-loops. With this in mind, we provide a simple reduction from the Universal language $\mathcal{L}_u$. We must convert a pair $(M, w)$ to a TM $M'$ such that, $M$ accepts $w$ iff there is a string $w'$ for which $M'$ remains on some state for two or more consecutive steps.

The reduction is simple. It constructs a TM $M'$ that ignores its input, and instead executes the "two-state" version of $M$, i.e., $M^2$ over the string $w$, with the addition that the accepting state is replaced with a new state $q'$ on which $M^2$ loops.

So, if $M$ accepts $w$, $M'$ executes $M^2$ which until the very end, never stays in the same state for more than one step, but when it reaches $q'$, it stays in the same state forever, and thus for two or more consecutive steps.

If $M$ does not accept $w$, the state $q'$ of $M^2$ is never reached, and thus, since $M^2$ has been modified as we discussed before, it never stays in the same state for more than one step. Hence $\bar{\mathcal{P}}_2 \in \text{RE} \setminus \text{R}$. We conclude that $\mathcal{P}_2 \notin \text{RE}$, because if it was, since also its complement is in RE, it would imply that both $\mathcal{P}_2$ and its complement are in R, which is not possible, as we have just shown $\bar{\mathcal{P}}_2$ is not in R.

**$\mathcal{P}_3$.** Is $\mathcal{P}_3$ trivial? No, because we have a TM $M_1$ that always accepts, but also some TM $M_2$ that accepts not all strings, like the one accepting $\mathcal{L}_{01}$. Is $\mathcal{P}_3$ semantic? Consider two arbitrary TMs $M_1, M_2$ such that $L(M_1) = L(M_2)$, and assume $< M_1 >\in \mathcal{P}_3$. Since $< M_1 >\in \mathcal{P}_3$, $L(M_1)$ is the set of all strings, which means also $L(M_2)$ is the set of all strings, and thus also $< M_2 >\in \mathcal{P}_3$. Thus, $\mathcal{P}_3$ is semantic. Hence, since $\mathcal{P}_3$ is non-trivial and semantic, by Rice's Theorem, $\mathcal{P}_3$ is undecidable.

Let us consider one last property of TMs.

$$\mathcal{P} = \{< M >| \; M \text{ halts after an even number of steps, when run on } \epsilon\}.$$

$\mathcal{P}$ is clearly in RE, because we can just simulate $M$ with input $\epsilon$ and count the number of steps. When it ends, we check. Clearly, this procedure only accepts TMs $M$ that are in $\mathcal{P}$, but if $M$ is not in $\mathcal{P}$, and thus it might not halt at all with input $\epsilon$, our procedure will loop as well, rather than rejecting.

Let us see if $\mathcal{P}$ is undecidable. Is $\mathcal{P}$ trivial? No, because we have a TM $M_1$ that in one step accepts, regardless of the input, and thus halts in one step also with $\epsilon$, and we have $M_2$ that accepts in two steps, regardless of the input. So, $M_1$ is not in $\mathcal{P}$, and $M_2$ is in $\mathcal{P}$. Is $\mathcal{P}$ semantic? No, because the above two TMs accept the same language (i.e., all strings), but one is in $\mathcal{P}$ and the other is not.

We show $\mathcal{P}$ is not in R, by reducing HALT-$\epsilon$ to it. Recall that HALT-$\epsilon$ is the language of all TMs encodings $< M >$ such that $M$ halts with input $\epsilon$. We must construct, given a TM $M$, a TM $M'$ such that $M$ halts with $\epsilon$ iff $M'$ halts with $\epsilon$ in an even number steps. The trick is to modify $M$ to perform one additional step, whenever it performs one. That is, if the transition function $\delta$ of $M$ has a transition $\delta(q, \alpha) = (q', \beta, \star)$, we change it with two transitions. The first one does nothing, and simply goes to an auxiliary state $q^*$, i.e., $\delta(q, \alpha) = (q^*, \alpha, S)$. The second does what the original transition was doing, i.e., $\delta(q^*, \alpha) = (q', \beta, \star)$. Thus, we doubled the number of steps of $M$, for whatever input, and thus also $\epsilon$. If $M$ halts with input $\epsilon$, then the new machine $M'$ also halts, but with double the steps, and thus $M'$ is in $\mathcal{P}$.

If $M$ does not halt with input $\epsilon$, $M'$ does not halt with input $\mathcal{P}$, let alone, within an even number of steps. So, $\mathcal{P} \in \text{RE} \not\in \text{R}$.

# 23 Exercises on NP−completeness

We prove more **NP**−complete languages, that will be both useful to have in your "database" of **NP**−complete problems (the more, the easier should be to find the right language to use when proving another language is **NP**−complete), but will also serve as further practice.

## 23.1 SUBSETSUM

We consider first the problem SUBSETSUM. You are given a list of (not necessarily distinct) natural numbers $c_1, \ldots, c_n$, and an additional natural number $k \in \mathbb{N}$. The question is whether there is a subset $I \subseteq \{1, \ldots, n\}$ of indices, such that $\sum_{i \in I} c_i = k$, i.e., you can select certain numbers from the list once, in such a way that their sum equals $k$. We thus, formally define the corresponding language as:

$$\begin{aligned}\text{SUBSETSUM} = \{(c_1, \ldots, c_n, k) \mid {} & c_1, \ldots, c_n, k \in \mathbb{N} \text{ and} \\ & \text{there is } I \subseteq \{1, \ldots, n\} \text{ s.t. } \sum_{i \in I} c_i = k\}.\end{aligned}$$

**Theorem 46.** *SUBSETSUM is* **NP**−complete.

*Proof.* Showing membership in **NP** is easy. Guess a set $I \subseteq \{1, \ldots, n\}$ by simply iterating over each number $c_i$ in the list, and guessing if $i$ should belong to $I$. Then, sum all the numbers with index in $I$ and verify the sum equals $k$. The set $I$ is of size at most linear in the input, and summation is clearly a polynomial-time operation.

We prove **NP**−hardness by reducing EXACT-3-SAT to SUBSETSUM. Consider a Boolean formula in CNF of the form:

$$\varphi = C_1 \wedge \cdots \wedge C_m,$$

where each clause $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$ contains three literals.

Let $x_1, \ldots, x_n$ be the Boolean variables of $\varphi$. We construct the list of numbers as follows. For each Boolean variable $x_i$, two numbers, that we call $t_i$ and $t_f$, are in the list. The idea is to represent choosing "true" for $x_i$ with choosing the number $t_i$, and "false" with choosing $f_i$.

The number $t_i$ is made of $n + m$ digits in base 10, i.e., it is of the form

$$a_1 \cdots a_n p_1 \cdots p_m,$$

where each symbol is a digit in $\{0, 9\}$. We set digit $a_i = 1$, and the other $a_j = 0$. Moreover, we set $p_j = 1$ if the clause $C_j$ contains $x_i$ without negation, otherwise $p_j = 0$. The number $f_i$ is defined in a similar way, it is of the form:

$$f_i = a_1 \cdots a_n p_1 \cdots p_m.$$

The symbols $a_1, \ldots, a_n$ are defined in the same way as for $t_i$. Instead, we have that $p_j = 1$ if $x_i$ appears negated in clause $C_j$, otherwise $p_j = 0$.

For example, if

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3),$$

the set list is made of the following numbers, with 3+4=7 digits:

| | $a_1$ | $a_2$ | $a_3$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|
| $t_1 =$ | 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 |

Let $\tau$ be some truth assignment for the variables $x_1, \ldots, x_n$, if $\tau(x_i)$ = true, we choose the number $t_i$, otherwise we choose $f_i$. Note that choosing the numbers in the list in this way, the sum of these numbers corresponds to a number of the form

$$\underbrace{11\cdots 1}_{n} s_1 \cdots s_m.$$

That is, the first $n$ digits are all 1, moreover $s_j$ denotes how many literals make the clause $C_j$ true, because of this assignment. For example, the assignment $x_1$ = false, $x_2$ = false, $x_3$ = true corresponds to choosing $f_1, f_2, t_3$. Thus

$$f_1 + f_2 + t_3 = 1112203.$$

The above means the assignment makes 2 literals of $C_1$ true, 2 literals of $C_2$ true, 0 literals of $C_3$ true, and 3 literals of $C_4$, true. Clearly, as far as the first $n$ digits are all 1's and the last $m$ digits are all non-zero, the choice represents an assignment satisfying the formula.

The problem is that we need this sum to be a precise number, and not any number of the form described above. Since each clause has 3 literals, the last $m$ digits $s_1, \ldots, s_m$ are no greater than 3. So, it is enough to add to the list some "dummy" numbers, that we can select to "help" the last $m$ digits become all 3 (without of course making them equal to 3 trivially), and at the same time, should not affect the first $n$ digits of the final sum.

For this, we add, for each clause $C_j$, two copies $u_j, v_j$ of the same number, which has all digits equal to 0, except for the $n + j$-th digit, which is 1. So, our final list, for the example formula is:

| | $a_1$ | $a_2$ | $a_3$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|
| $t_1 =$ | 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 |
| | | | | | | | |
| $u_1 =$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $v_1 =$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $u_2 =$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $v_2 =$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $u_3 =$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| $v_3 =$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| $u_4 =$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| $v_4 =$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Note that the new numbers alone cannot make any of the last $m$ digits equal to 3. They still need the "help" of some other number corresponding to choosing a value for a variable that makes the corresponding clause true. Moreover, to have all first digits to 1, we still need to choose one of $t_i$ or $f_i$ for each variable. So, the number $k$ we need to construct is (in base 10):

$$k = \underbrace{11\cdots 1}_{n}\underbrace{33\cdots 3}_{m}.$$

□

## 23.2 KNAPSACK

A famous generalization of SUBSETSUM, is the so-called KNAPSACK problem. Here you are given $n$ items, in the form of two lists of $n$ natural numbers: $w_1, \dots, w_n \in \mathbb{N}$ are the *weights* that each item has, and $v_1, \dots, v_n \in \mathbb{N}$ are the *values* of each item (i.e., how much they are worth, or important, etc.). Then, you are given a maximum weight $W \in \mathbb{N}$, and a minimum value $V \in \mathbb{N}$, and the question is, can you choose a subset of the items, i.e., a set $I \subseteq \{1, \dots, n\}$ such that

$$\sum_{i \in I} w_i \leq W \qquad \sum_{i \in I} v_i \geq V.$$

That is, we don't exceed the maximum weight and we also have an overall value at least as large as the minimum.

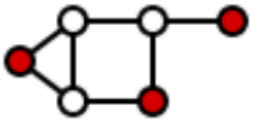
$$\begin{aligned}\text{KNAPSACK} = \{&(w_1, \dots, w_n, v_1, \dots, v_n, W, V) \mid \\ &w_1, v_1, \dots, w_n, v_n, W, V \in \mathbb{N}, \text{ and there is } I \subseteq \{1, \dots, n\} \text{ s.t.} \\ &\sum_{i \in I} w_i \leq W \text{ and } \sum_{i \in I} v_i \geq V\}.\end{aligned}$$

**Theorem 47.** *KNAPSACK is* **NP**−complete

*Proof.* Membership in **NP** is straightforward. Guess the indices in $I$, and verify. For the hardness, it is enough to observe that SUBSETSUM is a special case of KNAPSACK. In fact, an instance of SUBSETSUM $(c_1, \ldots, c_n, k)$ can be seen as an instance of KNAPSACK where each weight $w_i$ and value $v_i$ coincide with $c_i$, and $W = V = k$. □

## 23.3 DOMINATINGSET

We now move to a language over graphs. It is very similar to vertex cover. Consider an undirected graph $G = (V, E)$. A *dominating set* of $G$ is a set of nodes $D \subseteq V$ such that every *node* is "touched" by some node in $D$, i.e., for each $v \in V \setminus D$, there is a node $u \in D$ such that $\{u, v\} \in E$. The difference with vertex cover is that a vertex cover must touch every *edge* of the graph, while a dominating set touches every node of the graph. The following example shows that the two notions do not necessarily coincide:



The red nodes are a dominating set, but they are not a vertex cover, because the first vertical edge, from the left, is not touched by the red nodes. Note that every graph $G$ has a dominating set, i.e., the complete set of nodes $V$. So, what is interesting to ask is if there is a dominating set of size *at most* some $k$.

$$\text{DOMINATINGSET} = \{(G, k) \mid G \text{ is an undirected graph, and } G \text{ has a dominating set } D \text{ such that } |D| \leq k\}.$$

**Theorem 48.** *DOMINATINGSET is* **NP**−complete

*Proof.* We can decide DOMINATINGSET by guessing a set $D \subseteq V$, and then verify that every node in $v \in V \setminus D$ is connected by an edge with a node in $D$. $|D|$ is of size at most linear, and the second check requires $|V|$ iterations, and in each iteration we need to try each node $u$ in $D$ and verify there is an edge between $u$ and the current node. So, we need $O(|V| \cdot |D| \cdot |E|) = O(|V|^2 \cdot |E|)$ steps, hence polynomial.

We prove **NP**−hardness via a reduction from VCOVER. Indeed, there are quite some similarities between a vertex cover and a dominating set.

We must convert a pair $(G, k)$ to a pair $(G', k')$ such that $G$ has a vertex cover of size at most $k$ iff $G'$ has a dominating set of size at most $k'$.

The reduction first sets $k' := k$. Note that if a graph $G$ has no isolated nodes, than, every vertex cover $VC$ of $G$ is also a dominating set of $G$. This is because $VC$ touches all edges, and since there are no isolated nodes, every node is connected to some edge, that $VC$ necessarily "touches", and thus, every node outside $VC$ is connected to some node in $VC$. So, our reduction first removes all isolated vertices in $G$.

However, as we have seen in the example above, even in graphs without isolated nodes, the other direction might not be true, i.e., we can have a dominating set that is not a vertex cover. So, we need to force, in our graph $G'$ that a dominating set contains at least one node for each edge. To do this, we further modify $G$, by converting each edge $e = \{u, v\} \in E$ in a triangle, i.e., we connect $u$ and $v$ to a new special node $v_e$, not connected to anything else. We so obtain $G'$. In this way, if a dominating set "wants" to dominate the new nodes, it must choose at least one of the nodes in an original edge of the graph $G$.

So, if $G$ has a vertex cover $VC$ of size at most $k$, then when removing from $VC$ all isolated nodes, we obtain a set $VC'$ of size necessarily at most $k$, and $VC'$ must be a vertex cover of for the subgraph of $G$ that has no isolated nodes. From what we discussed, we conclude that $VC'$ is also a dominating set for such a subgraph. Thus, in the final graph $G'$, since the new nodes $v_e$ are connected to some node of this subgraph, we conclude that $VC'$ is also a dominating set for $G'$.

Assume $G'$ has as dominating set $D$ such that $|D| \leq k'$. Note that each special node $v_e$ is connected to both nodes of the edge $e = \{u, v\}$, and thus dominates $u$ and $v$. However, if $D$ contain $v_e$ and we replace it with $u$ in the set $D$, $D$ now dominate $v$ and $v_e$. So, if the dominating set $D$ contains a special node $v_e$, we can safely remove it from $D$, and add $u$ or $v$ to $D$ instead (if it is not in $D$ already), and still obtain a dominating set $D'$ (i.e., no nodes are left out). The size of $D'$ can only decrease, with this operation, and thus it is still of size at most $k'$. Now $D'$ contains only nodes of the original graph $G$. Since $D'$ is dominating, and thus must be connected to all nodes not in $D'$, among the nodes not in $D'$ we have *all* the special nodes $v_e$. The only way for $D'$ to be connected to such nodes, is to contain at least one node for each edge of the original graph $G$, and thus it is a vertex cover of $G$ of size at most $k' = k$. □

# 24 More exercises on NP−completeness and Search problems

**RESTAURANT.** Consider the following problem, called RESTAURANT. You are given a set of $n$ people $P = \{1, \dots, n\}$, a set of $k$ tables $T = \{1, \dots, k\}$, and for each person $i \in P$, you are given a set $L_i$ of people that $i$ does not like. Your job, as the owner of the restaurant, is to assign to each person a table, without placing two people at the same table, where one dislikes the other.

Provide a formal definition of RESTAURANT in terms of languages, and prove RESTAURANT is **NP**−complete (hint: VCOLORING is **NP**−complete).

So, a valid input is a tuple $(P, T, L_1, \dots, L_n)$, where $P$ is a set of $n$ integers, $T$ a set of $m$ integers, and $L_i$ is a set of integers of size at most $n$. We define a *table assignment* as a function $\tau$ from people $P$ to tables $T$ such that, for every two people $i, j \in P$ such that $i \in L_j$ or $j \in L_i$, $\tau(i) \neq \tau(j)$

$$\begin{aligned}\text{RESTAURANT} = \{(P, T, L_1, \dots, L_n) \mid (P, T, L_1, \dots, L_n) \text{ is a valid input} \\ \text{which has a table assignment}\}.\end{aligned}$$

People in the restaurant can be seen as nodes of a graph, that are connected, whenever one does not like the other. A table is a color that we do not want to assign to two people that do not like each other.

**Theorem 49.** *RESTAURANT is* **NP**−complete.

*Proof.* To show that RESTAURANT is in **NP** we provide the following non-deterministic procedure. Given $(P, T, L_1, \dots, L_n)$, guess for each person $i \in P$, a table $t_i \in T$. Then verify that for every person $i \in P$, and every person $j \in L_j$, $t_i \neq t_j$. The algorithm needs to guess $|P|$ tables, one per person. Then, verifying the latter check, requires trying for every person $i$ in $P$ at most $|P|$ people in $L_i$, so $O(|P|^2)$. Hence, the procedure is polynomial. The procedure is correct, because if there is a table assignment, the procedure will guess that assignment, and the final check will succeed. If there is not table assignment, whatever tables the procedure guesses for each person, will not pass the final check, and thus all computation paths are rejecting.

As discussed above, there is quite a similarity between VCOLORING and RESTAURANT. Recall that VCOLORING is the language of pairs $(G, k)$, where $G$ is an undirected graph, and $k$ a number of available colors, such that there is a $k$-coloring for the nodes of $G$, i.e., there is a function $\mu : V \to \{1, \dots, k\}$ that maps nodes to colors such that, for every two nodes $u, v \in V$, if $\{u, v\} \in E$, then $\mu(v) \neq \mu(u)$.

We construct, from $(G, k)$, with $G = (V, E)$, a string $(P, T, L_1, \dots, L_n)$ such that $G$ has a $k$-coloring iff there is a table assignment for $(P, T, L_1, \dots, L_n)$. As discussed already, we let $P = V$, and for each $i \in P$, we let $L_i$ be the set of nodes of $G$ to which $i$ is connected. Finally, $T = \{1, \dots, k'\}$, where $k' = k$ is the number of colors.

Note that the above procedure is *not* polynomial. The reason is because it writes the set $T = \{1, \dots, k'\}$, which contains $k' = k$ elements. So, the procedure requires time linear in $k$. However, $k$ is an integer encoded in binary. So, if it requires $m$ bits, in the worst case, $k = 2^m - 1$. Thus, the procedure requires time that is $O(2^m)$, and thus exponential in the *size of the encoding of the input.* To avoid this problem, we modify our procedure as follows:

1. If $k \geq |V|$, then set $k' := |V|$.

2. else $k' := k$.

3. Construct the string $(P, T, L_1, \dots, L_n)$ as discussed before.

What the reduction does is to first check if the number of colours is more or the same as $|V|$. If this is the case, note that $G$ trivially has a $k$-coloring, i.e., assign a different colour to each node. In particular, we still have a coloring, even if we reduce the number of colours to exactly $|V|$. Thus, if the input $(G, k)$ has more colours than needed, also the tables would be more than needed. Thus, we simply do not use all of them, but only at most $|V|$. Hence, in the second phase, the reduction constructs a set $T$ which will always contain at most $|V|$ tables, and thus will always require polynomial time.

Assume $G$ has a $k$-coloring $\mu$, and in particular, assume $k \leq |V|$.[25] Then, consider the function $\tau = \mu$. Since $\mu$ assigns a colour from $\{1, \dots, k\}$ to each node, $\tau$ assigns a table from $T$ to each person, by construction of $P$ and $T$. Then, since for every two nodes $i, j \in V$, $\mu(i) \neq \mu(j)$, whenever $\{i, j\} \in E$, and since there is an edge $\{i, j\} \in E$ iff $i \in L_j$ and $j \in L_i$, for every two people $i, j \in P$, $\tau(i) \neq \tau(j)$, whenever $i \in L_j$, or $j \in L_i$. Thus, $\tau$ is a table assignment.

Assume $\tau$ is a table assignment for $(P, T, L_1, \dots, L_n)$. Note that for every two pepole $i, j \in P$, if $i \in L_j$ or $j \in L_i$, then $\{i, j\} \in E$. Thus, since for every two people $i, j \in P$, $\tau(i) \neq \tau(j)$,when $i \in L_j$ or $j \in L_i$, it implies that $\mu = \tau$ is also a $k'$-coloring for $G$, where $k' = k$. □

**LABELS.** Consider the following problem, called LABELS. You are given as input an undirected graph $G = (V, E)$, two integers $k, k'$, and a set $X$ of integers, called *labels*. The question is whether we can assign a label from $X$ to each node $v \in V$, call it $p_v$, such that the sum of the labels of all nodes is strictly smaller than $k$, i.e., $\sum_{v \in V} p_v < k$, and the sum of the *cost* of each edge $e \in E$ of $G$, call it $C_e$, is at least $k'$, i.e., $\sum_{e \in E} C_e \geq k'$. The cost of an edge $e = \{u, v\} \in E$ is simply the maximum label between $p_u$ and $p_v$, i.e., $C_e = \max\{p_u, p_v\}$.

Provide a formal definition of LABELS in terms of languages, and prove LABELS is **NP**−complete (hint: VCOVER is **NP**−complete).

We can define a valid input of LABELS as a string of the form $(G, k, k', X)$, where $G$ is an undirected graph, $k, k'$ are integers, and $X$ is a set of integers.

[25] This assumption is without loss of generality, since if $k \geq |V|$, we know that we also have another coloring using just $|V|$ colors.

We define a *labeling* of $(G, k, k', X)$, with $G = (V, E)$, as a function $\ell : V \to X$ such that $\sum_{v \in V} \ell(v) < k$ and $\sum_{\{u,v\} \in E} \max\{\ell(u), \ell(v)\} \geq k'$.

$$\text{LABELS} = \{(G, k, k', X) \mid (G, k, k', X) \text{ is a valid input that has a labeling}\}.$$

**Theorem 50.** *LABELS is* **NP**−complete*.*

*Proof.* given $(G, k, k', X)$, a non-deterministic procedure that decides if $(G, k, k', X) \in$ LABELS is the following. For each node $v \in V$, guess a value $p_v \in X$. Verify that the values sum up to a number strictly smaller than $k$. Then, for each edge $\{u, v\} \in E$, compute its cost and sum it up to a count. Verify it is at least $k'$. The total labels, one for each node, take up $|V| \cdot m$ space, where $m$ is the number of bits used to encode a number in $X$. Thus, they can be guessed in polynomial time. Then, the first summation requries summing $|V|$ numbers. The second summation requires summing up $|E|$ numbers, where each number can be computed by simply comparing the two edge's nodes. So, the overall procedure is polynomial. The procedure is correct, because if a labeling exists, the procedure will guess it, and properly verify it. If a labeling does not exist, whatever labels it guesses for the nodes, they will not pass the later checks, and thus it will reject on all its computation paths.

We now prove LABELS is **NP**−hard. There is a high similarity between LABELS and VCOVER. A vertex cover is a set of nodes that "covers" all edges of the graph, i.e., each edge has at least one end point part of the vertex cover. LABELS is a generalization of this idea. In fact, if $X = \{0, 1\}$, i.e., we can only label a node with only 0 or 1, assigning a label to a node, means choosing the node, or discarding it. Hence, the first summation $\sum_{v \in V} p_v$ is essentially counting how many nodes we have chosen, and the second summation $\sum_{\{u,v\} \in E} \max\{p_u, p_v\}$ is counting how many egdes are "touched" by the chosen nodes.

Thus, given a string $(G, k)$, we construct a string $(\bar{G}, \bar{k}, \bar{k}', X)$ such that $G$ has a vertex cover of size at most $k$ iff $(\bar{G}, \bar{k}, \bar{k}', X)$ has a labeling, as follows. $\bar{G} = G$, the set of labels is $X = \{0, 1\}$. Since the first sum must be strictly smaller than $\bar{k}$, and we said it counts how many nodes we selected, we let $\bar{k} = k + 1$. Finally, since the second summation counts the number of touched edges, we want to touch all of them, and thus $\bar{k}' = |E|$.

The reduction is clearly polynomial, since we are essentially copying $G$, and counting the number of nodes and edges of $G$.

Assume $(G, k) \in$ VCOVER, i.e., $G$ has a vertex $VC$ cover of size at most $k$. Then, consider the function $\ell$ such that $\ell(v) = 1$ if $v \in VC$, and 0 otherwise. Thus, $\sum_{v \in V} \ell(v) = |VC|$. Since $|VC| \leq k$, and since $\bar{k} = k + 1$, the summation is strictly less than $\bar{k}$. Moreover, since $VC$ is a vertex cover, i.e., for each $\{u, v\} \in E$, either $u$ or $v$ is in $VC$, it means that the cost of each edge in $\bar{G}$ is 1. Hence, $\sum_{\{u,v\} \in E} \max\{\ell(u), \ell(v)\} = |E|$, which is greater or equal than $\bar{k}' = |E|$. Hence, $\ell$ is a labeling.

Assume now that $\ell$ is a labeling of $(\bar{G}, \bar{k}, \bar{k}', X)$. Thus, let $VC$ be the set of all nodes $v$ of $\bar{G}$ such that $\ell(v) = 1$. Since $\ell$ is a labeling, and $X = \{0, 1\}$, the first summation $\sum_{v \in V} \ell(v)$ coincides with the size of $VC$ and it is strictly less than $\bar{k}$. But, $\bar{k} = k + 1$, and thus $|VC| \leq k$. It remains to show that $VC$ is a vertex cover of $G$. Again, since $X = \{0, 1\}$, the second summation $\sum_{\{u,v\} \in E} \max\{\ell(u), \ell(v)\}$ is the number of edges having at least an end point in $VC$, and since $\ell$ is a labeling, this summation is equal to $\bar{k}' = |E|$, and thus, every edge has an end point in $VC$, which implies $VC$ is a vertex cover. □

Consider now the search problem MAX-LABELS. The input is an undirected graph $G$, an integer $k$ and a set of integers $X$. The output is the largest $k'$ such that $(G, k, k', X) \in$ LABELS. Find the smallest possible complexity class in which MAX-LABELS belongs.

We can solve MAX-LABELS with a polynomial-time TM using an oracle for LABELS. In particular, the procedure does the following. Let $(G, k, X)$ be the input.

```
1. Let $a = 0$ and let $b = |E| \cdot x$, where $x$ is the largest label in $X$;
2. while $a < b$, do
3.  (a) Let $k' = \lfloor (a + b)/2 \rfloor$;
    (b) If $(G, k, k' + 1, X) \in$ LABELS, then let $a = k' + 1$;
    (c) Else if $(G, k, k') \in$ LABELS, then write $k'$ and halt;
    (d) Else let $b = k' - 1$;
4. Write $\perp$.
```

The above algorithm performs $O(\log_2 C)$ iterations, where $C$ is the value $|E| \cdot x$, which essentially is the maximum value the sum of all edges costs can be. Summing two numbers, one having $n$ bits and the other $m$ bits can lead at most to a number using $\max\{m, n\} + 1$ bits. So, if the labels in $X$ are encoded in binary, using $m$ bits each, the number $C = |E| \cdot x$ requires at most $|E| + m$ bits. Thus, $C \in O(2^{|E|+m})$. Hence, the iterations are $O(|E| + m)$, i.e., linear in the size of the input. Hence, MAX-LABELS is in $\mathbf{FP}^{\mathbf{NP}}$.

MAX-LABELS cannot be in **FP**, unless $\mathbf{P} = \mathbf{NP}$, because we can decide LABELS as follows. Given $(G, k, k', X)$, compute via the above algorithm, using input $(G, k, X)$ the maximum cost $C$. Then, verify if $C \geq k'$, in which case accept, otherwise reject. Hence, if MAX-LABELS were in **FP**, we would solve the **NP**−complete language LABELS in polynomial time, implying $\mathbf{P} = \mathbf{NP}$.

**NOAH'S ARK.** We now consider another decision problem, called NOAH'S ARK. We are given a number $m$, denoting the total number of species of animals. We would like to have one animal for each species in our ark, to save them from the great flood. However, are only allowed to place at most $k$ different species in the ark. So, you are also given an $m \times m$ matrix $S$ of numbers, that encodes the *similarity* between two species, i.e.,g $S[i][j]$ is the similarity between species $i$ and $j$ (the matrix $S$ is symmetric).

Given an additional number $\ell$, a representative set of species is a set $X \subseteq \{1,\dots,m\}$ of species that is smaller than $k$, i.e., $|X| \leq k$, but still, each species $j$ in $\{1,\dots,m\}$ is represented by some species $i \in X$, with a similarity of at least $\ell$. That is, for each species $j \in \{1,\dots,m\}$, either $j \in X$, or there is $i \in X$ such that $S[i][j] \geq \ell$.

The goal, given the $m \times m$ matrix $S$, the ark capacity $k$, and the similarity threshold $\ell$, is to decide whether there exists a representative set of species $X$, as defined above.

Prove that NOAH'S ARK is **NP**−complete (hint: reduce from DOMINATING SET).

**Theorem 51.** *NOAH'S ARK is* **NP**−complete.

*Proof.* The language is in **NP**, as we can simply guess a set $X \subseteq \{1,\dots,m\}$, which is of polynomial size. Verify $|X| \leq k$ is easy. To verify that each species $j \in \{1,\dots,m\}$ is represented, we need to iterate $j$ over $m$ elements, and for each such element, iterate all elements $i$ in $X$ and verify if $i = j$ or $S[i][j] \geq \ell$. So, it requires at most iterations, and at each iteration, we need to access the entry $S[i][j]$. Accessing the entry in a matrix by a TM is not a constant time operation, because it has to scan the matrix to find the right position. So, it requires $m^2$ steps as well, and thus the overall check requires $O(m^4)$, which is anyway polynomial.

Following the suggestion, let us see if there are some similarities between DOMINATING SET and NOAH'S ARK. In an undirected graph $G = (V, E)$, a dominating set, is a set of nodes $D$ that is able to reach every other node, with a single edge, i.e., $\forall v \in V \setminus D$, there is an edge between $v$ and a node of $D$. A representative set of species $X$ essentially plays the role of a dominating set, because it must "reach", via a minimum similarity, all the other species left out.

Our reduction does the following. Given a pair $(G, k)$, it constructs a triple $(S, k', \ell)$ such that $G$ has a dominating set of size at most $k$ iff there is a representative set of species $X$ for $(S, k', \ell)$. Let $G = (V, E)$. We let the matrix $S$ be a $|V| \times |V|$ matrix, where $S[i][j] = 1$, if there is an edge between nodes $i, j \in V$, otherwise $S[i][j] = 0$. Essentially, $S$ represents the adjacency matrix of the graph $G$. Since $X$ will represent our Dominating set of size at most $k$, we let $k' = k$. Finally, since we are only interested in "reaching" every other species, without caring on the amount of similarity, we let $\ell = 1$.

The reduction is polynomial, as it requires $O(|V|^2)$ steps to build $S$. Setting $k'$ and $\ell$ is straightforward.

Assume $G$ has a dominating set $D$ of size at most $k$. Then, the set of species $X = D$ is surely such that $|X| \leq k' = k$. Moreover, since every other node $v \in V \setminus D$ is connected with an edge to some node $i$ in $D$, for every species $j \in \{1,\dots,m\}$, when $j \notin X$, we have that $S[i][j] = 1$, by construction of $S$. Since $\ell = 1$, $S[i][j] \geq \ell$, and thus $X$ is a representative set of species.

Assume $X$ is a representative set of species for $(S, k', \ell)$. Let $D = X$. Since $X$ is a representative set, $|X| \leq k'$, and thus $|D| \leq k' = k$. Moreover, for every species $j \in \{1, \dots, m\}$, when they are not in $X$, are such that $S[i][j] \geq \ell = 1$, for some species $i \in X$. This means that for every node $j \in V$, when $j \notin V \setminus D$, there is an edge between $j$ and a node $i \in D$. This is the very definition of dominating set, and the claim follows. □

Consider now the search problem MAX-NOAH'S ARK. The input is the matrix $S$ and the ark capacity $k$. The output is the largest similarity $\ell$ for which we can find a representative set of species, i.e., $\ell$ is the maximum number such that $(S, k, \ell) \in$ NOAH'S ARK.

Once again, we can employ a TM with an oracle for NOAH'S ARK. We simply need to decide the right initial interval. Since we are looking for the largest similarity, the initial interval is $[0, C]$, where $C$ is the maximum number that appears in a cell of the matrix $S$. This is the maximum similarity we can hope to get. The binary search algorithm is then identical to the one shown for LABELS, and since $C$ is a number of the input encoded in binary, the *value* of $C$ is exponential, and thus MAX-NOAH'S ARK is in $\mathbf{FP^{NP}}$.

It turns out, however, that this time, even if $C$ is very large, we can find a clever way to reduce the number of calls to the oracle. Note that it is not really important to know the actual values that are in the matrix $S$, as far as we are able to compare them with the current maximum $\ell$. Indeed, there can be at most $m \times m$ different numbers in the matrix $S$. Let us say the following are the unique numbers in the matrix $S$ (ordered from smallest to largest):

$$a_1, a_2, \dots, a_p,$$

with $p \leq m^2$. No matter how many bits they contain, we can replace each $a_i$ in the matrix with the number $i$. Thus, we map the above numbers to the numbers:

$$1, 2, \dots, p.$$

Since $p \leq m^2$, a number in $1, \dots, p$ can be encoded using $O(\log_2 m^2)$ bits. So, assume we first convert $S$ as discussed, and then run our binary search algorithm. Now, the initial interval is $[0, m^2]$ in the worst case, hence it requires $O(\log_2 m^2)$ iterations. However, when the algorithm finally finds the optimal value $\ell$, this is a number among $1, \dots, p$. But we need to output the right number w.r.t. the original matrix $S$. Since $\ell$ is among the number in $1, \dots, p$, we just convert $\ell$ back to its corresponding number $a_\ell$, and output it. Hence, MAX-NOAH'S ARK is in $\mathbf{FP^{NP[\log_2 n]}}$.

**Remark.** Why did we not apply the same trick for LABELS? The reason why this approach will not work for LABELS is because there, the labels of the nodes, are not simply compared between each other, but are also *summed* together. For example, If we had labels $X = \{10, 34\}$, and we "scale them down" to labels 1,2, even though sum of the labels $10 + 34 = 44$ and $10 + 10 + 10 = 30$ are different numbers, summing the corresponding labels, we obtain the numbers $1 + 2 = 3$ and $1 + 1 + 1 = 3$, which are the same. So, if we apply this approach, and find the maximum value $k'$, w.r.t. the new scale, and say we obtain $k' = 3$, since we have to convert $k'$ back to a value that is in the scale of the original labels, we do not know exactly to which value $k'$ corresponds. Is it 44 or 30?