Introduction to computability

Pierre Wolper

Email: Pierre.Wolper@ulg.ac.be

URL: http: //www.montefiore.ulg.ac.be/~pw/
http: //www.montefiore.ulg.ac.be/
~pw/cours/calc.html

References

Pierre Wolper, *Introduction à la calculabilité - 3ième édition*, Dunod, 2006.

Michael Sipser, Introduction to the Theory of Computation, Second Edition, Course Technology, 2005

Chapter 1

Introduction

1.1 Motivation

- To understand the limits of computer science.
- To distinguish problems that are solvable by algorithms from those that are not.
- To obtain results that are independent of the technology used to build computers.

1.2 Problems and Languages

- Which problems can be solved by a program executed by a computer?

  We need to be more precise about:

  - the concept of problem,
  - the concept of program executed by a computer.

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.”

The halting problem

```
recursive function threen ($n$: integer):integer;
begin
if ($n = 1$) then 1
              else if even($n$) then threen($n \div 2$)
                                 else threen($3 \times n + 1$);
end;
```

1.3 Formalizing problems

How could one represent problem instances?

Alphabets and words

Alphabet : finite set of symbols.

Examples

- $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$
- $\{\alpha, \beta, \gamma\}$
- $\{1, 2, 3\}$
- $\{\clubsuit, \diamondsuit, \heartsuit\}$

Word on an alphabet : *finite* sequence of elements of the alphabet.

Examples

- *a, abs, zt, bbbssnbnzzyyyyddtrra, grosseguindaille* are words on the alphabet $\{a, \ldots, z\}$.

- $4\clubsuit3\diamondsuit5\heartsuit2\spadesuit, 12765, \clubsuit\heartsuit$ are words on the alphabet $\{0, \ldots, 8, \clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$.

Empty word: represented by e, ε, or λ.

Length of a word w : $|w|$

$w = aaabbaaaabb$
$w(1) = a$, $w(2) = a$,. . . , $w(11) = b$

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 3 subsets:

- *positive* instances: the answer is *yes* ;
- *negative* instances: the answer is *no*;
- words that do not represent an instance of the problem.

Alternatively:

- the words encoding instances of the problem for which the answer is *yes*, *the positive instances* ;

- the words that do not encode and instance of the problem, or that encode an instance for which the answer is *no* , *the negative instances*.

Languages

Language: set of words defined over the same alphabet.

Examples

- $\{aab, aaaa, \varepsilon, a, b, abababababbbbbbbbbbb\}$, $\{\varepsilon, aaaaaaa, a, bbbbbb\}$ 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.

1.4 Describing languages

Operations on languages

Let L_1 and L_2 be languages.

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

Regular Languages

The set $\mathcal{R}$ of regular languages over an alphabet Σ is the smallest set of languages such that:

1. $\emptyset \in \mathcal{R}$and $\{\varepsilon\} \in \mathcal{R}$,
2. $\{a\} \in \mathcal{R}$ for all $a \in \Sigma$, and
3. if $A, B \in \mathcal{R}$, then $A \cup B$, $A \cdot B$ and $A^* \in \mathcal{R}$.

Regular expressions

A notation for representing regular languages.

1. $\emptyset$, ε and the elements of Σ are regular expressions;

2. If α and β are regular expressions, then $(\alpha\beta)$, $(\alpha \cup \beta)$, $(\alpha)*$ are regular expressions.

The set of regular expressions is a language over the alphabet $\Sigma' = \Sigma \cup \{), (, \emptyset, \cup, *, \varepsilon\}$.

The language represented by a regular expression

1. $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$,

2. $L(a) = \{a\}$ pour tout $a \in \Sigma$,

3. $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$,

4. $L((\alpha\beta)) = L(\alpha) \cdot L(\beta)$,

5. $L((\alpha)*) = L(\alpha)^*$.

Theorem

A language is regular

if and only if

it can be represented by a regular expression.

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

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

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.

1.5 Languages that are not regular

Fact

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 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

$$
\begin{array}{c|cccccc}
 & a_0 & a_1 & a_2 & a_3 & a_4 & \cdots \\
\hline
s_0 & \times & \times & & \times & & \\
s_1 & \times & \square & & \times & & \\
s_2 & & \times & \times & & \times & \\
s_3 & \times & & \times & \square & & \\
s_4 & & \times & & \times & \square & \\
\vdots & & & & & &
\end{array}
$$

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

Conclusion

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

1.6 To follow . . .

- The notion of effective procedure (automata).
- Problems that cannot be solved by algorithms.
- Problems that cannot be solved efficiently.

Chapter 2
Finite Automata

2.1 Introduction

- 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.

- The finite state hypothesis and its consequences: finite or cyclic sequences of states.

- The problem of representing the data: only a finite number of different data sets can be represented since there exists only a finite number of initial states.

Representing data.

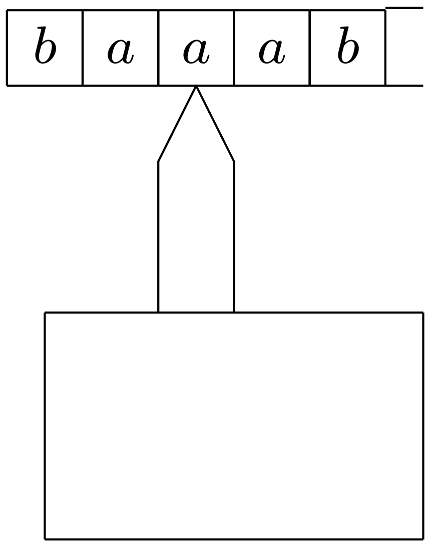
- Problem: to recognize a language.
- Data: a word.
- We will assume that the word is fed to the machine character by character, one character being handled at each cycle and the machine stopping once the last character has been read.

2.2 Description

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

tape :

head :



2.3 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,
- $\delta\ : Q \times \Sigma \rightarrow Q$ 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 $L(M)$:

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

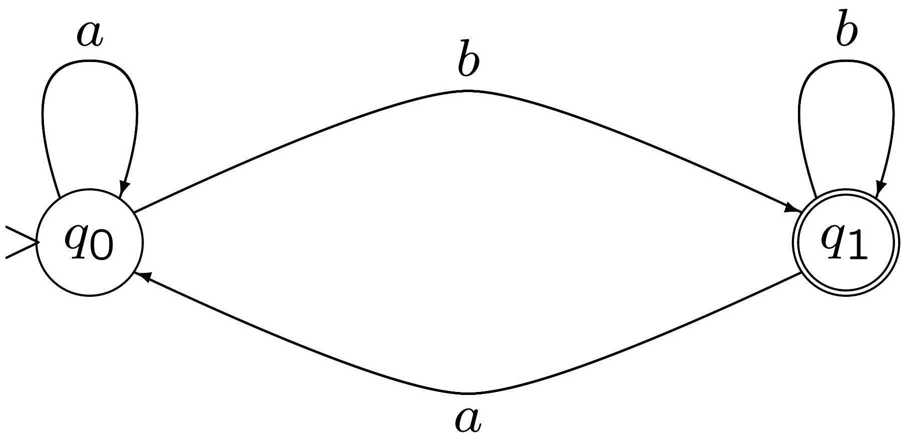
2.4 Examples

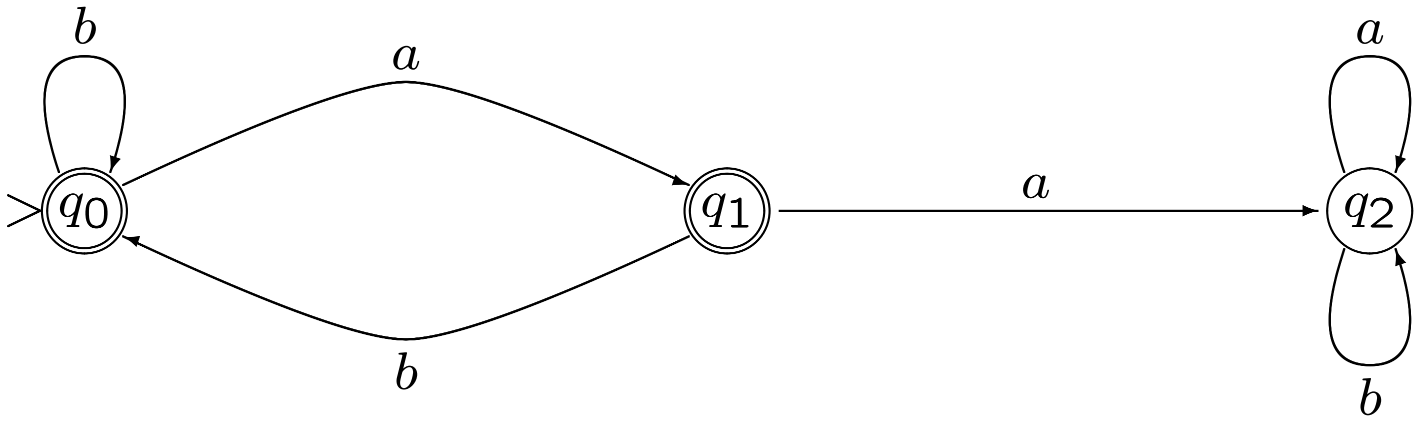
Words ending with b :

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$\}$.

2.5 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.

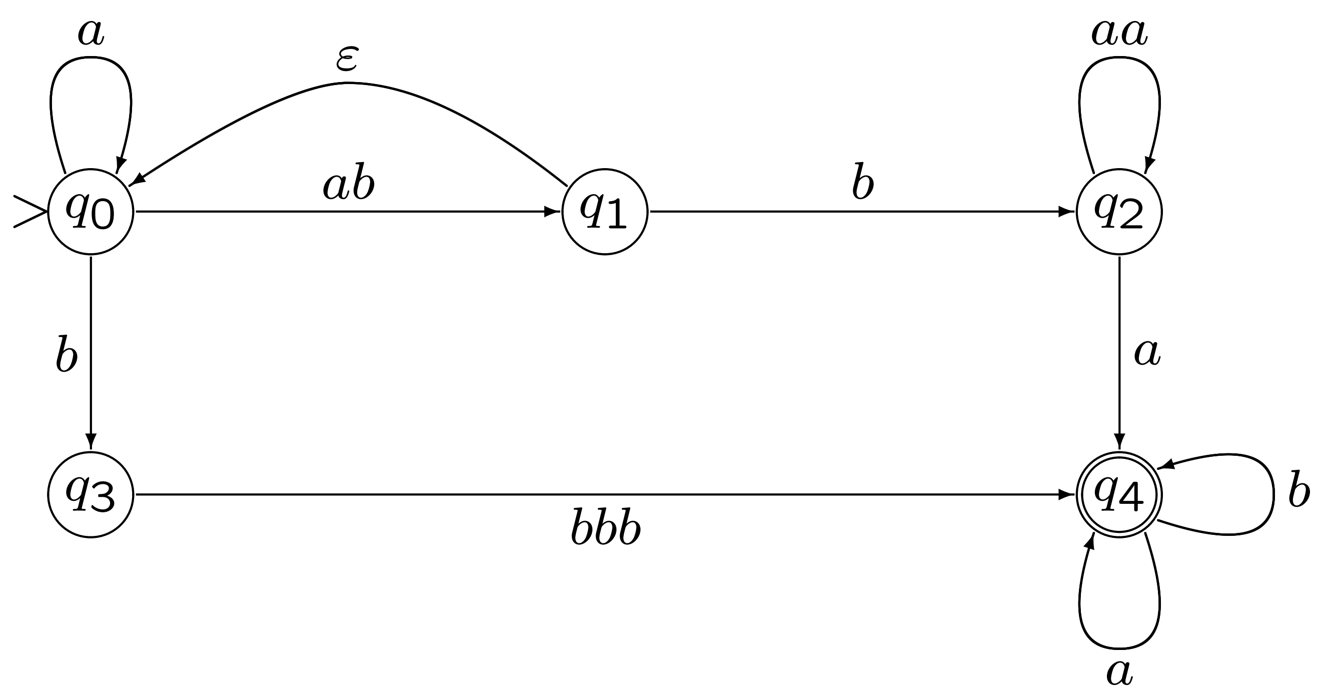
Defining the accepted language

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

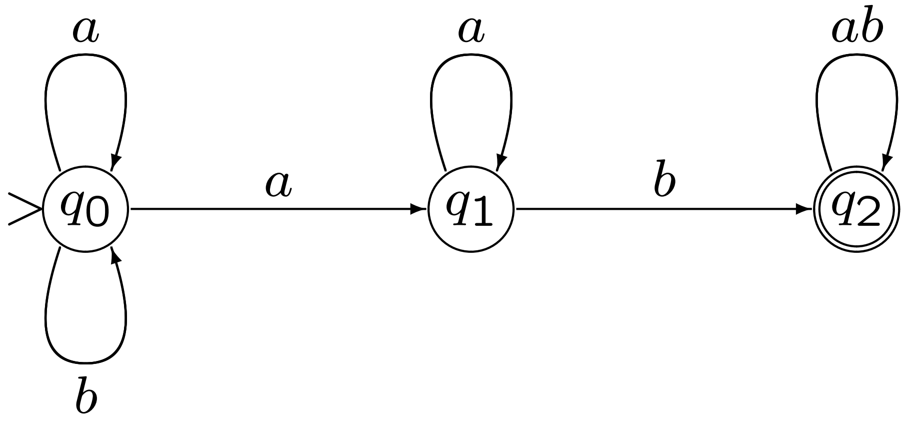
- $w = uw'$ (word w begins with a prefix $u \in \Sigma^*$),
- $(q, u, q') \in \Delta$ (the three-tuple (q, u, q') is in the transition relation Δ).

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

Examples



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



$$
\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "a"] \arrow[loop below, "b"] \arrow[r, "a"] & q_1 \arrow[loop above, "a"] \arrow[r, "b"] & q_2 \arrow[loop above, "ab"]
\end{tikzcd}
$$

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

Words ending with at least one repetition of ab.

2.6 Eliminating non determinism

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

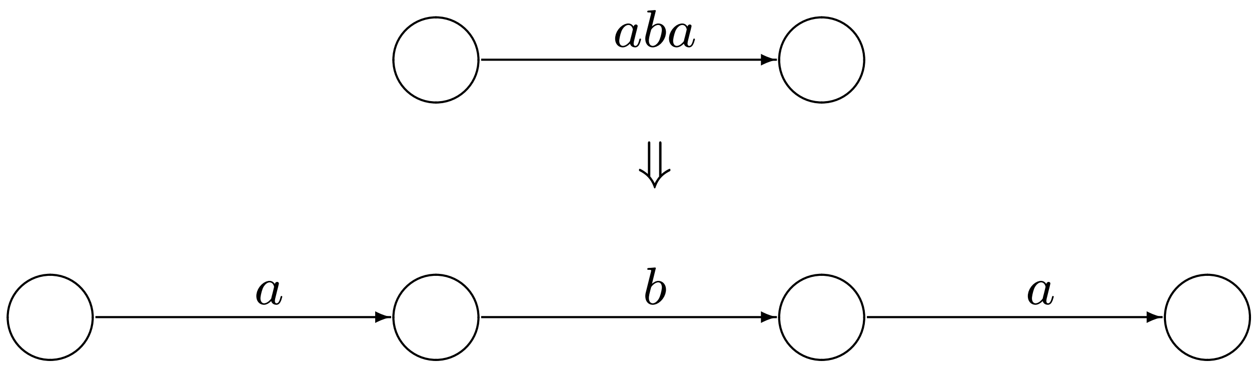
Theorem
Given any nondeterministic finite automaton, it is possible to build an equivalent deterministic finite automaton.

2.6 Idea of the construction

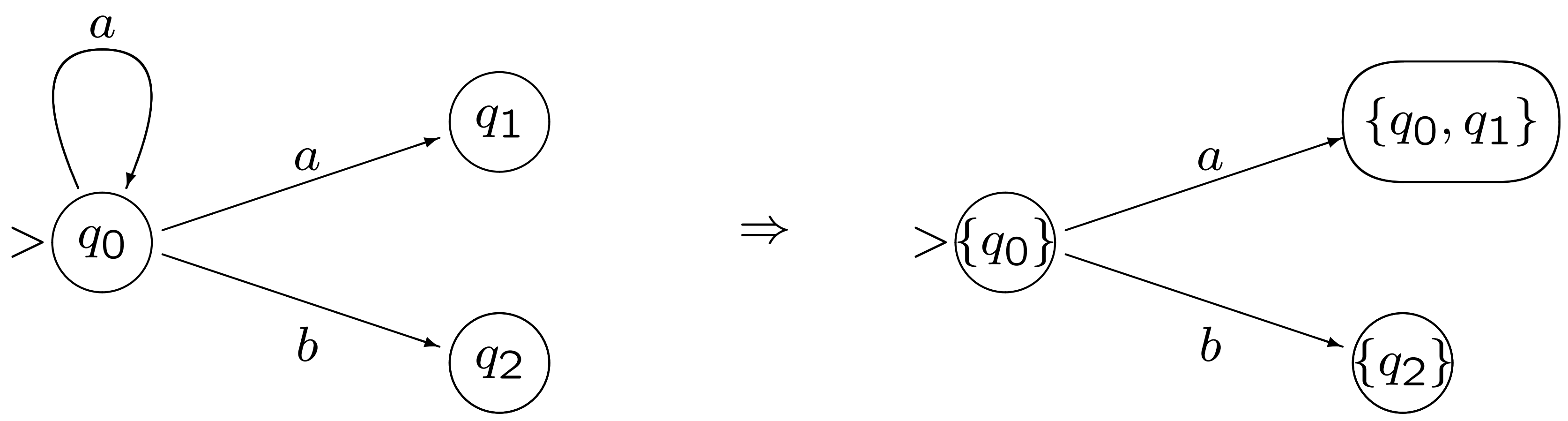
1. Eliminate transitions of length greater than 1.

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

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

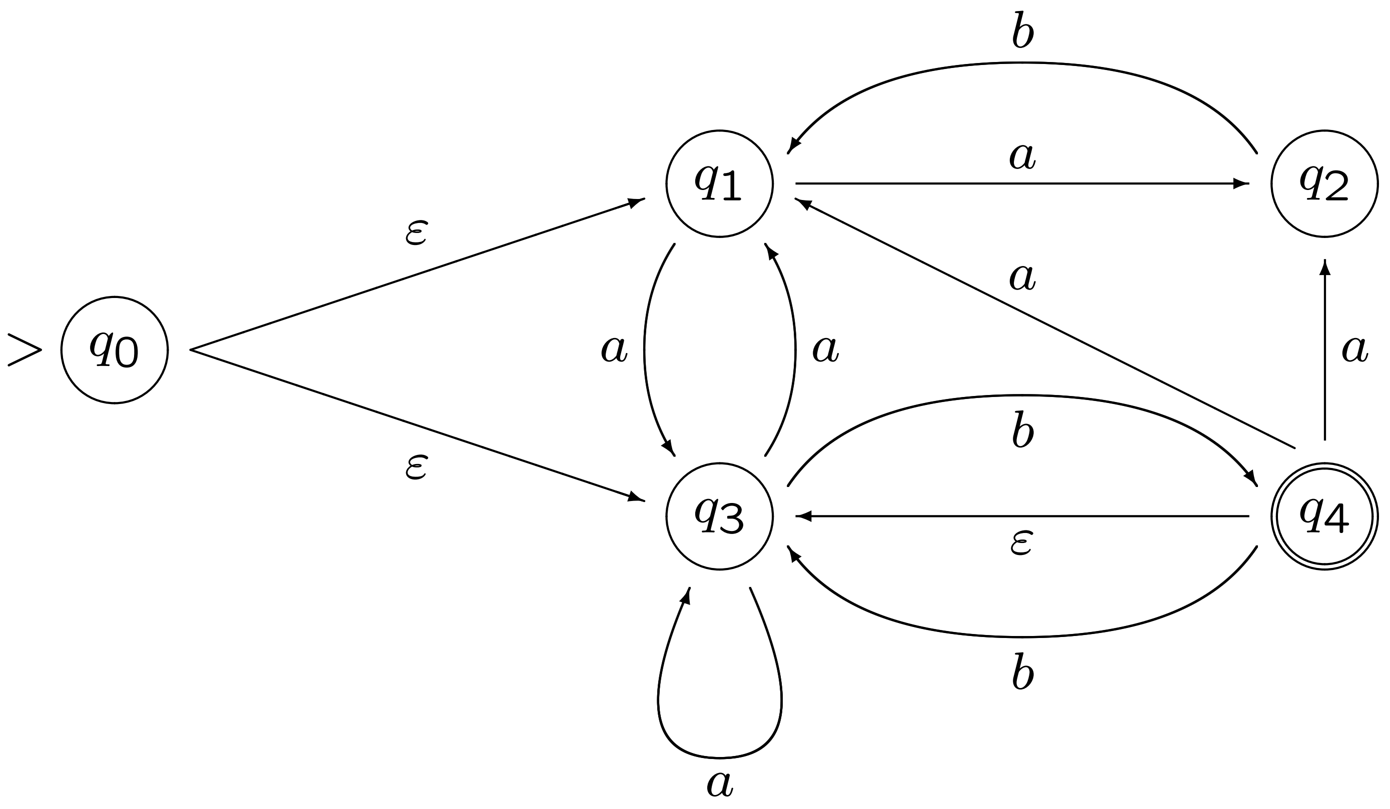
Formalization

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



$$\begin{tikzcd}
 & & \{q_1,q_2,q_3\} \arrow[loop above, "a"] \arrow[r, shift left, "b"] & \{q_1,q_3,q_4\} \arrow[l, shift left, "a"] \arrow[ddl, "b"] \\
{} \arrow[r] & \{q_0,q_1,q_3\} \arrow[ur, "a"] \arrow[dr, "b"'] & & \\
 & & \{q_3,q_4\} \arrow[uu, "a"'] \arrow[loop below, "b"] &
\end{tikzcd}$$

Alternative construction: Only accessible states

1. Initially Q' contains the initial state s'.

2. The following operations are then repeated until the set of states Q' is no longer modified.

   (a) Choose a state $\mathbf{q} \in Q'$ to which the operation has not yet been applied.

   (b) For each letter $a \in \Sigma$ compute the state $\mathbf{p}$ such that $\mathbf{p} = \delta'(\mathbf{q}, a)$. The state $\mathbf{p}$ is added to Q'.

2.7 Finite automata and regular expressions

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

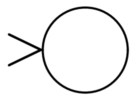
We will prove:

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

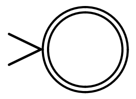
2. If a language is accepted by a non deterministic finite automaton, it is regular.

From expressions to automata

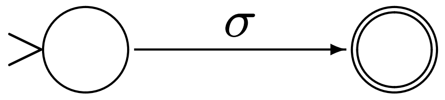
- $\emptyset$



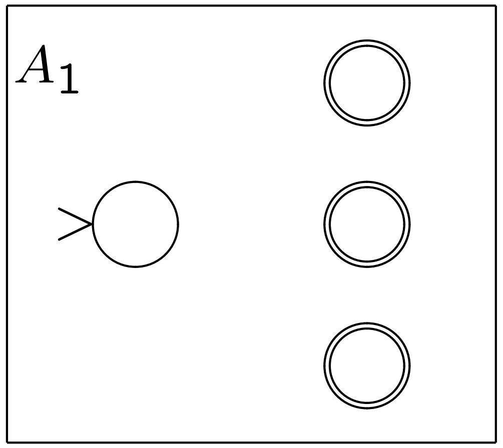
- ε

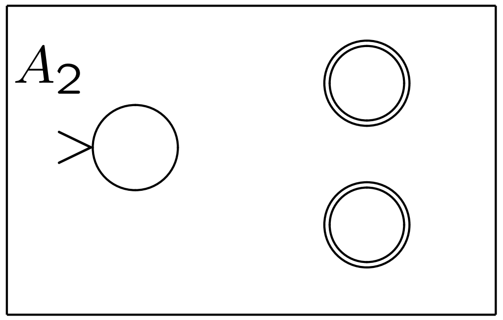


- $\sigma \in \Sigma$

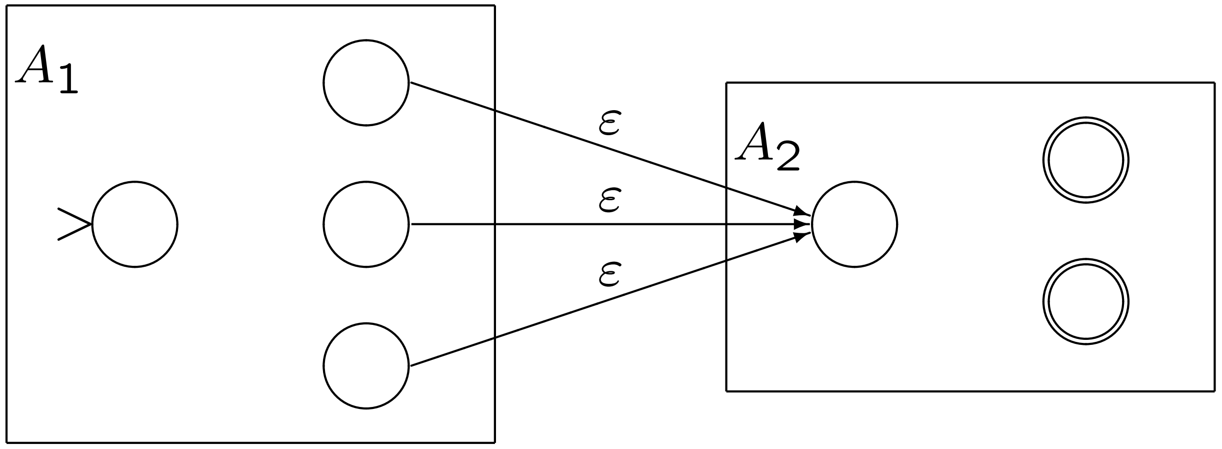


$$\alpha_1 : A_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$$
$$\alpha_2 : A_2 = (Q_2, \Sigma, \Delta_2, s_2, F_2)$$





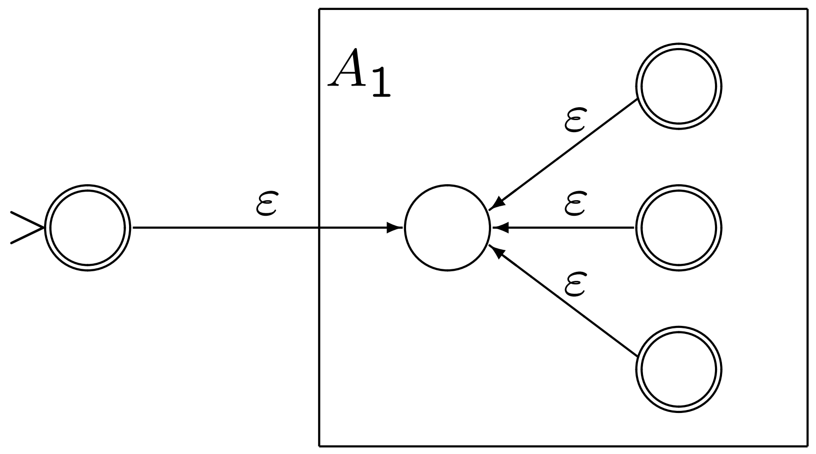
- $\alpha_1 \cdot \alpha_2$



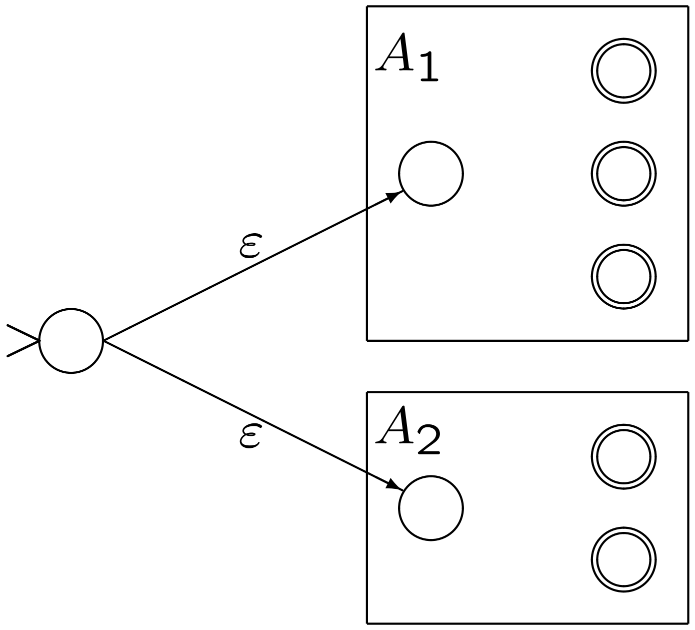
Formally, $A = (Q, \Sigma, \Delta, s, F)$ où

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

- $\alpha = \alpha_1^*$



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



From automata to regular languages

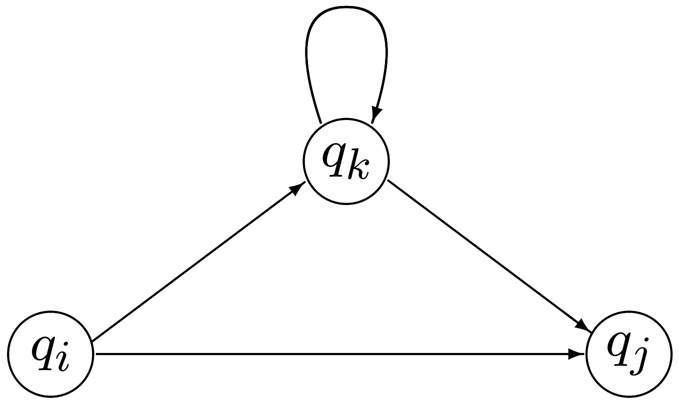
Intuitive idea:

- Build a regular expression for each path from the initial state to an accepting state.
- Use the $*$ operator to handle loops.

Definition
Let M by 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 $\{q_1, \ldots, q_{k-1}\}$.

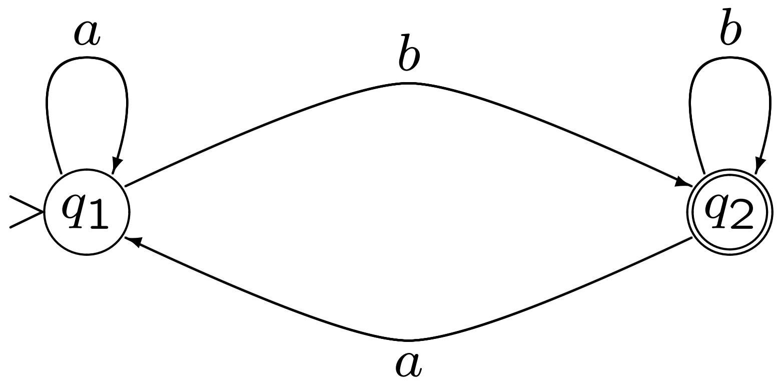
$$R(i,j,1) = \begin{cases} \{w \mid (q_i, w, q_j) \in \Delta\} & \text{si } i \neq j \\ \{\varepsilon\} \cup \{w \mid (q_i, w, q_j) \in \Delta\} & \text{si } i = j \end{cases}$$

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



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

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

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

Chapter 3

Regular grammars

3.1 Introduction

Other view of the concept of language:

- not the formalization of the notion of effective procedure,
- but set of words satisfying a given set of rules
- Origin : formalization of natural language.

Example

- a `phrase` is of the form `subject verb`
- a `subject` is a `pronoun`
- a `pronoun` is **he** or **she**
- a verb is **sleeps** or **listens**

Possible phrases:

1. **he listens**
2. **he sleeps**
3. **she sleeps**
4. **she listens**

Grammars

- Grammar: *generative* description of a language
- Automaton: *analytical* description
- Example: programming languages are defined by a grammar (BNF), but recognized with an analytical description (the parser of a compiler),
- Language theory establishes links between analytical and generative language descriptions.

3.2 Grammars

A grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where

- V is an alphabet,
- $\Sigma \subseteq V$ is the set *terminal symbols* ($V - \Sigma$ is the set of *nonterminal symbols*),
- $R \subseteq (V^+ \times V^*)$ is a finite set of *production rules* (also called simply rules or productions),
- $S \in V - \Sigma$ is the *start symbol.*

Notation:

- Elements of $V - \Sigma$: $A, B, \ldots$
- Elements of Σ : $a, b, \ldots$.
- Rules $(\alpha, \beta) \in R$: $\alpha \rightarrow \beta$ or $\alpha \underset{G}{\rightarrow} \beta$.
- The start symbol is usually written as S.
- Empty word: ε.

Example :

- $V = \{S, A, B, a, b\}$,
- $\Sigma = \{a, b\}$,
- $R = \{S \to A, S \to B, B \to bB, A \to aA, A \to \varepsilon, B \to \varepsilon\}$,
- S is the start symbol.

Words generated by a grammar: example

$aaaa$ is in the language generated by the grammar we have just described:

S		
A	rule	$S \longrightarrow A$
aA		$A \longrightarrow aA$
aaA		$A \longrightarrow aA$
$aaaA$		$A \longrightarrow aA$
$aaaaA$		$A \longrightarrow aA$
$aaaa$		$A \longrightarrow \varepsilon$

Generated words: definition

Let $G = (V, \Sigma, R, S)$ be a grammar and $u \in V^+$, $v \in V^*$ be words. The word v *can be derived in one step from* u by G (notation $u \underset{G}{\Rightarrow} v$) if and only if:

- $u = xu'y$ (u can be decomposed in three parts x, u' et y ; the parts x and y being allowed to be empty),
- $v = xv'y$ (v can be decomposed in three parts x, v' et y),
- $u' \underset{G}{\rightarrow} v'$ (the rule (u', v') is in R).

Let $G = (V, \Sigma, R, S)$ et $u \in V^+$, $v \in V^*$ be a grammar. The word v *can be derived in several steps from* u (notation $u \overset{*}{\underset{G}{\Rightarrow}} v$) if and only if
$\exists k \geq 0$ et $v_0 \ldots v_k \in V^+$ such that

- $u = v_0$,
- $v = v_k$,
- $v_i \underset{G}{\Rightarrow} v_{i+1}$ for $0 \leq i < k$.

- Words generated by a grammar G: words $v \in \Sigma^*$ (containing only terminal symbols) such that

$$S \underset{G}{\overset{*}{\Rightarrow}} v.$$

- The language generated by a grammar G (written $L(G)$) is the set

$$L(G) = \{v \in \Sigma^* \mid S \underset{G}{\overset{*}{\Rightarrow}} v\}.$$

Example :
The language generated by the grammar shown in the example above is the set of all words containing either only a's or only b's.

Types of grammars

Type 0: no restrictions on the rules.

Type 1: *Context sensitive* grammars.

The rules

$$\alpha \to \beta$$

satisfy the condition

$$|\alpha| \leq |\beta|.$$

Exception: the rule

$$S \to \varepsilon$$

is allowed as long as the start symbol S does not appear in the right hand side of a rule.

Type 2: *context-free* grammars.

Productions of the form

$$A \rightarrow \beta$$

where $A \in V - \Sigma$ and there is no restriction on β.

Type 3: *regular* grammars.

Productions rules of the form

$$\begin{array}{l} A \rightarrow wB \\ A \rightarrow w \end{array}$$

where $A, B \in V - \Sigma$ et $w \in \Sigma^*$.

3.3 Regular grammars

Theorem:
A language is regular if and only if it can be generated by a regular grammar.

A. If a language is regular, it can be generated by a regular grammar.

If L is regular, there exists

$$M = (Q, \Sigma, \Delta, s, F)$$

such that $L = L(M)$. From M, one can easily construct a regular grammar

$$G = (V_G, \Sigma_G, S_G, R_G)$$

generating L.

G is defined by:

- $\Sigma_G = \Sigma$,
- $V_G = Q \cup \Sigma$,
- $S_G = s$,
- $R_G = \left\{ \begin{array}{ll} A \rightarrow wB, & \text{for all} (A, w, B) \in \Delta \\ A \rightarrow \varepsilon & \text{for all} A \in F \end{array} \right\}$

B. If a language is generated by a regular grammar, it is regular.

Let

$$G = (V_G, \Sigma_G, S_G, R_G)$$

be the grammar generating L. A nondeterministic finite automaton accepting L can be defined as follows:

- $Q = V_G - \Sigma_G \cup \{f\}$ (the states of M are the nonterminal symbols of G to which a new state f is added),
- $\Sigma = \Sigma_G$,
- $s = S_G$,
- $F = \{f\}$,
- $\Delta = \left\{ \begin{array}{ll} (A, w, B), & \text{for all} A \rightarrow wB \in R_G \\ (A, w, f), & \text{for all} A \rightarrow w \in R_G \end{array} \right\}.$

3.4 The regular languages

We have seen four characterizations of the regular languages:

1. regular expressions,
2. deterministic finite automata,
3. nondeterministic finite automata,
4. regular grammars.

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.

$L_1 \cap L_2$ regular ?

$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$$

Alternatively, if $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ accepts L_1 and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ accepts L_2, the following automaton, accepts $L_1 \cap L_2$:

- $Q = Q_1 \times Q_2$,
- $\delta((q_1, q_2), \sigma) = (p_1, p_2)$ if and only is $\delta_1(q_1, \sigma) = p_1$ et $\delta_2(q_2, \sigma) = p_2$,
- $s = (s_1, s_2)$,
- $F = F_1 \times F_2$.

- 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 $w = w_1 \ldots w_k \in \Sigma^*$, $\pi(w) = \pi(w_1) \ldots \pi(w_k)$.

  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:

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

3.5 Beyond regular languages

- Many languages are regular,
- But, all languages cannot be regular for cardinality reasons.
- We will now prove, using another techniques that some specific languages are not regular.

Basic Observations

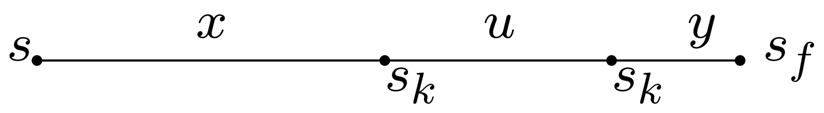
1. All finite languages (including only a finite number of words) are regular.

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 and 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 $xu^ny \in L \ \forall n \geq 0$.

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 $u \neq \varepsilon$ et $|xu| \leq |Q|$ such that $xuy = w$ and, $\forall n$, $xu^ny \in L$.

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 et 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 et 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

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

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.

Applications of regular languages II: handling arithmetic

- A number written in base r is a word over the alphabet $\{0, \ldots, r-1\}$ ($\{0, \ldots, 9\}$ in decimal, $\{0, 1\}$ en binary).

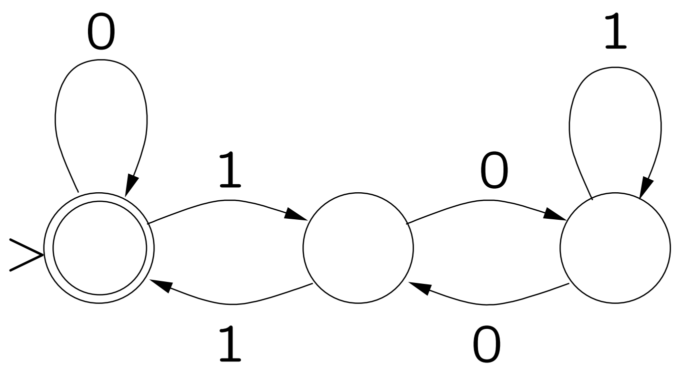
- The number represented by a word $w = w_0 \ldots w_l$ is $nb(w) = \sum_{i=0}^{l} r^{l-i} nb(w_i)$

- Adding leading 0's to the representation of a number does not modify the represented value. A number thus has a infinite number of possible representations. Number encodings are read most significant digit first, and all possible encodings will be taken into account.

- **Exemple:** The set of binary representations of 5 is the language 0^*101.

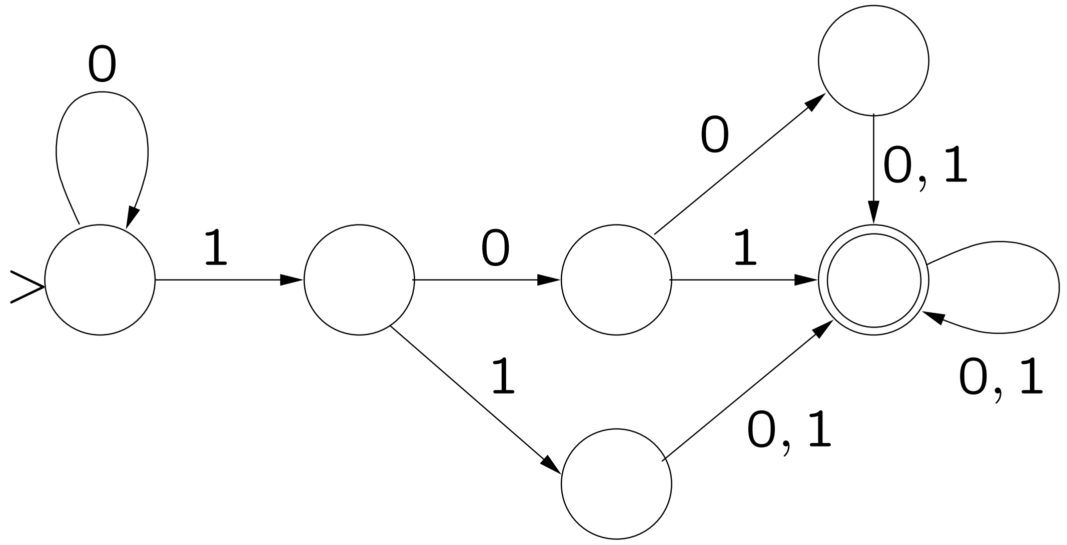
Which sets of numbers can be represented by regular languages?

- Finite sets.
- The set of multiples of 2 is represented by the language $(0 \cup 1)^*0$.
- The set of powers of 2 is represented by the language 0^*10^*, but is not representable in base 3.
- The set of multiples of 3 is represented by the following automaton.



Set of numbers represented by regular languages (continued)

- The set of numbers $x \geq 5$ is represented by the automatoné



- More generally, one can represent sets of the form $\{ax \mid x \in N\}$ or $\{x \geq a \mid x \in N\}$ pour any given value of a.

Set of numbers represented by regular languages (continued II)

- Combining the two types of sets: sets of the form $\{ax + b \mid x \in N\}$, for any given a and b.

- Union of such sets: the *ultimately periodic sets*.

- Intersection and complementation add nothing more.

- The only sets that can be represented in all bases are the ultimately periodic sets.

Representing vectors of numbers

- Each number is represented by a word, and bits in identical positions are read together.

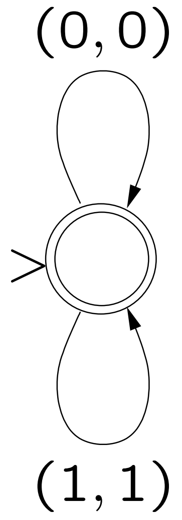
- **Example:**

  - the vector $(5, 9)$ is encoded by the word $(0, 1)(1, 0)(0, 0)(1, 1)$ defined over the alphabet $\{0, 1\} \times \{0, 1\}$.

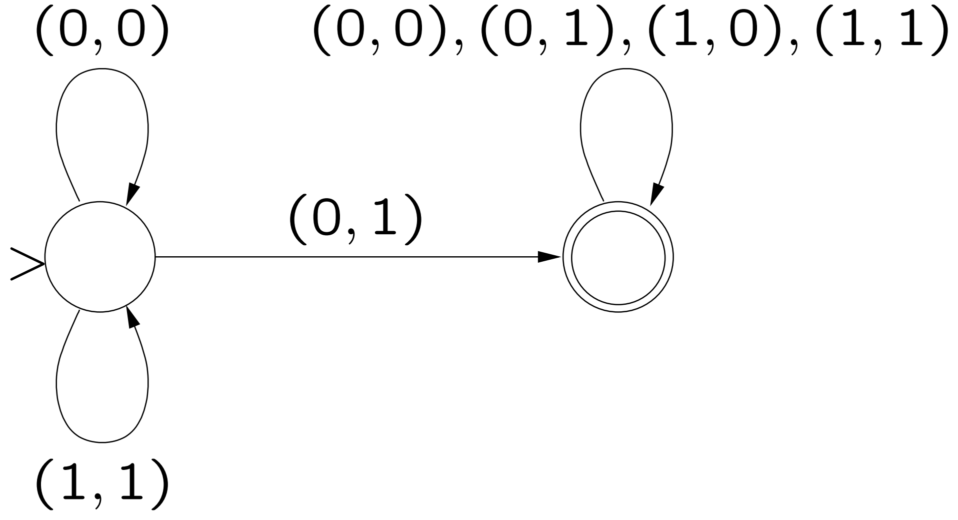
  - The set of binary encodings of the vector $(5, 9)$ is $(0, 0)^*(0, 1)(1, 0)$ $(0, 0)(1, 1)$.

Which sets of number vectors can be represented by regular languages?

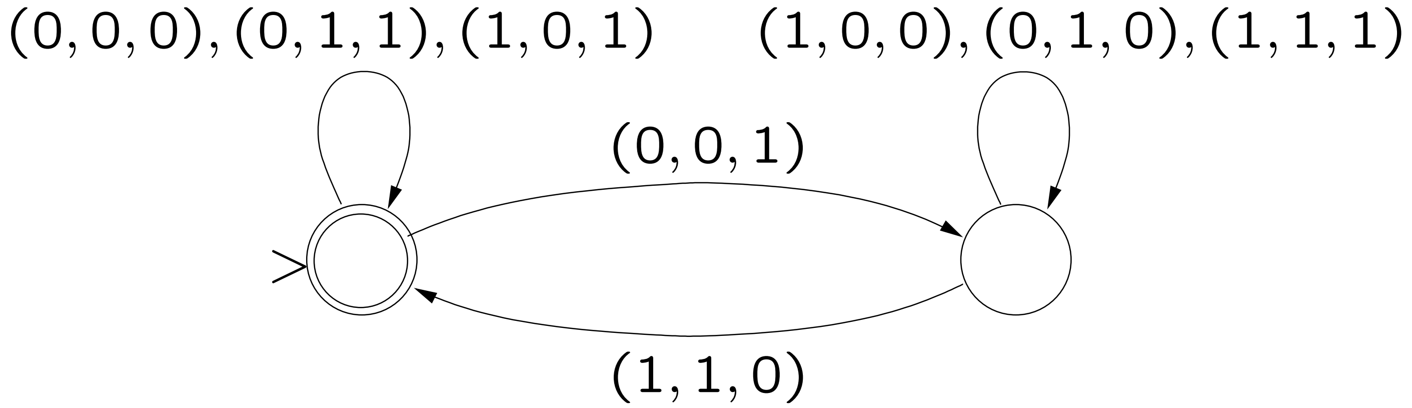
- The set of binary encodings of the vectors (x, y) such that $x = y$ is accepted by the automaton



- Vectors (x, y) such that $x < y$

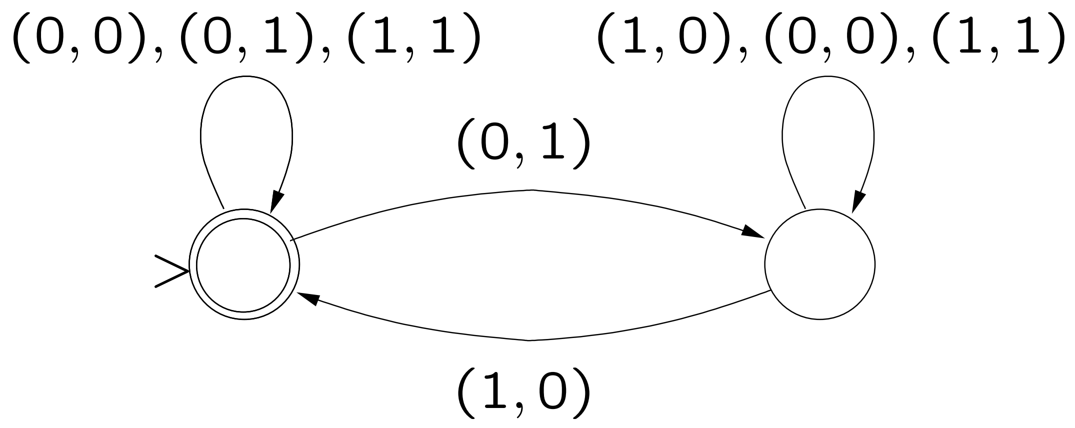


- Three-dimentional vectors (x, y, z) such that $z = x + y$

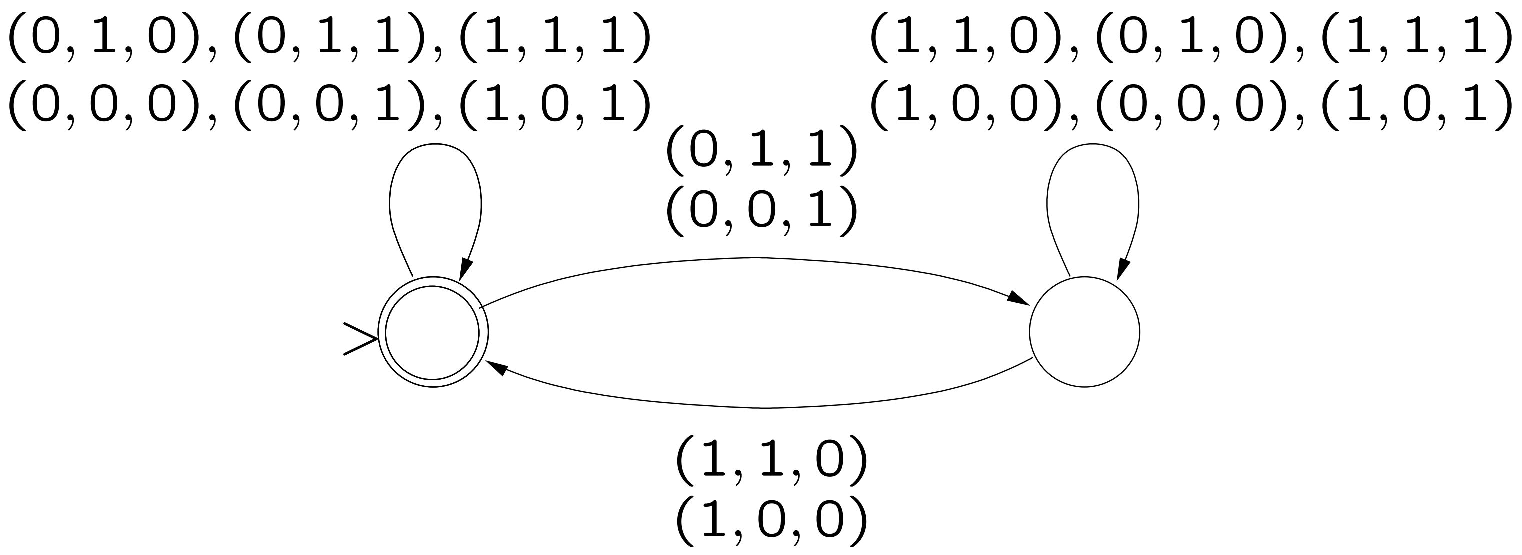


Definable sets of number vectors (continued)

- Intersection, union, complement of representable sets (closure properties of regular languages).
- Modifying the number of dimensions: projection and the inverse operation.
- **Remark:** projection does not always preserve the determinism of the automaton.
- **Example:** $\{(x, z) \mid \exists y\ x + y = z\}$ $(x \leq z)$.



- Adding a dimension to the previous automaton yields



- which is not equivalent to the automaton to which projection was applied.

Representable sets of vectors: conclusions

- Linear equality and inequality constraints
- **Example:** an automaton for $x + 2y = 5$ can be obtained by combing the automata for the following constraints:

$$\begin{aligned} z_1 &= y \\ z_2 &= y + z_1 \\ z_3 &= x + z_2 \\ z_3 &= 5. \end{aligned}$$

- There exists also a more direct construction.

Representable vector sets: conclusions (continued)

- Boolean combinations of linear constraints
- Existential quantification can be handled with projection ($\exists x$).
- For universal quantification, one uses $\forall x f \equiv \neg \exists \neg f$
- **Example:** It is possible to build an automaton accepting the representations of the vectors (x, y) satisfying the arithmetic constraint

$$\forall u \exists t[(2x + 3y + t - 4u = 5) \vee (x + 5y - 3t + 2u = 8)]$$

- This is Presburger arithmetic, which corresponds exactly to the sets representable by automata in all bases.

Chapter 4
Pushdown automata and context-free languages

Introduction

- The language a^nb^n cannot be accepted by a finite automaton
- On the other hand, $L_k = \{a^nb^n \mid n \leq k\}$ is accepted for any given n.
- Finite memory, infinite memory, extendable memory.
- Pushdown (stack) automata: LIFO memory.

4.1 Pushdown automata

- Input tape and read head,
- finite set of states, among which an initial state and a set of accepting states,
- a transition relation,
- an unbounded pushdown stack.

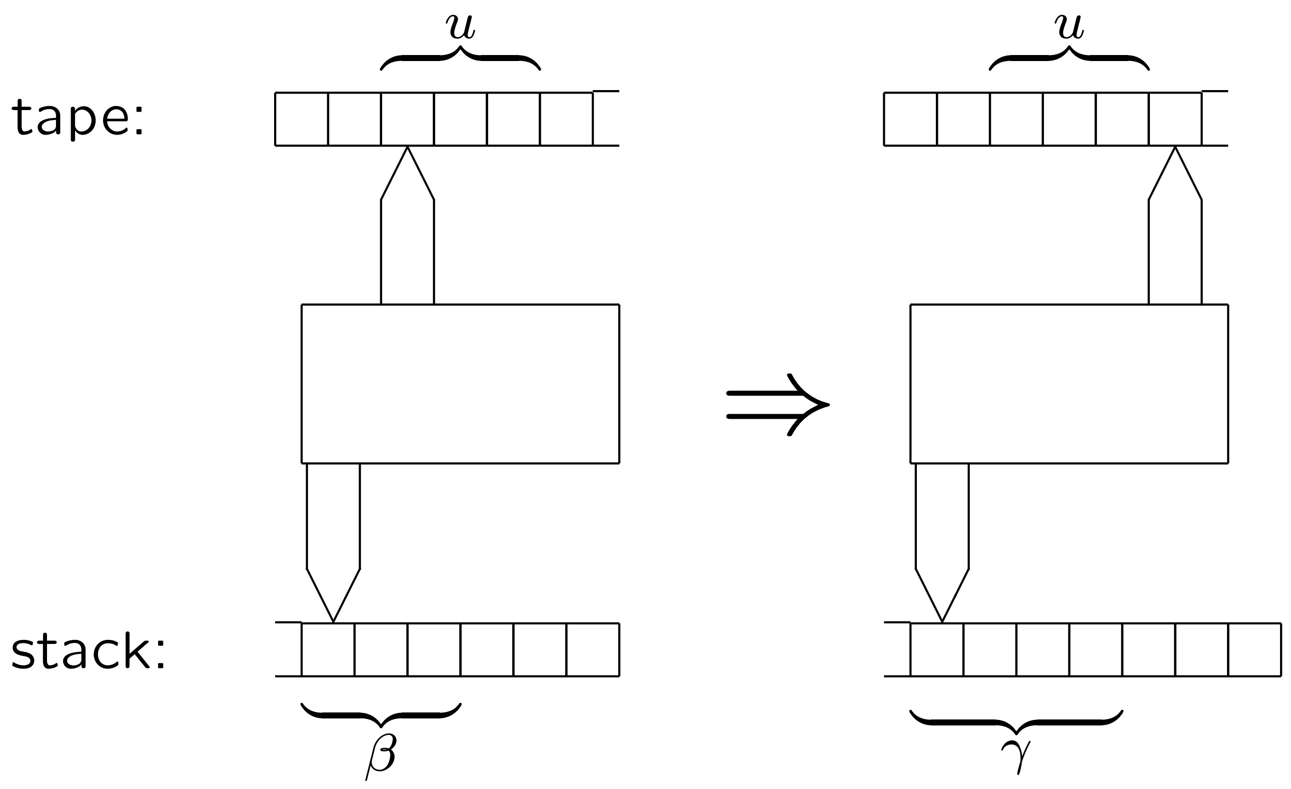
Formalization

7-tuple $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$, where

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

Transitions

$$((p, u, \beta), (q, \gamma)) \in \Delta$$



Executions

The configuration (q', w', α') is *derivable in one step* from the configuration (q, w, α) by the machine M (notation $(q, w, \alpha) \vdash_M (q', w', \alpha')$) if

- $w = uw'$ (the word w starts with the prefix $u \in \Sigma^*$),
- $\alpha = \beta\delta$ (before the transition, the top of the stack read from left to right contains $\beta \in \Gamma^*$),
- $\alpha' = \gamma\delta$ (after the transition, the part β of the stack has been replaced by γ, the first symbol of γ is now the top of the stack),
- $((q, u, \beta), (q', \gamma)) \in \Delta$.

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

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

An *execution of a pushdown automaton* on a word w is a sequence of configurations

$$(s, w, Z) \vdash (q_1, w_1, \alpha_1) \vdash \cdots \vdash (q_n, \varepsilon, \gamma)$$

where s is the initial state, Z is the initial stack symbol, and ε represents the empty word.

A word w is *accepted* by a pushdown automaton $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ if

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

Examples

$$\{a^n b^n \mid n \geq 0\}$$

- $Q = \{s, p, q\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{A\}$,
- $F = \{q\}$ and Δ contains the transitions

$$(s, a, \varepsilon) \to (s, A)$$
$$(s, \varepsilon, Z) \to (q, \varepsilon)$$
$$(s, b, A) \to (p, \varepsilon)$$
$$(p, b, A) \to (p, \varepsilon)$$
$$(p, \varepsilon, Z) \to (q, \varepsilon)$$

The automaton $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ described below accepts the language

$$\{ww^R\}$$

- $Q = \{s, p, q\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{A, B\}$,
- $F = \{q\}$ and Δ contains the transitions

$$(s, a, \varepsilon) \to (s, A)$$
$$(s, b, \varepsilon) \to (s, B)$$
$$(s, \varepsilon, \varepsilon) \to (p, \varepsilon)$$
$$(p, a, A) \to (p, \varepsilon)$$
$$(p, b, B) \to (p, \varepsilon)$$
$$(p, \varepsilon, Z) \to (q, \varepsilon)$$

Context-free languages

Definition:
A language is context-free if there exists a context-free grammar that can generate it.

Examples

The language $a^n b^n$, $n \geq 0$, is generated by the grammar whose rules are

1. $S \rightarrow aSb$
2. $S \rightarrow \varepsilon$.

The language containing all words of the form ww^R is generated by the grammar whose productions are

1. $S \to aSa$
2. $S \to bSb$
3. $S \to \varepsilon$.

The language generated by the grammar whose productions are

1. $S \rightarrow \varepsilon$
2. $S \rightarrow aB$
3. $S \rightarrow bA$
4. $A \rightarrow aS$
5. $A \rightarrow bAA$
6. $B \rightarrow bS$
7. $B \rightarrow aBB$

is the language of the words that contain the same number of a's and b's in any order

Relation with pushdown automata

Theorem
A language is context-free if and only if it is accepted by a pushdown automaton.

Properties of context-free languages

Let L_1 and L_2 be two context-free languages.

- The language $L_1 \cup L_2$ is context-free.
- Le language $L_1 \cdot L_2$ is context-free.
- L_1^* is context-free.
- $L_1 \cap L_2$ and $\overline{L_1}$ are not necessarily context-free!
- If L_R is a regular language and if the language L_2 is context-free, then $L_R \cap L_2$ is context-free.

Let $M_R = (Q_R, \Sigma_R, \delta_R, s_R, F_R)$ be a deterministic finite automaton accepting L_R and let $M_2 = (Q_2, \Sigma_2, \Gamma_2, \Delta_2, Z_2, s_2, F_2)$ be a pushdown automaton accepting the language L_2. The language $L_R \cap L_2$ is accepted by the pushdown automaton $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ for which

- $Q = Q_R \times Q_2$,
- $\Sigma = \Sigma_R \cup \Sigma_2$,
- $\Gamma = \Gamma_2$,
- $Z = Z_2$,
- $s = (s_R, s_2)$,
- $F = (F_R \times F_2)$,

- $(((q_R, q_2), u, \beta), ((p_R, p_2), \gamma)) \in \Delta$ if and only if

  $(q_R, u) \vdash^*_{M_R} (p_R, \varepsilon)$ (the automaton M_R can move from the state q_R to the state p_R, while reading the word u, this move being done in one or several steps) and

  $((q_2, u, \beta), (p_2, \gamma)) \in \Delta_2$ (The pushdown automaton can move from the state q_2 to the state p_2 reading the word u and replacing β by γ on the stack).

4.3 Beyond context-free languages

- There exist languages that are not context-free (for cardinality reasons).

- We would like to show that some specific languages are not context-free.

- For this, we are going to prove a from of pumping lemma.

- This requires a more abstract notion of derivation.

Example

1. $S \rightarrow SS$
2. $S \rightarrow aSa$
3. $S \rightarrow bSb$
4. $S \rightarrow \varepsilon$

Generation of $aabaab$:

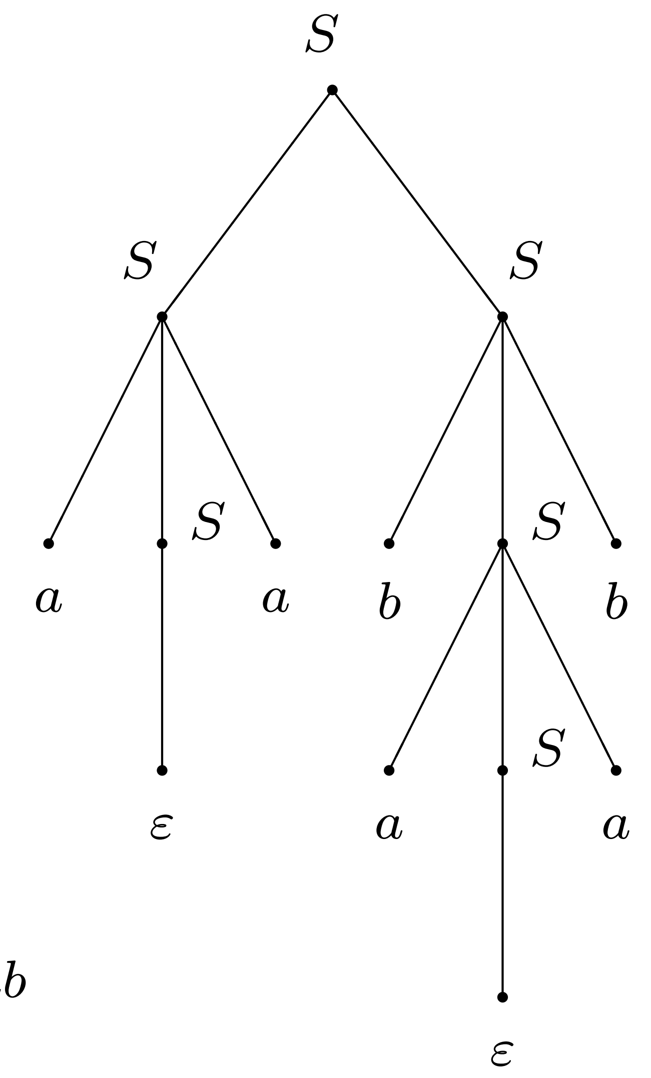
$$\begin{gathered} S \Rightarrow SS \Rightarrow aSaS \Rightarrow aaS \\ \Rightarrow aabSb \Rightarrow aabaSab \Rightarrow aabaab \end{gathered}$$

$$\begin{gathered} S \Rightarrow SS \Rightarrow SbSb \Rightarrow SbaSab \\ \Rightarrow Sbaab \Rightarrow aSabaab \Rightarrow aabaab \end{gathered}$$

and 8 other ways.

We need a representation of derivations that abstract from the order in which production rules are applied.

The notion of parse tree



Parse tree for $aabaab$

Definition

A parse tree for a context-free grammar $G = (V, \Sigma, R, S)$ is a tree whose nodes are labeled by elements of $V \cup \varepsilon$ and that satisfies the following conditions.

- The root is labeled by the start symbol S.
- Each interior node is labeled by a non-terminal.
- Each leaf is labeled by a terminal symbol or by ε.

- For each interior node, if its label is the non-terminal A and if its direct successors are the nodes $n_1, n_2, \ldots, n_k$ whose labels are respectively $X_1, X_2, \ldots, X_k$, then

$$A \rightarrow X_1 X_2 \ldots X_k$$

  must be a production of G.

- If a node is labeled by ε, then this node must be the only successor of its immediate predecessor (this last constraints aims only at preventing the introduction of unnecessary copied of ε in the parse tree).

Generated word

The word generated by a parse tree is the one obtained by concatenating its leaves from left to right

Theorem
Given a context-free grammar G, a word w is generated by G ($S \overset{*}{\underset{G}{\Rightarrow}} w$) if and only if there exists a parse tree for the grammar G that generates w.

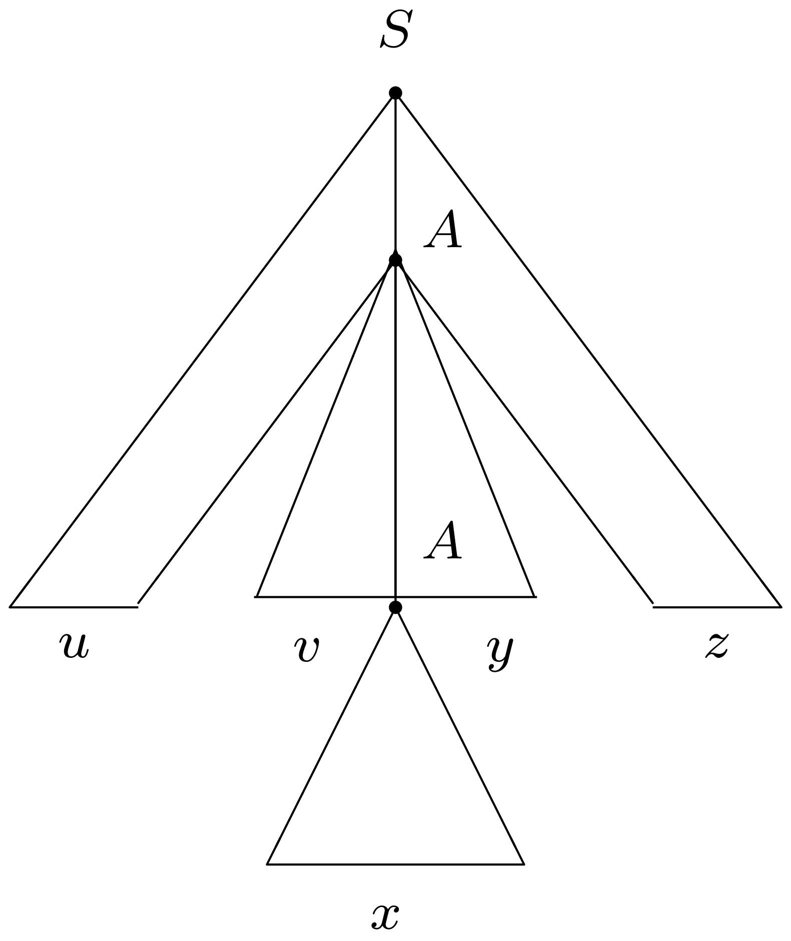
Le pumping lemma

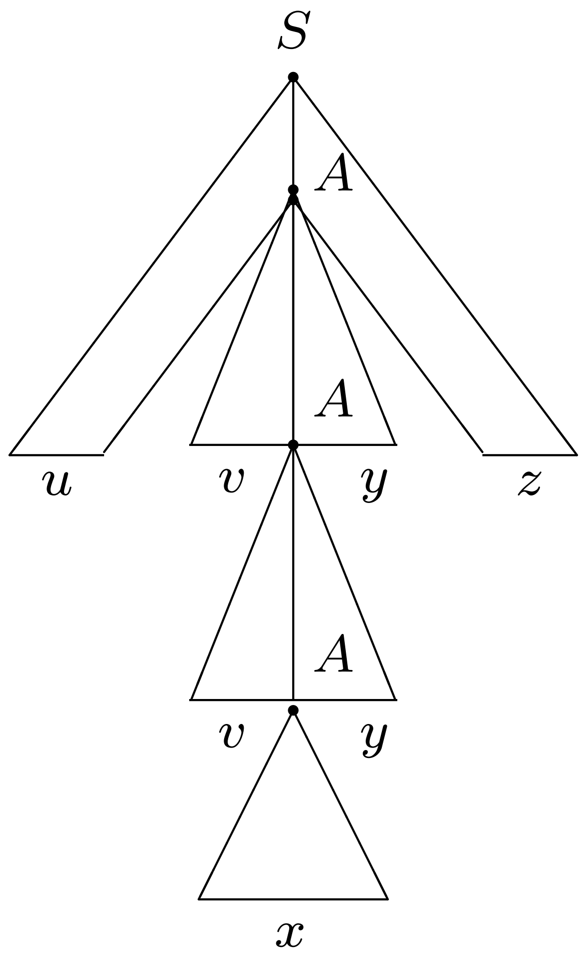
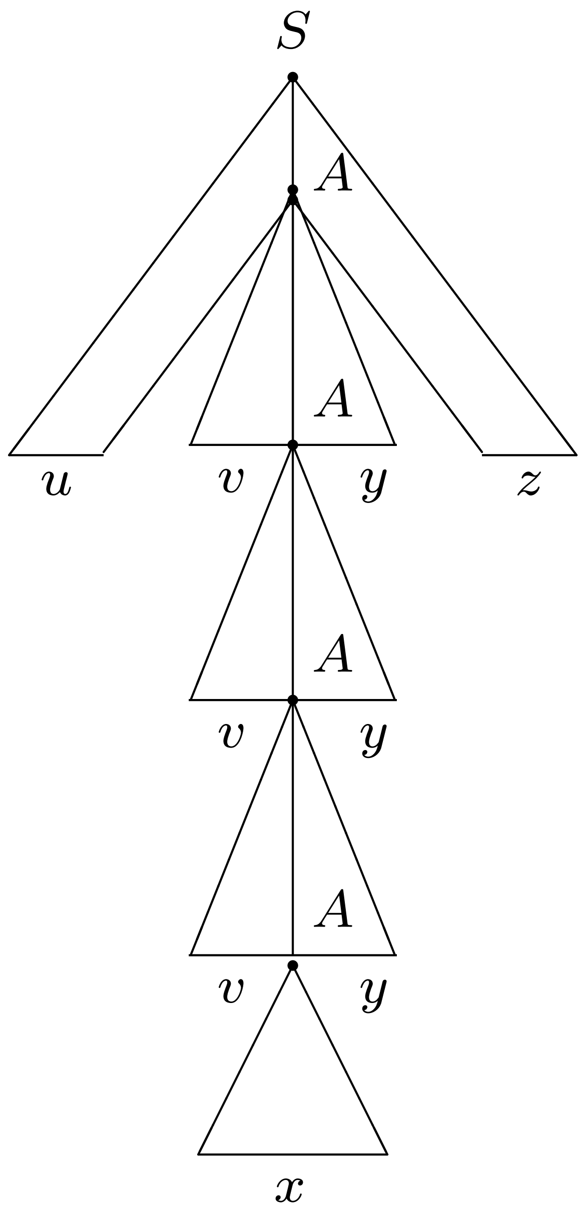
lemma

Let L be a context-free language. Then there exists, a constant K such that for any word $w \in L$ satisfying $|w| \geq K$ can be written $w = uvxyz$ with v or $y \neq \varepsilon$, $|vxy| \leq K$ and $uv^nxy^nz \in L$ for all $n > 0$.

Proof

A parse tree for G generating a sufficiently long word must contain a path on which *the same non-terminal appears at least twice*.


S
A
A
u
v
y
z
x


S
A
A
A
u
v
y
z
v
y
x

S
A
A
A
A
u
v
y
z
v
y
v
y
x

Choice of K

- $p = max\{|\alpha|, A \rightarrow \alpha \in R\}$
- The maximal length of a word generated by a tree of depth i is p^i.
- We choose $K = p^{m+1}$ where $m = |\{V - \Sigma\}|$.
- Thus $|w| > p^m$ and the parse tree contains paths of length $\geq m + 1$ that must include the same non terminal at least twice.
- Going back up one of these paths, a given non terminal will be seen for the second time after having followed at $m + 1$ arcs. Thus one can choose vxy of length at most $p^{m+1} = K$.
- Note: v and y cannot both be the empty word for all paths of length greater than $m + 1$. Indeed, if this was the case, the generated word would be of length less than p^{m+1}.

Applications of the pumping lemma

$L = \{a^n b^n c^n\}$ is not context-free.

Proof
There is no decomposition of $a^n b^n c^n$ in 5 parts u, v, x, y and z (v or y nonempty) such that, for all $j > 0$, $uv^j xy^j z \in L$. Thus the pumping lemma is not satisfied and the language cannot be context-free.

- v and y consist of the repetition of a unique letter. Impossible
- v and y include different letters. Impossible.

1. There exist two context-free languages L_1 St L_2 such that $L_1 \cap L_2$ is not context-free :

   - $L_1 = \{a^n b^n c^m\}$ is context-free,
   - $L_2 = \{a^m b^n c^n\}$ is context-free, but
   - $L_1 \cap L_2 = \{a^n b^n c^n\}$ is not context-free !

2. The complement of a context-free language is not necessarily context-free. Indeed, the union of context-free languages is always a context-free language. Thus, if the complement was context-free, so would be intersection:

$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}.$$

Algorithms for context-free languages

Let L be a context-free language (defined by a grammar or a pushdown automaton).

1. Given a word w, there exists an algorithm for checking whether $w \in L$.

2. There exists an algorithm for checking if $L = \emptyset$.

3. There is no algorithm for checking if $L = \Sigma^*$.

4. If L' is also a context-free language, there is no algorithm that can check if $L \cap L' = \emptyset$.

Theorem
Given context-free grammar G, there exists an algorithm that decides if a word w belongs to $L(G)$.

Proof

- Pushdown automaton? No, since these are nondeterministic and contain transitions on the empty word.

- Idea: bound the length of the executions. This will be done in the context of grammars (bound on the length of derivations).

Hypothesis: bounded derivations

To check if $w \in L(G)$:

1. One computes a bound k on the number of steps that are necessary to derive a word of length $|w|$.
2. One then explores systematically all derivations of length less than or equal to k. There is a finite number of such derivations.
3. If one of these derivations produces the word w, the word is in $L(G)$. If not, the word cannot be produced by the grammar and is not in $L(G)$.

Grammars with bounded derivations

Problem:

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow A \end{aligned}$$

Solution: Grammar satisfying the following constraints

1. $A \rightarrow \sigma$ with σ terminal, or
2. $A \rightarrow v$ with $|v| \geq 2$.
3. Exception: $S \rightarrow \varepsilon$

Bound: $2 \times |w| - 1$

Obtaining a grammar with bounded derivations

1. Eliminate rules of the form $A \to \varepsilon$.

If $A \to \varepsilon$ and $B \to vAu$ one adds the rule $B \to vu$. The rule $A \to \varepsilon$ can then be eliminated.

If one eliminates the rule $S \to \varepsilon$, one introduces a new start symbol S' and the rules $S' \to \varepsilon$, as well as $S' \to \alpha$ for each production of the form $S \to \alpha$.

2. Eliminating rules of the form $A \to B$.

For each pair of non-terminals A and B one determines if $A \overset{*}{\Rightarrow} B$.

If the answer is positive, for each production of the form $B \to u$ ($u \notin V - \Sigma$), one adds the production $A \to u$.

All productions of the form $A \to B$ can then be eliminated.

Theorem
Given a context-free grammar G, there exists an algorithm for checking if $L(G) = \emptyset$.

- Idea: search for a parse tree for G.
- One builds parse trees in order of increasing depth.
- The depth of the parse trees can be limited to $|V - \Sigma|$.

Deterministic pushdown automata

Two transitions $((p_1, u_1, \beta_1), (q_1, \gamma_1))$ and $((p_2, u_2, \beta_2), (q_2, \gamma_2))$ are compatible if

1. $p_1 = p_2$,
2. u_1 and u_2 are compatible (which means that u_1 is a prefix of u_2 or that u_2 is a prefix of u_1),
3. β_1 and β_2 are compatible.

A pushdown automaton is deterministic if for every pair of compatible transitions, theses transitions are identical.

Deterministic context-free languages

Let L be a language defined over the alphabet Σ, the language L is deterministic context-free if and only if it is accepted by a deterministic pushdown automaton.

- All context-free languages are not deterministic context-free.
- $L_1 = \{wcw^R \mid w \in \{a,b\}^*\}$ is deterministic context-free.
- $L_2 = \{ww^R \mid w \in \{a,b\}^*\}$ is context-free, but not deterministic context-free.

Properties of deterministic context-free languages

If L_1 and L_2 are deterministic context-free languages,

- $\Sigma^* - L_1$ is also deterministic context-free.
- There exists context-free languages that are not deterministic context-free.
- The languages $L_1 \cup L_2$ and $L_1 \cap L_2$ are not necessarily deterministic context-free.

Applications of context-free languages

- Description and syntactic analysis of programming languages.
- Restriction to deterministic context-free languages.
- Restricted families of grammars:LR.

Chapter 5

Turing Machines

5.1 Introduction

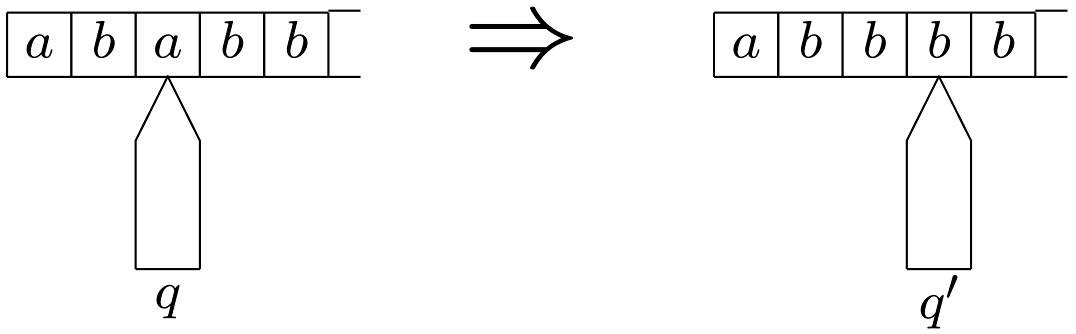
- The language $a^n b^n c^n$ cannot be accepted by a pushdown automaton.
- Machines with an infinite memory, which is not restricted to LIFO access.
- Model of the concept of effective procedure.
- Justification : extensions are not more powerful; other formalizations are equivalent.

5.2 Definition

- 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:

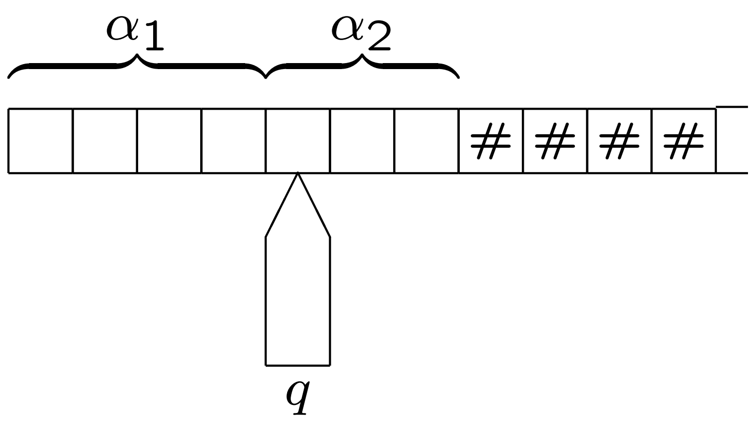
1. The state,
2. the tape content,
3. the position of the read head.

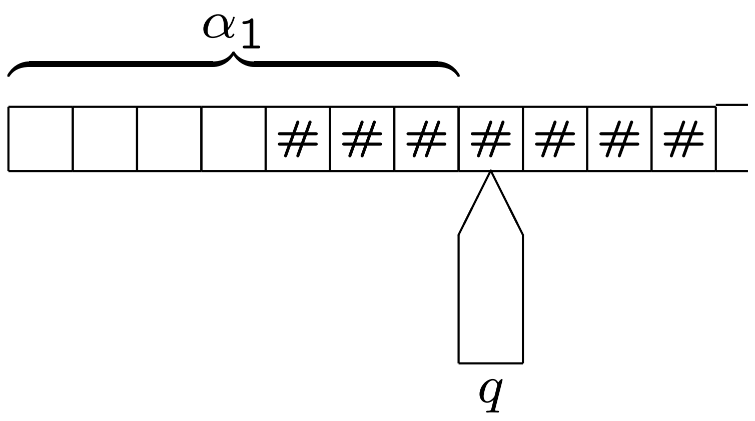
Representation : 3-tuple containing

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

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

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

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

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

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.

Decided Language

Deterministic Automata!

- Deterministic finite automata: the accepted and decided languages are the same.
- Nondeterministic finite automata: meaningless.
- Nondeterministic pushdown automata: meaningless, but the context-free languages can be decided by a Turing machine.
- Deterministic pushdown automata : the accepted language is decided, except if infinite executions exist (loops with only ε transitions).

Other definitions of Turing machines

1. Single *stop state* and a transition function that is defined everywhere. In the stop state; the result is placed on the tape: tape : "accepts" (1) or "does not accept" (0).

2. Two stop states: q_Y and q_N, and a transition function that is defined everywhere.

Recursive and recursively enumerable languages

A language is *recursive* if it is decided by a Turing machine.

A language is *recursively enumerable* if it is accepted by a Turing machine.

The Turing-Church thesis

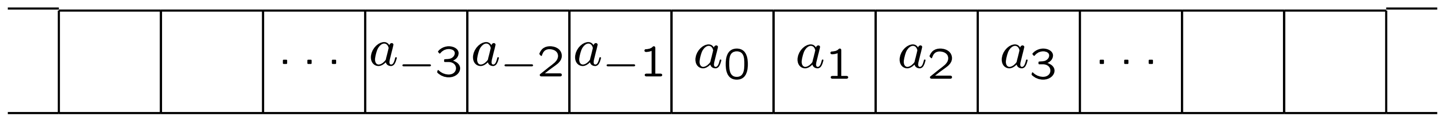
The languages that can be recognized by an effective procedure are those that are decided by a Turing machine.

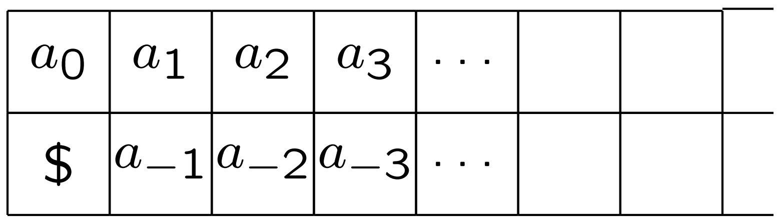
Justification.

1. If a language is decided by a Turing machine, it is computable: clear.
2. If a language is computable, it is decided by a Turing machine:
   - Extensions of Turing machines and other machines.
   - Other models.

Extensions of Turing machines

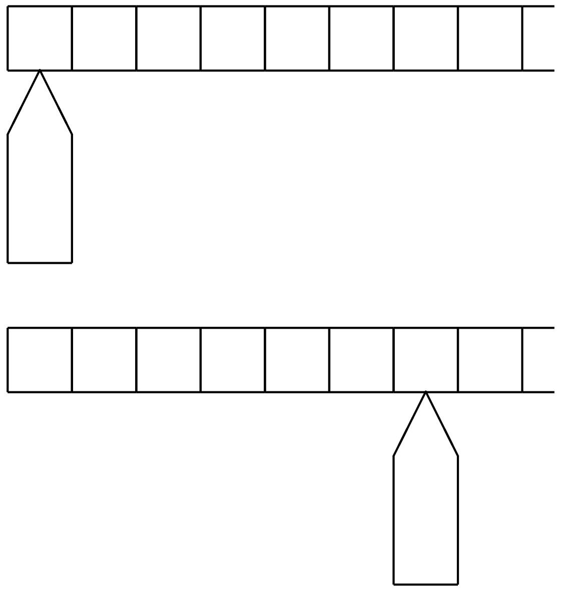
Tape that is infinite in both directions





Multiple tapes

Several tapes and read heads:

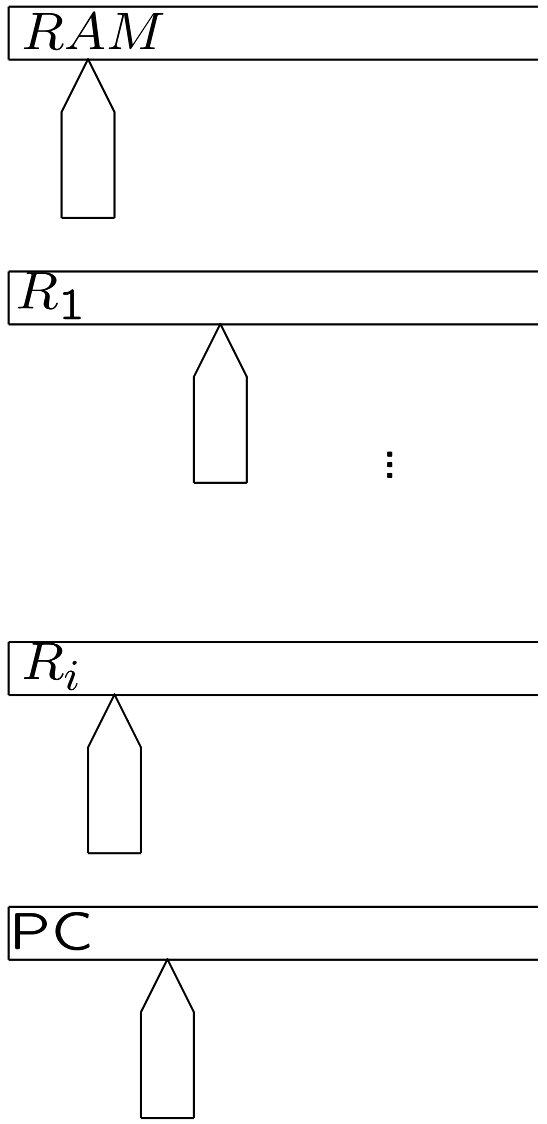


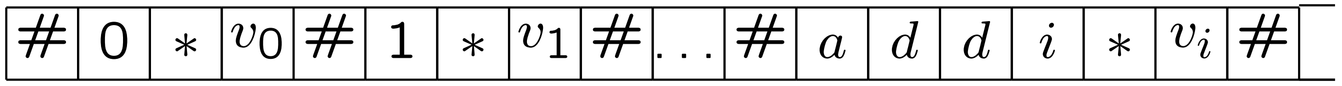
Simulation (2 tapes) : alphabet = 4-tuple

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

Machines with RAM

One tape for the memory, one for each register.



Simulation : 
$\# \; 0 \; * \; v_0 \; \# \; 1 \; * \; v_1 \; \# \; \ldots \; \# \; a \; d \; d \; i \; * \; v_i \; \#$

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.

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.

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

Chapter 6
Recursive functions

6.1 Introduction

- Other formalization of the concept of effective procedure: computable functions over the natural numbers.

- Computable functions?
  - Basic functions.
  - Function composition.
  - Recursion mechanism.

6.2 Primitive recursive functions

Functions in the set $\{N^k \rightarrow N \mid k \geq 0\}$.

1. Basic primitive recursive functions.

 1. $\mathbf{0}()$

 2. $\pi_i^k(n_1, \ldots, n_k)$

 3. $\sigma(n)$

2. Function composition.

- Let g be a function with ℓ arguments,
- $h_1, \ldots, h_\ell$ functions with k arguments.
- $f(\overline{n}) = g(h_1(\overline{n}), \ldots, h_\ell(\overline{n}))$ is the composition of g and of the functions h_i.

3. Primitive recursion.

- Let g be a function with k arguments and h a function with $k+2$ arguments.

- $$\begin{array}{lcl} f(\overline{n},0) & = & g(\overline{n}) \\ f(\overline{n},m+1) & = & h(\overline{n},m,f(\overline{n},m)) \end{array}$$

  is the function defined from g and h by primitive recursion.

- Remark: f is computable if g and h are computable.

Definition
The *Primitive recursive functions* are :

- the basic primitive recursive functions ;

- all functions that can be obtained from the basic primitive recursive functions by using composition and primitive recursion any number of times.

Examples

Constant functions :

$$\mathbf{j}() \; = \; \overbrace{\sigma(\sigma(\ldots\sigma(}^{j}\mathbf{0}())))$$

Addition function:

$$\begin{array}{lcl} plus(n_1, 0) & = & \pi_1^1(n_1) \\ plus(n_1, n_2 + 1) & = & \sigma(\pi_3^3(n_1, n_2, plus(n_1, n_2))) \end{array}$$

Simplified notation :

$$\begin{array}{lcl} plus(n_1, 0) & = & n_1 \\ plus(n_1, n_2 + 1) & = & \sigma(plus(n_1, n_2)) \end{array}$$

Evaluation of $plus(4,7)$:

$$\begin{aligned} plus(7,4) &= plus(7,3+1) \\ &= \sigma(plus(7,3)) \\ &= \sigma(\sigma(plus(7,2))) \\ &= \sigma(\sigma(\sigma(plus(7,1)))) \\ &= \sigma(\sigma(\sigma(\sigma(plus(7,0))))) \\ &= \sigma(\sigma(\sigma(\sigma(7)))) \\ &= 11 \end{aligned}$$

Product function :

$$\begin{aligned} n \times 0 &= 0 \\ n \times (m+1) &= n + (n \times m) \end{aligned}$$

Power function:

$$\begin{array}{lcl} n^0 & = & 1 \\ n^{m+1} & = & n \times n^m \end{array}$$

Double power :

$$\begin{array}{lcl} n \uparrow\uparrow 0 & = & 1 \\ n \uparrow\uparrow m+1 & = & n^{n \uparrow\uparrow m} \end{array}$$

$$n \uparrow\uparrow m = n^{n^{\cdot^{\cdot^{\cdot^{n}}}}} \}m$$

Triple power:

$$\begin{array}{lcl} n \uparrow\uparrow\uparrow 0 & = & 1 \\ n \uparrow\uparrow\uparrow m+1 & = & n \uparrow\uparrow (n \uparrow\uparrow\uparrow m) \end{array}$$

k-power :

$$\begin{array}{lcl} n \uparrow^k 0 & = & 1 \\ n \uparrow^k m+1 & = & n \uparrow^{k-1} (n \uparrow^k m). \end{array}$$

If k is an argument:

$$f(k+1, n, m+1) \; = \; f(k, n, f(k+1, n, m)).$$

Ackermann's function:

$$\begin{array}{lcl} Ack(0, m) & = & m+1 \\ Ack(k+1, 0) & = & Ack(k, 1) \\ Ack(k+1, m+1) & = & Ack(k, Ack(k+1, m)) \end{array}$$

Factorial function:

$$\begin{array}{lcl} 0! & = & 1 \\ (n+1)! & = & (n+1).n! \end{array}$$

Predecessor function:

$$\begin{array}{lcl} pred(0) & = & 0 \\ pred(m+1) & = & m \end{array}$$

Difference function:

$$\begin{array}{lcl} n \dot{-} 0 & = & n \\ n \dot{-} (m+1) & = & pred(n \dot{-} m) \end{array}$$

Sign function:

$$\begin{array}{lcl} sg(0) & = & 0 \\ sg(m+1) & = & 1 \end{array}$$

Bounded product:

$$f(\overline{n}, m) = \prod_{i=0}^{m} g(\overline{n}, i)$$

$$\begin{array}{lcl} f(\overline{n}, 0) & = & g(\overline{n}, 0) \\ f(\overline{n}, m+1) & = & f(\overline{n}, m) \times g(\overline{n}, m+1) \end{array}$$

6.3 Primitive recursive predicates

A predicate P with k arguments is a subset of N^k (the elements of N^k for which P is true).

The *characteristic function* of a predicate $P \subseteq N^k$ is the function $f : N^k \rightarrow \{0, 1\}$such that

$$f(\overline{n}) = \begin{cases} 0 & \text{si } \overline{n} \notin P \\ 1 & \text{si } \overline{n} \in P \end{cases}$$

A predicate is *primitive recursive* if its characteristic function is primitive recursive.

Examples

Zero predicate :

$$\begin{array}{lcl} zerop(0) & = & 1 \\ zerop(n+1) & = & 0 \end{array}$$

$<$ predicate :

$$less(n,m) \; = \; sg(m \dot{-} n)$$

Boolean predicates :

$$\begin{array}{lcl} and(g_1(\overline{n}), g_2(\overline{n})) & = & g_1(\overline{n}) \times g_2(\overline{n}) \\ or(g_1(\overline{n}), g_2(\overline{n})) & = & sg(g_1(\overline{n}) + g_2(\overline{n})) \\ not(g_1(\overline{n})) & = & 1 \dot{-} g_1(\overline{n}) \end{array}$$

$=$ predicate :

$$equal(n,m) \; = \; 1 \dot{-} (sg(m \dot{-} n) + sg(n \dot{-} m))$$

Bounded quantification :

$$\forall i \leq m \ p(\overline{n}, i)$$

is true if $p(\overline{n}, i)$ is true for all $i \leq m$.

$$\exists i \leq m \ p(\overline{n}, i)$$

is true if $p(\overline{n}, i)$ is true for at least one $i \leq m$.

$\forall i \leq m p(\overline{n}, i)$:

$$\prod_{i=0}^{m} p(\overline{n}, i)$$

$\exists i \leq m p(\overline{n}, i)$:

$$1 \dot{-} \prod_{i=0}^{m} (1 \dot{-} p(\overline{n}, i)).$$

Definition by case :

$$f(\overline{n}) = \begin{cases} g_1(\overline{n}) & \text{if } p_1(\overline{n}) \\ \vdots & \\ g_\ell(\overline{n}) & \text{if } p_\ell(\overline{n}) \end{cases}$$

$$f(\overline{n}) = g_1(\overline{n}) \times p_1(\overline{n}) + \ldots + g_\ell(\overline{n}) \times p_\ell(\overline{n}).$$

Bounded minimization :

$$\mu i \leq m\ q(\overline{n}, i) = \begin{cases} \text{the smallest } i \leq m \text{ such that } q(\overline{n}, i) = 1, \\ 0 \text{ if there is no such } i \end{cases}$$

$$\begin{array}{rcl} \mu i \leq 0\ q(\overline{n}, i) & = & 0 \\ \mu i \leq m+1\ q(\overline{n}, i) & = & \end{array}$$

$$\begin{cases} 0 & \text{if } \neg\exists i \leq m+1\ q(\overline{n}, i) \\ \mu i \leq m\ q(\overline{n}, i) & \text{if } \exists i \leq m\ q(\overline{n}, i) \\ m+1 & \text{if } q(\overline{n}, m+1) \\ & \text{and } \neg\exists i \leq m\ q(\overline{n}, i) \end{cases}$$

6.4 Beyond primitive recursive functions

Theorem
There exist computable functions that are not primitive recursive.

$$\begin{array}{c|cccccc}
A & 0 & 1 & 2 & \ldots & j & \ldots \\
\hline
f_0 & f_0(0) & f_0(1) & f_0(2) & \ldots & f_0(j) & \ldots \\
f_1 & f_1(0) & f_1(1) & f_1(2) & \ldots & f_1(j) & \ldots \\
f_2 & f_2(0) & f_2(1) & f_2(2) & \ldots & f_2(j) & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\
f_i & f_i(0) & f_i(1) & f_i(2) & \ldots & f_i(j) & \ldots \\
\vdots & \vdots & \vdots & \vdots & & \vdots & \ddots
\end{array}$$

$$g(n) = f_n(n) + 1 = A[n, n] + 1.$$

is not primitive recursive, but is computable.

6.4 The μ-recursive functions

Unbounded minimization :

$$\mu i\ q(\overline{n}, i) = \begin{cases} \text{the smallest } i \text{ such that } q(\overline{n}, i) = 1 \\ 0 \text{ if such an } i \text{ does not exist} \end{cases}$$

A predicate $q(\overline{n}, i)$ is said to be *safe* if

$$\forall \overline{n}\ \exists i\ q(\overline{n}, i) = 1.$$

The μ-recursive functions and predicates are those obtained from the basic primitive recursive functions by :

- composition, primitive recursion, and
- unbounded minimization of safe predicates.

μ-recursive functions and computable functions

Numbers and character strings :

Lemma
There exists an effective representation of numbers by character strings.

Lemma
There exists an effective representation of character strings by natural numbers.

Alphabet Σ of size k. Each symbol of Σ is represented by an integer between 0 and $k-1$. The representation of a string $w = w_0 \ldots w_l$ is thus:

$$gd(w) = \sum_{i=0}^{l} k^{l-i} gd(w_i)$$

Example : $\Sigma = \{abcdefghij\}$.

$$\begin{array}{rcl} gd(a) & = & 0 \\ gd(b) & = & 1 \\ & \vdots & \\ gd(i) & = & 8 \\ gd(j) & = & 9 \end{array}$$

$$gd(aabaafgj) = 00100569.$$

This encoding is ambiguous :

$$\begin{array}{l} gd(aaabaafgj) = 000100569 = \\ 00100569 = gd(aabaafgj) \end{array}$$

Solution: use an alphabet of size $k+1$ and do not encode any symbol by 0.

$$gd(w) = \sum_{i=0}^{l} (k+1)^{l-i} gd(w_i).$$

From μ-recursive functions To Turing machines

Theorem
Every μ-recursive function is computable by a Turing machine..

1. The basic primitive recursive functions are Turing machine computable;

2. Composition, primitive recursion and bounded minimization applied to Turing computable functions yield Turing computable functions.

From Turing machines to μ-recursive functions

Theorem
Every Turing computable functions is μ-recursive.

Let M be a Turing machine. One proves that there exists a μ-recursive f such that

$$f_M(w) = gd^{-1}(f(gd(w))).$$

Useful predicates :

1. $init(x)$ initial configuration of M.

2. $next\_config(x)$

3.

$$\mathit{config}(x,n)\begin{cases}\mathit{config}(x,0)=x\\ \mathit{config}(x,n+1)=\\ \mathit{next\_config}(\mathit{config}(x,n))\end{cases}$$

4. $\mathit{stop}(x)=\begin{cases}1 & \text{if } x \text{ final}\\ 0 & \text{if not}\end{cases}$

5. $\mathit{output}(x)$

We then have :

$$f(x)=\mathit{output}(\mathit{config}(\mathit{init}(x),\mathit{nb\_of\_steps}(x)))$$

où

$$\mathit{nb\_of\_steps}(x)=\mu i\ \ \mathit{stop}(\mathit{config}(\mathit{init}(x),i)).$$

Partial functions

A partial function $f\ :\Sigma^* \to \Sigma^*$ is computed by a Turing machine M if,

- for every input word w for which f is defined, M stops in a configuration in which $f(w)$ is on the tape,
- for every input word w for which f is not defined, M does not stop or stops indicating that the function is not defined by writing a special value on the tape.

A partial function $f\ : N \to N$ is μ-recursive is it can be defined from basic primitive recursive functions by

- composition,
- primitive recursion,
- unbounded minimization.

Unbounded minimization can be applied to unsafe predicates. The function $\mu i\ p(\overline{n}, i)$ is undefined when there is no i such that $p(\overline{n}, i) = 1$.

Theorem
A partial function is μ-recursive if and only if it is Turing computable.

Chapter 7

Uncomputability

7.1 Introduction

- Undecidability of concrete problems.
- First undecidable problem obtained by diagonalisation.
- Other undecidable problems obtained by means of the reduction technique.
- Properties of languages accepted by Turing machines.

7.2 Proving undecidability
Undecidability classes

Correspondence between a problem and the language of the encodings of its positive instances.

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.

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)

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	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

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

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

$$L_0 = \{w | w = w_i \wedge A[M_i, w_i] = \text{N}\}.$$

is not in the class RE.

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:

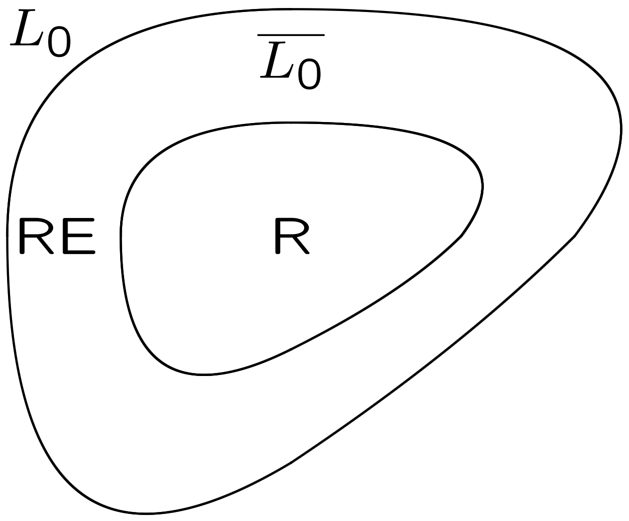
1. L and $\overline{L} \in$ R,
2. $L \notin$ RE and $\overline{L} \notin$ RE,
3. $L \notin$ RE and $\overline{L} \in \text{RE} \cap \overline{\text{R}}$.

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 ($L_2 \not\in$ 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.

Le *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 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 : $\overline{\text{UL}} \not\in \text{RE}$

More undecidable problems

The halting problem

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

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.

The problem of deciding if a Turing machine stops when its input word is the empty word (*the empty-word halting problem*) is undecidable. This is proved by reduction from the halting problem.

1. For an instance $< M, w >$ of the halting problem, one builds a Turing machine M' that has the following behaviour:
    - it writes the word w on its input tape;
    - it then behaves exactly as M.
2. One solves the empty-word halting problem for M' et uses the result as answer.

The problem of deciding if a Turing machine stops for at least one input word (*the existential halting problem*) is undecidable. One proceeds by reduction from the empty-word halting problem.

1. For an instance M of the empty-word halting problem, one builds a Turing machine M'that behaves as follows:
    - it erases the content of its input tape;
    - it then behaves as M.
2. One solve the existential halting problem for M' and uses the result as answer.

The problem of deciding if a Turing machine stops for every input word (*the universal halting problem* is undecidable. The reduction proceeds from the empty-word halting problem and is identical to the one used for the existential halting problem. The only difference is that one solves the universal halting problem for M', rather than the existential halting problem.

Determining if the language accepted by a Turing machine is empty (*empty accepted language*) is undecidable. Reduction from $\overline{\text{UL}}$.

1. For an instance $< M, w >$ of $\overline{\text{UL}}$, one builds a Turing machine M' that
   - simulates the execution of M on w ignoring its own input word;
   - if M accepts w, it accepts is input word, whatever it is.
   - if M does not accept w (rejects or has an infinite execution) it does not accept any word.
2. One solves the empty accepted language problem for M' eand uses the result as answer.

This reduction is correct given that

- $L(M') = \emptyset$ exactly when M does not accept w, i.e., when $< M, w >\in \overline{\mathsf{UL}}$;

- $L(M') = \Sigma^* \neq \emptyset$ exactly when M accepts w, i.e. when $< M, w >\not\in \overline{\mathsf{UL}}$.

Determining if the language accepted by a Turing machine is recursive (*recursive accepted language*) is undecidable. Reduction from $\overline{\text{UL}}$.

1. For an instance $< M, w >$ of $\overline{\text{UL}}$, one builds a Turing machine M' that
   - simulates the execution of M on w ignoring its own input word;
   - if M accepts w, it behaves on its own input word as a universal turing machine.
   - if M does not accept w (rejects or has an infinite execution) it does not accept any word.
2. One solves the *recursive accepted language problem* for M' and uses the result as answer.

This reduction is correct since

- $L(M') = \emptyset$ and is recursive exactly when M does not accept w, i. e. when $< M, w > \in \overline{\text{UL}}$;

- $L(M') = \text{UL}$ and is not recursive exactly when M accepts w, i.e. when $< M, w > \notin \overline{\text{UL}}$.

Determining if the language accepted by a Turing machine is not recursive (undecidable) (*undecidable accepted language*) is undecidable. Reduction from $\overline{\text{UL}}$.

1. For an instance $< M, w >$ of $\overline{\text{UL}}$, one builds a Turing machine M' that

   - simulates the execution of M on w, without looking at its own input word x;
   - simultaneously (i.e. interleaving the executions), the machine M' simulates the universal Turing machine on its own input word x;
   - As soon as one of the executions accepts, (i.e., if M accepts w or if the input word is in UL), M' accepts.

2. If neither of the two executions accepts (i.e., if M does not accept w, or if the input word $x \notin$ UL), M' does not accept.

3. One solves the undecidable accepted language problem for M' and uses the result as answer.

This reduction is correct since

- $L(M') =$ UL and is undecidable exactly when M does not accept w, i.e., when $< M, w > \in \overline{\text{UL}}$;

- $L(M') = \Sigma^*$ and is decidable exactly when M accepts w, i.e. when $< M, w > \notin \overline{\text{UL}}$.

In the preceding reductions, the language accepted by the machine M' is either UL, or $\emptyset$, or Σ^*. These proofs can thus also be used to establish that the problem of determining if the language accepted by a Turing machine is regular (or non regular) is undecidable. Indeed, $\emptyset$ and Σ^* are regular languages, whereas UL is not a regular language.

7.4 Properties of recursively enumerable languages

The recursively enumerable languages are :

- The languages computed by a Turing machine,
- the languages generated by a grammar,
- The languages that can be enumerated by an effective procedure (which explains why they are called “recursively enumerable”).

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 languages 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.

Let L be a language computed by a Turing machine M. The nondeterministic Turing machine described below accepts this language.

1. The machine M' first memorises its input word w.
2. Thereafter, it generates nondeterministically a word u.
3. The machine M' then simulates the behaviour of M on u.
4. If M stops on u, M' compares w to $f_M(u)$ and accepts w if $w = f_M(u)$.
5. If M does not stop on u, M' does not accept w.

The languages generated by a grammar

Theprem
A language is generated by a grammar if and only if it is recursively enumerable.

Let $G = (V, \Sigma, R, S)$, The Turing machine M described below accepts the language generated by G.

1. The machine M starts by memorising its input word (we can assume it uses a second tape to do so).
2. Then, it erases its tape and writes on it the start symbol S of the grammar.

3. The following cycle is then repeated :
   (a) nondeterministically, the machine chooses a rule R and a string appearing on its tape;
   (b) if the selected string is identical to the left-hand side of the rule, it is replaced by the right-hand side;
   (c) the content of the tape is compared to the memorised input word, and if they are identical the machine accepts; if not it carries on with its execution.

Let $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$ be a Turing machine. One builds a grammar

$$G_0 = (V_{G_0}, \Sigma_{G_0}, R_{G_0}, S_{G_0})$$

such that $S_{G_0} \stackrel{*}{\Rightarrow} w$ with $w \in (Q \cup \Gamma)^*$ if and only if w describes a configuration (q, α_1, α_2) of M written as $\alpha_1 q \alpha_2$.

The grammar G_0 is defined by

- $V_{G_0} = Q \cup \Gamma \cup \{S_{G_0}, A_1, A_2\}$,
- $\Sigma_{G_0} = \Sigma$,
- R_{G_0} is the set of rules below.

1. Initial configuration of M :

$$\begin{array}{lll} S_{G_0} & \rightarrow & sA_1 \\ A_1 & \rightarrow & aA_1 \quad \forall a \in \Sigma \\ A_1 & \rightarrow & A_2 \\ A_2 & \rightarrow & BA_2 \\ A_2 & \rightarrow & \varepsilon. \end{array}$$

2. Transitions. For all $p, q \in Q$ and $X, Y \in \Gamma$ such that

$$\delta(q, X) = (p, Y, R)$$

   we include the rule

$$qX \rightarrow Yp.$$

   Similarly, for all $p, q \in Q$ and $X, Y, Z \in \Gamma$ such that

$$\delta(q, X) = (p, Y, L)$$

   we include the rule

$$ZqX \rightarrow pZY.$$

Problem: the input word is lost.

Solution: simulate a Turing machine with two tapes.

$G_1 = (V_{G_1}, \Sigma_{G_1}, R_{G_1}, S_{G_1})$ where

- $V_{G_1} = \Sigma \cup Q \cup ((\Sigma \cup \{e\}) \times \Gamma) \cup \{S_{G_1}, A_1, A_2\}$ (we represent an element of $((\Sigma \cup \{e\}) \times \Gamma)$ by a pair $[a, X]$),

- $\Sigma_{G_1} = \Sigma$,

- R_{G_1} is the set of rules described below.

1. Initial configuration of M :

$$\begin{array}{lll} S_{G_1} & \rightarrow & sA_1 \\ A_1 & \rightarrow & [a,a]A_1 \quad \forall a \in \Sigma \\ A_1 & \rightarrow & A_2 \\ A_2 & \rightarrow & [e,B]A_2 \\ A_2 & \rightarrow & \varepsilon. \end{array}$$

2. Transitions. For all $p, q \in Q$, $X, Y \in \Gamma$ and $a \in \Sigma \cup \{e\}$ such that

$$\delta(q, X) = (p, Y, R)$$

we include the rule

$$q[a, X] \rightarrow [a, Y]p.$$

Similarly, for all $p, q \in Q$, $X, Y, Z \in \Gamma$ and $a, b \in \Sigma \cup \{e\}$ such that

$$\delta(q, X) = (p, Y, L)$$

we include the rule

$$[b, Z]q[a, X] \rightarrow p[b, Z][a, Y].$$

3. For all $q \in F$, $X \in \Gamma$ and $a \in \Sigma \cup \{e\}$, we include the rules

$$\begin{array}{rcl} q[a,X] & \rightarrow & qaq \\ [a,X]q & \rightarrow & qaq \end{array}$$

   if $a \neq e$ and

$$\begin{array}{rcl} q[a,X] & \rightarrow & q \\ [a,X]q & \rightarrow & q \end{array}$$

   if $a = e$. These rules propagate a copy of q next to each nonterminal $[a, X]$ and extract its first component. Finally, we add

$$q \rightarrow \varepsilon$$

   that allows the copies of the state q to be removed.

The languages enumerated by an effective procedure

Turing machine that enumerates the words accepted by M.

- Generate all words in lexicographical and increasing length order,
- simulate M on each newly generated word and keep this word only if it is accepted by M.

Incorrect: the Turing machine can have infinite executions.

Solution: other enumeration order.

$$\begin{tikzcd}
w \setminus n & 1 & 2 & 3 & 4 \\
w_1 & (w_1,1) \arrow[r] & (w_1,2) \arrow[dl] & (w_1,3) \arrow[r] & (w_1,4) \arrow[dl] \\
w_2 & (w_2,1) \arrow[d] & (w_2,2) \arrow[ur] & (w_2,3) \arrow[dl] & \\
w_3 & (w_3,1) \arrow[ur] & (w_3,2) \arrow[dl] & (w_3,3) & \\
w_4 & (w_4,1) \arrow[d] & & & \\
 & {} & & &
\end{tikzcd}$$

- One considers the pairs (w, n) in the order of their enumeration.
- For each of these pairs, one simulates the execution of M on w, but limits the execution to n steps. On produces the word w if this execution accepts w.
- On then moves to the next pair (w, n).

7.5 Other undecidable problems

The problem of determining if a word w is in the language generated by a grammar G is undecidable.

Reduction from the problem UL. Let $< M, w >$ be an instance of the problem UL. It can be solved as follows:

1. one builds the grammar G generating the language accepted by M
2. one determines if $w \in L(G)$ and uses the result as answer.

The problem of deciding if two grammars G_1 and G_2 generate the same language is undecidable.

Reduction from the membership problem for the language generated by a grammar. An instance $< w, G >$ of this problem can be solved as follows:

1. Let $G = (V, \Sigma, R, S)$. One builds the grammars $G_1 = G$ and $G_2 = (V, \Sigma, R', S')$, with
$$R' = R \cup \{S' \to S, S' \to w\}.$$
2. One checks if $L(G_1) = L(G_2)$ and uses the result as answer.

One has indeed that $L(G_2) = L(G_1) \cup \{w\}$ and thus that $L(G_2) = L(G_1)$ if and only if $w \in L(G)$.

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

The problem of determining the *universality of a context-free language*, i.e., the problem of determining if for a context-free grammar G one has $L(G) = \Sigma^*$ is undecidable.

The problem of determining the *emptiness of the intersection of context-free languages* is undecidable.

The problem is to determine if, for two context-free grammars G_1 and G_2, one has $L(G_1) \cap L(G_2) = \emptyset$.

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.

Noncomputable functions

A total function

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

is computable if and only if the following questions are decidable.

1. Given $n \in N$ and $w \in \Sigma_1^*$, do we have that $|f(w)| > n$?
2. Given $k \in N$, $w \in \Sigma_1^*$ and $a \in \Sigma_2$, do we have that $f(w)_k = a$? (is the k^{th} letter of $f(w)$ a?).

The situation is similar in the case of a partial function. A function

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

is a partially computable function if and only if the following conditions are satisfied.

1. Checking if for a given word w, $f(w)$ is defined is partially decidable.

2. For $n \in N$ and $w \in \Sigma_1^*$ such that $f(w)$ is defined, checking if $|f(w)| > n$ is decidable.

3. For $k \in N$, $a \in \Sigma_2$ and $w \in \Sigma_1^*$ such that $f(w)$ is defined, checking if $f(w)_k = a$ is decidable.

Chapter 8
Complexity

8.1 Introduction

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

8.2 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.

8.3 Polynomial problems

- Influence of the encoding.
- Graph example.
- Reasonable encodings:
  - no padding,
  - polynomial decoding,
  - unary representation of numbers not allowed.

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.

8.4 Polynomial transformations

- Diagonalisation is not adequate to prove that problems are not in P.

- Another approach: comparing problems.

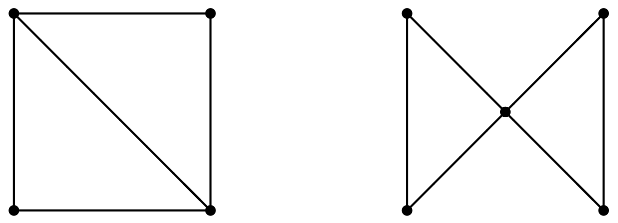
The Travelling Salesman (TS)

- Set C of n Cities.
- Distances $d(c_i, c_j)$.
- Constante b.
- Permutation if 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.



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 $f : \Sigma_1^* \rightarrow \Sigma_2^*$ that satisfies the following conditions :

1. it is computable in polynomial time,
2. $f(x) \in L_2$ if and only if $x \in L_1$.

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. $b = |V|$.

Properties of $\propto$

If $L_1 \propto L_2$, then

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

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

- $L_1 \propto L_3$.

Polynomially equivalent problems

Déeinition
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

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

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

Largest class in NP : NPC

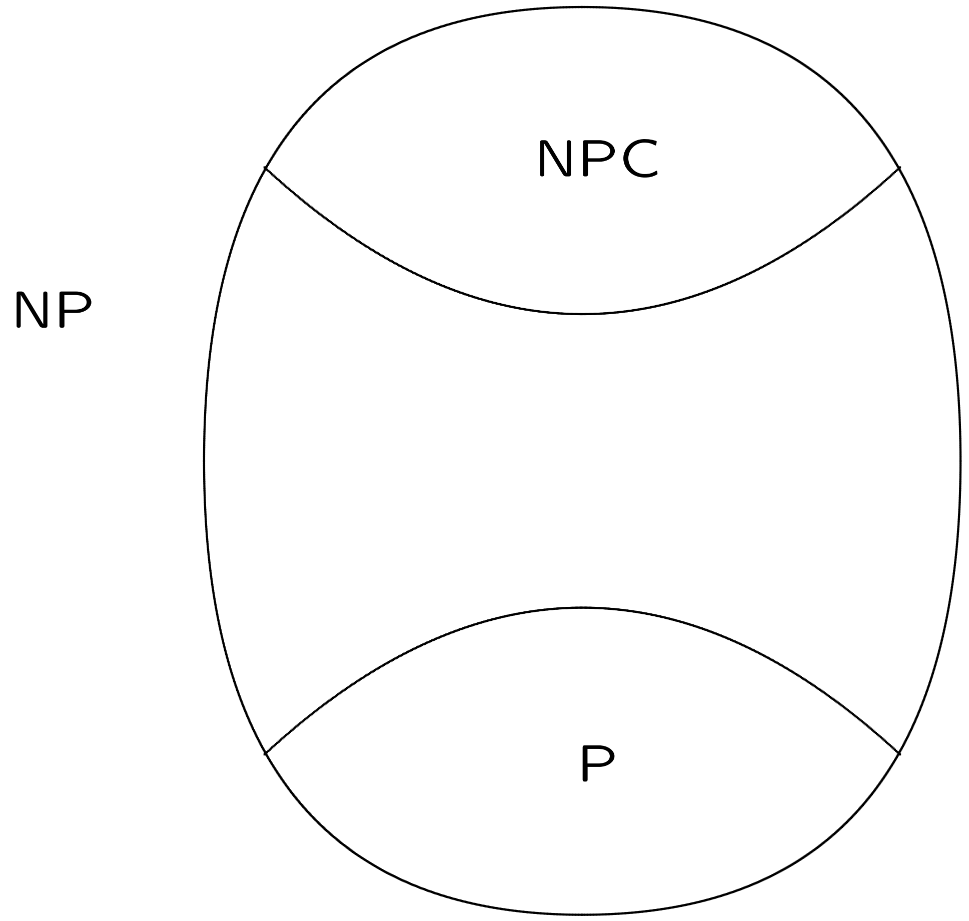
A language L is NP-complete if

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


NPC
NP
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 ($L \in \text{NP}$),

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

or, alternatively,

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

Concept of NP-hard problem.

A first NP-complete problem
propositional calculus

Boolean calculus :

p	$\neg p$
0	1
1	0

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

p	q	$p \supset q$
0	0	1
0	1	1
1	0	0
1	1	1

- Boolean expression: $(1 \wedge (0 \vee (\neg 1))) \supset 0$.

- Propositional variables and propositional calculus :
  $(p \wedge (q \vee (\neg r))) \supset s$.

- Interpretation function. Valid formula, satisfiable formula.

- Conjunctive normal form: conjunction of disjunctions of literals.

Cook's theorem

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

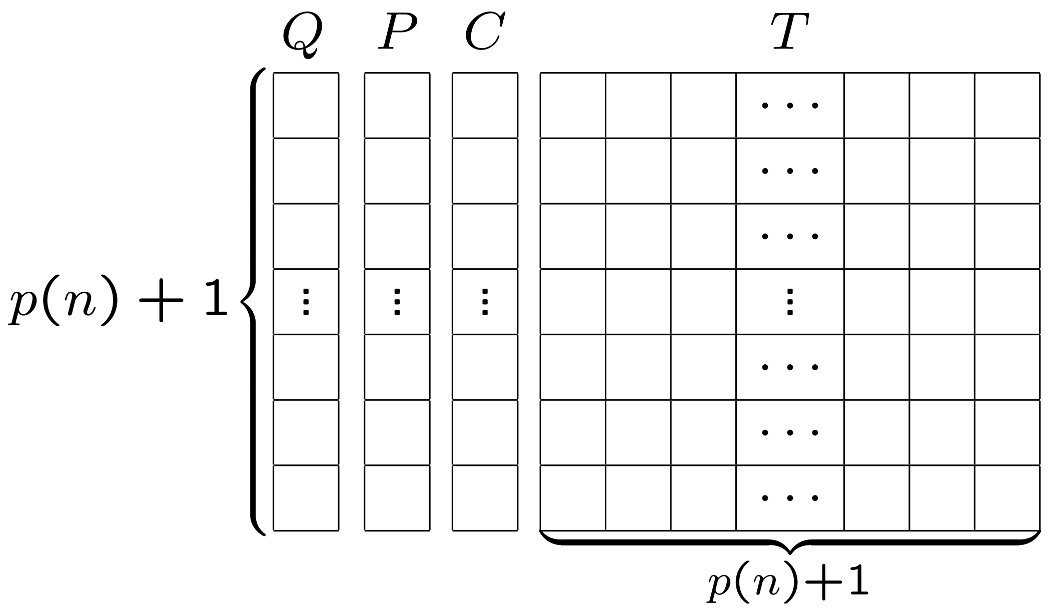
Theorem
The SAT problem is NP-complete

Proof

1. SAT is in NP.
2. There exists a polynomial transformation from every language in NP to L_{SAT}.
    - Transformation with two arguments : word and language.
    - The languages of NP are characterised by a polynomial-time nondeterministic Turing machine.

Word w ($|w| = n$) and nondeterministic polynomial Turing machine $M = (Q, \Gamma, \Sigma, \Delta, s, B, F)$ (bound $p(n)$).

Description of an execution of M (T : tape; Q : state; P : position; C : choice.)



Representing an execution with propositional variables:

1. A proposition $t_{ij\alpha}$ for $0 \leq i, j \leq p(n)$ and $\alpha \in \Gamma$.
2. A proposition $q_{i\kappa}$ for $0 \leq i \leq p(n)$ and $\kappa \in Q$.
3. A proposition p_{ij} for $0 \leq i, j \leq p(n)$.
4. A proposition c_{ik} for $0 \leq i \leq p(n)$ and $1 \leq k \leq r$.

Formula satisfied only by an execution of M that accepts the word w : conjunction of the following formulas.

$$\bigwedge_{0 \leq i,j \leq p(n)} \left[(\bigvee_{\alpha \in \Gamma} t_{ij\alpha}) \wedge \bigwedge_{\alpha \neq \alpha' \in \Gamma} (\neg t_{ij\alpha} \vee \neg t_{ij\alpha'}) \right]$$

One proposition for each tape cell. Length $O(p(n)^2)$.

$$\bigwedge_{0 \leq i \leq p(n)} \left[(\bigvee_{0 \leq j \leq p(n)} p_{ij}) \wedge \bigwedge_{0 \leq j \neq j' \leq p(n)} (\neg p_{ij} \vee \neg p_{ij'}) \right]$$

One proposition for each position. Length $O(p(n)^3)$.

$$\left[\bigwedge_{0\leq j\leq n-1} t_{0jw_{j+1}} \wedge \bigwedge_{n\leq j\leq p(n)} t_{0jB}\right] \wedge q_{0s} \wedge p_{00}$$

Initial state. Length $O(p(n))$

$$\bigwedge_{\substack{0\leq i<p(n)\\ 0\leq j\leq p(n)\\ \alpha\in\Gamma}} [(t_{ij\alpha} \wedge \neg p_{ij}) \supset t_{(i+1)j\alpha}]$$

$$\bigwedge_{\substack{0\leq i<p(n)\\ 0\leq j\leq p(n)\\ \alpha\in\Gamma}} [\neg t_{ij\alpha} \vee pij \vee t_{(i+1)j\alpha}]$$

Transitions, tape not modified. Length $O(p(n)^2)$.

$$\bigwedge_{\substack{0\leq i<p(n)\\0\leq j\leq p(n)\\\alpha\in\Gamma\\1\leq k\leq r}}\left[\begin{array}{l}((q_{i\kappa}\wedge p_{ij}\wedge t_{ij\alpha}\wedge c_{ik})\supset q_{(i+1)\kappa'})\wedge\\((q_{i\kappa}\wedge p_{ij}\wedge t_{ij\alpha}\wedge c_{ik})\supset t_{(i+1)j\alpha'})\wedge\\((q_{i\kappa}\wedge p_{ij}\wedge t_{ij\alpha}\wedge c_{ik})\supset p_{(i+1)(j+d)})\end{array}\right]$$

$$\bigwedge_{\substack{0\leq i<p(n)\\0\leq j\leq p(n)\\\alpha\in\Gamma\\1\leq k\leq r}}\left[\begin{array}{l}(\neg q_{i\kappa}\vee\neg p_{ij}\vee\neg t_{ij\alpha}\vee\neg c_{ik}\vee q_{(i+1)\kappa'})\wedge\\(\neg q_{i\kappa}\vee\neg p_{ij}\vee\neg t_{ij\alpha}\vee\neg c_{ik}\vee t_{(i+1)j\alpha'})\wedge\\(\neg q_{i\kappa}\vee\neg p_{ij}\vee\neg t_{ij\alpha}\vee\neg c_{ik}\vee p_{(i+1)(j+d)})\end{array}\right]$$

Transitions, modified part. Length $O(p(n)^2)$.

$$\bigvee_{\substack{0 \le i \le p(n) \\ \kappa \in F}} [q_{i\kappa}]$$

Final state reached. Length $O(p(n))$.

- Total length of the formula $O(p(n)^3)$.

- The formula can be built in polynomial time.

- Thus, we have a transformation that is polynomial in terms of $n = |w|$.

- The formula is satisfiable if and only if the Turing machine M accepts.

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

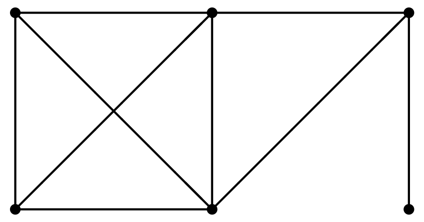
$$(x_1 \vee x_2 \vee \cdots \vee x_i \vee \cdots \vee x_{\ell-1} \vee x_\ell)$$

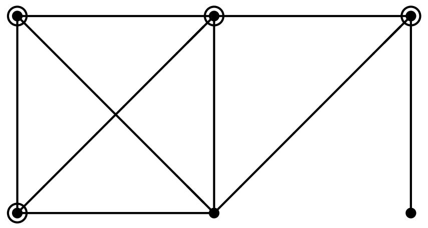
with $\ell \geq 4$ literals is replaced by

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

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 is 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}, $1 \leq i \leq k$,

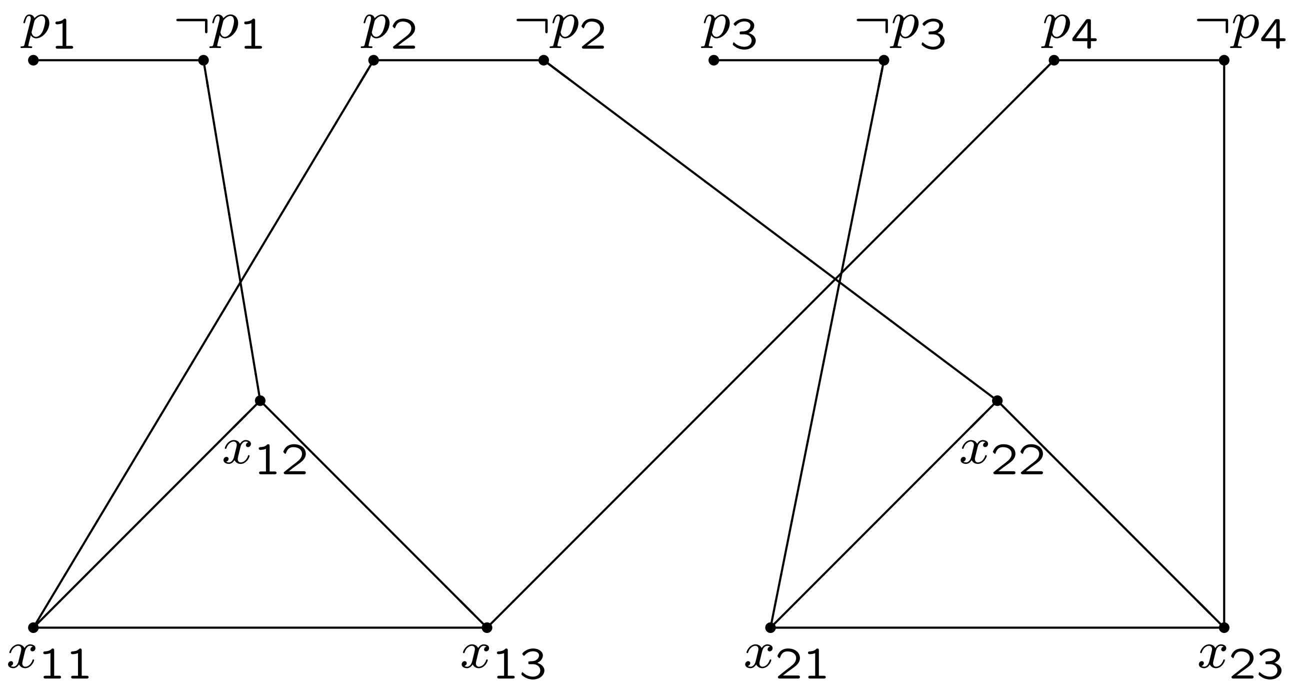
   (c) an edge between each vertex x_{ij} and the vertex p or $\neg p$ representing the corresponding literal.

   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.

8.8 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.

8.9 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 language 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 language accepted by a nondeterministic Turing machine whose space complexity is bounded by a polynomial.

$$\mathrm{P} \subseteq \begin{array}{c}\mathrm{NP} \\ \text{co-NP}\end{array} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME}.$$