What is a computer?

A short computability intro [1].

Pablo Arrighi

U. Paris-Saclay

8 octobre 2021

1. Based on Pierre Wolpert's *Introduction à la calculabilité*, Dunod.

Why the question ?

- To understand the limits of computer science.
- To abstract away from the technology used to build computers, and become "hardware independent".
- To distinguish problems that are solvable by algorithms or not.
  - Solvable with all the time and space "in the world" $\rightsquigarrow$ computability.
  - Solvable efficiently $\rightsquigarrow$ complexity.

Computability

The concept of problem

Problem: *generic* question.

Examples :

- to sort an array of numbers;
- to decide if a program written in C stops for all possible input values; (halting problem) ;
- to decide if an equation with integer coefficients has integer solutions (Hilbert's 10th problem).

The concept of program

Effective procedure: program that can be executed by a computer.

Examples :

- Effective procedure : program written in JAVA ;
- Not an effective procedure: "to solve the halting problem, one must just check that the program has no infinite loops or recursive call sequences."

Formalizing problems

How could one represent problem instances?

Languages

Language: set of words defined over the same alphabet.

Examples

- $\{aab, aaaa, \varepsilon, a, b, abababababbbbbbbbbbbb\}$, $\{\varepsilon, aaaaaaa, a, bbbbbb\}$ and $\emptyset$ (the empty set) are languages over the alphabet $\{a, b\}$.
- for the alphabet $\{0, 1\}$,

  $\{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100,$
  $101, 110, 111, \ldots\}$ is the language containing all words.
- language $\emptyset \neq$ language $\{\varepsilon\}$.
- the set of words encoding C programs that always stop.

Representing problems

Encoding a problem

Let us consider a binary problem whose instances are encoded by words defined over an alphabet Σ. The set of all words defined on Σ can be partitioned in 2 subsets :

- *The positive instances* : the words encoding instances of the problem for which the answer is yes ;
- *The negative instances* : the words that do not encode and instance of the problem, or that encode an instance for which the answer is no.

$\hookrightarrow$ that defines a language.

Binary problems $\equiv$ Languages

Operations on languages

- $L_1 \cup L_2 = \{w | w \in L_1 \text{ or } w \in L_2\}$;
- $L_1 \cdot L_2 = \{w | w = xy, x \in L_1 \text{ and } y \in L_2\}$;
- $L_1^* = \{w | \exists k \geq 0 \text{ and } w_1, \ldots, w_k \in L_1 \text{ such that } w = w_1 w_2 \ldots w_k\}$;
- $\overline{L_1} = \{w | w \notin L_1\}$.

The grammar of regular expressions

Let us define a small "language to describe languages" called regular expressions. We proceed with the same three steps that are used to define a programming language.

Regular expressions are defined by their

1. *Lexicon*

$$\emptyset \ \varepsilon \ \Sigma \ \cup \ \cdot \ * \ (\ )$$

2. *Grammar*

$$e, e' ::= \emptyset \mid \varepsilon \mid \Sigma \mid e \cup e' \mid e \cdot e' \mid e*$$

3. *Semantics*

A language $L(e)$ obtained recursively as follows...

The language represented by a regular expression

1. $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$,
2. $L(a) = \{a\}$ for each $a \in \Sigma$,
3. $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$,
4. $L((\alpha\beta)) = L(\alpha) \cdot L(\beta)$,
5. $L((\alpha)*) = L(\alpha)^*$.

Regulars languages : examples

- The set of all words over $\Sigma = \{a_1, \ldots, a_n\}$ is represented by $(a_1 \cup \ldots \cup a_n)^*$ (or Σ^*).

- The set of all nonempty words over $\Sigma = \{a_1, \ldots, a_n\}$ is represented by $(a_1 \cup \ldots \cup a_n)(a_1 \cup \ldots \cup a_n)^*$ (or $\Sigma\Sigma^*$, or Σ^+).

- the expression $(a \cup b)^* a (a \cup b)^*$ represents the language containing all words over the alphabet $\{a, b\}$ that contain at least one "a".

Regulars languages : more examples

$$(a^*b)^* \cup (b^*a)^* = (a \cup b)^*$$

Proof

- $(a^*b)^* \cup (b^*a)^* \subset (a \cup b)^*$ since $(a \cup b)^*$ represents the set of all words built from the characters "a" and "b".

- Let us consider an arbitrary word

$$w = w_1 w_2 \dots w_n \in (a \cup b)^*.$$

One can distinguish 4 cases ...

1. $w = a^n$ and thus $w \subset (\varepsilon a)^* \subset (b^*a)^*$;
2. $w = b^n$ and thus $w \subset (\varepsilon b)^* \subset (a^*b)^*$;
3. w contains both a's and b's and ends with a b

$$w = \underbrace{a \dots ab}_{a^*b} \underbrace{\dots b}_{(a^*b)^*} \underbrace{a \dots ab}_{a^*b} \underbrace{\dots b}_{(a^*b)^*}$$

$\Rightarrow w \in (a^*b)^* \cup (b^*a)^*$;

4. w contains both a's and b's and ends with an $a \Rightarrow$ similar decomposition.

Finite automata

- Finite automata: a first model of the notion of effective procedure. (They also have many other applications).

- The concept of finite automaton can be derived by examining what happens when a program is executed on a computer: state, initial state, transition function.

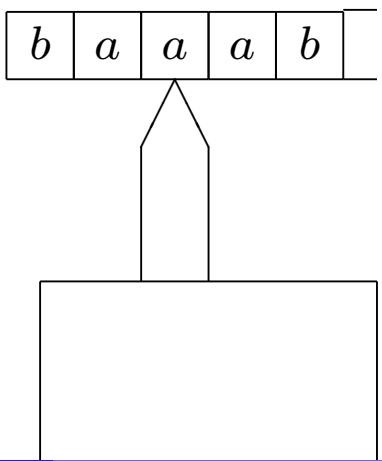
Representing data

- Problem: to recognize a language.
- Data: a word.
- We will assume that the word is fed to the machine character by character, one character being handled at each cycle and the machine stopping once the last character has been read.

Description

- Input tape.
- Set of states:
  - initial state,
  - accepting states.
- Execution step:

tape :



head :

Formalization

A deterministic finite automaton is defined by a five-tuple $M = (Q, \Sigma, \delta, s, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $s \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

Defining the language accepted by a finite automaton

- Configuration : $(q, w) \in Q \times \Sigma^*$.
- Configuration derivable in one step: $(q, w) \vdash_M (q', w')$.
- Derivable configuration (multiple steps) : $(q, w) \vdash_M^* (q', w')$.
- Execution:

$$(s, w) \vdash (q_1, w_1) \vdash (q_2, w_2) \vdash \cdots \vdash (q_n, \varepsilon)$$

- Accepted word:

$$(s, w) \vdash_M^* (q, \varepsilon)$$

  and $q \in F$.

- Accepted language $L(M)$:

$$\{w \in \Sigma^* \mid (s, w) \vdash_M^* (q, \varepsilon) \text{ avec } q \in F\}.$$

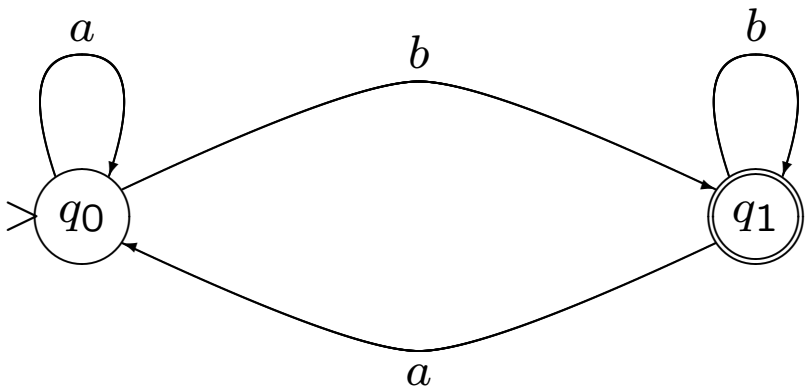
Example

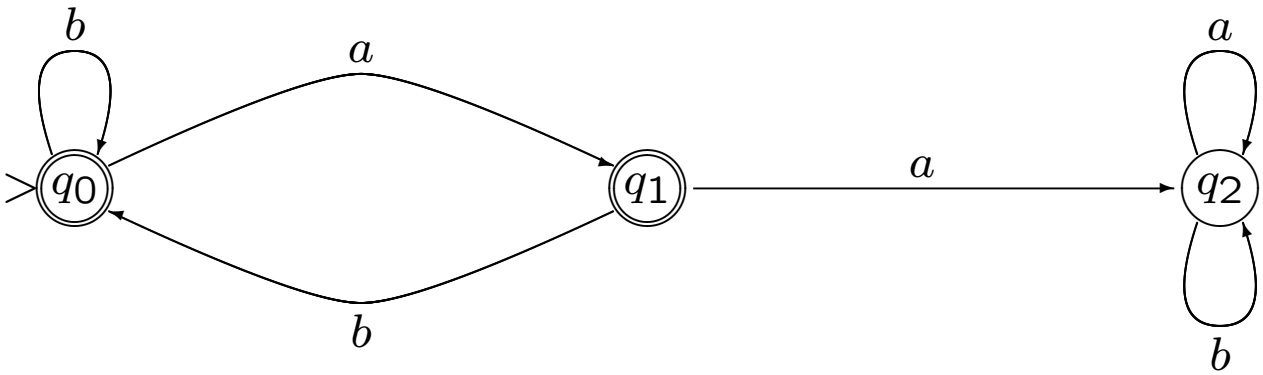
Words ending with b :

δ :

q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_0
q_1	b	q_1

$Q = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$s = q_0$
$F = \{q_1\}$





$\{w \mid w$ does not contain 2 consecutive a's$\}$.

Nondeterministic finite automata

Automata that can *choose* among several transitions.

Motivation :

- To examine the consequences of generalizing a given definition.
- To make describing languages by finite automata easier.
- The concept of non determinism is generally useful.

Description

Nondeterministic finite automata are finite automata that allow:

- several transitions for the same letter in each state,
- transitions on the empty word (i.e., transitions for which nothing is read),
- transitions on words of length greater than 1 (combining transitions).

Nondeterministic finite automata accept if a least one execution accepts.

Formalization

A nondeterministic finite automaton is a five-tuple $M = (Q, \Sigma, \Delta, s, F)$, where

- Q is a finite set of states,
- Σ is an alphabet,
- $\Delta \subset (Q \times \Sigma^* \times Q)$ is the transition relation,
- $s \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

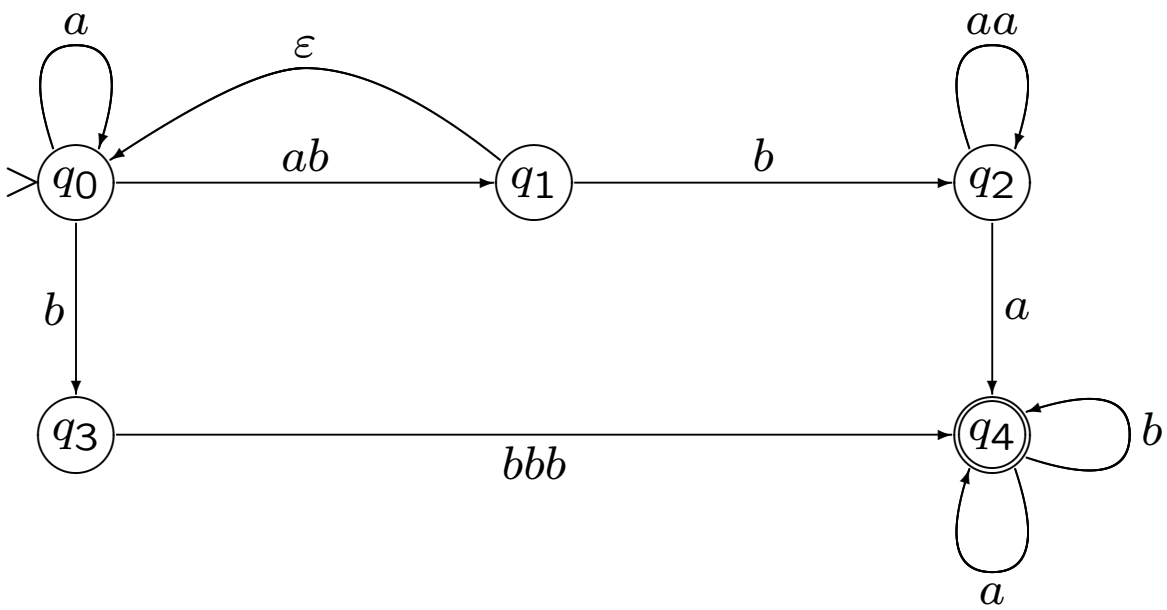
Defining the accepted language

A configuration (q', w') is derivable in one step from the configuration (q, w) by the machine M ($(q, w) \vdash_M (q', w')$) if

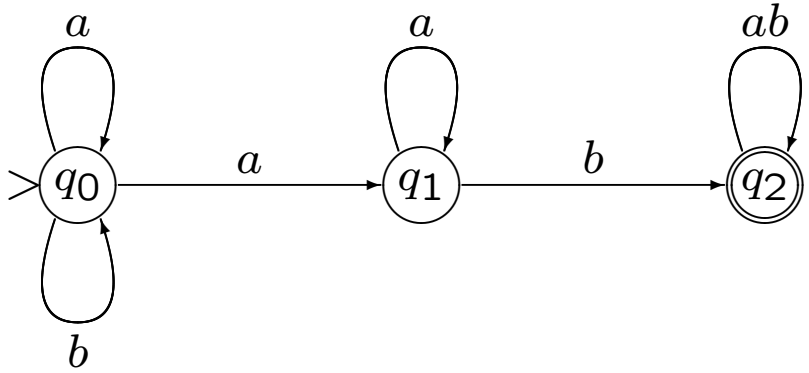
- $w = uw'$ (word w begins with a prefix $u \in \Sigma^*$),
- $(q, u, q') \in \Delta$ (the three-tuple (q, u, q') is in the transition relation Δ).

A word is accepted if *there exists* an execution (sequence of derivable configurations) that leads to an accepting state.

Examples



$$L(M) = ((a \cup ab)^* bbbb\Sigma^*) \cup ((a \cup ab)^* abb(aa)^* a\Sigma^*)$$



$$L(M) = \Sigma^* ab(ab)^*$$

Words ending with at least one repetition of ab.

Eliminating nondeterministism

Definition

Two automata M_1 and M_2 are equivalent if they accept the same language, i.e. if $L(M_1) = L(M_2)$.

Theorem

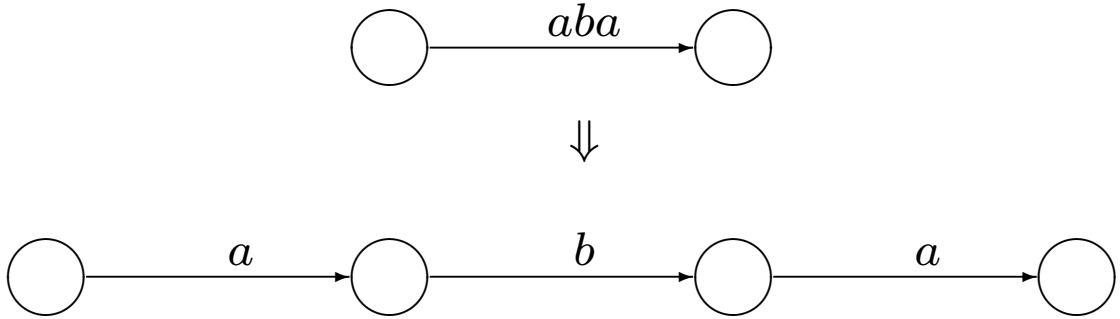
Given any nondeterministic finite automaton, it is possible to build an equivalent deterministic finite automaton.

Idea of the construction

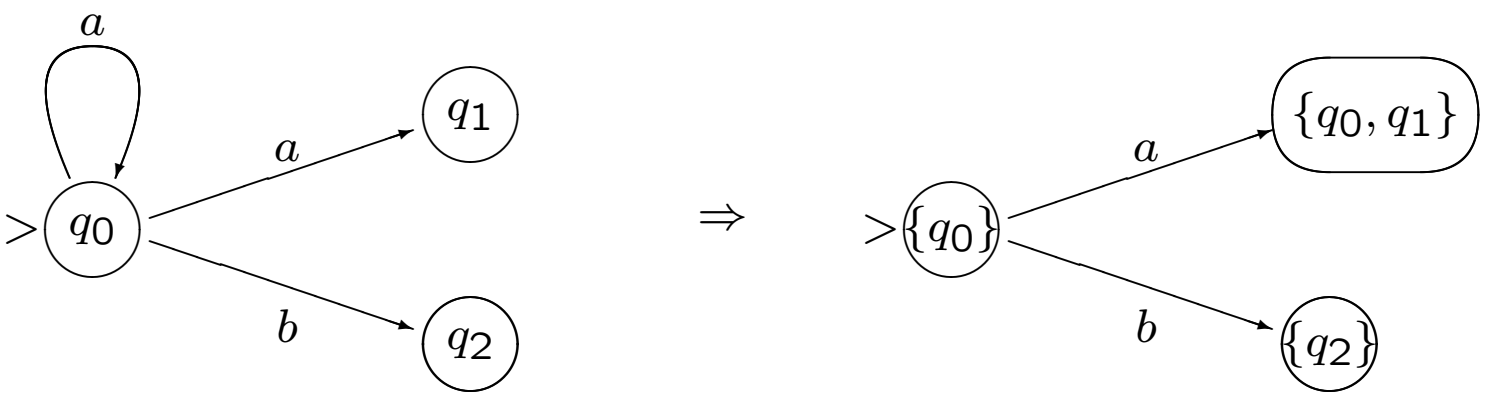
1. Eliminate transitions of length greater than 1.

2. Eliminate compatible transitions

Transitions of length greater than 1



Compatible transitions



Formalization

Non deterministic automaton $M = (Q, \Sigma, \Delta, s, F)$. Build an equivalent non deterministic automaton $M' = (Q', \Sigma, \Delta', s, F)$ such that $\forall (q, u, q') \in \Delta',\ |u| \leq 1$.

- Initially $Q' = Q$ and $\Delta' = \Delta$.
- For each transition $(q, u, q') \in \Delta$ with $u = \sigma_1 \sigma_2 \dots \sigma_k$, $(k > 1)$:
  - remove this transition from Δ',
  - add new states $p_1, \dots, p_{k-1}$ to Q',
  - add new transitions (q, σ_1, p_1), (p_1, σ_2, p_2), $\dots$, (p_{k-1}, σ_k, q') to Δ'

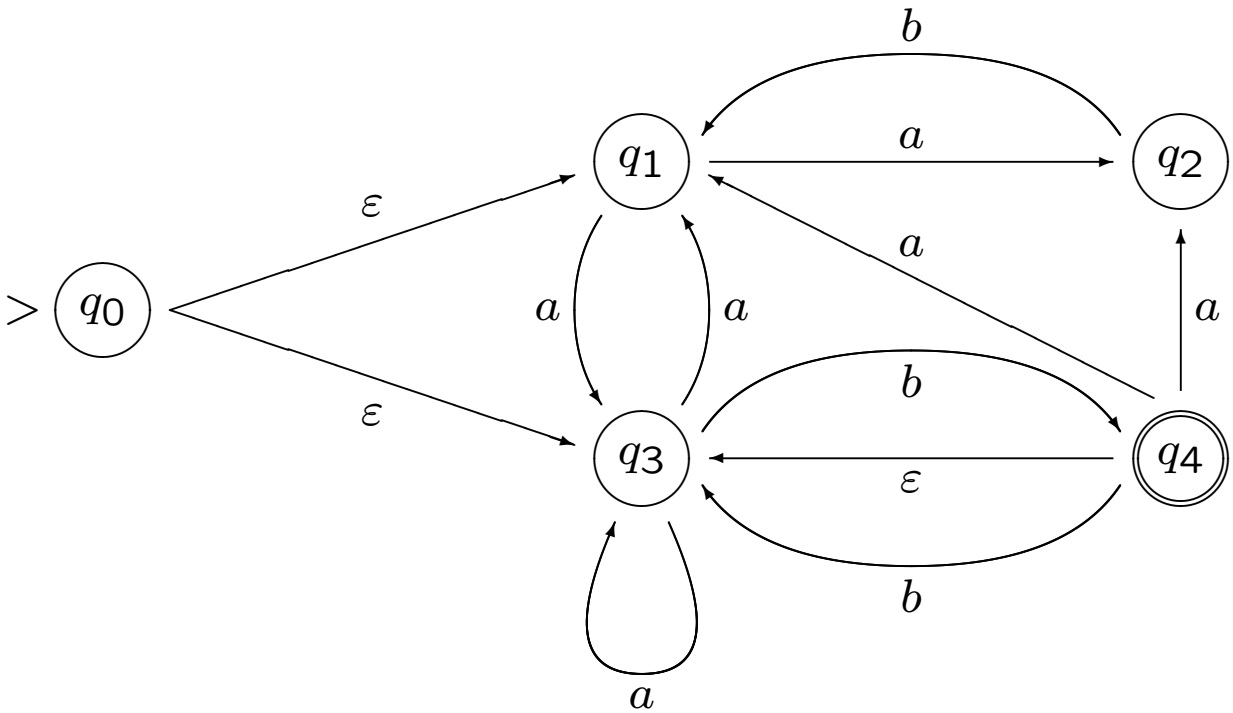
Formalization

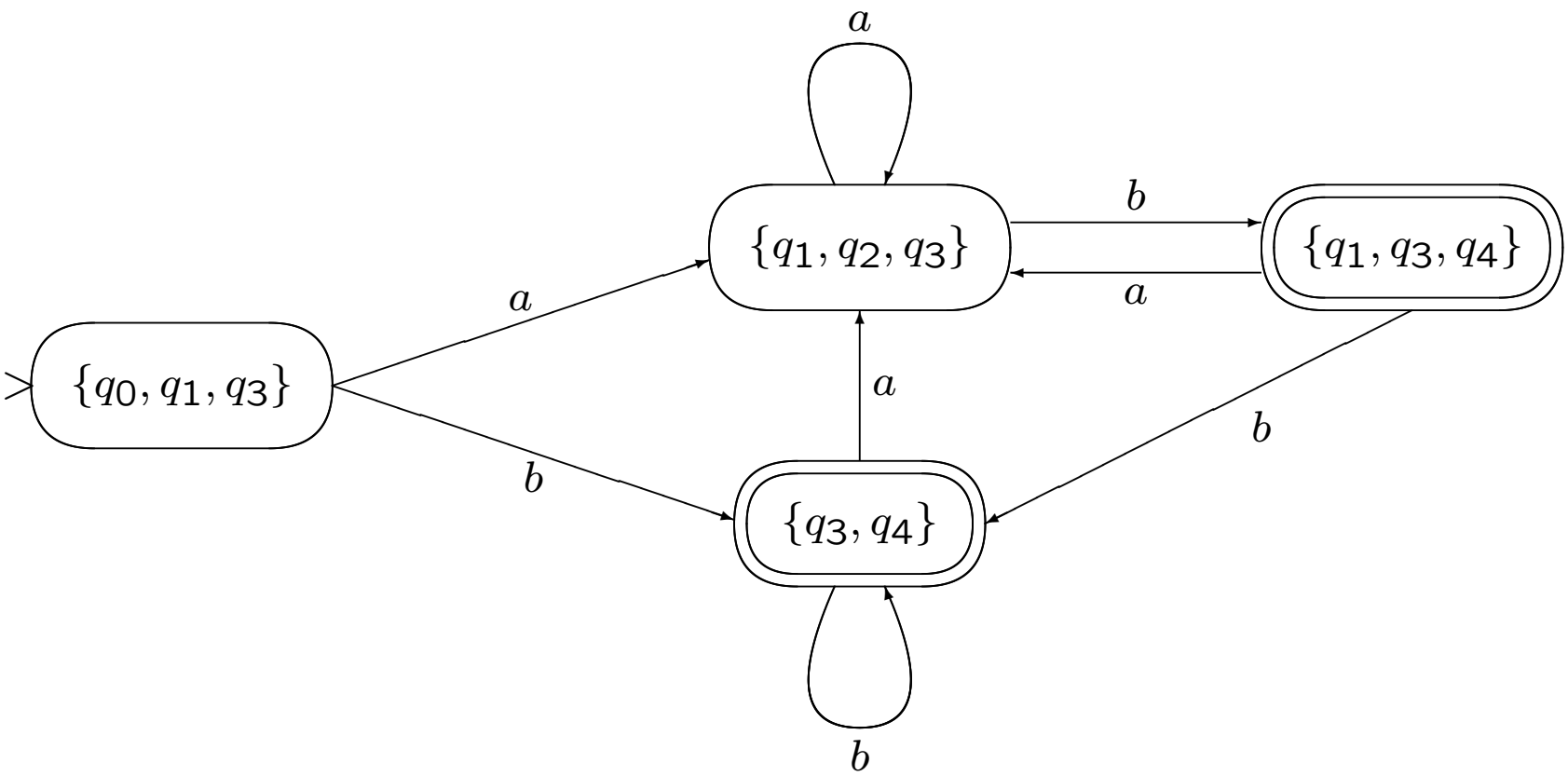
Non deterministic automaton $M = (Q, \Sigma, \Delta, s, F)$ such that $\forall (q, u, q') \in \Delta',\ |u| \leq 1$. Build an equivalent deterministic automaton $M' = (Q', \Sigma, \delta', s', F')$.

$$E(q) = \{p \in Q \mid (q, w) \vdash_M^* (p, w)\}.$$

- $Q' = 2^Q$.
- $s' = E(s)$.
- $\delta'(\mathbf{q}, a) = \bigcup\{E(p) \mid \exists q \in \mathbf{q} \ : (q, a, p) \in \Delta\}$.
- $F' = \{\mathbf{q} \in Q' \mid \mathbf{q} \cap F \neq \emptyset\}$.

Example




a
b
a
a
a
b
b
$\{q_0, q_1, q_3\}$
$\{q_1, q_2, q_3\}$
$\{q_1, q_3, q_4\}$
$\{q_3, q_4\}$

Finite automata and regular expressions

Theorem
A language is regular if and only if it is accepted by a finite automaton.

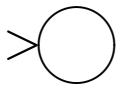
We will prove:

1. If a language can be represented by a regular expression, it is accepted by a non deterministic finite automaton.

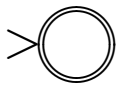
2. If a language is accepted by a non deterministic finite automaton, it is regular.

From expressions to automata

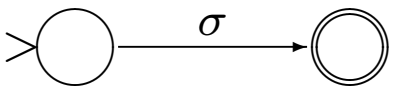
- $\emptyset$



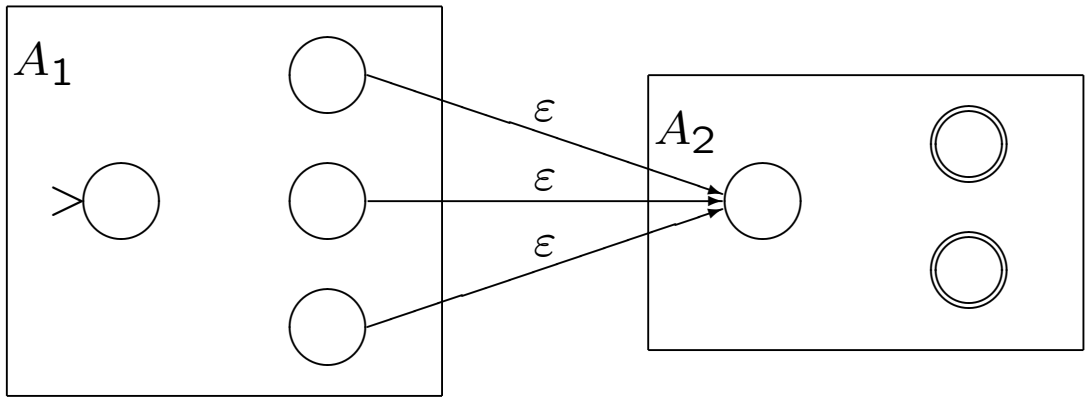
- ε



- $\sigma \in \Sigma$



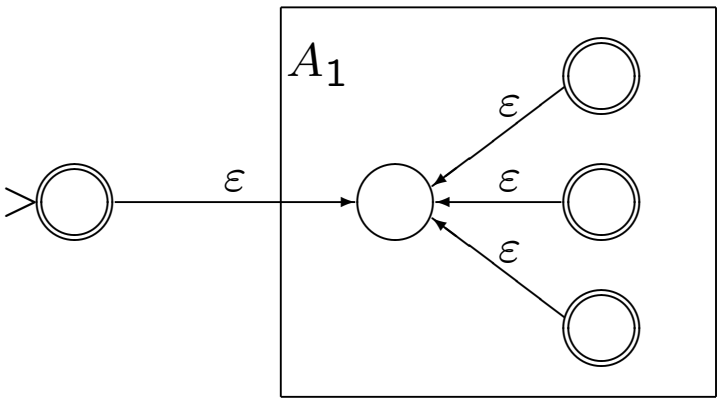
- $\alpha_1 \cdot \alpha_2$



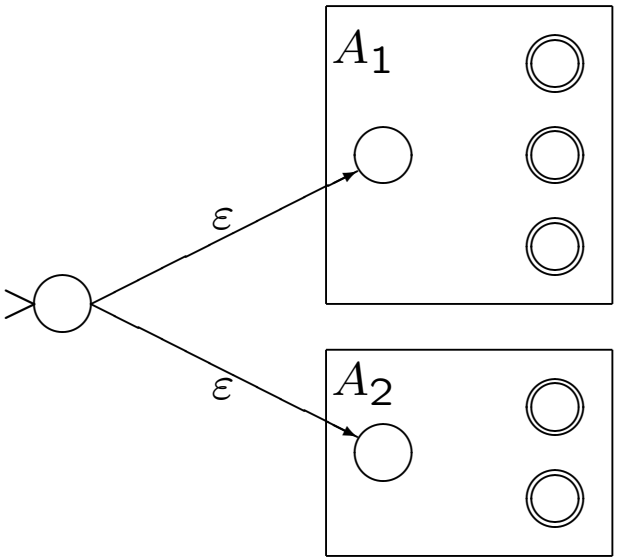
Formally, $A = (Q, \Sigma, \Delta, s, F)$ where

- $Q = Q_1 \cup Q_2$,
- $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q, \varepsilon, s_2) \mid q \in F_1\}$,
- $s = s_1$,
- $F = F_2$.

- $\alpha = \alpha_1^*$



- $\alpha = \alpha_1 \cup \alpha_2$



From automata to regular languages

Intuitive idea:

- Build a regular expression for each path from the initial state to an accepting state.

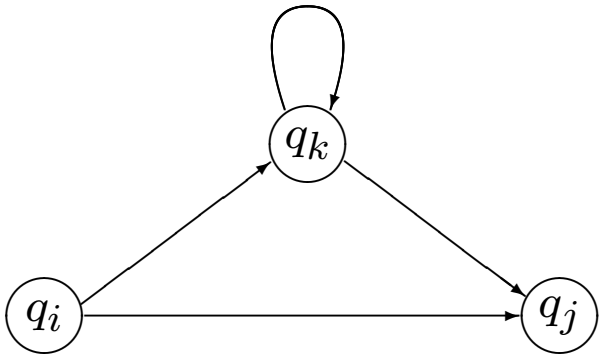
- Use the $*$ operator to handle loops.

Definition

Let M be an automaton and $Q = \{q_1, q_2, \ldots, q_n\}$ its set of states. We will denote by $R(i, j, k)$ the set of words that can lead from the state q_i to the state q_j, going only through states in $\{q_1, \ldots, q_{k-1}\}$.

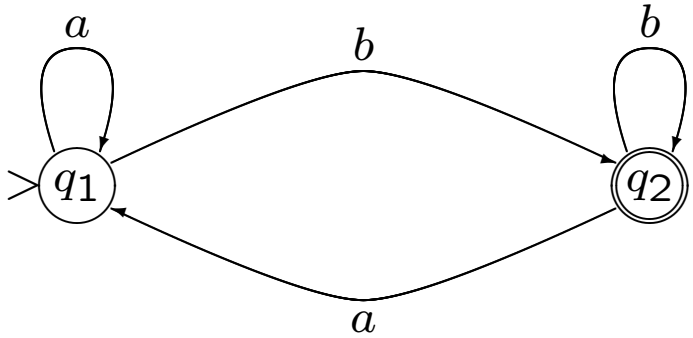
$$R(i,j,1) = \begin{cases} \{w \mid (q_i, w, q_j) \in \Delta\} & \text{si } i \neq j \\ \{\varepsilon\} \cup \{w \mid (q_i, w, q_j) \in \Delta\} & \text{si } i = j \end{cases}$$

$$\begin{aligned} R(i,j,k+1) &= R(i,j,k) \cup \\ & \quad R(i,k,k)R(k,k,k)^*R(k,j,k) \end{aligned}$$



$$L(M) = \bigcup_{q_j \in F} R(1,j,n+1).$$

Example



	$k=1$	$k=2$
$R(1,1,k)$	$\varepsilon \cup a$	$(\varepsilon \cup a) \cup (\varepsilon \cup a)(\varepsilon \cup a)^*(\varepsilon \cup a)$
$R(1,2,k)$	b	$b \cup (\varepsilon \cup a)(\varepsilon \cup a)^*b$
$R(2,1,k)$	a	$a \cup a(\varepsilon \cup a)^*(\varepsilon \cup a)$
$R(2,2,k)$	$\varepsilon \cup b$	$(\varepsilon \cup b) \cup a(\varepsilon \cup a)^*b$

The language accepted by the automaton is $R(1,2,3)$, which is

$$[b \cup (\varepsilon \cup a)(\varepsilon \cup a)^*b] \quad \cup \quad \begin{array}{l} [b \cup (\varepsilon \cup a)(\varepsilon \cup a)^*b] \\ [(\varepsilon \cup b) \cup a(\varepsilon \cup a)^*b]^* \\ [(\varepsilon \cup b) \cup a(\varepsilon \cup a)^*b] \end{array}$$

So far 3 characterizations of regular languages

1. regular expressions,

2. deterministic finite automata,

3. nondeterministic finite automata,

Properties of regular languages

Let L_1 and L_2 be two regular languages.

- $L_1 \cup L_2$ is regular.
- $L_1 \cdot L_2$ is regular.
- L_1^* is regular.
- L_1^R is regular.
- $\overline{L_1} = \Sigma^* - L_1$ is regular.
- $L_1 \cap L_2$ is regular.

- Let Σ be the alphabet on which L_1 is defined, and let $\pi : \Sigma \to \Sigma'$ be a function from Σ to another alphabet Σ'.

  This fonction, called a *projection function* can be extended to words by applying it to every symbol in the word, *i.e.* for $w = w_1 \dots w_k \in \Sigma^*$, $\pi(w) = \pi(w_1) \dots \pi(w_k)$.

  If L_1 is regular, the language $\pi(L_1)$ is also regular.

Algorithms

Les following problems can be solved by algorithms for regular languages:

- $w \in L$?
- $L = \emptyset$?
- $L = \Sigma^*$? $\qquad (\overline{L} = \emptyset)$
- $L_1 \subseteq L_2$? $\qquad (\overline{L_2} \cap L_1 = \emptyset)$
- $L_1 = L_2$? $\qquad (L_1 \subseteq L_2$ and $L_2 \subseteq L_1)$

Applications of regular languages

Problem : To find in a (long) character string w, all ocurrences of words in the language defined by a regular expression α.

1. Consider the regular expression $\beta = \Sigma^* \alpha$.

2. Build a nondeterministic automaton accepting the language defined by β

3. From this automaton, build a *deterministic* automaton A_β.

4. Simulate the execution of the automaton A_β on the word w.
   Whenever this automaton is in an accepting state, one is at the end of an occurrence in w of a word in the language defined by α.

Languages that are not regular ?

There are not enough regular expressions to represent all languages!

Definition

Cardinality of a set. . .

Example

The sets $\{0, 1, 2, 3\}$, $\{a, b, c, d\}$, $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ all have the same size. There exists a one-one correspondence (bijection) between them, for example $\{(0, \clubsuit), (1, \diamondsuit), (2, \heartsuit), (3, \spadesuit)\}$.

Denumerable (countably infinite) sets

Definition

An infinite set is *denumerable* if there exists a bijection between this set and the set of natural numbers.

Remark

Finite sets are all countable in the usual sense, but in mathematics countable is sometimes used to mean precisely countably infinite.

Denumerable sets: examples

1. The set of even numbers is denumerable:

$$\{(0,0),(2,1),(4,2),(6,3),\ldots\}.$$

2. The set of words over the alphabet $\{a,b\}$ is denumerable :
   $\{(\varepsilon,0),(a,1),(b,2),(aa,3),(ab,4),(ba,5),$
   $(bb,6),(aaa,7)\ldots\}$.

3. The set of rational numbers is denumerable:
   $\{(0/1,0),(1/1,1),(1/2,2),(2/1,3),(1/3,4),$
   $(3/1,5),\ldots\}$.

4. The set of regular expressions is denumerable.

The diagonal argument

Theorem The set of subsets of a denumerable set is not denumerable.

Proof

	a_0	a_1	a_2	a_3	a_4	$\cdots$
s_0	$\times$	$\times$		$\times$		
s_1	$\times$	$\square$		$\times$		
s_2		$\times$	$\times$		$\times$	
s_3	$\times$		$\times$	$\square$		
s_4		$\times$		$\times$	$\square$	
$\vdots$						

$$D = \{a_i \mid a_i \notin s_i\}$$

D is not amongts the s_i since :

- $a_i \in s_i \Rightarrow a_i \notin D \Rightarrow D \neq S_i$
- $a_i \notin s_i \Rightarrow a_i \in D \Rightarrow D \neq S_i$

So, there are nonregular languages.

- The set of languages is not denumerable.
- The set of regular languages is denumerable.
- Thus there are (many) more languages than regular languages

So, there are nonregular languages. But which ?

- We will now prove, using another techniques that some specific languages are not regular.

Basic Observations

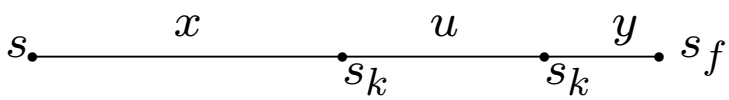
1. All finite languages (including only a finite number of words) are regular.

2. A non regular language must thus include an infinite number of words.

3. If a language includes an infinite number of words, there is no bound on the size of the words in the language.

4. Any regular language is accepted by a finite automaton that has a given number number m of states.

5. Consider an infinite regular language and an automaton with m states accepting this language. For any word whose length is greater than m, the execution of the automaton on this word must go through an identical state s_k at least twice, a nonempty part of the word being read between these two visits to s_k.



6. Consequently, all words of the form xu^*y are also accepted by the automaton and thus are in the language.

The "pumping" lemmas (theorems)

First version

Let L be an infinite regular language. Then there exists words $x, u, y \in \Sigma^*$, with $u \neq \varepsilon$ such that $xu^n y \in L \ \forall n \geq 0$.

Second version :

Let L be a regular language and let $w \in L$ be such that $|w| \geq |Q|$ where Q is the set of states of a determnistic automaton accepting L. Then $\exists x, u, y$, with $u \neq \varepsilon$ and $|xu| \leq |Q|$ such that $xuy = w$ and, $\forall n$, $xu^n y \in L$.

Applications of the pumping lemmas

The langage

$$a^n b^n$$

is not regular. Indeed, it is not possible to find words x, u, y such that $xu^k y \in a^n b^n \ \forall k$ and thus the pumping lemma cannot be true for this language.

$u \in a^*$: impossible.

$u \in b^*$: impossible.

$u \in (a \cup b)^* - (a^* \cup b^*)$: impossible.

The language

$$L = a^{n^2}$$

is not regular. Indeed, the pumping lemma (second version) is contradicted.

Let $m = |Q|$ be the number of states of an automaton accepting L. Consider a^{m^2}. Since $m^2 \geq m$, there must exist x, u and y such that $|xu| \leq m$ and $xu^n y \in L \ \forall n$. Explicitly, we have

$$\begin{array}{ll} x = a^p & 0 \leq p \leq m-1, \\ u = a^q & 0 < q \leq m, \\ y = a^r & r \geq 0. \end{array}$$

Consequently $xu^2y \notin L$ since $p + 2q + r$ is not a perfect square. Indeed,

$$m^2 < p + 2q + r \leq m^2 + m < (m+1)^2 = m^2 + 2m + 1.$$

The language

$$L = \{a^n \mid n \text{ is prime}\}$$

is not regular. The first pumping lemma implies that there exists constants p, q and r such that $\forall k$

$$xu^k y = a^{p+kq+r} \in L,$$

in other words, such that $p + kq + r$ is prime for all k. This is impossible since for $k = p + 2q + r + 2$, we have

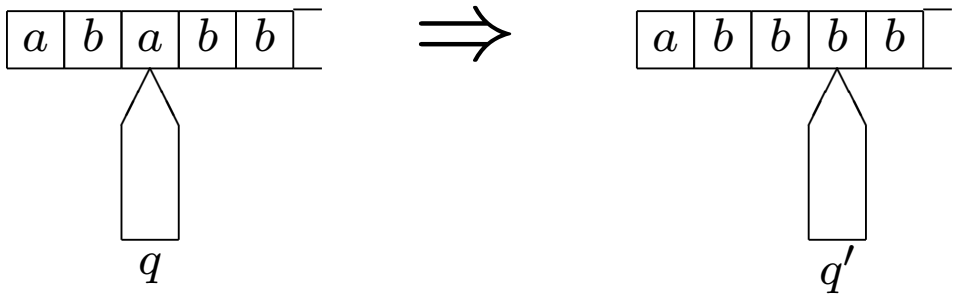
$$p + kq + r = \underbrace{(q+1)}_{>1}\underbrace{(p+2q+r)}_{>1},$$

The Turing machine

- Infinite memory viewed as a tape divided into cells that can hold one character of a tape alphabet.
- Read head.
- Finite set of states, accepting states.
- transition function that, for each state and tape symbol pair gives
  - the next state,
  - a character to be written on the tape,
  - the direction (left or right) in which the read head moves by one cell.

Execution

- Initially, the input word is on the tape, the rest of the tape is filled with "blank" symbols, the read head is on the leftmost cell of the tape.
- At each step, the machine
  - reads the symbol from the cell that is under the read head,
  - replaces this symbol as specified by the transition function,
  - moves the read head one cell to the left or to the right, as specified by the transition function.
  - changes state as described by the transition function,
- the input word is accepted as soon as an accepting state is reached.


a b a b b
q
a b b b b
q′

Formalization

7-tuple $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$, where:

- Q is a finite set of states,
- Γ is the tape alphabet,
- $\Sigma \subseteq \Gamma$ is the input alphabet,
- $s \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states,
- $B \in \Gamma - \Sigma$ is the "blank symbol" (#),
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.

Configuration

The required information is:

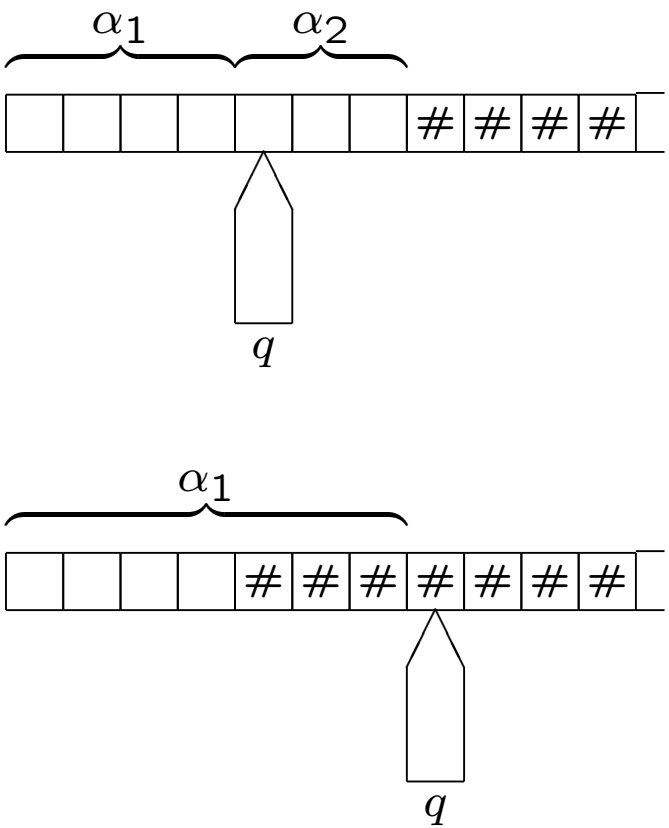
1. The state,
2. the tape content,
3. the position of the read head.

Representation : 3-tuple containing

1. the state of the machine,
2. the word found on the tape up to the read head,
3. the word found on the tape from the read head on.

Formally, a configuration is an element of $Q \times \Gamma^* \times (\varepsilon \cup \Gamma^*(\Gamma - \{B\}))$.

Configurations (q, α_1, α_2) and $(q, \alpha_1, \varepsilon)$.



Derivation

Configuration (q, α_1, α_2) written as $(q, \alpha_1, b\alpha_2')$ with $b = \#$ if $\alpha_2 = \varepsilon$.

- If $\delta(q, b) = (q', b', R)$ we have

$$(q, \alpha_1, b\alpha_2') \vdash_M (q', \alpha_1 b', \alpha_2').$$

- If $\delta(q, b) = (q', b', L)$ and if $\alpha_1 \neq \varepsilon$ and is thus of the form $\alpha_1' a$ we have

$$(q, \alpha_1' a, b\alpha_2') \vdash_M (q', \alpha_1', ab'\alpha_2').$$

Derivation

A configuration C' is derivable in several steps from the configuration C by the machine M ($C \vdash_M^* C'$) if there exists $k \geq 0$ and intermediate configurations $C_0, C_1, C_2, \ldots, C_k$ such that

- $C = C_0$,
- $C' = C_k$,
- $C_i \vdash_M C_{i+1}$ for $0 \leq i < k$.

The language $L(M)$ accepted by the Turing machine is the set of words w such that

$$(s, \varepsilon, w) \vdash_M^* (p, \alpha_1, \alpha_2), \text{ with } p \in F.$$

Example

Turing machine $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$ with

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- $\Sigma = \{a, b\}$,
- $s = q_0$,
- $B = \#$,
- δ given by

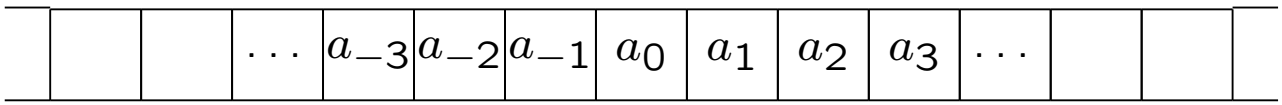
	a	b	X	Y	$\#$
q_0	(q_1, X, R)	$-$	$-$	(q_3, Y, R)	$-$
q_1	(q_1, a, R)	(q_2, Y, L)	$-$	(q_1, Y, R)	$-$
q_2	(q_2, a, L)	$-$	(q_0, X, R)	(q_2, Y, L)	$-$
q_3	$-$	$-$	$-$	(q_3, Y, R)	$(q_4, \#, R)$
q_4	$-$	$-$	$-$	$-$	$-$

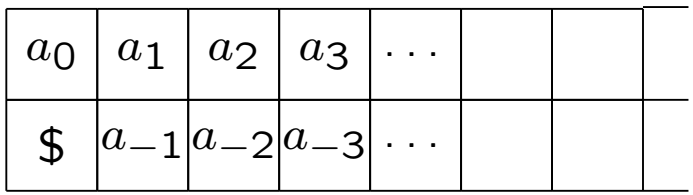
M accepts $a^n b^n$. For example, its execution on $aaabbb$ is

$(q_0, \varepsilon, aaabbb)$	$\vdots$
$(q_1, X, aabbb)$	$(q_1, XXXYY, b)$
$(q_1, Xa, abbb)$	$(q_2, XXXY, YY)$
(q_1, Xaa, bbb)	(q_2, XXX, YYY)
$(q_2, Xa, aYbb)$	$(q_2, XX, XYYY)$
$(q_2, X, aaYbb)$	(q_0, XXX, YYY)
$(q_2, \varepsilon, XaaYbb)$	$(q_3, XXXY, YY)$
$(q_0, X, aaYbb)$	$(q_3, XXXYY, Y)$
$(q_1, XX, aYbb)$	$(q_3, XXXYYY, \varepsilon)$
	$(q_4, XXXYYY\#, \varepsilon)$

Extensions of Turing machines

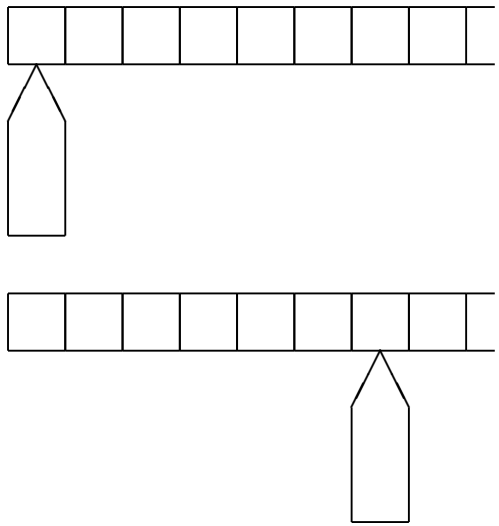
Tape that is infinite in both directions





Multiple tapes

Several tapes and read heads:



Simulation (2 tapes) : alphabet = 4-tuple

- Two elements represent the content of the tapes,
- Two elements represent the position of the read heads.

Nondeterministic Turing machines

Transition relation :

$$\Delta : (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$$

The execution is no longer unique.

Eliminating non-determinism

Theorem

Any language that is accepted by a nondeterministic Turing machine is also accepted by a deterministic Turing machine.

Proof

Simulate the executions in increasing-length order.

$$r = \max_{q \in Q, a \in \Gamma} |\{((q, a), (q', x, X)) \in \Delta\}|.$$

Three tape machine:

1. The first tape holds the input word and is not modified.
2. The second tape will hold sequences of numbers less than r.
3. The third tape is used by the deterministic machine to simulate the nondeterministic one.

The deterministic machine proceeds as follows.

1. On the second tape it generates all finite sequences of numbers less than r. These sequences are generated in increasing length order.

2. For each of these sequences, it simulates the nondeterministic machine, the choice being made according to the sequence of numbers.

3. It stops as soon as the simulation of an execution reaches an accepting state.

Accepted language
Decided language

Turing machine = effective procedure ? Not always. The following situation are possible.

1. The sequence of configurations contains an accepting state.
2. The sequence of configurations ends because either
   - the transition function is not defined, or
   - it requires a left move from the first cell on the tape.
3. The sequence of configurations never goes through an accepting state and is infinite.

In the first two cases, we have an effective procedure, in the third not.

The *execution* of a Turing machine on a word w is the maximal sequence of configurations

$$(s, \varepsilon, w) \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_k \vdash_M \cdots$$

i.e., the sequence of configuration that either

- is infinite,
- ends in a configuration in which the state is accepting, or
- ends in a configuration from which no other configuration is derivable.

Decided language: A language L is decided by a Turing machine M if

- M accepts L,
- M has no infinite executions.

Turing machine accepts and terminates

Definition
The decidability class R is the set of languages that can be decided by a Turing machine.

The class R is the class of languages (problems) that are

- decided by a Turing machine,
- recursive, decidable, computable,
- algorithmically solvable.

Turing machine accepts but may loop

Definition

The decidability class RE is the set of languages that can be accepted by a Turing machine.

The class RE is the class of languages (problems) that are

- accepted by a Turing machine,
- partially recursive, partially decidable, partially computable,
- partially algorithmically solvable,
- recursively enumerable.

Lemma

The class R is contained in the class RE ($R \subseteq RE$)

The languages computed by a Turing machine

Definition

Let M be a Turing machine.If M stops on an input word u, let $f_M(u)$ be the word computed by M for u. The language computed by M is then the set of words

$$\{w \mid \exists u \text{ such that } M \text{ stops for } u \text{ and } w = f_M(u)\}.$$

Theorem

A language is computed by a Turing machine if and only if it is recursively enumerable (accepted by a Turing machine).

Let L be a language accepted by a Turing machine M. The Turing machine M' described below computes this language.

1. The machine M' first memorises its input word (one can assume that it uses a second tape for doing this).

2. Thereafter, it behaves exactly as M.

3. If M accepts, M' copies the memorised input word onto its tape.

4. If M does not accept, M' keeps running forever.

A first undecidable language

A	w_0	w_1	w_2	$\dots$	w_j	$\dots$
M_0	Y	N	N	$\dots$	Y	$\dots$
M_1	N	N	Y	$\dots$	Y	$\dots$
M_2	Y	Y	N	$\dots$	N	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
M_i	N	N	Y	$\dots$	N	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

- $A[M_i, w_j] = \mathsf{Y}$ (yes) if the Turing machine M_i accepts the word w_j ;

- $A[M_i, w_j] = \mathsf{N}$ (no) if the Turing machine M_i does not accept the word w_j (loops or rejects the word).

$L_0 = \{w \mid w = w_i \wedge A[M_i, w_i] = N\}$ is not accepted by an M_i since :

- $w_i \in L(M_i) \Rightarrow w_i \notin L_0 \Rightarrow L_0 \neq L(M_i)$
- $w_i \notin L(M_i) \Rightarrow w_i \in L_0 \Rightarrow L_0 \neq L(M_i)$

A second undecidable language

Lemma
The complement of a language in the class R is also in the class R.

Lemma
If a language L and its complement $\overline{L}$ are both in the class RE, then both L and $\overline{L}$ are in R.

Three situations are thus possible:

1. L and $\overline{L} \in \mathsf{R}$,
2. $L \notin \mathsf{RE}$ and $\overline{L} \notin \mathsf{RE}$,
3. $L \notin \mathsf{RE}$ and $\overline{L} \in \mathsf{RE} \cap \overline{\mathsf{R}}$.

Lemma

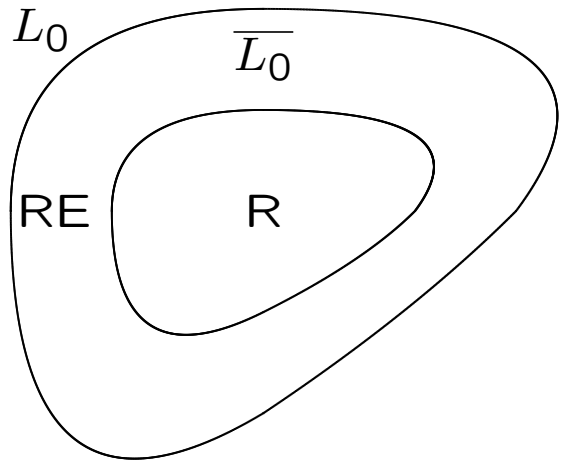
The language

$$\overline{L_0} = \{w | w = w_i \wedge M_i \text{ accepts } w_i\}$$

is in the class RE.

Theorem

The language $\overline{L_0}$ is undecidable (is not in R), but is in RE.



The reduction technique

1. One proves that, if there exists an algorithm that decides the language L_2, then there exists an algorithm that decides the language L_1. This is done by providing an algorithm (formally a Turing machine that stops on all inputs) that decides the language L_1, using as a sub-program an algorithm that decides L_2. This type of algorithm is called a *reduction*from L_1 to L_2. Indeed, it reduces the decidability of L_1 to that of L_2.

2. If L_1 is undecidable, one can conclude that L_2 is also undecidable ($L_2 \notin \mathrm{R}$). Indeed, the reduction from L_1 to L_2 establishes that if L_2 was decidable, L_1 would also be decidable, which contradicts the hypothesis that L_1 is an undecidable language.

The *universal language* UL

$$\text{UL} = \{< M, w > | \ M \text{ accepts } w\}$$

is undecidable.

Reduction from $\overline{L_0}$: to check if a word w is in $\overline{L_0}$, proceed as follows.

1. Find the value i such that $w = w_i$.

2. Find the Turing machine M_i.

3. Apply the decision procedure for UL to the wprd $< M_i, w_i >$: if the result is positive, w is accepted, if not it is rejected.

Note : $\overline{\text{UL}} \notin \text{RE}$

More undecidable problems

The halting problem

$$\mathsf{H} = \{< M, w >| \ M \text{ stops on } w\}$$

is undecidable. Reduction from UL.

1. Apply the algorithm deciding H to $< M, w >$.
2. If the algorithm deciding H gives the answer "no" (*i.e.* the machine M does not stop), answer "no" (in this case, we have indeed that $< M, w > \notin$ UL).
3. If the algorithm deciding H gives the answer "yes, simulate the execution of M on w and give the answer that is obtained (in this case, the execution of M on w terminates and one always obtains an answer).

The problem of determining if a program written in a commonly used programming language (for example C or, Java) stops for given input values is undecidable. This is proved by reduction from the halting problem for Turing machines.

1. Build a C program P that, given a Turing machine M and a word w, simulates the behaviour of M on w.

2. Decide if the program P stops for the input $< M, w >$ and use the result as answer.

Undecidability in logics

The problem of determining *validity in the predicate calculus* is undecidable

Undecidability in arithmetics

Hilbert's tenth problem is undecidable. This problem is to determine if an equation

$$p(x_1, \ldots, x_n) = 0$$

where $p(x_1, \ldots, x_n)$ is an integer coefficient polynomial, has an integer solution.

Turing machine computable functions

A Turing machine computes a function $f : \Sigma^* \to \Sigma^*$ if, for any input word w, it always stops in a configuration where $f(w)$ is on the tape.

The Turing-Church thesis

The functions that are computable by an effective procedure are those that are computable by a Turing machine

Justification

- All reasonable models of computations proposed can be simulated by a Turing machine.
- No phenomena in nature has allowed us to compute beyond what the Turing machine can do.

Whatever function can be computer by Physics,
can be computed by a Turing Machine.

Universal Turing machines

A Turing machine that can simulate any Turing machine.

- Turing machine M.
- Data for M : M' and a word w.
- M simulates the execution of M' on w.

Complexity

Main points

- Solvable problems versus efficiently solvable problems.
- Measuring complexity: complexity functions.
- Polynomial complexity.
- NP-complete problems.

Measuring complexity

- Abstraction with respect to the machine being used.
- Abstraction with respect to the data (data size as only parameter).
- O notation.
- Efficiency criterion: polynomial.

Complexity and Turing machines

Time complexity of a Turing machine that always stops:

$$T_M(n) = \max \{m \mid \exists x \in \Sigma^*, |x| = n \text{ and the execution of } M \text{ on } x \text{ is } m \text{ steps long}\}.$$

A Turing machine is polynomial if there exists a polynomial $p(n)$ such that

$$T_M(n) \leq p(n)$$

for all $n \geq 0$.

The class P is the class of languages that are decided by a polynomial Turing machine.

Polynomial transformations

- Diagonalisation is not adequate to prove that problems are not in P.
- Another approach: comparing problems.

The Travelling Salesman (TS)

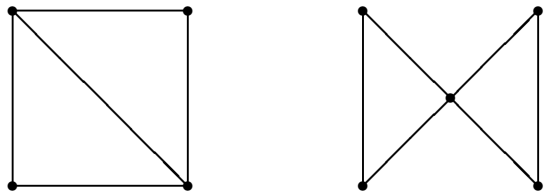
- Set C of n Cities.
- Distances $d(c_i, c_j)$.
- A constant b.
- Is there a permutation of the towns such that:

$$\sum_{1 \leq i < n} d(c_{p_i}, c_{p_{i+1}}) + d(c_{p_n}, c_{p_1}) \leq b.$$

Hamiltonian Circuit (HC)

- Graph $G = (V, E)$

- Is there a closed circuit in the graph that contains each vertex exactly once.



Solving HC by solving TS

HC $\propto$ TS

- The set of cities is identical to the set of vertices of the graph, i.e. $C = V$.
- The distances are the following $(c_i, c_j) = \begin{cases} 1 & \text{si } (c_i, c_j) \in E \\ 2 & \text{si } (c_i, c_j) \notin E \end{cases}$.
- The constant b is equal to the number of cities, i.e. $b = |V|$.

Definition of polynomial transformations

Goal : to establish a link between problems such as HC and TS (one is in P if and only if the other is also in P).

Definition :
Consider languages $L_1 \in \Sigma_1^*$ and $L_2 \in \Sigma_2^*$. A *polynomial transformation* from L_1 to L_2 (notation $L_1 \propto L_2$) is a function $f : \Sigma_1^* \to \Sigma_2^*$ that satisfies the following conditions :

1. it is computable in polynomial time,

2. $f(x) \in L_2$ if and only if $x \in L_1$.

Properties of $\propto$

If $L_1 \propto L_2$, then

- if $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}$,
- if $L_1 \notin \mathsf{P}$ then $L_2 \notin \mathsf{P}$.

If $L_1 \propto L_2$ et $L_2 \propto L_3$, then

- $L_1 \propto L_3$.

Polynomially equivalent problems

Definition

Two languages L_1 and L_2 are *polynomially equivalent* (notation $L_1 \equiv_P L_2$) if and only if $L_1 \propto L_2$ and $L_2 \propto L_1$.

- Classes of polynomially equivalent problems: either all problems in the class are in P, or none is.
- Such an equivalence class can be built incrementally by adding problems to a known class.
- We need a more abstract definition of the class containing HC and TS.

The class NP

- The goal is to characterise problems for which it is necessary to examine a very large number of possibilities, but such that checking each possibility can be done quickly.

- Thus, the solution is fast, if enumerating the possibilities does not cost anything.

- Modelisation : nondeterminism.

The complexity of nondeterministic Turing machines

The execution time of a nondeterministic Turing machine on a word w is given by

- the length of the *shortest* execution accepting the word, if it is accepted,
- the value 1 if the word is not accepted.

The time complexity of M (non deterministic) is the function $T_M(n)$ defined by

$$T_M(n) = \max \{m \mid \exists x \in \Sigma^*, |x| = n \text{ and the execution time of } M \text{ on } x \text{ is } m \text{ steps long}\}.$$

The definition of NP

Définition

The class NP (from *N*ondeterministic *P*olynomial) is the class of languages that are accepted by a polynomial nondeterministic Turing machine.

Exemple

HC and TS are in NP.

Theorem

Consider $L \in$ NP. There exists a deterministic Turing machine M and a polynomial $p(n)$ such that M decides L and has a time complexity bounded by $2^{p(n)}$.

Let M_{nd} be a nondeterministic machine of polynomial complexity $q(n)$ that accepts L. The idea is to simulate all executions of M_{nd} of length less than $q(n)$. For a word w, the machine M must thus:

1. Determine the length n of w and compute $q(n)$.

2. Simulate each execution of M_{nd} of length $q(n)$ (let the time needed be $q'(n)$). If r is the largest number of possible choices within an execution of M_{nd}, there are at most $r^{q(n)}$ executions of length $q(n)$.

3. If one of the simulated executions accepts, M accepts. Otherwise, M stops and rejects the word w.

Complexity : bounded by $r^{q(n)} \times q'(n)$ and thus by $2^{\log_2(r)(q(n)+q'(n))}$, which is of the form $2^{p(n)}$.

The structure of NP

Definition A polynomial equivalence class C_1 is smaller than a polynomial equivalence class C_2 (notation $C_1 \preceq C_2$) if there exists a polynomial transformation from every language in C_1 to every language in C_2.

Smallest class in NP : P

- The class NP contains the class P ($\mathsf{P} \subseteq \mathsf{NP}$).
- The class P is a polynomial equivalence class.
- For every $L_1 \in \mathsf{P}$ and for every $L_2 \in \mathsf{NP}$, we have $L_1 \propto L_2$.

Largest class in NP : NPC

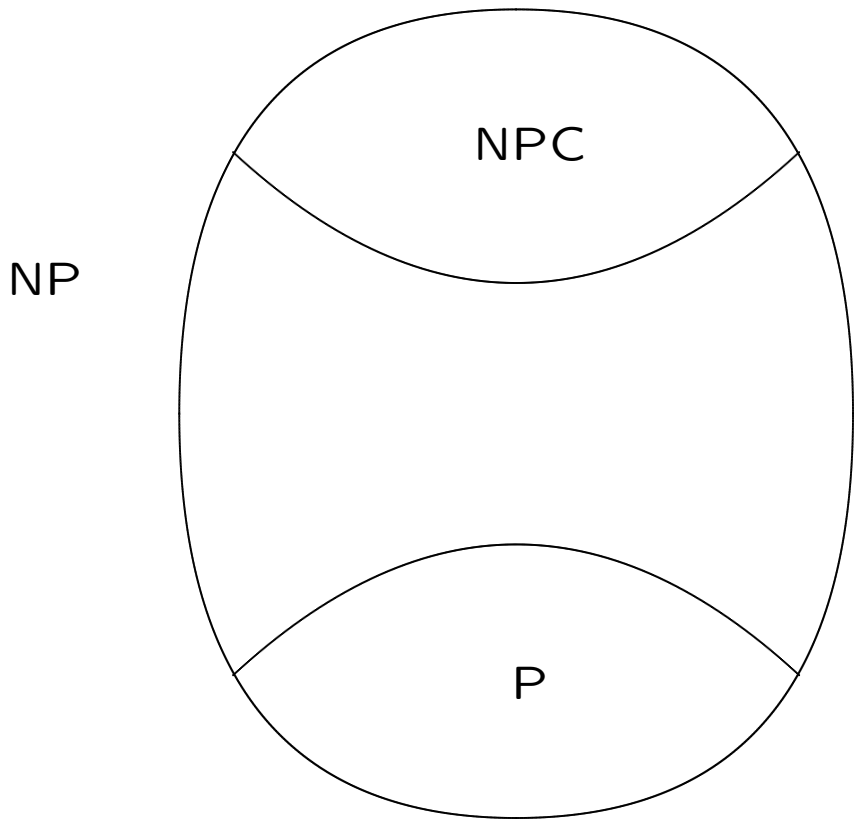
A language L is NP-complete if

1. $L \in$ NP,

2. for every language $L' \in$ NP, $L' \propto L$.

Theorem
If there exists an NP-complete language L decided by a polynomial algorithm, then all languages in NP are polynomially decidable, i.e. P = NP.

Conclusion : An NP-complete problem does not have a polynomial solution if and only if P $\neq$ NP


NP
NPC
P

Proving NP-completeness

To prove that a language L is NP-complete, one must establish that

1. it is indeed in the class NP ($L \in$ NP),
2. for every language $L' \in$ NP, $L' \propto L$,

or, alternatively,

3. There exists $L' \in$ NPC such that $L' \propto L$.

Concept of NP-hard problem.

A first NPC problem

SAT Problem : satisfiability of conjunctive normal form propositional calculus formulas.

Theorem
The SAT problem is NP-complete

Other NP-complete problems

3-SAT : satisfiability for conjunctive normal form formulas with exactly 3 literals per clause.

SAT $\propto$ 3-SAT.

1. A clause $(x_1 \vee x_2)$ with two literals is replaced by

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$$

2. A clause (x_1) with a single literal is replaced by

$$\begin{array}{ccccc} (x_1 \vee y_1 \vee y_2) & \wedge & (x_1 \vee y_1 \vee \neg y_2) & \wedge \\ (x_1 \vee \neg y_1 \vee y_2) & \wedge & (x_1 \vee \neg y_1 \vee \neg y_2) & \end{array}$$

3. A clause

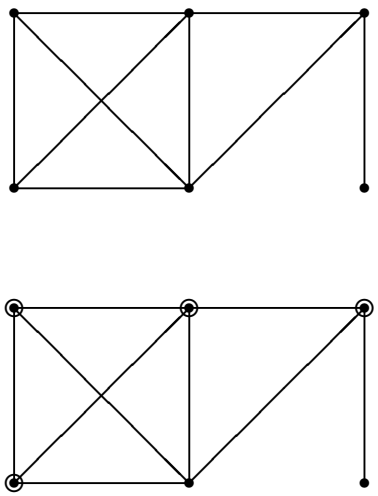
$$(x_1 \vee x_2 \vee \cdots \vee x_i \vee \cdots \vee x_{\ell-1} \vee x_\ell)$$

with $\ell \geq 4$ literals is replaced by

$$\begin{aligned}(x_1 \vee x_2 \vee y_1) \quad & \wedge (\neg y_1 \vee x_3 \vee y_2) \\ & \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \cdots \\ & \wedge (\neg y_{i-2} \vee x_i \vee y_{i-1}) \wedge \cdots \\ & \wedge (\neg y_{\ell-4} \vee x_{\ell-2} \vee y_{\ell-3}) \\ & \wedge (\neg y_{\ell-3} \vee x_{\ell-1} \vee x_\ell)\end{aligned}$$

The *vertex cover* problem (VC) is NP-complete.

Given a graph $G = (V, E)$ and an integer $j \leq |V|$, the problem is to determine if there exists a subset $V' \subseteq V$ such that $|V'| \leq j$ and such that, for each edge $(u, v) \in E$, either u, or $v \in V'$.



3-SAT $\propto$ VC

Instance of 3-SAT :

$$E_1 \wedge \cdots \wedge E_i \wedge \cdots \wedge E_k$$

Each E_i is of the form

$$x_{i1} \vee x_{i2} \vee x_{i3}$$

where x_{ij} is a literal. The set of propositional variables is

$$\mathcal{P} = \{p_1, \ldots, p_\ell\}.$$

The instance of VC that is built is then the following.

1. The set of vertices V contains

   (a) a pair of vertices labeled p_i and $\neg p_i$ for each propositional variable in $\mathcal{P}$,

   (b) a 3-tuple of vertices labeled x_{i1}, x_{i2}, x_{i3} for each clause E_i.

   The number of vertices is thus equal to $2\ell + 3k$.

2. The set of edges E contains

   (a) The edge $(p_i, \neg p_i)$ for each pair of vertices $p_i, \neg p_i$, $1 \leq i \leq \ell$,

   (b) The edges (x_{i1}, x_{i2}), (x_{i2}, x_{i3}) et (x_{i3}, x_{i1}) for each 3-tuple of vertices x_{i1}, x_{i2}, x_{i3}, $1 \leq i \leq k$,

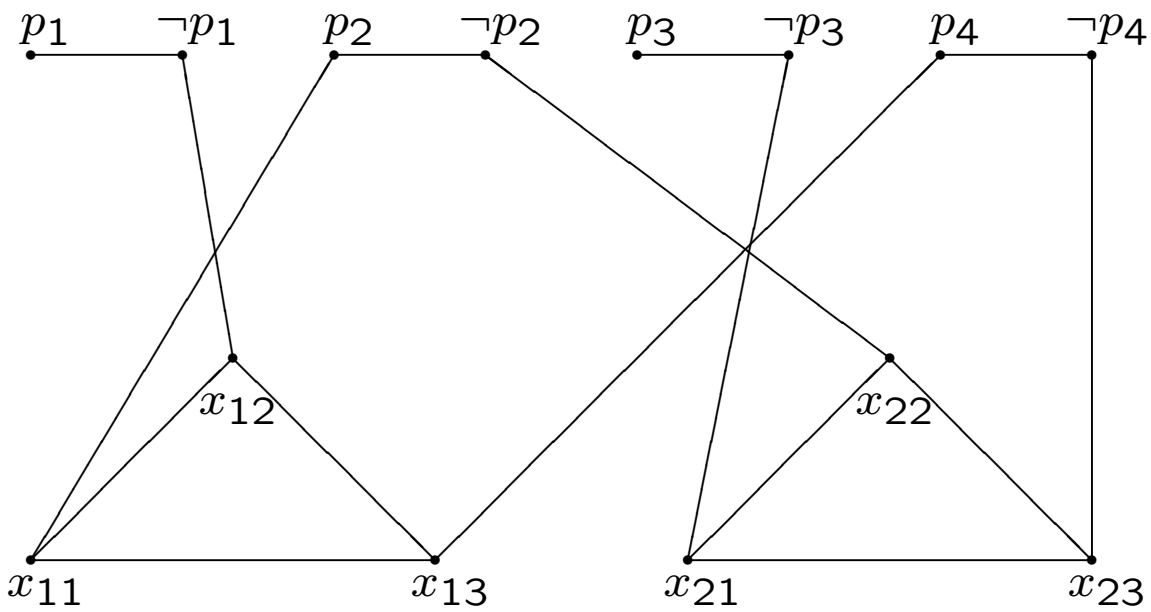
   (c) an edge between each vertex x_{ij} and the vertex p or $\neg p$ representing the corresponding literal.

   The number of edges is thus $\ell + 6k$.

3. The constant j is $\ell + 2k$.

Example

$$(p_2 \vee \neg p_1 \vee p_4) \wedge (\neg p_3 \vee \neg p_2 \vee \neg p_4)$$



Other examples

The Hamiltonian circuit (HC) and travelling salesman (TS) problems are NP-complete.

The *chromatic number* problem is NP-Complete. Given a graph G and a constant k this problem is to decide whether it is possible to colour the vertices of the graph with k colours in such a way that each pair of adjacent (edge connected) vertices are coloured differently.

The *integer programming* problem is NP-compete. An instance of this problem consists of

1. a set of m pairs $(\overline{v_i}, d_i)$ in which each $\overline{v_i}$ is a vector of integers of size n and each d_i is an integer,
2. a vector $\overline{d}$ of size n,
3. a constant b.

The problem is to determine if there exists an integer vector $\overline{x}$ of size n such that $\overline{x} \cdot \overline{v_i} \leq d_i$ for $1 \leq i \leq m$ and such that $\overline{x} \cdot \overline{d} \geq b$.

Over the rationals this problem can be solved in polynomial time (linear programming).

The problem of checking the equivalence of nondeterministic finite automata is NP-hard. Notice that there is no known NP algorithm for solving this problem. It is complete in the class PSPACE.

Interpreting NP-completeness

- Worst case analysis. Algorithms that are efficient "on average" are possible.
- Heuristic methods to limit the exponential number of cases that need to be examined.
- Approximate solutions for optimisation problems.
- The "usual" instances of problems can satisfy constraints that reduce to polynomial the complexity of the problem that actually has to be solved.

Other complexity classes

The class co-NP is the class of languages L whose complement $(\Sigma^* - L)$ is in NP.

The class EXPTIME is the class of languages decided by a deterministic Turing machine whose complexity function is bounded by an exponential function ($2^{p(n)}$ where $p(n)$ is a polynomial).

The class PSPACE is the class of languages decided by a deterministic Turing machine whose space complexity (the number of tape cells used) is bounded by a polynomial.

The class NPSPACE is the class of languages accepted by a nondeterministic Turing machine whose space complexity is bounded by a polynomial.

$$\mathsf{P} \subseteq \begin{matrix}\mathsf{NP}\\ \text{co-NP}\end{matrix} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}.$$