Theory of computation
Prof. Mika Göös — EPFL

Notes by Joachim Favre

Computer science bachelor — Semester 4
Spring 2023

Version 2023–06–13

To Gilles Castel, whose work has
inspired me this note taking method.

Rest in peace, nobody
deserves to go so young.

Contents

List of lectures

Chapter 1

Summary by lecture

Lecture 1 : Deterministic finite automatons — Monday 20th February 2023 ____ *p. 13*

- Definition of deterministic finite automatons, and many examples.
- Explanation of how to prove that a given language is the one accepted by the finite automaton, and some examples.

Lecture 2 : Killing cats — Monday 27th February 2023 ____ *p. 18*

- Definition of regular languages.
- Proof that the complement of a regular language, and the union and intersection of any two regular languages, are also regular, and explanation of the DFA constructions.
- Definition of NFAs.
- Proof that any NFA is equivalent to a DFA, and explanation of the construction.
- Proof that the concatenation of any two regular languages is also regular, and explanation of its NFA construction.

Lecture 3 : "Pigeon collisions" — Monday 6th March 2023 ____ *p. 24*

- Explanation and proof of the pumping lemma.
- Examples of application of the pumping lemma to show that some languages are not regular.

Lecture 4 : Turing ☺ — Monday 13th March 2023 ____ *p. 27*

- Definition of Turing machines and their configurations.
- Definition of Turing-recognisable and Turing-decidable languages.

Lecture 5 : Halting problem — Monday 20th March 2023 ____ *p. 30*

- Proof of the existence of undecidable languages.
- Proof that $HALT$ and A_{TM} are undecidable.
- Proof that $\overline{HALT}$ is unrecognisable.

Chapter 2

Introduction

History In the 1930s, people were asking themselves what can be mathematically proved (meaning, whether is maths limited), and what can be computed. Gödel answered the first one, and Turing the second one.
There are many people who work on theoretical computer science. Alan Turing is seen as the father of computer science: he defined mathematically what is an algorithm thanks to the Turing's machine, and he immediately proved limitations of it. Jack Edmonds introduced the class P, algorithms that can be solved efficiently (meaning in polynomial time). Stephen Cook and Leonid Levid introduced the concept of NP-complete problems, which we aim to understand in this course.

Theoretical computer science In theoretical computer science, we consider different problems and see how much time or space we would need to solve them. However, we don't stop there, and we try as well to make link between problems: see if one problem is easier than another, if randomness or quantum computers would help, if solving one implied another, and so on.
For instance, we want to understand efficiently solvable problem (P), efficiently verifiable problems (NP) and undecidable problems.

Chapter 3

Finite automatons

3.1 Deterministic finite automaton

Fixed memory

We will want to solve some problems using a fixed number of memory. This means that we scan the input from left to right and, for every symbol seen, we can only change the memory using a sate based on the current state and the current symbol.

Definition: DFA

A **deterministic finite automaton** (DFA) M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set called the **states** (which must not depend on the length of the input).
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma \mapsto Q$ is the **transition function**.
4. $q_0 \in Q$ is the **start state**.
5. $F \subseteq Q$ is the set of **accepting states** (where we allow $F = \emptyset$).

Note that $\delta(q, \sigma)$ encodes the state we go to from q when reading $\sigma \in \Sigma$. We extend this definition such that, for a string s, we use $\delta(q, s)$ to denote the state obtained by reading all of s starting in state q.

Observation We notice that, for the function δ to be complete, there must be one and exactly one new state for any state and any character.

Remark Note that this may seem a bit abstract, but this will become very clear using the following examples and their diagrams. This abstraction will allow us to reason on it.

Language of a machine

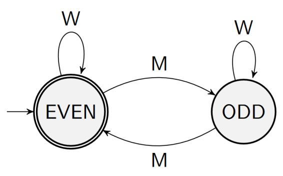
We let A to be the set of all string that the machine M accepts. We call A the **language of the machine** M and write $L(M) = A$. We say that M **recognises** or **accepts** A. For instance, if the machine accepts no string, it recognises the empty language $\emptyset$.

Definition: Empty string

The **empty string**, meaning the string which does not contain any character, is written ε.

Example 1: Parity

Let's say we receive as input a string s made of symbols M and W. We want to output yes if M appears in s an even number of times, and no otherwise.
We want to solve this problem using 1-bit of memory and, to do so, we use a DFA. The alphabet is $\Sigma = \{M, W\}$ (given by the problem description), and we have some states (the circles), transition functions (the arrows), a starting state (the additional arrow pointing at even) and some accepting state (circles which are circled, "even" in this case). We can make a diagram of this:



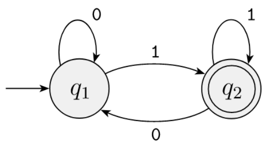
We can draw the evolution of the state with the following table, if we for instance read MWWMWMM:

Read	Initial	M	W	W	M	W	M	M
State	E	O	O	O	E	E	O	E

Since we ended up in the accepting state, we return `true`.

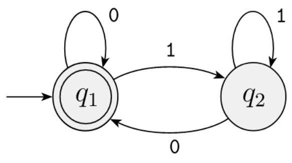
Remark Note that our DFA accepts the empty string ε.

Example 2 Let's consider the following DFA:



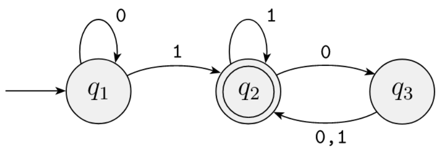
We want to know what it does. To do so, it is a good idea to try it on a few strings. For instance, 000 and 010 would not be accepted, but 011 would be. Spending a bit more time on it, we can see that when we read a 0, we always go to q_1. This means that it accepts all strings that end with a 1.

Example 3 Let's consider the following DFA:



It is very close to the last example. This one accepts all string ending with a 0, and the empty string ε.

Example 4 Let's consider the following DFA, which accepts strings with at least a 1 and that end in an even number of 0s:



We can describe formally with $M = (Q, \Sigma, \delta, q_1, F)$ where $Q = \{q_1, q_2, q_3\}$, q_1 is the start state, $\Sigma = \{0, 1\}$, $F = \{q_2\}$ and δ is described as:

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

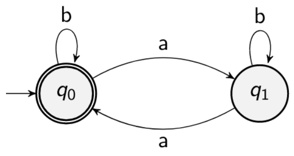
Then, we say that M recognises the language:

$L(M) = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}$

Note that this formal definition and the diagram above are completely equivalent.

Proving correctness To prove the correctness of a DFA, that it indeed recognises a given set, we will use induction.

Example 1 Let us consider again the following DFA:



We want to prove that M accepts exactly L where:

$$L = \{w \mid \text{count}(w, a) \text{ is even}\}$$

where $\text{count}(s, c)$ returns the number of times c appears in s.
The following proof will be very pedantic, but it is a good way to understand how to make really formal proofs.

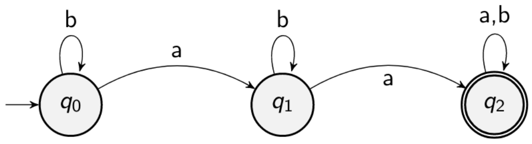
Proof We want to show that, for all strings w, $\delta(q_o, w) = q_0$ (M accepts w) if and only if $\text{count}(w, a)$ is even.
Let's begin with the base case, $w = \varepsilon$. We know that $\delta(q_0, \varepsilon) = q_0$. However, for the empty string, $\text{count}(\varepsilon, a) = 0$, which is indeed even.
Now, let's do the inductive case. We assume that our claim holds for an arbitrary string x, and we want to show that the claim works for $w = x.\sigma$ where σ is a symbol (a or b) and the dot represents appending a character to a string. We have four different cases for $\delta(q_0, x)$, which can be either equal to q_0 or q_1, and σ can either be a or b. We will only consider the case where $\delta(q_0, x) = q_0$ and $\sigma = b$, the others are similar.
By the inductive hypothesis, $\text{count}(x, a)$ is even. By definition of δ, and since $\delta(q_0, x) = q_0$ (we just took this hypothesis when selecting one of the four cases):

$$\delta(q_0, x.b) = \delta(\delta(q_0, x), b) = \delta(q_0, b) = q_0$$

However, since adding a b does not change the number of a's a string contains, $\text{count}(x.b, a) = \text{count}(x, a)$, which is even by inductive hypothesis. This finishes our proof.

□

Example 2 Let's consider the following DFA:



We want to show that:

$$L(M) = \{w \mid w \text{ contains at least two } a\text{'s}\}$$

Wrong proof We want to prove that, for all strings w, $\delta(q_0, w) = q_2$ if and only if $\text{count}(w, a)$ is at least 2. This proof will fail.
Let's start with the base case, $w = \varepsilon$. By definition, $\delta(q_0, \varepsilon) = q_0$. Also, $\text{count}(w, a) = 0 < 2$. Both sides of our equivalence are false, showing that base case holds.
Let's now make the inductive step, and let $\sigma \in \{a, b\}$. Let's consider the case where $\delta(q_0, x) = q_0$ and $\sigma = a$. In this case, by induction hypothesis, $\text{count}(x, a) < 2$. However, it would not break our

hypothesises that $\text{count}(x, a) = 1$ (this would never happen in practice since the count must be 0 in q_0, but our hypotheses do not catch this). In this case, we would have $\text{count}(x.a, a) = 2$ but $\delta(q_0, x.a) = \delta(q_0, a) = q_1$, meaning that our claim does not hold.

Better proof Let's make a better proof. To do so, we need to prove something stronger than our claim, something that tells us more about the automata. We will thus try to prove:

$$\delta(q_0, w) = \begin{cases} q_0 & \text{if count}(w, a) = 0 \\ q_1 & \text{if count}(w, a) = 1 \\ q_2 & \text{if count}(w, a) \geq 2 \end{cases}$$

The base case holds rather trivially. Concerning the inductive step, we have even more cases to consider. However, doing it is not too hard and completely solves the problem we have just had.

Remark We can always prove the correctness of any automata using strong induction. Let's see the general recipe.
Let's say that we are given an arbitrary language L described by a mathematical constraint, and a DFA $M = (\{q_0, \ldots, q_n\}, \Sigma, \delta, q_0, F)$.
To prove $L(M) = L$, we need to start by finding a precise description of the sets T_i of strings that make the machine go to state q_i, for all $i = 0, \ldots, n$. For instance, in our example above, T_0 was the set of strings such that $\text{count}(w, a) = 0$, and similarly for T_1 and T_2. Naturally, the language L should match the sets corresponding to the accepting states.
Then, we need to prove by induction that, for any string w, $\delta(q_0, w) = q_i$ if w is in set T_i; for all $i = 0, \ldots, n$.

Monday 27th February 2023 — Lecture 2 : Killing cats

Definition: Regular language $A \subseteq \Sigma^*$ is named a **regular language** if there exists at least a DFA M such that $A = L(M)$ (meaning that it is the language of some DFA).

Example For instance, the following language is a regular language, as we have already seen a DFA for it:

$$L = \left\{w \in \{0, 1\}^* \mid w \text{ contains an even number of 1's}\right\}$$

Formally, we would need to prove the correctness of the DFA.

Definition: Complement The **complement** of some language L is defined as:

$$\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$$

Theorem: Complement Let L be some regular language and $M = (Q, \Sigma, \delta, q_0, F)$ be a machine recognising it.
Then, the complement of L, $\overline{L}$, is regular and is the language of the following machine:

$$M' = \left(Q, \Sigma, \delta, q_0, \overline{F} = Q \setminus F\right)$$

Remark We just swap the accepting state and the non-accepting state. This naturally means that strings that were accepted are no longer, and inversely.

Definition: Union The **union** of two languages L_1 and L_2 is:

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}$$

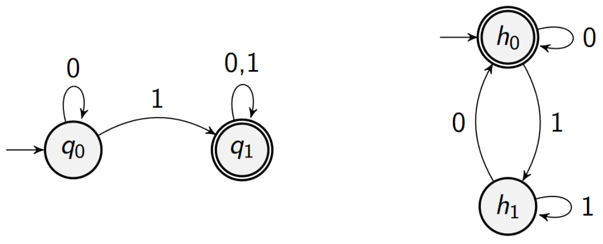
Theorem: Union Let L_1 and L_2 be the languages of machines $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, respectively.
Then, the union of L_1 and L_2, $L_1 \cup L_2$ is regular, and it is accepted by the machine $M = (Q, \Sigma, \delta, q_0, F)$, where:

$$Q = Q_1 \times Q_2, \quad \delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$$

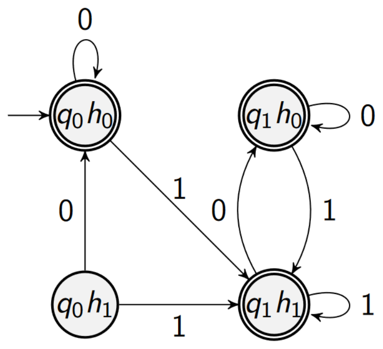
$$q_0 = (q_1, q_2), \quad F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$$

Intuition We are basically simulating the two machines at the same time, keeping track of the two states as a tuple. We then consider that we end up in an accepting state if either machine ended up in one.

Example Let us consider the following two machines:



Then, their union is:



Definition: Intersection The **intersection** of two languages L_1 and L_2 is:

$$L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\}$$

Theorem: Intersection Let L_1 and L_2 be the languages of machines $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, respectively.
Then, the intersection of L_1 and L_2, $L_1 \cap L_2$ is regular, and it is accepted by the machine $M = (Q, \Sigma, \delta, q_0, F)$, where:

$$Q = Q_1 \times Q_2, \quad \delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$$

$$q_0 = (q_1, q_2), \quad F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ and } r_2 \in F_2\}$$

Intuition This is the *exact* same construction as for intersection, except that we consider the accepting states to be the ones where both DFA ended up in one.

Definition: Concatenation The **concatenation** of two languages L_1 and L_2 is:

$$L_1 \circ L_2 = \{w \in \Sigma^* \mid w = w_1.w_2, w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

Observation It is very hard to know if concatenating two regular languages always yield a regular language. The main problem is that we do not know when to switch from the first machine to the second one: a string can be obtained through concatenation in many ways ($101 = 1 \circ 01 = 10 \circ 1 = 101 \circ \varepsilon$).

We will see that the concatenation of two languages is indeed regular but, for this, we need to do some non-determinisim.

3.2 Non-deterministic finite automaton

Definition: Power set Let Q be some set. We note 2^Q the power set of Q, meaning the set of all the subsets of Q.

Example For instance, let us consider $Q = \{q_1, q_2\}$. Then:

$$2^Q = \{\emptyset, \{q_1\}, \{q_2\}, \{q_1, q_2\}\}$$

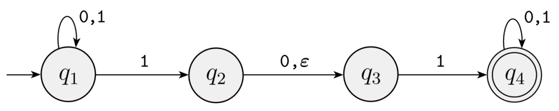
Definition: NFA A **non-deterministic finite automaton** (NFA) M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set called the states.
- Σ is a finite set called the alphabet.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \mapsto 2^Q$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept states, where we allow $F = \emptyset$.

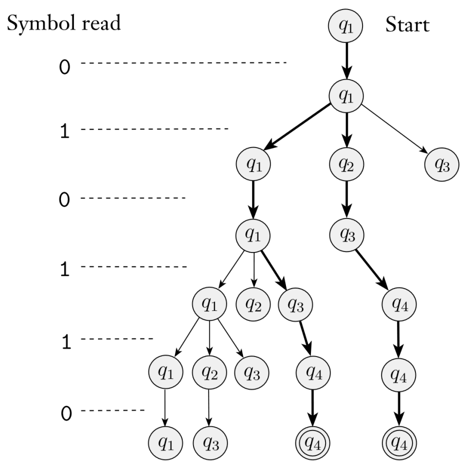
A NFA has multiple ways to compute the final state for some string. It accepts the word if there exists a set of choices, that make the word to be accepted. In other words, if we run a parallel computer that follows all possibilities in parallel, we consider the word to be accepted if any of them are in an accepting state at the end.

Comparison with DFA A NFA has two main difference to DFA.
First, it has the ability to transition to more than one state on a given symbol, or no symbol at all: the codomain of δ allows any subset of Q, meaning that it could map to $\emptyset$ or $\{q_1, q_2\}$. Note that if at any point we reach a state where we cannot transition out (because $\delta(\sigma, q_i) = \emptyset$), then the path ends and does not continue further. This path is not considered accepted even though it died on an accepting state.
Second, it has the possibility to take a step in-between reading symbols, thanks to ε transition. In other words, if $\delta(\varepsilon, q_i) = \{q_j\}$, then the machine can decide to transition form q_i to q_j even without reading any character.

Example 1 Let's consider the state diagram of the following NFA:



As we can see, there are indeed some states which can transition to more than one state on a given symbol, some which have no transition, and some which have an ε-transition.
A good way to see the computation of an NFA is a tree where any level represents all the possible' states reachable at that point. In this case, if we read input 010110, then we get:



Note that, as mentioned before, a thread which cannot take transition is *killed* (the fact that the professor represented the different parallel threads as cats makes this really violent). Also, we can see that the ε transitions can be taken in-between reading symbols, this was mentioned above as well.

Since, in this example, we have some threads which end up in an accepting state at the end of the string, then it is accepted.

Observation Note that, to show an input is accepted, we don't need to draw the whole tree, only the branch going to an accepting state. This is named guess and verify.

Remark In fact, we can convince ourselves that this NFA recognises the following language:

$$L = \left\{w \in \{0,1\}^* \mid w \text{ contains 11 or 101 as a substring}\right\}$$

Formal definition The formal definition of our machine is $M = (Q, \Sigma, \delta, q_1, F)$, where:

$$Q = \{q_1, q_2, q_3, q_4\}, \quad \Sigma = \{0,1\}, \quad F = \{q_4\}$$

Also, the function δ is defined as:

δ	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	q_3	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

Just as for DFA, this formal definition and the state diagram above are completely equivalent.

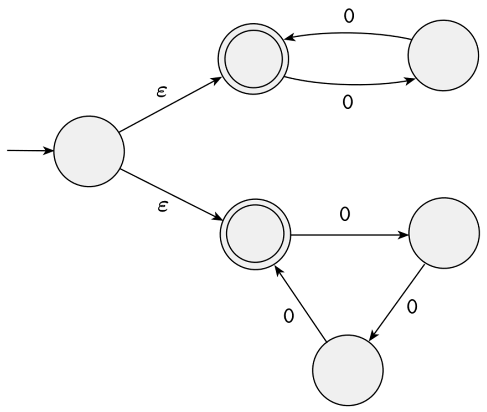
Example 2 Let's consider the following NFA:

$$\begin{tikzcd} \arrow[r] & q_1 \arrow[loop above, "{0,1}"] \arrow[r, "1"] & q_2 \arrow[r, "{0,1}"] & q_3 \arrow[r, "{0,1}"] & \boxed{q_4} \end{tikzcd}$$

We notice that, for instance, "1000" is not accepted since the only thread which reaches q_4 dies before the end of the string.
We can convince ourselves that it recognises the following language:

$$L = \left\{w \in \{0,1\}^* \mid w \text{ contains a 1 in the third position from the end}\right\}$$

Example 3 Let us now consider an NFA over a unary language:



The beginning with the two ε-transitions, the only non-deterministic part of the machine, allows to have a non-deterministic choice of starting state.
This machine is the union of two regular languages: $w \in \{0\}^*$ has a length which is a multiple of 2, and $w \in \{0\}^*$ has a length which is a multiple of 3. This construction is more efficient in size, and it gives:

$$L = \left\{w \in \{0\}^* \mid w \text{ has a length which is multiple of 2 or 3}\right\}$$

Remark An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. In fact, they seem really more powerful than DFA, but this is a wrong intuition as we will see with the following theorem, for which we will need some lemma beforehand.

Theorem Every non-deterministic finite automaton has an equivalent deterministic finite automaton.

Proof We will only prove the case without ε-transitions since it slightly complicates the initial states. However, the proof with them is very similar, and we will show in the second series of exercises that any NFA with ε-transitions can be translated to any NFA without such transitions.
Let $N = (Q_N, \Sigma, \delta_N, q_0, F_n)$ be an arbitrary NFA. We want to construct a DFA M such that:

$$L(N) = L(M)$$

The idea is to consider that a state of the DFA represents the set of states from the NFA which could be reached by one of the threads. In other words, the transition function maps a set of states into another set of all the possible states the threads could reach in one step. Mathematically, we let $M = \left(2^Q, \Sigma, \widetilde{\delta}, \{q_0\}, \widetilde{F}\right)$, where:

$$\widetilde{\delta}(A, a) = \bigcup_{q \in A} \delta(q, a)$$

Also, we let $\widetilde{F}$ to contain all the sets of states which contain an accepting state:

$$\widetilde{F} = \left\{A \in 2^Q \mid A \cap F \neq \emptyset\right\}$$

We now want to show that M's state after $x \in \Sigma^*$ is $\widetilde{\delta}(\{q_0\}, x) = A$ if and only if A is N's possible states after reading x. We will prove this by induction.
Let's begin with the base case, taking $x = \varepsilon$. We see that $\widetilde{\delta}(\{q_0\}, \varepsilon) = \{q_0\}$ by definition of the starting state, which is indeed

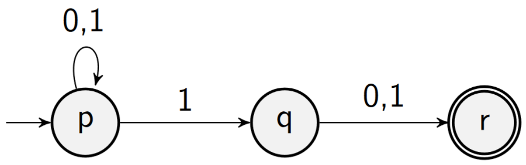
N's possible states after ε. Note that this is where appears our hypothesis that our machine does not have an ε-transition, since, else, we may not have the equality hereinabove and it thus complicates the proof.
Let us now consider the inductive step. As usual, we assume that the claim holds for x, and we want to prove it for $x.a$. We see that:

$$\widetilde{\delta}(\{q_0\}, x.a) = \widetilde{\delta}\Big(\widetilde{\delta}(\{q_0\}, x), a\Big) \overset{\text{déf}}{=} \bigcup_{q \in \widetilde{\delta}(\{q_0\}, x)} \delta(q, a)$$

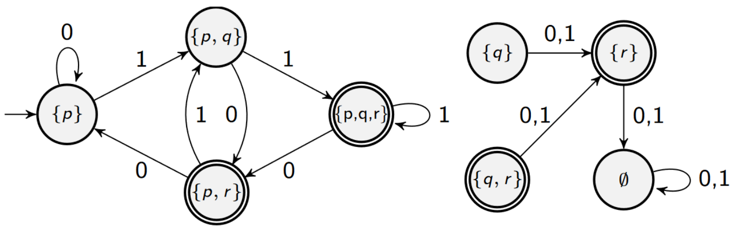
where $\widetilde{\delta}(\{q_0\}, x)$ is N's possible states after reading all the characters of x, as usual. However, we are taking the union of all the states we can reach from those those possible states, keeping our inductive property.
In other words, since we indeed had all the possible states of N for x, then we still have all the possible states of N after $x.a$, as required.

□

Example Let's consider this construction for the following NFA:



We can construct a new DFA which considers all the 8 subsets of $\{p, q, r\}$ as states, and we consider an accepting state to be any set containing r:

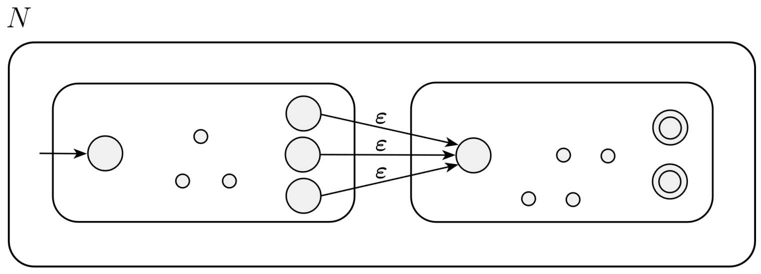


For instance, if we are in any of the state q or r and we read a 1, we can go to any of p, q and r. This indeed means that $\delta(\{p, q\}, 1) = \{p, q, r\}$.
We can see that the this DFA can never reach some of its states, such as $\emptyset$.

Corollary A language is regular if and only if some NFA recognises it.

Theorem: Concatenation Let L_1 and L_2 be regular languages.
Then, the concatenation of L_1 and L_2, $L_1 \circ L_2 = L$, is regular.

Proof Concatenation is very easy with non-determinism. Indeed, we just need to link the accepting states of the first machine to the initial state of the second machine using ε-transition. In other words, we allow to switch to the second machine as soon as we recognise a word, but also to stay on the first one.



However, by the corollary above, since we have a NFA which language is the concatenation of L_1 and L_2 is indeed regular.

□

Monday 6th March 2023 — **Lecture 3 : "Pigeon collisions"**

3.3 Non-regular languages

Existence of non-regular languages

We have seen that many languages are regular: using a finite automaton, we can verify a number has a given substring, check if it is divisible modulo a given number, skip some prefix and suffix, do complements, unions, intersections, and so on.
We now wonder if all languages are regular. In fact, sadly, no. Let us consider the following language:

$$B = \{0^n 1^n \mid n \geq 0\} = \{\varepsilon, 01, 0011, 000111, \ldots\}$$

The problem is that we need some kind of memory depending on the size of the input: we need to recall how many zeroes we had when we are counting the ones.

Proof

Let us prove that B is indeed not regular.
Let's suppose for contradiction that B is recognised by a DFA with p states. Note that p has to be finite.
Now, let us consider the countable infinite set $\left\{0^1, 0^2, \ldots\right\}$. By the pigeon hole principle, since there is a finite number of states p but an infinite number of 0^k, there must exist i, j such that 0^i and 0^j end up in the same state. We have a pigeon collision.

However, this means that if $0^i 1^i$ is accepted, then $0^j 1^i$ is also accepted. However, $0^j 1^i$ should not be accepted, which is our contradiction.

□

Remark

Note that, to show a language is not regular, we cannot just say "it needs memory depending on the size of the input" blindly. Let us consider the following language:

$$D = \{w \mid w \text{ has an equal number of occurrences of 01 and 10 as substrings}\}$$

In fact, we can show that it is equivalent to:

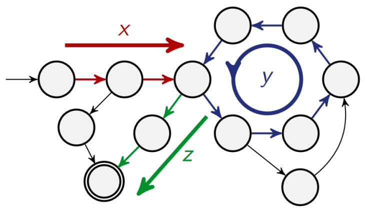
$$D = \{w \mid w \text{ begins and ends with the same character, or length}(w) \leq 1\}$$

Thus, D is regular.

Pumping lemma If A is a regular language, then there is a number p (named the pumping length) such that, for every string s in A of length at least p, there exists a division of s into three pieces, $s = xyz$, such that:

1. For any $i \geq 0$, $xy^i z \in A$.
2. $|y| \geq 1$.
3. $|xy| \leq p$.

Intuition It is much better to remember the proof of this lemma, instead of the lemma itself.
In fact, it just generalises the argument we had before. Let $p = |Q|$. By the pigeon hole principle, if we have a string of length at least p, we need to have a loop: let y be a string which loops from a state q_i to the same state q_i. Then, we also need x to be a string bringing from the initial state to q_i, and z to be a string bringing from q_i to an accepting state. We can then go from the initial step to q_i, loop any number of times we want (even 0 times) and go to an accepting state.



$|xy| \leq p$ just means that the state collision (which leads to the loop) has to take place before p.

Proof Let M be an arbitrary DFA accepting A, and let $p = |Q|$ be the number of states in M. Also, let s be any arbitrary string of length $|s| \geq p$.
By the pigeonhole principle, since we have p states and we visit $p+1$ states when reading a string of length p, this means that we must always have some collision for any string which length is less than or equal to p. Thus, when reading the p first characters of s, there must be a loop: there must be some x, y, z such that $|xy| \leq p$ (the loop must happen in the first p characters of s), $|y| \geq 1$ (because of the loop) and:

$$\delta(q_0, x) = q, \quad \delta(q_0, xy) = q, \quad \delta(q, z) = r \in F$$

In other words, x takes us to some state q; then, y loops us back to this same state; and finally z brings us to an accepted state.
However, this definitely means that:

$$\delta\left(q_0, xy^i z\right) = r, \quad \forall i \geq 0$$

□

Example 1 Let us consider the following language:

$$F = \left\{ww \mid w \in \{0,1\}^*\right\}$$

In other words, we only want words which left part is equal to its right part.
Let's suppose for contradiction that this language is regular. By the pumping lemma, it must have some pumping length p. Now, any long string can be pumped. Let's pick $s = 0^p 1^p 0^p 1^p \in F$ (note that the choice of which string to pick is the whole game when doing such proofs).
By the pumping lemma, we know that there exists some x, y, z such that:

$$s = xyz, \quad |xy| \leq p, \quad |y| \geq 1, \quad xy^i z \in F, \ \forall i \geq 0$$

However, since $|xy| \leq p$, this necessarily means that $xy = 0^j$ for some j by construction. Moreover, since $|y| \geq 1$, we get that $y = 0^k$ for some $k \geq 1$.
According to the pumping lemma, we must have $xy^2z \in F$. However:

$$xy^2z = 0^{p+k}1^p0^p1^p \notin F$$

since the left part of this new string does not match the right one.
This is our contradiction.

Example 2 Let us consider the following language:

$$E = \{0^m1^n \mid m > n\}$$

We want to show that E is not regular. Let us suppose for contradiction that E is regular, with p as pumping length.
Let us consider the following string:

$$s = 0^{p+1}1^p \in E$$

Since we know that $|xy| \leq p$ and $|y| \geq 1$, we necessarily have that $y = 0^k$ for some $k \geq 1$. However, we see that:

$$xy^0z = xz = 0^{p+1-k}1^p \notin E$$

since $p + 1 - k \leq p$. By the pumping lemma, we should have $xz \in E$, this is our contradiction.

Example 3 Let us consider the following language:

$$C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$$

We again want to show that it is not regular. Thus, let's suppose for contradiction that C is regular, and that p is its pumping length.
Let us consider the following string:

$$s = 0^p1^p \in C$$

Since we know that $|xy| \leq p$ and $|y| \geq 1$, we necessarily have that $y = 0^k$ for some $k \geq 1$. Moreover:

$$xy^2z = 0^{p+k}1^p \notin C$$

However, since the pumping lemma tells us that $xy^2z \in C$, we got our contradiction.

Other proof We can use an easier approach than the pumping lemma.
Let us consider again a language we have proven not to be regular at the start of this lecture:

$$B = \{0^n1^n \mid n \geq 0\} = \{\varepsilon, 01, 0011, 000111, \ldots\}$$

However, we can see that:

$$B = C \cap \{0^m1^n \mid m, n \geq 0\}$$

Let us now suppose for contradiction that C is regular. However, we know that $\{0^m1^n \mid m, n \geq 0\}$ is a regular language, and the intersection of two regular language is also regular. This thus means that B is regular, which is a contradiction.
This shows that the pumping lemma, though very powerful, is not the always the only way to go.

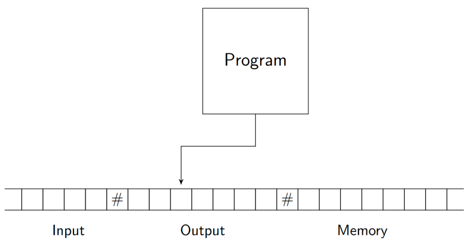
Chapter 4

Turing machines

4.1 Formal definition

Introduction So far, it feels like we miss memory: we would need some kind of memory depending on the size of the input (while keeping a fixed-size program). This is where Turing machines come in.
The idea is that we have a tape on which we can write and read things. Thus, we have like a finite automaton where we can also take into account what is written on the tape. We still have a finite number of states (the program is finite), but the tape is infinite (allowing for different inputs to use different size of memory).

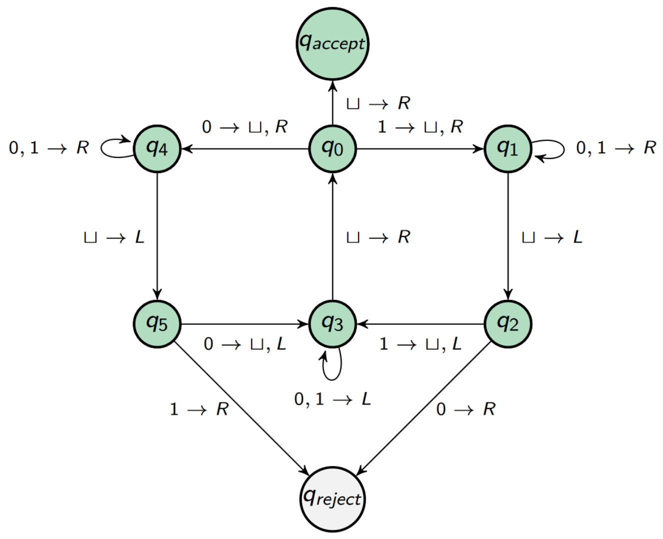
Observation We only need a single tape since we can reserve some part of the memory for the input and the output.



It is possible to show that having multiple tapes or a single one is completely equivalent.

Capabilities Even though the tape is infinite, the Turing Machine can only see a specific symbol, where its current head is.
At any given step, it acts depending on what is written on the tape under the head and the current state. Just as for DFA, it goes to some state. However, moreover, it can move the tape right or left, and write any symbol on it (including a new symbol, ⊔, representing an empty cell).
As soon as it reaches the accept or reject state, the machine stops immediately, unlike DFA. Because of that we can have a single accept and reject states (one of each), since it would not be useful to have more. Moreover, this means that it may never halt: it may never reach an accept or reject state.

Example Let us consider the following diagram:



Let's say the head begins at the first character of "1001". We start at q_0 and read a 1, so we transition to state q_1, empty the current cell, and move the tape one to the right. We can continue this until the accepting state.
We can in fact convince ourselves that this Turing machine accepts even length binary palindromes.

Remark This is a very inefficient algorithm to find palindromes. However, in this part of the course, we do not care about running time.

Definition: Turing Machine A **Turing Machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_o, q_{accept}, q_{reject})$ (with Γ and q_{reject} being the new elements with respect to DFA), where:

1. Q is the set of states.
2. Σ is the input alphabet, not containing the blank symbol $\sqcup$.
3. Γ is the tape alphabet, where $\Sigma \cup \{\sqcup\} \subseteq \Gamma$ (and it potentially contains more characters).
4. $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $q_{accept} \in Q$ is the accept state.
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Definition: TM configuration Let's consider the Turing Machine $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. We still need to define how it computes.
We write uqv where $u, v \in \Gamma^*$ and $q \in Q$ for the configuration where:

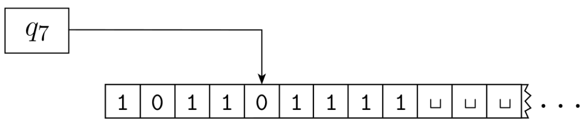
1. The current state is q.
2. The current tape content is uv.
3. The curent head location is the first symbol of v.
4. The cells whose content are unspecified are blank, they contain $\sqcup$.

Given a configuration uaq_ibv where $a, b \in \Gamma$, $u, v \in \Gamma^*$ and $q_i \in Q$, we move to:

$$\begin{cases} uq_jacv, & \text{if } \delta(q_ib) = (q_j, c, L) \\ uacq_jv, & \text{if } \delta(q_i, b) = (q_j, c, R) \end{cases}$$

We start at some configuration $C_1 = q_0w$ for some $w \in \Sigma^*$. We then obtain new configurations $C_2, C_3, \ldots$ by valid transitions. We accept the word w if the machine halts and if the state q_{accept} is reached, but we reject it if it halts and the state q_{reject} is reached.
Naturally, the program may not terminate. We can even show that some loops may happen even though no configurations are repeated (for instance if the machine writes a 1 to the current cell and moves to the right).

Example For instance, a Turing Machine with configuration $1011q_701111$ would look like:



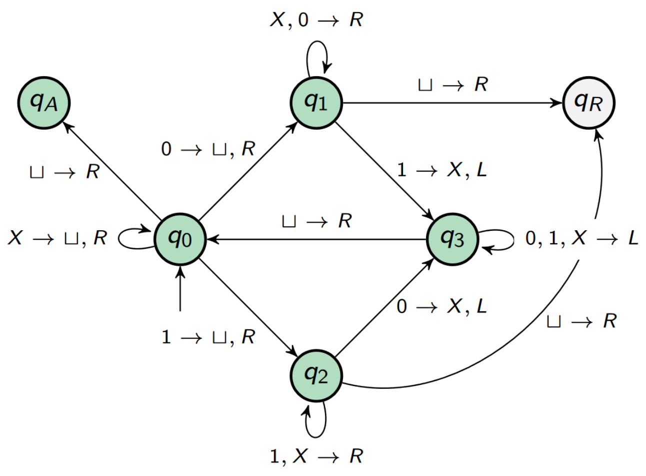
Example We want to make a Turing Machine which recognises the following language:

$$L = \{w \mid w \text{ has an equal number of 0s and 1s}\}$$

We have an input alphabet $\Sigma = \{0, 1\}$. We consider a tape alphabet which has an extra character X, which represents a crossed off letter:

$$\Gamma = \{0, 1, \sqcup, X\}$$

The idea is to scan the input from left to right. Whenever we encounter an uncrossed 0 or 1, we replace it by a cross, and proceed to the right to find a corresponding 1 or 0 that we cross. We then go back at the start of the input, and start again. If we end up crossing out every characters, we accept it, but if we can't find a corresponding 1 or 0 to a 0 or 1, we reject it.



Note that we are doing something slightly more intelligent here: when we are at the left of the string and scan for the first non-X character, we also replace all X characters by $\sqcup$, in order to simplify the detection when we finished considering the whole word.

4.2 Decidable and undecidable languages

Definition: Turing-Recognisable A language L is **Turing-Recognisable** (or Recognisable) if there exists a Turing Machine M which recognises L.
In other words, all $w \in L$ must be accepted by M and all $w \notin L$ do not make M stop or M rejects w.

Definition: Turing-Decidable A language L is **Turing-Decidable** (or Decidable) if there exists a Turing Machine M which decides L.
In other words, all $w \in L$ must be accepted by M and all $w \notin L$ must be rejected by M.

Observation This second definition is stronger: if a machine decides a language, then it recognises it. However, the converse may not be true. The machine may not stop on some words $w \notin L$.

Church-Turing thesis In 1936, two scientists gave different definitions for the concept of algorithms. Church defined it using λ-calculus, in an intuitive way. Turing defined it using his machines. However, we can show that the two definitions are equivalent. This means that anything we can write in programming languages (such as C, Java, and so on) or with quantum computers can also be executed on Turing Machines; and conversely. Thus, when we want to show that a language is decidable or recognisable, we do not necessarily mean to make a formal Turing Machine, a program is sufficient.

Monday 20th March 2023 — Lecture 5 : Halting problem

Definition: Encoding Let S be any object. We write its binary encoding $\langle S\rangle$.

Example 1 A Turing Machine can thus, for instance, decide whether a graph is connected, or find a shortest path.
To give a graph G as an input to a Turing Machine, we just give $\langle G\rangle$.

Example 2 We want to make a Turing machine which checks if a DFA is empty, if it rejects every word.
The first idea may be to try all possibilities $s \in \{\varepsilon, 0, 1, 00, \ldots\}$ and see if the DFA accepts it. The problem with this approach is that, if the DFA accepts nothing, then we will never halt.
A second, better idea, is to check if there exists a set of transitions bringing from the initial state to an accepting state. This can be done using BFS, considering the DFA as a graph; which always halts.

Example 3 We want to make a Turing Machine which decides whether two DFAs recognise the same language.
We notice that $L(D) = L(D')$ if and only if $L(D) \oplus L(D') = \emptyset$, where $\oplus$ is the symmetric difference of state (an element is in the symmetric difference if it is in one set xor the other).
Moreover, we can see that:

$$L(D) \oplus L(D') = \left(L(D) \cap \overline{L(D')}\right) \cup \left(\overline{L(D)} \cap L(D')\right)$$

We saw that complements, unions and intersections of regular languages were also regular, so $L(D) \oplus L(D')$ is regular and there exists a DFA $D^{\oplus}$ accepting it.
So, we can make a Turing Machine to create $D^{\oplus}$ from D and D', and then verify if it is empty using the program made right before. This always halts, so this is perfect.

Example 4: Halting problem We now receive a Turing Machine and some input as input, and we want to output whether the machine halts.
It is rather easy to show that this problem is recognisable. We can just simulate the Turing machine, and see if it halts. The problem is that it seems very hard to decide. We will develop some more theory and come back on this problem.

Definition: Cardinality of sets Two sets A and B have the same cardinality, written $|A| = |B|$ if there exists a bijective function $f : A \mapsto B$.

> *Example* For instance, we can show that $|\mathbb{N}| = |\mathbb{Q}|$ (and we did in AICC-1).

Definition: Countable set A set A is **countable** if either it is finite or it has the same cardinality as $\mathbb{N}$.

> *Example* As mentioned earlier, $\mathbb{Q}$ is countable.

Theorem: $\mathbb{R}$ is not countable The set of real numbers is not countable:

$$|\mathbb{R}| \neq |\mathbb{N}|$$

Proof The proof can be done using Cantor's diagonalisation. It was also done in AICC-1, but let's do it again.
Suppose for contradiction that there exists a bijection $f : \mathbb{N} \mapsto \mathbb{R}$. We want to find some $x \in \mathbb{R}$ for which there exists no $a \in \mathbb{N}$ such that $f(a) = x$.
We construct x by taking a number different from the n^{th} fractional digit of $f(n)$, effectively taking different digits over the diagonal. For instance, if we have the following f:

n	$f(n)$
1	$3.\underline{1}4159\ldots$
2	$55.5\underline{5}555\ldots$
3	$0.12\underline{3}45\ldots$
4	$0.500\underline{0}00\ldots$
$\vdots$	$\vdots$

Then we could pick $0.3323\ldots$
There cannot exist some m such that $f(m) = x$: the m^{th} digit of x is different from the m^{th} digit of $f(m)$, by construction. This thus finishes our proof.

□

Theorem There exists some language which is not decidable.

Proof We can build a table for Turing Machine: the rows list the Turing Machine (they are countable, so we can indeed make a list of them). The columns represent a binary representation of each machine, which we feed as an input to the machines (this is a bit meta, but this will be useful). Each entry of the table states if the machine accepts the given input, rejects it, or never halts.

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\langle M_5\rangle$	$\langle M_6\rangle$	$\ldots$
M_1	A	∞	R	A	A	R	$\ldots$
M_2	R	R	A	A	∞	A	$\ldots$
M_3	R	∞	A	∞	R	R	$\ldots$
M_4	A	∞	R	R	R	∞	$\ldots$
M_5	∞	∞	A	A	A	A	$\ldots$
M_6	R	A	R	∞	A	∞	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Now, we construct a language $DIAG$ such that it contains all inputs $\langle M_i\rangle$ such that the i^{th} machine M_i does not accept this input. In this example, we would have:

$$\begin{aligned} DIAG &= \{\langle M_i\rangle \mid M_i \text{ doesn't accept } \langle M_i\rangle\} \\ &= \{\langle M_2\rangle, \langle M_4\rangle, \langle M_6\rangle, \ldots\} \end{aligned}$$

We want to show that $DIAG$ is undecidable. Let's suppose for contradiction that there exists some Turing Machine M which decides $DIAG$. However, since we made the list of all possible machines, it means that $M = M_i$ for some $i \in \mathbb{N}$ in the table.
Now, we cannot have $\langle M_i\rangle \in L(M_i)$ since it would imply that the machine accepts itself and thus, by definition of $DIAG$, it would imply that $\langle M_i\rangle \notin DIAG$.
Similarly, we cannot have $\langle M_i\rangle \notin L(M_i)$ since it would imply that the machine does not accept itself and thus, by definition of $DIAG$, it would imply that $\langle M_i\rangle \in DIAG$.

We cannot have $\langle M_i \rangle \in L(M_i)$ and $\langle M_i \rangle \notin L(M_i)$. This is our contradiction.

□

Theorem The following language is undecidable:

$$HALT = \{\langle M, w \rangle : M \text{ is a Turing Machine which halts on } w\}$$

Proof Let's assume that there exists some Turing Machine H which decides $HALT$.
We construct a Turing Machine D which decides $DIAG$, defined in the proof of the previous theorem. On an input $\langle M \rangle$, we run H on the input $\langle M, \langle M \rangle \rangle$. If H rejects (meaning that M loops on $\langle M \rangle$), then we accept. If it accepts (meaning that M halts on $\langle M \rangle$), then we run M on the input $\langle M \rangle$, and output the opposite of M (accept if it rejects, and inversely).

```
machine D(M):
    if H(<M, <M>>) then  // meaning that M halts on input <M
        >
        return !M(<M>)  // output accepted if rejected, and
            inversely
    else
        return accepted  // it loops, so we accept
```

We notice that D halts on all inputs: we only run machines we are sure will end. Moreover, D accepts $\langle M \rangle$ is equivalent to M not accepting $\langle M \rangle$ and thus equivalent to $\langle M \rangle \in DIAG$.
This is a contradiction, since we showed that $DIAG$ was undecidable.

□

Theorem The following language is undecidable:

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing Machine and accepts } w\}$$

Proof Let's assume for contradiction that there exists a decider H for A_{TM}.
Again, we construct a decider D for $DIAG$ using H. To do so, we feed $M, \langle M \rangle$ to H, and output its opposite value (if it accepts, then we reject, and inversely).

```
machine D(M):
    return !H(<M, <M>>)
```

As before, D halts on all inputs and D accepting $\langle M \rangle$ is equivalent to $\langle M \rangle \in DIAG$ (we output rejected if and only if the machine accepts itself). This indeed shows that D decides $DIAG$, which is our contradiction.

□

Observation We have seen undecidable languages, but there also exists unrecognisable languages.

Theorem A language L is decidable if and only if it is recognisable and its complement is also recognisable.

Proof $\implies$ The proof is considered trivial and left as an exercise to the reader.

Proof $\impliedby$ Let L and $\overline{L}$ be recognisable, and M_1, M_2 be their corresponding recognisers.
We construct a machine M which runs both M_1 and M_2 in parallel (we alternate between running 1 step of M_1 and one step of M_2),

and stop as soon as M_1 or M_2 accepts w. If it is M_1 which accepted, then we output accept; if it is M_2, we output reject.
We notice that M must always halt: $w \in L$ or $w \in \overline{L}$, so one of the two machines must halt and output accept. Thus, M decides L.

Corollary The language $\overline{HALT}$ is unrecognisable.

Proof We know that $HALT$ is undecidable but recognisable.
If $\overline{HALT}$ was recognisable, then it would mean that $HALT$ would be decidable by our theorem above, which is our contradiction.
□

Monday 27th March 2023 — **Lecture 6 : Reductions**

4.3 Reductions

Goal The goal of this section is to make a more abstract-type of proofs, more systematic way, to show that a language is not recognisable or not decidable.
The idea is to compare the difficulty of languages.

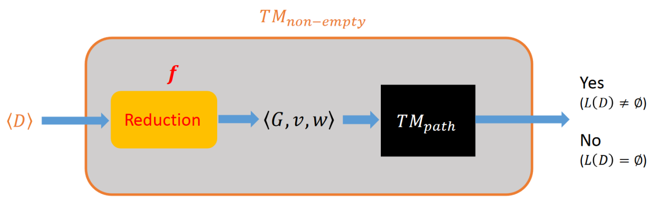
Example Let us consider the language where, given the encoding of some DFA, we output whether it accepts a non-empty language:

$$NE_{DFA} = \{\langle D\rangle \mid L(D) \neq \emptyset\}$$

We can also consider the language such that, given the encoding of some graph and two of its vertices, we output whether there exists a path between those two in the graph:

$$L_{path} = \{\langle G, v, w\rangle \mid v, w \in V(G) \text{ and there exists some path from } v \text{ to } w \text{ in } G\}$$

We notice that solving this second problem allows to solve the first too. If there is a path from the initial state to an accepting state, then the language of the DFA is non-empty. We have done a reduction: we transformed our problem into another one using some function $f(\langle D\rangle) = \langle G, v, w\rangle$, which is easier to solve.



Definition: Computable function A function $f : \Sigma^* \mapsto \Sigma^*$ is a **computable function** if there exists some TM M such that, on every input w, halts with just $f(w)$ on its tape.

Definition: Mapping reducible Language A is **mapping reducible** to language B, written $A \leq_m B$, if there exists some computable function $f : \Sigma^* \mapsto \Sigma^*$ named **reduction** such that, for every $w \in \Sigma^*$:

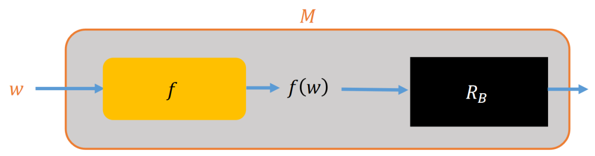
$$w \in A \iff f(w) \in B$$

In other word, f does not need to be bijective, but any element in A must land in B, and any element in $\overline{A}$ must end in $\overline{B}$.

Theorem If $A \leq_m B$ and B is decidable, then A is decidable.

Proof We know that B is decidable, thus let R_B be a decider for it. Also, let f be a reduction from A to B.

We can then define a Turing Machine M such that, it takes any input w, feed $f(w)$ to R_B, and output whatever R_B outputs.



However, by the definition of reduction:

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

This is equivalent to:

$$w \notin A \iff f(w) \notin B$$

Thus, we indeed have that M is a decider for A: it accepts all words inside A and rejects all inside $\overline{A}$.

□

Corollary If $A \leq_m B$ and A is undecidable, then B is undecidable.

Example 1 The following reduction is possible:

$$A_{TM} \leq_m HALT$$

And thus, $HALT$ is undecidable.

Proof Our goal is to find some reduction f from A_{TM} to $HALT$.
Given some input $x = \langle M, w\rangle$, our reduction outputs (without running the machine we construct) $f(x) = \langle M', w\rangle$, where M' receives some input y, and runs it on M. If M rejects y, M' enter an infinite loop; and if M accepts y, then M' accepts y. We can indeed show that this f is computable.
We constructed M' such that it halts on w if and only if M accepts w. Thus, we showed that:

$$\langle M, w\rangle \in A_{TM} \iff f(\langle M, w\rangle) = \langle M', w\rangle \in HALT$$

We have found a reduction, which ends our proof.

□

Example 2 We consider the following language

$$REG_{TM} = \{\langle N\rangle \mid L(N) \text{ is regular}\}$$

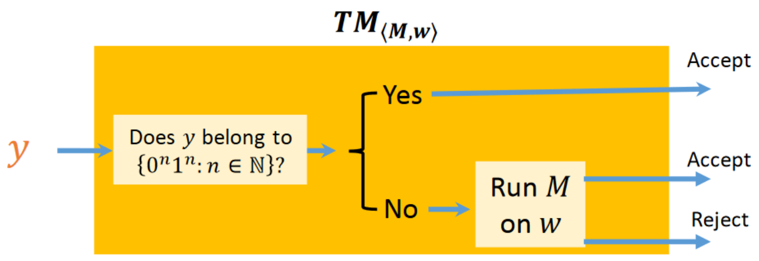
We have that:

$$A_{TM} \leq_m REG_{TM}$$

And thus, REG_{TM} is undecidable.

Proof Our goal is to map any (M, w) pair to a regular language if M accepts w and to a non-regular language is M rejects or does not halt on w.
The idea is to create a Turing Machine M' (without simulating it) using some kind of switch. First, if $y \in \{0^n 1^n \mid n \in \mathbb{N}\}$, the machine will directly accept it. If not, this is where the switch appears: the machine M' runs M on w and, depending on the result, accepts y or not. If M accepts w, then M' accepts y; meaning that, overall, it accepts any y. If M rejects w, then M' rejects y; meaning that overall it only accepts $\{0^n 1^n \mid n \in \mathbb{N}\}$. If M loops on w then, well,

M' loops on w and thus does not accept it; meaning that, overall, it also only accepts $\{0^n 1^n \mid n \in \mathbb{N}\}$.
The machine we constructed (but did not run, it is the job of the machine for REG_{TM} to analyse it without halting), M', can be represented as:


$TM_{\langle M,w \rangle}$
y
Does y belong to $\{0^n 1^n : n \in \mathbb{N}\}$?
Yes
No
Run M on w
Accept
Accept
Reject

We have thus constructed a Turing Machine which language depends on the acceptance of w by M: if M accepts w, then M' has a regular language but, if M halts or rejects w, then M' does not have a regular language. Thus:

$$\langle M, w \rangle \in A_{TM} \iff f(\langle M, w \rangle) = \langle M' \rangle \in REG_{TM}$$

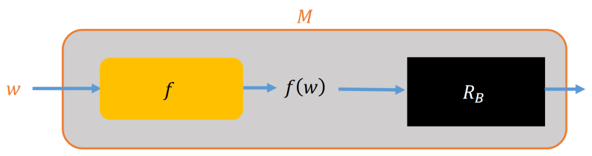
This is our reduction, finishing our proof.

□

Theorem If $A \leq_m B$ and B is recognisable, then A is recognisable.

Contrapositive If $A \leq_m B$ and A is unrecognisable, then B is unrecognisable.

Proof We can, as usual, consider the following machine:



We know that M accepting w is equivalent to R_B accepting $f(w)$, by creation of our machine. Then, this is equivalent to $f(w) \in B$ because R_B is a recogniser for B. This is finally equivalent to $w \in A$ since $A \leq_m B$.

$$M \text{ accepts } w \iff R_B \text{ accepts } f(w) \iff f(w) \in B \iff w \in A$$

This thus shows that M is a recogniser for A.

□

Example Let us consider the following language, which states whether two Turing Machine decide the same language.:

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are Turing Machines such that } L(M_1) = L(M_2)\}$$

Then:

$$\overline{A_{TM}} \leq_m EQ_{TM}$$

Thus, EQ_{TM} is unrecognisable.

Proof We define a reduction f such that, given an input $x = \langle M, w \rangle$, it outputs $f(x) = \langle M_1, M_2 \rangle$. The goal is that both machines have the same language if and only if $\langle M, w \rangle \in A_{TM}$. Thus, we construct M_1 to ignore its input and just runs M on w, and accepts if M accepts or enter an infinite loop otherwise. M_2 rejects any input y.

We notice that if $\langle M, w\rangle \in \overline{A_{TM}}$, then M_1 loops on all inputs and thus:

$$L(M_1) = \emptyset = L(M_2) \implies f(\langle M, w\rangle) = \langle M_1, M_2\rangle \in EQ_{TM}$$

If $\langle M, w\rangle \notin \overline{A_{TM}}$, then M accepts w and thus M_1 accepts every string. This means that:

$$L(M_1) = \Sigma^* \neq \emptyset = L(M_2) \implies \langle M_1, M_2\rangle \notin EQ_{TM}$$

We have indeed show that f is a reduction, and thus that $\overline{A_{TM}} \leq_m EQ_{TM}$.

□

Chapter 5

Complexity classes

5.1 P complexity class

Observation We have already noticed that having a decider for some language is better than a recogniser. Now, amongst deciders, we would like to distinguish which is better.

Definition: Running time Let M be a Turing Machine that halts on all inputs (in other words, it is a decider). The **running time** (or time complexity) of M is the function $t : \mathbb{N} \mapsto \mathbb{N}$ where:

$$t(n) = \max_{w \in \Sigma^* \mid |w| = n} \text{number of steps } M \text{ takes on } w$$

Intuition Intuitively, we consider the worst case for all inputs of size 1, then the worst case for all inputs of size 2, and so on.

Asymptotic growth We may do this for two Turing Machine and get two functions, where one is sometimes smaller than the other, and the other is sometimes smaller. We need a way to compare them , so we will only consider asymptotic growth.

Definition: Big-O Let $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$. We say that $f(n) = O(g(n))$ if $\exists C > 0$ and $\exists n_0 \in \mathbb{N}$ such that:

$$\forall n \geq n_0, f(n) \leq Cg(n)$$

Intuition This means that, asymptotically, f grows at most as fast as g.

Example For instance, we have:

$$5n^3 + 1 = O(2^n)$$

$$5n^3 + 1 \neq O(20n + 5)$$

Definition: Small-o Let $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$. We say $f(n) = o(g(n))$ if, for all $c > 0$, $\exists n_0 \in \mathbb{N}$ such that:

$$\forall n \geq n_0, f(n) < cg(n)$$

Intuition This is like saying that, asymptotically, f grows strictly slower than g.

Example For instance, we have:

$$\sqrt{n} = o(n)$$

$$f(n) \neq o(f(n))$$

Definition: Time complexity class We define **complexity classes** as:

$$\text{TIME}(t(n)) = \{L \subset \Sigma^* \mid L \text{ is decided by a TM with running time } O(t(n))\}$$

Property We notice that:

$$\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \ldots \subseteq \text{TIME}(2^n) \subseteq \ldots$$

Also, we notice that we have:

$$\text{REGULAR} \subseteq \text{TIME}(n)$$

Definition: Complexity class P **P** is the class of languages that are decidable in polynomial time on a deterministic Turing machine. In other words:

$$\text{P} = \bigcup_{k=1}^{\infty} \text{TIME}(n^k)$$

Note that this definition is robust to different model of computations. Thus, describing an algorithm working on a Turing Machine, or making one in C++, are equivalent.

Example For instance, sorting an array and traversing a graph using BFS are algorithms in the class P.

Extended Church-Turing thesis The extended Church-Turing thesis states that P corresponds to the class of problems that are "realistically" solvable in our universe. However, this claim is a little controversial since randomisation and quantum may help solving problems faster than anything doable on a Turing Machine. However, we don't know if this is really true, we just found some algorithms running faster on a quantum computer, though we did not prove that regular computers could not have any faster algorithms.

5.2 NP complexity class

SAT-verify and SAT We want to define some language SAT-verify and some harder language SAT. For it, we need to first define some other objects.

CNF A CNF (conjunctive normal form) is an expression with sub-expression composed of OR and NOT, which are composed using AND. For instance:

$$\varphi_1 = (\overline{x} \lor \overline{y} \lor z_0) \land (x \lor \overline{y} \lor z_1) \land (\overline{x} \lor y \lor z_2) \land (x \lor y \lor z_3)$$

$$\varphi_2 = \overline{x}_1 \land (x_1 \lor \overline{x}_2) \land (x_1 \lor x_2 \lor \overline{x}_3) \land (x_1 \lor x_2 \lor x_3 \lor \overline{x}_4)$$

$$\varphi_3 = \overline{x}_1 \land (x_1 \lor \overline{x}_2) \land (x_1 \lor x_2)$$

Satisfiable sentence A satisfying assignment is a set of variables which make the variable true. A sentence is satisfiable if there exists at least one satisfying assignments. For instance, φ_1 has 32 satisfying assignments, φ_2 has only one and φ_3 has zero.

SAT-verify We define the following language:

$$\text{SAT-verify} = \{\langle \varphi, C \rangle \mid C \text{ is a satisfying assignment of } \varphi\}$$

It is clearly in P since we can just substitute for the literals according to C, and see if the result is true.

SAT We now define the following, harder, language:

$$\begin{aligned}\text{SAT} &= \left\{\left\langle \varphi\right\rangle \mid \varphi \text{ is satisfiable}\right\} \\ &= \left\{\left\langle \varphi\right\rangle \mid \exists C \text{ such that } \left\langle \varphi, C\right\rangle \in \text{SAT-verify}\right\}\end{aligned}$$

We notice that we can iterate over all possible assignments C, and see if $\left\langle \varphi, C\right\rangle \in$ SAT-verify. However, this runs in exponential time. There are lots of much more complicated algorithms which are a bit better, but they all run in expected exponential time.

GI-verify and GI We want to define some languages GI-verify and GI. We will need to define some objects first.

Graph isomorphism A graph isomorphism is a bijection $f : V(G_1) \mapsto V(G_2)$ which preserves adjacency:

$$\left\{u, v\right\} \in E(G_1) \iff \left\{f(u), f(v)\right\} \in E(G_2)$$

Two graphs are isomorphic if they have at least one graph ismorphism. In other words, they are isomorphic if we can just re-label the vertices, to get one another.

GI-verify We define the following language:

$$\begin{aligned}&\text{GI-verify} \\ &= \left\{\left\langle G_1, G_2, C\right\rangle \mid C : V(G_1) \mapsto V(G_2) \text{ is a graph ismorophism}\right\}\end{aligned}$$

We notice that GI-verify is P. Indeed, we only need to iterate over the edges (which is quadratic in the number of vertices) and check that:

$$\left\{u, v\right\} \in E(G_1) \iff \left\{C(u), C(v)\right\} \in E(G_2)$$

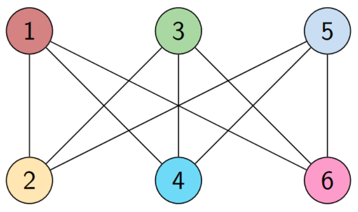
GI Now, let's consider the following language:

$$\begin{aligned}\text{GI} &= \left\{\left\langle G_1, G_2\right\rangle \mid G_1 \text{ and } G_2 \text{ are isomorphic}\right\} \\ &= \left\{\left\langle G_1, G_2\right\rangle \mid \exists C \text{ such that } \left\langle G_1, G_2, C\right\rangle \in \text{GI-verify}\right\}\end{aligned}$$

Again, this problem is very hard to solve, and we don't have any solution in polynomial time. However, the naive way, to try all the possible isomorphisms, is not the best asymptotic one known so far. The best known runs in quasi-polynomial time, $n^{O(\log(n))}$, but this is still not polynomial.

INDSET-verify and INDSET

Independent set In a graph, an independent set is a subset $S \subseteq V(G)$ such that no two vertices in S are adjacent in G.
For instance, in the following graph, $\left\{1, 3, 5\right\}$, $\left\{6\right\}$ and $\varnothing$ are indepedent sets.



INDSET-verify Let us consider the following language:

$$\begin{aligned}&\text{INDSET-verify} \\ &= \left\{\left\langle G, k, C\right\rangle \mid C \text{ is an independent set of size } k \text{ in } G\right\}\end{aligned}$$

This can be checked in polynomial time. Indeed, we can first check that $|C| = k$. Then, we loop for each $u, v \in C$, and we verify that:

$$\{u, v\} \notin E(G)$$

INDSET Let us now define the following, more complicated language:

$$\begin{aligned}\text{INDSET} &= \{\langle G, k\rangle \mid G \text{ has an independent set of size } k\} \\ &= \{\langle G, k\rangle \mid \exists C \text{ such that } \langle G, k, C\rangle \in \text{INDSET-verify}\}\end{aligned}$$

Again, the naive solution of looping over all possible inputs of size k takes exponential time: $\binom{n}{k} = n^{O(k)}$ (which is not polynomial since k is not constant; we might have $k = \frac{n}{2}$). As for the two other problems, we do not know any solution solving this in polynomial time.

Definition: Verifier A **verifier** for a language L is a TM M such that for each $x \in \Sigma^*$:

$$x \in L \implies \exists C \text{ such that } M \text{ accepts } \langle x, C\rangle$$

$$x \notin L \implies \forall C,\ M \text{ rejects } \langle x, C\rangle$$

We call those C a **certificate** (or witness).

Definition: Polynomial time verifier A **polynomial time verifier** (poly-time verifier) is a verifier which has a running time in x on any $\langle x, C\rangle$ is polynomial in x.

Remark This allows to know that $|C|$ is polynomial in $|x|$, since we will need to read the whole certificate and it must thus not be longer than the time we are allowed to take.

Observation We can turn any verifiable language to a decidable language, by iterating over all the possible inputs. However, for a polynomial verifiable language, this gives a decider running in exponential time.

Definition: NP **NP** is the class of languages that have polynomial-time verifiers.

Observations We trivially notice that:

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

Indeed, we can make a verifier which just uses the decider, and ignores the given certificate. It indeed runs in polynomial time.
Now, we don't know if $\text{P} = \text{NP}$. This is actually one of the 7 millennium problems, showing its difficulty. However, most researchers actually think that $\text{P} \neq \text{NP}$, since, otherwise, it would for instance be very easy to prove theorems (since we can verify that a proof is correct in polynomial time).

Monday 24th April 2023 — **Lecture 8 : Turning satisfying assignments to graphs**

Proposition: SAT SAT is in NP.

Proof We construct a verifier for SAT: given some input $\langle \varphi, C\rangle$, we interpret C as a truth assignment to the variables of φ, and check if our sentence indeed reduces to true. We can thus indeed write our language as:

$$\text{SAT} = \{\langle \varphi\rangle \mid \exists C \text{ such that the above verifier accepts } \langle \varphi, C\rangle\}$$

Proposition: GI GI is in NP.

Proof We construct a verifier for GI: given some input $\langle G_1, G_2, C\rangle$, we interpret C as function $V(G_1) \mapsto V(G_2)$ and check that it is both a bijection and preserves adjacency. This is a poly-time verifier, indeed showing this is in NP.

Proposition: INDSET INDSET is in NP.

Proof We construct a verifier for the language INDSET: on input $\langle G, k, C\rangle$, we interpret C as a subset $C \subseteq V(G)$, check that $|C| = k$ and check that, for all $u, v \in C$, we have $\{u, v\} \notin E(G)$.
Since this verifier runs in polynomial time, we have indeed shown INDSET is in NP.

Definition: Nondeterministic Turing Machines In a regular Turing machine, the transition function is $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$. In a Nondeterministic Turing Machine (NTM), we instead let it to be:

$$\delta : Q \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

We thus allow several transitions for any given state and tape symbol, just as for NFAs.

Observation Those are not really computable efficiently, we would require an exponential amount of parallel threads.

Multithreading Note that NTMs are weaker than multithreaded computers: threads cannot communicate between each other. This is more some kind of “quantum parallel worlds”.

Definition: Nondeterministic decider A **nondeterministic decider** for language L is an NTM N such that for each $x \in \Sigma^*$, every computation of N on x halts. Moreover, if $x \in L$, then some threads of N on x accepts and, if $x \notin L$, then every computation of N on x rejects.

Definition: Polynomial time NTM An NTM is a **polynomial time NTM** if the running time of its longest computation on x is polynomial in $|x|$.

Theorem Let $L \subseteq \Sigma^*$ be an arbitrary language.
L has a nondeterministic poly-time decider if and only if L has a poly-time verifier.

Proof $\Longrightarrow$ We know by hypothesis that there exists some nondeterministic polynomial-time decider N for L
We construct a verifier V for L. It interprets the certificate as a set of choices for which thread to choose. In other words, on input $\langle x, C\rangle$, it simulates $N(x)$ with nondeterministic choices given by C.
We know that there exists a path in our decision tree which has a polynomial length by definition of polynomial time NTM. Thus, we know this path can be encoded by a certificate with polynomial length.

Proof $\Longleftarrow$ We know by hypothesis that there exists a verifier V for L.
We construct a nondeterministic poly-timer decider. Using nondeterminism, we can try all certificates in parallel. In other words, at each step, we split into three threads: one which appends a 0 to the current certificate, one that appends a 1, and one that uses the current certificate (which runs V on $\langle x, C\rangle$). If there exists a certificate which works, we will find it in polynomial time. This gives us an algorithm which runs in polynomial time for each thread, which is thus a polynomial-time NTM.

□

Definition: NTIME complexity class We define the NTIME complexity class as:

$$\text{NTIME}(t(n)) = \{L \mid L \text{ has a nondeterministic } O(t(n)) \text{ time decider}\}$$

Equivalent definition: NP Class **NP** can be equivalently defined as:

$$\text{NP} = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k)$$

Remark This explains why NP is called nondeterministic P: a language in NP has a nondeterministic polynomial-time decider.

5.3 Polynomial-time reductions

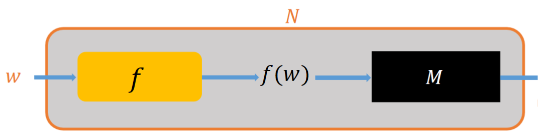
Definition: Poly-time computable function A function $f : \Sigma^* \mapsto \Sigma^*$ is a **poly-time computable function** if there exists some poly-time TM M such that, on every input w, it halts with just $f(w)$ on its tape.

Definition: Poly-time reduction A language A is **poly-time mapping reducible** to language B, written $A \leq_P B$, if there exists a poly-time computable function $f : \Sigma^* \mapsto \Sigma^*$ such that, for every $w \in \Sigma^*$:

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

Theorem If $A \leq_P B$ and B is in P, then A is in P.

Proof We assume by hypothesis that M is an $O(n^p)$-time decider for B, and f is an $O(n^q)$-time reduction from A to B.
We can then construct a TM N as:



In other words, on input w, it computes $y = f(w)$, runs M on y and outputs whatever M outputs. f outputs something which is at most its running time, so $|y| = O(|w|^q)$. Thus, our machine runs in $O(|w|^{pq})$.
We have indeed construct a poly-time TM for our problem A, showing it is in P.

□

Intuition This is another instance of the intuition behind reductions: if A reduces to B and B is easy, then A is easy.

Corollary If $A \leq_P B$ and A is not in P, then B is not in P.

Theorem: Transistivity If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof The proof is completely analogous to the one we have just done: we can show that running f_{AB} and then f_{BC} runs in $O(|w|^{pq})$.

Definition: NP-hardness A language L is said to be **NP-hard** if every language L' in NP is such that:

$$L' \leq_P L$$

Definition: NP-completeness A language L is said to be **NP-complete** if L is in NP and L is NP-hard.

Observation If any of the NP-complete language has a polynomial time decider, then every language in NP have a polynomial time decider and thus $P = NP$. Thus, NP-complete problems are the hardest NP problems. In other words, if we show that a problem is NP-complete, we showed that, even if we are not able to find an efficient algorithm, neither 40 years of research could.

Cook-Levin Theorem SAT is NP-complete.

Remark In practice, from now on, to show that a language is NP-complete we will always follow the same recipe.
We first need to show that L belongs to NP, and thus we need to give a poly-time verifier for L. Then, for NP-hardness, we only need to show $H \leq_P L$ for some NP-complete language H. Indeed, since we know that $\forall L' \in NP$ we have $L' \leq_P H$ and we show $H \leq_P L$, we indeed get that $L' \leq_P L$.

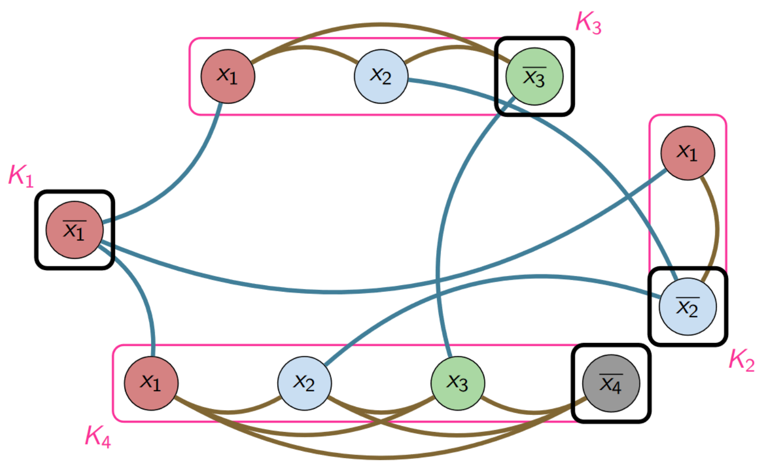
Proposition: INDSET We can make the following reduction:

$$\text{SAT} \leq_P \text{INDSET}$$

Since INDSET is in NP, this implies that INDSET is NP-complete.

Proof We are given as input a CNF formula φ, and we want to turn it into a graph for INDSET in polynomial time.
The idea is to construct a vertex per literal per clause, grouping the literals within a clause. We then link with an edge every vertex within a group (in brown) and every conflicting literals (so x and $\overline{x}$; in blue). For instance, we could have:

$$\varphi = \underbrace{\overline{x_1}}_{K_1} \land \underbrace{(x_1 \lor \overline{x_2})}_{K_2} \land \underbrace{(x_1 \lor x_2 \lor \overline{x_3})}_{K_3} \land \underbrace{(x_1 \lor x_2 \lor x_3 \lor \overline{x_4})}_{K_4}$$



Our reduction outputs $f(\varphi) = (G, m)$, where m is the number of clauses ($m = 4$ in our example, since we have $K_1, \ldots, K_4$). This can definitely be done in polynomial time, so we now want to show that:

$$\varphi \in \text{SAT} \iff f(\varphi) \in \text{INDSET}$$

Let's suppose that $\varphi \in$ SAT, and thus that there exists some satisfying assignment C of φ. In C, there is at least a true literal for each clause. For each group, we can take one of those literals, and construct an independent set of size m. Note that, because of the construction of $f(\varphi)$, two vertices in this independent set cannot be linked by an edge. Indeed, we took a single variable per clause

(therefore there cannot be any brown edge), and we cannot have both x_i and $\overline{x}_i$ true (therefore there cannot be a blue edge either). We have thus indeed shown that $\varphi \in \text{SAT} \implies f(\varphi) \in \text{INDSET}$
Let's suppose that $f(\varphi) \in \text{INDSET}$, meaning that there exists an independent set C in G of size $|C| = m$. Again, by construction, C contains a vertex from each group. We can thus set the corresponding literal to true, obtaining a satisfiable assignment for φ. Indeed, this allows to have one literal set to true by group, and no contradiction (it is impossible to need $x_i = \overline{x}_i = T$, since there would be a blue edge between our elements and thus it would not be an independent set to start with). We have also shown that $f(\varphi) \in \text{INDSET} \implies \varphi \in \text{SAT}$, finishing our proof.

□

Remark This proof is really beautiful: we managed to make a link between independent sets and satisfiable CNF formulas.

Monday 1st May 2023 — Lecture 9 : I hope the rest of the course will not be like this

Definition: kSAT We define kSAT as:

$$k\text{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is satisfiable and each clause of } \varphi \text{ contains at most } k \text{ literals}\}$$

Remark We notice that kSAT is in NP for any k, since we can use the exact same verifier as for SAT.

Theorem: kSAT Let $k \geq 3$. We can make the following reduction:

$$\text{SAT} \leq_P k\text{SAT}$$

Since kSAT is in NP for all k, it is NP-complete for $k \geq 3$.

Proof We do the proof for $k = 3$, a general $k \geq 3$ is completely analogous. We have some input φ, and we want to turn it into some input for 3SAT.
Thus, let us say that φ has a clause with too many variables, for instance:

$$K = (\ell_1 \lor \ell_2 \lor \ldots \lor \ell_m), \quad m > 3$$

We can replace this clause by the two following clauses, introducing a new variable z:

$$K_1 = (\ell_1 \lor \ell_2 \lor z), \quad K_2 = (\overline{z} \lor \ell_3 \lor \ldots \lor \ell_m)$$

We notice replacing K by $K_1 \land K_2$ we keep a CNF, while preserving satisfiability. If K is satisfiable, we know that there is at least a satisfying assignment in ℓ_1, ℓ_2 or in $\ell_3, \ldots, \ell_m$; both cannot have no satisfying assignment at the same time. If both ℓ_1 and ℓ_2 are false, we can set $z = T$; if all $\ell_3, \ldots, \ell_m$ are false we can set $\overline{z} = T$. On the other hand, if $K_1 \land K_2$ is satisfiable, it means that there must be a true literal in the clause where z or $\overline{z}$ is false, which allows K to be satisfied.
We can do this iteratively, until all our clauses have at most 3 literals. Each clause needs to be iterated over at most a linear number of times, meaning that our reduction can indeed be done in polynomial time. Satisfiability is preserved throughout the transformation, finishing our reduction.

□

Remark This proof cannot be done for $k = 2$. Indeed, let's say that we have:

$$K = (x_1 \lor x_2 \lor x_3)$$

Now, we try doing our transformation on it, giving:

$$K_1 = (x_1 \lor z_1), \quad K_2 = (\overline{z}_1 \lor x_2 \lor x_3)$$

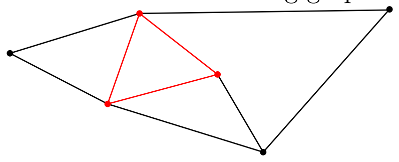
We have not reduced the number of literals in K_2, so our proof cannot be applied. In fact, 2SAT is in P (this proof is non-trivial), and we don't expect an NP-complete problem to reduce to a P problem (since it would imply P = NP).

Definition: *k*-clique Let G be a graph. A k-clique is a subset of k vertices which are all pairwise connected.

Definition: CLIQUE We define the language CLIQUE as:

$$\text{CLIQUE} = \{\langle G, k\rangle \mid G \text{ has a clique of size } k\}$$

Example For instance, let us consider the following graph:



Then, $\langle G, 3\rangle \in$ CLIQUE (a 3-clique is highlighted in red) but $\langle G, 4\rangle \notin$ CLIQUE.

Remark We can easily show that CLIQUE is in NP: we consider the certificate as a subset of vertices, and we verify that it is indeed a clique (which only requires $\Theta\left(k^2\right)$ operations).

Definition: Complement of graph Let G be a graph. The complement of G is:

$$\overline{G} = \left(V, \overline{E}\right)$$

where $\overline{E}$ is the complement of E, meaning:

$$(u, v) \in E \iff (u, v) \notin \overline{E}$$

Theorem We can make the following reduction:

$$\text{INDSET} \leq_P \text{CLIQUE}$$

Since CLIQUE is in NP, it means that CLIQUE is NP-complete.

Proof We notice that a clique is the some kind of opposite to an independent set: all vertices are pairwise adjacent, instead of pairwise non-adjacent. Thus, let us consider the complement of G. We notice that, if we have some graph $G = (V, E)$, then a subset $S \subseteq V$ is an independent set of G if and only if S is a clique of $\overline{G}$.
Thus, we can take the following reduction:

$$f(\langle G, k\rangle) = \left\langle \overline{G}, k\right\rangle$$

By our observation, this indeed has the reduction property, finishing our proof.

□

Recall: Vertex incidence Let $G = (V, E)$ be a graph. An edge $(u_1, u_2) \in E$ is incident to a vertex $v \in V$ if the vertex is an endpoint of the edge, meaning that:

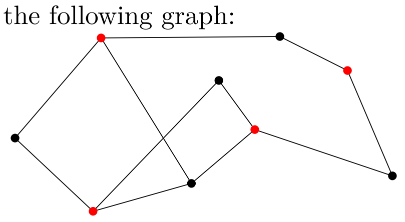
$$v = u_1 \quad \text{or} \quad v = u_2$$

Definition: Vertex cover Let $G = (V, E)$ be a graph. A **vertex cover** is a subset $S \subseteq V$ such that every edge of G is incident to at least one vertex in S.

Definition: VERTEX COVER language We define the following language:

$$\text{VERTEX COVER} = \{\langle G, k\rangle \mid G \text{ is a graph that has a vertex cover of size } k\}$$

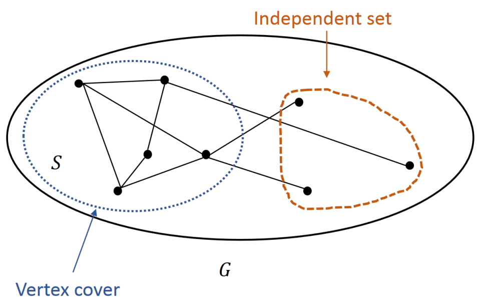
Example Let us consider the following graph:



We notice that we have $\langle G, 4\rangle \in$ VERTEX COVER, the highlighted vertices are an example. However, $\langle G, 3\rangle \notin$ VERTEX COVER. Indeed, we notice that a vertex has at most 3 incident edges in this graph. Thus, taking only 3 vertices gives at most 9 covered edges, but the graph has a total of 11 edges, so this is a contradiction.

Remark We can notice rather trivially that this language is in NP: we can just consider the certificate as S, and see if this is indeed a vertex cover.

Lemma Let G be a graph.
$S \subseteq V$ is a vertex cover if and only if $\overline{S} = V \setminus S$ is an independent set.



Proof $\implies$ We do this proof by the contrapositive.
If $\overline{S}$ is not an independent set, it means that there is an edge linking two of its vertices. However, this edge is not covered by a vertex of S and thus S is not a vertex cover.

Proof $\impliedby$ If $\overline{S}$ is an independent set, it means that none of its vertices are linked by an edge, and thus S is sufficient to cover all the edges.

□

Corollary Let G be a graph, and $k \in \mathbb{N}$.
G has an independent set of size k if and only if G has a vertex cover of size $n - k$.

Theorem We have the following reduction:

$$\text{INDSET} \leq_p \text{VERTEX COVER}$$

Since VERTEX COVER is in NP, it implies that it is NP-complete.

Proof We can pick the following reduction:

$$f(\langle G, k\rangle) = \langle G, n-k\rangle$$

This reduction is indeed computable in polynomial time, and works thanks to our corollary.

□

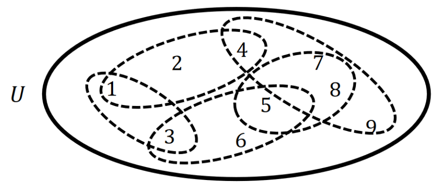
Definition: Set cover Let $U = \{1, \ldots, n\}$ and $\mathcal{F} = \{T_1, \ldots, T_m\}$ be a family of subsets (meaning that $T_i \subseteq U$ for all i).
A subset $\langle T_{i_1}, \ldots, T_{i_k}\rangle \subseteq \mathcal{F}$ is called a **set cover** of size k if:

$$\bigcup_{j=1}^{k} T_{i_j} = U$$

Definition: SET COVER language We define the following language:

$$\text{SET COVER} = \{\langle U, \mathcal{F}, k\rangle \mid \mathcal{F} \text{ contains a set cover of } U \text{ of size } k\}$$

Example Let us consider the following U and $\mathcal{F}$:



We notice that $\langle U, \mathcal{F}, 3\rangle \in \text{SET COVER}$ since we can take:

$$\{\{1, 2, 4\}, \{3, 6, 5\}, \{4, 7, 8, 9\}\}$$

However, $\langle U, \mathcal{F}, 2\rangle \notin \text{SET COVER}$. Indeed, we need a T_i containing 2, one containing 6 and one containing 9 (since they are each in an different set). We thus need at least 3 sets.

NP We notice that this language is in NP: it is easy to verify that a given partition indeed works.

Theorem We can make the following reduction:

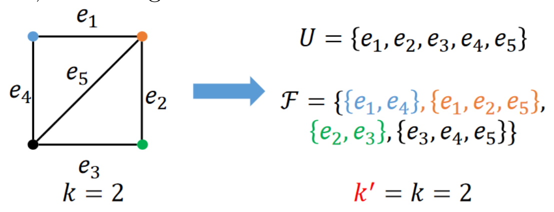
$$\text{VERTEX COVER} \leq_P \text{SET COVER}$$

Since SET COVER is in NP, it implies that SET COVER is NP-complete.

Proof We notice that VERTEX COVER is a special case of SET COVER. The idea is to take $U = E$: we consider the edges as the element of our set, since we want to cover them. Then, to construct $\mathcal{F}$, we construct sets S_v, which contain the set of edges which are incident to the vertex $v \in V$. Then, we can just take:

$$\mathcal{F} = \{S_v \mid v \in V\}$$

For instance, it could give:

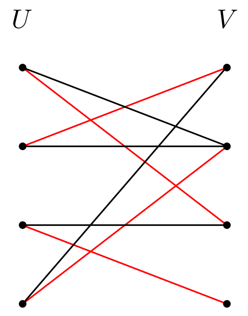

e_1
e_2
e_3
e_4
e_5
$k = 2$
$U = \{e_1, e_2, e_3, e_4, e_5\}$
$\mathcal{F} = \{\{e_1, e_4\}, \{e_1, e_2, e_5\}, \{e_2, e_3\}, \{e_3, e_4, e_5\}\}$
$k' = k = 2$

We notice that making a set cover is equivalent to making a vertex cover: each element of $\mathcal{F}$ represents a vertex, and thus making a set cover means that we found a set of vertices which covers the graph, and inversely. This reduction is also clearly computable in polynomial time.

$\square$

Monday 8th May 2023 — **Lecture 10 : More fun proofs**

Definition: Perfect matchings Let $G = (U \cup V, E)$, where $E \subseteq U \times V$, be a bipartite graph, meaning that there is some "left part" and some "right part" and edges can only go from left to right:



We call $M \subseteq E$ a **matching** if each $v \in U \cup V$ has at most one incident edge. It is moreover named a **perfect matching** (in red in the graph above) if each $v \in U \cup V$ has exactly one incident edge, which is equivalent to having M being a matching and $|M| = |U| = |V|$.

Definition: PERFECT-MATCHING We define the following language:

$$\text{PERFECT-MATCHING} = \{\langle G\rangle \mid G \text{ admits a perfect matching}\}$$

NP We notice that this problem is clearly in NP: if we are given some M, we can easily check that it is a matching and $|M| = |U| = |V|$ in polynomial time.

Complement We notice that the complement of PERFECT-MATCHING is also in NP.
We can define the neighbourhood of some $S \subseteq U$ as:

$$N(S) = \{v \in V \mid \exists u \in S, (u, v) \in E\}$$

In other words, it is the set of vertices reachable in V from S. Clearly, a very reasonable necessary condition for G not to admit any perfect matching is the existence of S such that if $|S| > |N(s)|$. However, Hall's theorem states that this is also a sufficient condition: G contains a perfect matching if and only if $S \leq |N(S)|$ for all $S \subseteq U$. So, we can use our certificate as a subset of U, and check whether Hall's condition is met. This is done in polynomial time, and indeed reaches the goals of certificates.
Note that this is an open question, but many people think that, if both a language and its complement are in NP, then the language is in P; we have always managed to show that languages with this property where in P so far. In fact, if we managed to prove that this is false, we would have shown that P $\neq$ NP since P is closed under complement.

Edmonds' theorem PERFECT-MATCHING is solvable efficiently:

$$\text{PERFECT-MATCHING} \in P$$

Proof We can for instance use max flows, as seen in the Algorithms course.

Definition: PERFECT-3-MATCHING Let $G = (U \cup V \cup W, E)$, where $E \subseteq U \times V \times W$, be a tripartie graph.
A **perfect 3-matching** is a subset $M \subseteq E$ where each $v \in U \cup V \cup W$ appears exactly once in M.
We then naturally define:

$$\text{PERFECT-3-MATCHING} = \{G \mid G \text{ admits a perfect 3-matching}\}$$

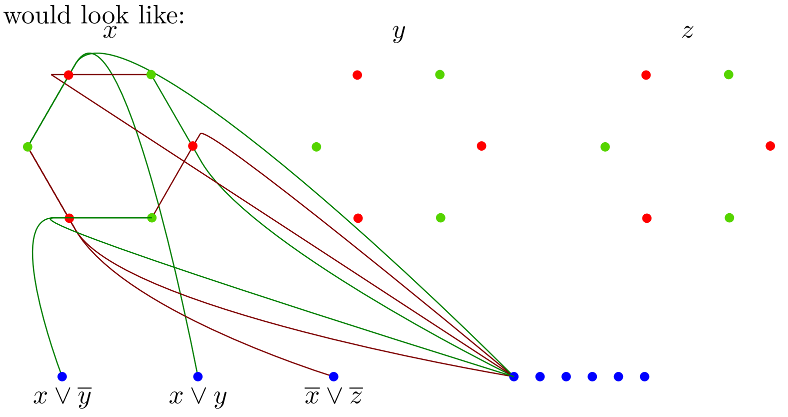
NP Again, this problem is definitely in NP.

Theorem We have the following reduction:

$$\text{SAT} \leq_p \text{PERFECT-3-MATCHING}$$

Since we know that PERFECT-3-MATCHING is in NP, it is also NP-complete.

Proof Let φ be a CNF with n clauses and m different variables.
The idea is to construct a graph using three types of vertices, with a total of $3mn$ vertices. The first n ones are in W, they each represent one of the clause. We then have $mn - n$ other vertices in W, which will be helper ones. We finally make m groups of $2n$ vertices: each group represents a different variable, and half of their vertices are in U, whereas the other half is in V.
The main thing to realise for this reduction is that we are able to link the vertices in each group using 2-edges from vertices of U to vertices of V so that there is only two choices for each variable (see the picture below, where vertices in green are in U, ones in red are in V and ones in blue are in W): in a given group, either we always go clockwise (choosing green edges), or always go counterclockwise (choosing red edges). If we decide to go once clockwise, and once counterclockwise, then it necessarily means that a vertex will not be reached, or that it will be reached more than once. That way, we can encode the choice of setting $x_i = T$ xor $x_i = F$.
Then, we can extend each 2-edge into multiple 3-edges, by happening some vertices from W. First, we use a different edge representing $x_i = T$ to link to each clause where we have x_i, and a different edge representing $x_i = F$ to each clause where we have $\overline{x}_i$. That way, to reach each of the vertices in W representing a clause, we need to make a (consistent) choice of variables to set to true. We then need to extend *every* 2-edge to reach every helper vertices in W, which allows variables not to appear in every clause.
A partial drawing of the graph representing $(x \vee \overline{y}) \wedge (x \vee y) \wedge (\overline{x} \vee \overline{z})$ would look like:



It is then possible to show that this reduction indeed has the reduction property:

$$\varphi \in \text{SAT} \iff f(\varphi) \in \text{PERFECT-3-MATCHING}$$

Definition: SUBSET-SUM Let X be a multiset of positive integers (meaning that elements can appear multiple times in it). We define the following language:

$$\text{SUBSET-SUM} = \{\langle X, s\rangle \mid X \text{ contains a subset whose elements sum to } s\}$$

Example For instance, let us consider:

$$X = \{1, 3, 4, 6, 13, 13\}$$

Then, we have $\langle X, 8\rangle \in$ SUBSET-SUM since we can take $T = \{1, 3, 4\}$. However, $\langle X, 12\rangle \notin$ SUBSET-SUM.

NP We can notice that this problem is in NP: if we are given a multiset of number, we can verify that it is indeed a subset and that its sum equals s in polynomial time.

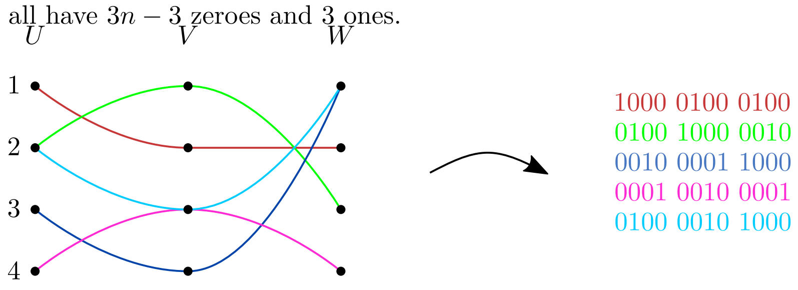
Remark We notice that we can encode the SUBSET-SUM problem in multiple ways: using binary (writing $101_2 = 5$) or unary (writing $11111_1 = 5$). Using n bits we can encode number up to 2^n in binary, whereas we can only encode numbers up to n in unary. This is important because we measure the speed of some algorithm as a function of the size of the input. For instance, encoding everything in SUBSET-SUM using unary, makes it go in P: we can make an algorithm which is polynomial in the value of the input. However, as we will se right after, if we encode everything using binary, SUBSET-SUM is in fact NP-complete: there probably does not exist any algorithm which running time is a polynomial of the number of binary digits of the input.
When we don't mention anything, binary encoding is naturally implicit.

Theorem We have the following reduction:

$$\text{PERFECT-3-MATCHING} \leq_p \text{SUBSET-SUM}$$

Since SUBSET-SUM is in NP, it implies that it is NP-complete.

Proof Let $n = |E|$ be the number of edges. The idea is to encode each 3-edge $e \in E \subseteq U \times V \times W$ as a $3n$-bit number in base $b = n + 1$, using only 0s and 1s.
Each vertex is represented by an n-digit number: it contains only zeroes except for a 1 at the i^{th} position. Since each edge goes through three vertices, we are concatenating three n-bits numbers, and we thus indeed get that each edges are $3n$-digits, and that they all have $3n - 3$ zeroes and 3 ones.



1000 0100 0100
0100 1000 0010
0010 0001 1000
0001 0010 0001
0100 0010 1000

Now, we can define the SUBSET-SUM target to be a $3n$-digits number containing only 1s. This models the fact that each node is picked exactly once.
Let's suppose that $x \in$ PERFECT-3-MATCHING. We know that there is a set of edges which hits each vertex exactly once. This means that, if we choose to sum their binary encoding, then there

will be exactly one 1 in each column and thus the sum will contain $3n$ ones. This indeed show that, then, $f(x) \in$ SUBSET-SUM.
Let's now suppose that $f(x) \in$ SUBSET-SUM. We may think that carry bits are a problem since, in binary, we would have $001 + 100 + 001 + 001 = 111$ even though we did not choose a set of edges which yields a unique 1 in each column. However, since we chose to work in base $|E| + 1$, there cannot be any carry when adding at most $|E|$ numbers which digits are all at most 1. This thus means that, indeed, we can interpret the result of SUBSET-SUM as a valid 3-matching, and thus $x \in$ PERFECT-3-MATCHING.

□

Observation We have seen the following reduction tree of NP-complete languages:

$$\begin{tikzcd}
 & & & \text{SAT} \arrow[dll] \arrow[d] \arrow[dr] & \\
 & \text{INDSET} \arrow[dl] \arrow[dr] & & k\text{SAT} & \text{PERFECT-3-MATCHING} \arrow[d] \\
\text{VERTEX-COVER} \arrow[d] & & \text{CLIQUE} & & \text{SUBSET-SUM} \\
\text{SET-COVER} & & & &
\end{tikzcd}$$

We can naturally use any of those NP-complete problems when we want to show another language is NP-complete.

Monday 15th May 2023 — **Lecture 11 : I thought this would never happen**

5.4 Cook-Levin theorem

Remark We have not proven the Cook-Levin problem; that SAT is NP-complete. We will finish this course by doing this.

Definition: WITNESS-EXISTENCE We define the following language:

$$\text{WITNESS-EXISTENCE} = \left\{\left\langle V, x, 1^t\right\rangle \mid \exists y, V(x, y) = 1 \text{ in } t \text{ steps}\right\}$$

where V is a verifier, x is an input to this TM, and 1^t is a running time t written in unary.

Observation This problem is definitely in NP: we can just consider the certificate to be y. This indeed runs in polynomial time in the size of the input, since it does one step per digit of 1^t. We notice that we absolutely need t to be written using unary, binary would not have allowed to do this computation in polynomial time.

Lemma: WITNESS-EXISTENCE WITNESS-EXISTENCE is NP-hard.
Since moreover it is in NP, it is NP-complete.

Proof Let $L \in \text{NP}$ be an arbitrary language in NP. We want to show that $L \leq_P$ WITNESS-EXISTENCE.
By definition of NP, there exists a polynomial-time verifier V for L. In other words:

$$L = \left\{x \mid \exists y \text{ such that } V(x, y) = 1 \text{ in time } |x|^k\right\}$$

Our reduction f is very trivial:

$$f(x) = \left\langle V, x, 1^{\left(|x|^k\right)}\right\rangle$$

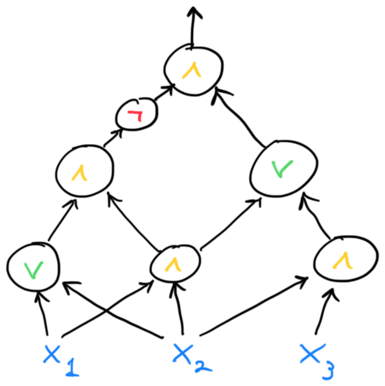
This can indeed be computed in polynomial time: we only need $|x|^k$ steps to write $1^{\left(|x|^k\right)}$, $|x|$ steps to write x and a constant time to write V. The rest of the proof directly comes from the definition of this reduction and of WITNESS-EXISTENCE.

□

Remark This problem is nice, but it is not very useful in this form. Let us thus introduce another concept to make it usable.

Observation Any computer is done using electronics circuits. Let us try to formalise this, and see where it can bring us.

Definition: Circuit A **circuit** is some acyclic directed graph which nodes are input variables (which do not have any incoming edge) or gates (which can only consist in $\vee$, $\wedge$ and $\neg$). The directed edges are wires. Finally, we have one or more output wires.



Remark The fact that we are asking an acyclic graph may seem problematic, since we sometimes use the output of a circuit as its own input. However, this can easily be bypassed, since we can just copy the circuit multiple times.

Definition: Size of a circuit The **size** of some circuit is its number of wires.

Proposition: DNF Let $f : \{0, 1\}^n \mapsto \{0, 1\}$ be an arbitrary boolean function.
Then, we can express f using a DNF of size $O(2^n)$.

Proof The idea is to just convert the truth table of f using a DNF:

$$f(x) = \bigvee_{y : f(y) = 1} \left(x \overset{?}{=} y\right) = \bigvee_{y : f(y) = 1} \bigwedge_{i=1}^{n} \begin{cases} x_i & \text{if } y_i = 1 \\ \overline{x}_i & \text{if } y_i = 0 \end{cases}$$

□

Proposition: CNF

Let $f : \{0,1\}^n \mapsto \{0,1\}$ be an arbitrary boolean function.
Then, we can express f using a CNF of size $O(2^n)$.

Proof We can just construct the DNF of $\neg f$, and then take its negation. Every $\wedge$ will be turned to an $\vee$, and inversely, thanks to Morgan's Laws. This indeed implies that we get a CNF.

Remark This is already great, but other circuits are often more expressive. We will in fact even show in exercises that there are some circuits which CNFs and DNFs are exponentially larger.

Theorem

Let M be some Turing machine, $n \in \mathbb{N}$ be an input length, and $t(n)$ be its runtime. Then, we can compute in $t(n)^{O(1)}$ (meaning in time polynomial in $t(n)$) a circuit C_n of size $O\left(t(n)^2\right)$ such that:

$$C_n(x) = M(x), \quad \forall x \in \{0,1\}^n$$

Proof sketch Let us consider a $t \times t$ table for M. It is constructed such that its ith row contains the configuration of the Turing machine at step i (recall that it encodes what is written on the tape, where the head is, and in what state the machine is in), including the input. In other words, the first row is an input to the circuit; the circuit then uses this input to propagate until the tth row, where we have the state of the machine which would run on this input at the tth step. Since the runtime is t, it means that the machine halted before iteration t. Moreover, it can only consider at most the tth first bits of input, since it considers a bit at a time. We can thus store everything important for the machine in this $t \times t$ table. Our goal is now to show that we can construct such a table efficiently using a circuit, which input variables are the n input squares in the first row.
We notice that any cell in the table is determined by 4 cells below (meaning in the previous iteration of the machine). Indeed, if we don't see the head in the four cells below, the cell cannot change. Else, the head can move in or out of this column, or modify it.
However, this means that we can always make a circuit which can use the bits of 4 cells to produce the bits of the next cell. Putting such a circuit everywhere, we get a constant amount of wires for each cells, meaning that we get a circuit of size $t^2 O(1) = O\left(t^2\right)$, as required.

Definition: CIRCUIT-SAT

We define the following language:

$$\text{CIRCUIT-SAT} = \left\{\langle C_n \rangle \mid \exists x \in \{0,1\}^n \text{ such that } C_n(x) = 1\right\}$$

where C_n is an arbitrary circuit.

NP We notice that this problem is definitely in NP: we can simply consider the certificate to be x.

Lemma: CIRCUIT-SAT

We have the following reduction:

$$\text{WITNESS-EXISTENCE} \leq_p \text{CIRCUIT-SAT}$$

Since WITNESS-EXISTENCE is NP-complete and CIRCUIT-SAT is in NP, this implies that CIRCUIT-SAT is NP-complete.

Proof Let $(V, x, 1^t)$ be an arbitrary input to WITNESS-EXISTENCE. The reduction makes the circuit C_V from V using an input length $|x|$ and simulating it for t steps. We hard-code x in the circuit, so that this is only a function of y, yielding:

$$f\left(V, x, 1^t\right) = C(y) = C_V(x, y)$$

where, by construction, $C_V(x, y) = V(x, y)$.
The rest of the reduction is considered trivial and left as an exercise to the reader.

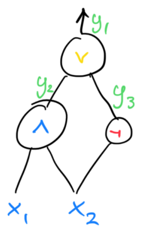
□

Lemma: SAT We have the following reduction:

$$\text{CIRCUIT-SAT} \leq_p \text{SAT}$$

Since CIRCUIT-SAT is NP-complete, it implies that SAT is NP-hard.

Proof Let C_n be an arbitrary circuit.
We notice that we are not allowed to convert the circuit to its CNF formula because the construction we have seen takes size $O(2^n)$.
The idea instead is to introduce new variables, one for which wire of C_n: $y_1, \ldots, y_m$. For instance, let us consider the following circuit:



We want to force $y_1 = x_2 \vee y_3$, which can be done by using an equivalence, $y_1 \leftrightarrow y_2 \vee y_3$. We can thus make our reduction by forcing every wire to match its definition, and adding the condition that the output wire must be true:

$$f(C_n) = y_1 \wedge (y_1 \leftrightarrow x_1 \wedge x_2) \wedge (y_3 \leftrightarrow \neg x_2) \wedge (y_2 \leftrightarrow y_1 \vee y_3)$$

We notice that we can make a CNF for any $y \leftrightarrow (x \wedge z)$, $y \leftrightarrow (x \wedge z)$ and $y \leftrightarrow \neg x$ in constant time. However, and-ing CNF formulas also yield a CNF formula. This means that we managed to turn our circuit into a CNF formula in linear time.
The rest of the reduction is considered trivial and left as an exercise to the reader.

□

Cook-Levin theorem SAT is NP-complete.

Proof We have already seen that SAT is in NP, and we have just shown that it is NP hard. Thus, it is indeed NP-complete. ☺

□