Izračunljivost in računska zahtevnost

Computability and Computational Complexity

Borut Robič

Faculty of Computer and Information Science
University of Ljubljana

Lectures

- Lectures in Slovenian
- Slides in English

Literature

- *These slides* (are a combination of the following:)
- J.E.Hopcroft, J.D.Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1st ed., 1979
- S.Arora, B.Barak. *Computational Complexity : A Modern Approach*, Cambridge University Press, 2009
- B.Robič. *The Foundations of Computability Theory*, Springer, 2015
- B.Robič. *O rešljivem, nerešljivem, obvladljivem in neobvladljivem*
  http://www.delo.si/znanje/znanost/o-resljivem-neresljivem-obvladljivem-in-neobvladljivem.html

Contents

Dictionary

Finite automata končni avtomati regular exspressions regularni izrazi context/free grammars kontekstno neodvisne gramatike pushdown automata skladovni avtomati context/free languages kontekstno neodvisni jeziki Turing machines Turingovi stroji undecidability neodločljivost Chomsky hierarchy hierarhija Chomskega computational complexity računska zahtevnost intractable problems neobvladljivi problemi approximation algorithms aproksimacijski algoritmi probabilistic (or randomized) algorithms verjetnostni (ali naključnostni) algoritmi parallel algorithms vzporedni algorithmi quantum algorithms kvantni algoritmi

Question

Why Theoretical Computer Science??

- Theoretical Computer Science is concerned with *modeling* computational problems and solving them *algorithmically*.
  It strives to distinguish what *can* be computed from what *cannot*.
  If a problem can be solved by an algorithm, it is important to know the amount of *space* and *time* needed.

1 Preliminaries

Contents

- Strings, alphabets, and languages
- Graphs and trees
- Inductive proofs
- Set notation
- Relations
- Synopsis of the course

1.1 Strings, Alphabets, and Languages

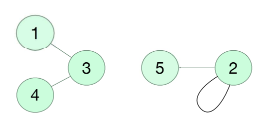
- A symbol is an abstract entity that we shall not define formally. Example. *Letters* and *digits* are frequently used symbols.
- A string (or word) is a finite sequence of symbols juxtaposed. Example. a, b, and c are symbols and $abcb$ is a string. The length of a string w, denoted $|w|$, is the number of symbols composing w. E.g., $abcb$ has length 4. The empty string, denoted by ε, is the string consisting of zero symbols. So $|\varepsilon|=0$.

- A prefix of a string is any number of leading symbols of that string, and a suffix is any number of trailing symbols. Example. abc has prefixes ε, a, ab, abc, and suffixes ε, c, bc, abc. A prefix or suffix of a string, other than the string itself, is called a proper prefix or suffix.
- The concatenation of two strings is the string formed by writing the first, followed by the second, with no intervening space. Example. The concatenation of *dog* and *house* is *doghouse*. Juxtaposition is used as the *concatenation operator*. That is, if w and x are strings, then wx is the concatenation of these two strings. The ε is the *identity* for the concatenation operator. That is, $\varepsilon w = w\varepsilon = w$ for each string w.

- An alphabet is a finite set of symbols.
- A (formal) language is a set of strings of symbols from some alphabet. The *empty set*, $\emptyset$, and the set consisting of the empty string, $\{\varepsilon\}$, are languages. They are distinct.
  - **Example**. The set of palindromes (words that read the same in both directions) over the alphabet $\{0, 1\}$ is an infinite language. Some of its members are ε, 0, 1, 00, 11, 010, 1101011.
- Another language is the set of all strings over a fixed alphabet Σ. We denote this language by Σ^*
  - **Example**. If $\Sigma = \{a\}$, then $\Sigma^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$. If $\Sigma = \{0,1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots\}$.

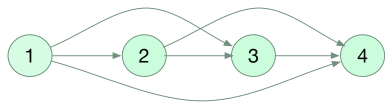
1.2 Graphs and Trees

- An undirected graph, denoted $G = (V, E)$, consists of a finite set V of vertices (or nodes) and a set E of pairs of vertices called edges.
  - Example. $V=\{1,2,3,4,5\}$, $E=\{\{n,m\} \mid n+m = 4 \text{ or } n+m = 7\}$.



- A path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$, $k \geq 1$, such that there is an edge $\{v_i, v_{i+1}\}$ for each i, $1 \leq i < k$. The length of the path is k--1. Example. 1, 3, 4 is a path in the above graph; so is 5 by itself. If $v_1 = v_k$, the path is a cycle.

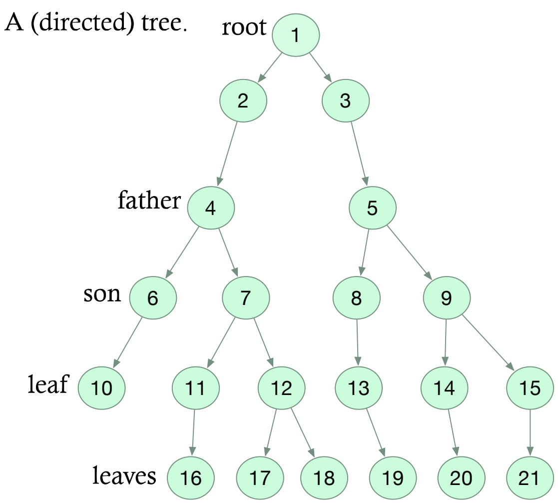
- A directed graph (or digraph), denoted $G = (V, A)$, consists of a finite set V of vertices and a set A of *ordered* pairs of vertices called arcs. We also denote an arc (u,v) by $u \to v$.
  - Example.



- A path in a digraph is a sequence of vertices $v_1, v_2, \ldots, v_k$, $k \geq 1$, such that $v_i \to v_{i+1}$ is an arc for each i, $1 \leq i < k$. We say the path is *from* v_1 *to* v_k. Example. $1 \to 2 \to 3 \to 4$ is a path from 1 to 4. If $u \to v$ is an arc we say that u is a predecessor of v (and v is a successor of u).

- A tree is a digraph with the following properties:
  - 1) There is *exactly one* vertex, called the root, that has *no predecessors* and from which there is a *path to every vertex*.
  - 2) Each vertex other than the root has *exactly one predecessor.*
  - 3) The *successors* of each vertex are *ordered* "from the left."
- A successor of a vertex is called a son, and the predecessor is called the father. If there is a path from v_i to v_j, then v_i is said to be ancestor of v_j (and v_j is a descendant of v_i). A vertex with no sons is a leaf, the other vertices are interior vertices.

Example. A (directed) tree.



1.3 Inductive Proofs

- Many theorems are proved by mathematical induction.
- The Principle of Mathematical Induction:

  Let $P(n)$ be a statement (proposition) about a natural number n. Then:

  If $P(0)$ holds and $P(k\text{-}1) \Rightarrow P(k)$ holds for any $k \geq 1$, then $P(n)$ holds for every $n \geq 0$.
- $P(0)$ is called the basis, $P(k\text{-}1)$ is the inductive hypothesis, and $P(k\text{-}1) \Rightarrow P(k)$ is the inductive step.

Example. Proposition: $P(n) \equiv \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

PROOF.

Basis [take $n = 0$ in $P(n)$] $\sum_{i=0}^{0} i^2 = 0 = \frac{0(0+1)(2\cdot 0+1)}{6}$. So $P(0)$ holds.

Inductive hypothesis [suppose $P(k\text{-}1)$ holds] $\sum_{i=0}^{k-1} i^2 = \frac{(k-1)k(2k-1)}{6}$ holds.

Inductive step [does $P(k\text{-}1) \Rightarrow P(k)$ hold? Apply *ind.hyp.* to answer the question.]

$$\sum_{i=0}^{k} i^2 = \sum_{i=0}^{k-1} i^2 + k^2 \overset{Ind.hyp.}{=} \frac{(k-1)k(2k-1)}{6} + k^2 = \ldots = \frac{k(k+1)(2k+1)}{6}$$

So, $P(k)$ holds. Hence $P(k-1) \Rightarrow P(k)$ holds.

Conclusion: $P(n)$ holds for every natural n.

QED.

1.4 Set Notation

- A set is a collection of objects (members) without repetition.
- *Finite* sets may be specified by *listing* their members between brackets. Example. $\{0, 1\}$ is a set; $\{a,b,c,d,e,f,g,h,i,j,k\}$ is a set.
- We also specify sets by set formers:

  $\{x \mid P(x)\}$ ...the set of objects x such that $P(x)$ is true

  or

  $\{x \in A \mid P(x)\}$ ...the set of x in A such that $P(x)$ is true
- Example.

  $\{i \in \mathbb{N} \mid \text{there is integer } j \text{ such that } i = 2j\}$

- If every member of A is a member of B, then we write $A \subseteq B$ and say A is contained in B. $B \supseteq A$ is synonymous with $A \subseteq B$.
- If $A \subseteq B$ but $A \neq B$, then we write $A \subsetneq B$ or $A \subset B$. In this case we say that A is properly contained in B.
- Sets A and B are equal if they have the same members. That is, $A=B$ *iff* $A \subseteq B$ and $B \subseteq A$.

  (Here, *iff* means 'if and only if.')

The usual operations on sets are:

- $A \cup B$, the union of A and B, is $\{x \mid x \in A \text{ or } x \in B\}$
- $A \cap B$, the intersection of A and B, is $\{x \mid x \in A \text{ and } x \in B\}$
- $A - B$, the difference of A and B, is $\{x \mid x \in A \text{ and } x \notin B\}$
- $A \times B$, the Cartesian product of A and B, is

  $$\{(x, y) \mid x \in A \text{ and } y \in B\}$$
- 2^A, the power set of A, is $\{X \mid X \subseteq A\}$

  (The alternative notation for the power set of A is $\mathcal{P}(A)$.)

- Sets A and B have the *same* cardinality if there is a *bijection* $f:A \rightarrow B$.
  - *Finite* sets:
    - If A is a finite set, then its cardinality is a *natural* number, which denotes the number of its members.
    - If A,B are finite sets and $A \subset B$, then A and B have *different* cardinalities.
  - *Infinite* sets:
    - If A,B are infinite and $A \subset B$, then A and B *may* have the *same* cardinality! Example. Let A = *Even integers*, and B = *Integers*. Although $A \subset B$, there is a bijection $f: A \rightarrow B$, namely $f(i) = i/2$.
    - Not all infinite sets have the same cardinality. Example. $\mathbb{N}$ and $\mathbb{R}$. Sets that can be injectively mapped into $\mathbb{N}$ are countable or countably infinite. Examples. $\mathbb{Q}$ and Σ^* are countably infinite. The set $2^{\mathbb{N}}$ (of all *subsets* of $\mathbb{N}$) and the set of all functions from $\mathbb{N}$ to $\{0,1\}$ have the same cardinality as $\mathbb{R}$, so they are *not countable*.

1.5 Relations

- A binary relation R is a *set* of *pairs*:

$$R = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

- The first component of each pair is chosen from a set A called the domain of R, and the second component of each pair is chosen from a (possibly different) set B called the range of R.
- When A and B are the same set S, we say the relation is *on* S. If R is a relation and (a, b) is a pair in R, we often write aRb.

There are important properties of relations that a relation R on S *may* or *may not* have. In particular, we say that a relation R on S is

- reflexive if aRa for all $a \in S$
- irreflexive if aRa is false for all $a \in S$
- transitive if aRb and bRc imply aRc
- symmetric if aRb implies bRa
- asymmetric if aRb implies that bRa is false

Note: any asymmetric relation is irreflexive.

- Example. $<$ on $\mathbb{Z}$ is transitive and asymmetric (hence irreflexive).

A relation R that is reflexive, symmetric, and transitive is said to be an equivalence relation.

- An equivalence relation R on a set S partitions S into *disjoint nonempty* equivalence classes.
- That is, $S = S_1 \cup S_2 \cup \ldots$,where for every i and $j \neq i$:
  - $S_i \cap S_j = \emptyset$
  - for each $a, b \in S_i$, aRb is true (i.e. aRb)
  - for each $a \in S_i$ and $b \in S_j$, aRb is false (i.e. $\neg(aRb)$)
- The sets S_i are called *equivalence classes*. *Note*: the number of equivalence classes may be infinite.

Example.

- Define the relation R on $\mathbb{Z}$ as follows: $R(i,j)$ *iff* $i = j \bmod m$.
- R is an equivalence relation on $\mathbb{Z}$. (Prove.)
- Equivalence classes of R are:
  - $\{\ldots, \ -m, \ \mathbf{0}, \ m, \ 2m, \ \ldots\}$
  - $\{\ldots, -(m-1), \ \mathbf{1}, \ m+1, \ 2m+1, \ldots\}$

    …
  - $\{\ldots \ -1, \ \mathbf{m-1}, \ 2m-1, \ 3m-1, \ \ldots\}$

Let P be a set of (some) *properties of relations*. The P-closure of a relation R is the smallest relation that contains R and has all the properties in P. In particular:

- Let $P = \{transitivity\}$. Then P-closure of a relation R is denoted by R^+, called the transitive closure of R, and defined by
  1) If aRb, then aR^+b.
  2) If aR^+b and bRc, then aR^+c.
  3) Nothing is in R^+ unless it so follows from 1) and 2).

  R^+ is the *smallest transitive* relation containing R.
- Let $P = \{reflexivity, transitivity\}$. Then P-closure of a relation R is denoted by R^*, called the reflexive and transitive closure of R, and defined by $R^* = R^+ \cup \{(a,a) \mid a \in S\}$.

1.6 Synopsis of the Course

Computer Science has two major areas:

1. Theoretical Computer Science (TCS), which investigates the *fundamental ideas and models underlying computing*;
2. Engineering Techniques, which are needed and/or applied in the *design of computing systems* (both *hardware* and *software*), especially in the *application of theory to design*.

This course is an intro to the 1st area, *theoretical computer science.* (But we shall remark briefly on the most important applications).

TCS has its roots in diverse fields:

- *Mathematics* (problems in the *foundations* of *math* and *logic*)
- *Linguistics* (grammars for *natural languages*)
- *Electrical Engineering* (switching theory in *hardware design*)
- *Biology* (models for *neuron nets*)
- *Quantum Physics* (quantum algorithms in *quantum mechanics*)

What are the goals of TCS? Who needs it?

- The goal of TCS is to analyze and formalize
  - what engineers have done,
  - what engineers could in principle do,
  - what engineers in principle cannot do.
- Such an analysis is carried out by describing and classifying various theoretical *models of computation*.

What is a model of computation?

- **Definition**. A model of computation is a formal definition of the *basic notions* of algorithmic computation. It characterizes
  - what is meant by the notion of the *algorithm,*
  - what is the *environment* required to execute the algorithm,
  - how the algorithm *executes* in this environment.

Out of studies in mathematics, linguistics, electrical engineering, and biology, emerged various models of computation. Some of them are central to TCS:

- *finite automata*
- *pushdown automata*
- *Turing machines*

Many other models of computation are important as well:

- *two-way finite automata, Moore machines, Mealy machines, …*
- *linear bounded automata, …*
- *register machine (RAM, RASP), μ - recursive functions, (general) recursive functions, λ-calculus, Post machines, Markov algorithms, cellular automata, DNA-calculus, quantum Turing machines, …*

Finite automata (stem from *neuron nets* and *switching circuits*)

- today they serve in the design of:
  - *Lexical analyzers* (the part of a compiler that groups characters into indivisible units)
  - *Pattern matching*
  - *Text editors*
  - other *Text-processing / File-searching programs*
  - *Theorem provers*

Pushdown automata

- today they serve in the design of
  - *Parsers* (the part of a compiler that checks the syntax of a program (= string of indiv. units))

  and in
  - *Formal specification of programming languages*

Turing machines (stem from math and logic)

- TMs reveal one of the fundamental problems of CS, namely that *there are more functions than algorithms for computing them.*
- But this means that *there exist functions that are not computable !*
- For such functions f the following holds:

  there is no computer program that can ever be written, which given an argument x for f produces the value f(x) and works for all possible x.

Turing machines (cont'd)

- We can do more than tell *whether or not* a problem is solvable.
- If it *is* solvable does *not* mean there is a *practical* algorithm for it.
- We will see that:
  - there are abstract problems that are *solvable* by computer *but* require *inordinate amounts* of *time* and/or *space* for their solution.
  - there are *many realistic/important* problems that fall in this category. Such problems are called *intractable* (*or hard*)
  - The corresponding theory of intractable problems influences profoundly how we think about problems.

1.7 Exercises

1. A palindrome can be defined as a string that reads the same forward and backward, or by the following definition:

a) ε is a palindrome.
b) If a is any symbol, then the string a is a palindrome.
c) If a is any symbol and x is a palindrome, then axa is a palindrome.
d) Nothing is a palindrome unless it follows from (1, 2, 3).

Prove by induction that the two definitions are equivalent.

2. The strings of balanced parentheses can be defined in at least two ways.

A string w over alphabet {(,)} is balanced if and only if:

1) a) w has an equal number of ('s and)'s, and
   b) any prefix of w has at least as many ('s as)'s.

2) a) ε is balanced.
   b) If w is a balanced string, then (w) is balanced.
   c) If w and x are balanced strings, then so is wx
   d) Nothing else is a balanced string.

Prove by induction on the length of a string that (1) and (2) define the same class of strings.

3. Show that the following are equivalence relations and give their equivalence classes.

a) The relation R_1 on integers defined by iR_1j if and only if $i = j$.

b) The relation R_2 on people defined by $pR_2\,q$ if and only if p and q were born at the same hour of the same day of some year.

c) The same as (b) but "of the same year" instead of "of some year."

4. Find the transitive closure, the reflexive and transitive closure, and the symmetric closure of the relation

$$\{(1,2), (2, 3), (3, 4), (5, 4)\}$$

1.8 Dictionary

Symbol simbol letter črka digit števka string niz word beseda length dolžina empty string prazna beseda prefix predpona suffix pripona proper pravi concatenation stik to juxtapose stakniti, pripeti juxtaposition stik alphabet abeceda formal language formalni jezik palindrome palindrom graph graf vertex, node vozlišče edge povezava path pot cycle cikel directed graph usmerjen graf predecessor predhodnik successor naslednik tree drevo ancestor prednik descendant potomec leaf list interior vertex notranje vozlišče mathematical induction matematična indukcija inductive hypothesis induktivna hipoteza basis osnova inductive step korak indukcije, indukcijski korak set množica member pripadnik, element contain vsebovati properly contain strogo vsebovati equal enak operation operacija union unija intersection presek difference razlika Cartesian product kartezični produkt power set potenčna množica cardinality moč countable števna countably infinite števno neskončna binary relation binarna relacija domain domena range zaloga vrednosti reflexive refleksivna irreflexive irefleksivna transitive tranzitivna symmetric simetrična asymmetric asimetrična equivalence relation ekvivalenčna relacija partition razbitje, razdelitev equivalence class ekvivalenčni razred transitive closure tranzitivna ovojnica (tr. zaprtje) reflexive and transitive closure refleksivna tranzitivna ovojnica (refl. tr. zaprtje) model of computation računski model finite automaton končni avtomat regular expression regularni izraz pushdown automaton skladovni avtomat context-free grammar kontekstno neodvisna gramatika Turing machine Turingov stroj intractable (or hard) problem neobvladljiv (oz. težek) problem

2
Finite Automata and Regular Expressions

Contents

- Finite state systems
- Deterministic finite automata, DFA
- Nondeterministic finite automata, NFA
- The equivalence of NFA and DFA
- Finite automata with ε-moves
- Equivalence of NFA with and without ε-moves

2.1 Finite State Systems

- A finite state system (FSS) is an object that can read *discrete inputs* and can be in any one of a *finite* number of *internal configurations* called *states.*
  - The state summarizes the information (concerning past inputs) needed to determine the behavior of the system on subsequent inputs.
  - The finite automaton is a *mathematical model of a FSS.*
- E.g., the **control mechanism of an elevator** is a FSS. That mechanism does not remember all previous requests for service but only the *current floor*, the *direction of motion* (up or down), and the *collection of not yet satisfied requests for service*.

- **Examples**. There are many examples of FSSs in CS.
  - Switching circuits (e.g., computer's CPU).
    - A switching circuit consists of a *finite* number of *gates*, each of which can be in one of *two* conditions, 0 and 1 (different voltage levels at the gate output).
    - The *state* of a circuit with n gates can be any one of 2^n assignments of 0 or 1 to the gates.
    - (Comment. The circuitry is so designed that only the two voltages corresponding to 0 and 1 are stable; other voltages immediately adjust themselves to one of these voltages. Switching circuits are intentionally designed in this way, *so that they can be viewed as FSSs*, thereby separating the *logical design* of a computer from the *electronic implementation*.)
  - Certain programs (e.g. text editors and lexical analyzers)
    - A *lexical analyzer* scans the symbols of a computer program to locate the strings of characters corresponding to *identifiers*, *numerical constants*, *reserved words*, and so on. In this process the lexical analyzer needs to remember only a *finite* amount of information (e.g., the length of a prefix of a reserved word that has seen since startup).
- The theory of finite automata is used in the design of such FSSs.

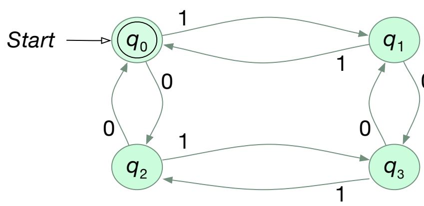
- **Question**. Does FSS capture the notion of *computation*?
  - *No.* FSS is *not* satisfying neither mathematically nor realistically.
  - *Why*? FSS places an artificial *limit on the memory capacity*, and because of that fails to capture the *essence of computation*.
  - To properly capture the *notion of computation* we need a *potentially infinite memory* (even though each real computer is finite).
- Such an infinite model is, for example, the Turing Machine. We'll describe it in Chapter 7.

2.2 Deterministic Finite Automata

- A deterministic finite automaton (DFA) consists of a finite set of states and a set of transitions from state to state that occur on reading symbols from an input alphabet Σ.
- For each input symbol there is *exactly one* transition out of each state.
- One state, denoted q_0, is the initial state. DFA starts in q_0. Some states are designated as final (or accepting) states.
- We say that a DFA accepts a word x if the sequence of transitions corresponding to the symbols of x leads from the initial state q_0 to an accepting state.

- A DFA is associated with a transition diagram (digraph) whose
  - vertices correspond to the *states* of the DFA.
  - arcs correspond to *transitions*: there is an arc $q_i \xrightarrow{a} q_j$ if DFA moves from state q_i to state q_j on reading input symbol a.

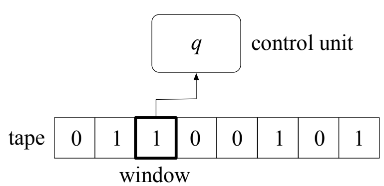
Example.



q_0 is the initial state. Final states are in double circles (here q_0).

- **Definition.** A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:
  - Q is a finite set of states,
  - Σ is a finite input alphabet,
  - $q_0 \in Q$ is the initial state,
  - $F \subseteq Q$ is the set of final states, and
  - δ is the transition function, i.e. $\delta : Q \times \Sigma \to Q$.
    That is, $\delta(q, a)$ *is a state* (for each state q and input symbol a).
- *Note*: δ is the *program* of DFA. Every DFA has its own, specific δ.

- We view a DFA as a consisting of a control unit that reads *input word* ($\in \Sigma^*$) from a tape, and during this changes its state.



- If it is *in state q* and *reads symbol a*, then the DFA, *in one* move,
  - 1) *enters* the next state which is $\delta(q, a)$,
  - 2) *shifts* its window *one symbol* to the *right*.
- If $\delta(q, a)$ is an *accepting* state, the DFA has accepted the prefix of the input word up to (not including) the current position of the window. A DFA may accept several prefixes. If the window has moved off the right end of the input word, DFA accepts the entire word.

- It is useful to extend δ so that it can be applied to a state and a *string* (instead just one *symbol*).
- We define a function $\hat{\delta} : Q \times \Sigma^* \to Q$ so that $\hat{\delta}(q, x)$ *is the state in which* DFA *is after reading x starting in state q.* So $\hat{\delta}(q, x)$ is *the state p such that there is a path in the diagram from q to p, labeled x.*
- The extended transition function $\hat{\delta}$ is defined as follows:
  - $\hat{\delta}(q, \varepsilon) = q$
  - $\hat{\delta}(q, wa) = \delta(\underline{\hat{\delta}(q, w)}, a)$, for all strings w and input symbols a

- Since $\hat{\delta}(q, a) = \delta(q, a)$ (prove!) there can be no disagreement between δ and $\hat{\delta}$. So, for *convenience* we will write δ instead of $\hat{\delta}$.

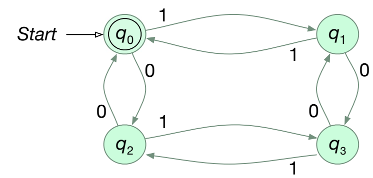
Definitions.

- A string x is said to be accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if $\delta(q_0, x) = p$ for some $p \in F$.
- The language accepted by a DFA M is defined as the set

$$L(M) = \{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$$

- A language is said to be a regular set (or just regular) if it is accepted by some DFA (i.e. if it is $L(M)$ for some DFA M).

Example.



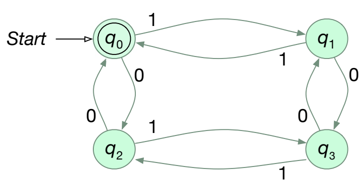
q_0 is the initial state. Final states are in double circles (here q_0).

- This is the transition diagram of $M = (Q, \Sigma, \delta, q_0, F)$, where

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$\Sigma = \{0, 1\}$$
$$F = \{q_0\}$$

δ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Example (cont'd).



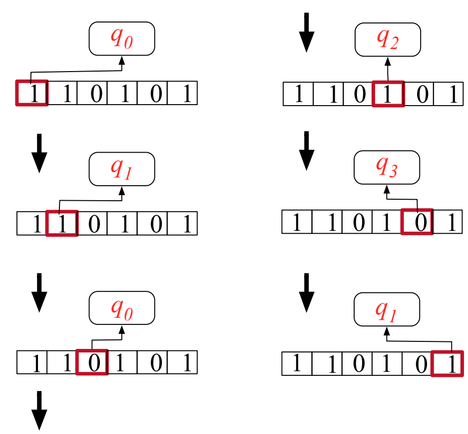
δ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

q_0 is the initial state. Final states are in double circles (here q_0).

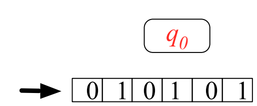
- Suppose $x = 110101$ is input to M. Is $x \in L(M)$?
  - We must compute the state $\delta(q_0, x) = \delta(q_0, 110101)$.
  - $$\begin{aligned} &\delta(q_0, 110101) \\ &= \delta(q_1, 10101) \\ &= \delta(q_0, 0101) \\ &= \delta(q_2, 101) \\ &= \delta(q_3, 01) \\ &= \delta(q_1, 1) = q_0 \in F \end{aligned}$$

Example (cont'd).

- Computation of M on $x = 110101$.

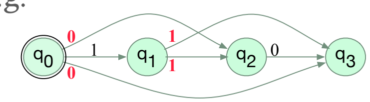


δ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



2.3 Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) is obtained from DFA by allowing *zero*, *one* or *more* transitions from a state on the *same* input symbol; e.g.



- An input word $a_1 a_2 \cdots a_n$ is accepted by a NFA if there **exists** a sequence of transitions, corresponding to the input word, that leads from the initial state to *some final* state.

- Thus in a DFA, for a given input string w and state q, there will be *exactly one* path labeled w starting at q. To determine if a string is *accepted* by a DFA it suffices to check *this one* path. In contrast, for an NFA there may be *many* paths labeled w, and in the worst case *all* must be checked to see if at least one ends in a final state.

- **Nondeterminism.**

- **Question:** Given an input word $a_1 a_2 \cdots a_n$, who decides *whether or not* there **exists** a sequence of transitions leading from initial to some final state?
Answer: NFA itself.

- **Question:** How does NFA do that?
Answer: The NFA is *not* a *realistic* model of computation: it is *assumed* that NFA can *always guess right*. That is, it is assumed that NFA has the *magic* capability of choosing, from any given set of options, the right option, i.e. the option that leads to a success (if such an option exists; otherwise, NFA halts).

  In particular, if there are several transitions from a state on the same input symbol, the NFA can immediately choose the one (if there is such) which eventually leads to some final state.

- This capability of prediction makes NFA unrealistic.

- **Nondeterminism** (cont'd).

- **Question:** If NFA is unrealistic, who needs it?
Answers:
  - NFA (and other nondeterministic models that we will see later) can be used to find *lower bounds* on the time required to solve computational problems. The reasoning is as follows: If a problem P requires time T to be solved by a *nondeterministic* model M, then solving this problem on *any deterministic* version D of the model M must require *at least* time T (because D lacks the ability of prediction).

    We will use this in chapters on *Computational Complexity*.
  - Often it is much*easier* to design a NFA (or some other *nondeterministic* model) for a given problem P. We then try to construct an *equivalent deterministic* version (equivalent in the sense that it solves P too, regardless of the time needed).

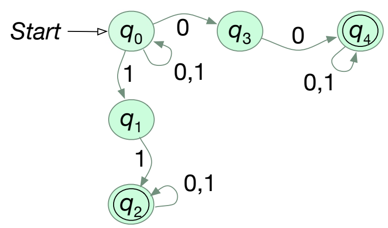
    We will see this soon.

- **Definition.** A nondeterministic finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$,where:
  - Q is a finite set of states,
  - Σ is a finite input alphabet,
  - $q_0 \in Q$ is the initial state,
  - $F \subseteq Q$ is the set of final states, and
  - δ is the transition function, i.e. $\delta : Q \times \Sigma \to 2^Q$.

    That is, $\delta(q, a)$ is the *set* of all states p such that there is a transition labeled a from q to p.

- *Note*: δ is the *program* of NFA. Every NFA has its own specific δ.

Example.

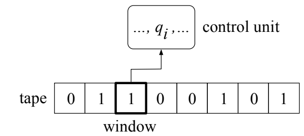


- This is the transition diagram of NFA $M = (Q, \Sigma, \delta, q_0, F)$, where

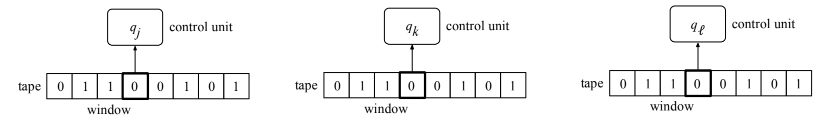
$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$
$$\Sigma = \{0, 1\}$$
$$F = \{q_2, q_4\}$$

δ	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

- We view a NFA similarly to DFA. It also reads an input tape, but the *control unit at any time can be in any number of states*.

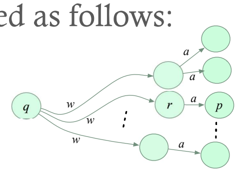


- When a *choice* of the next state can be made, we may *imagine* that duplicate *copies* of the automaton are made. For each *possible next state* there is *one copy* of the automaton whose control unit is in *that state*. Example. if $\delta(q_i, 1) = \{q_j, q_k, q_\ell\}$, we imagine three copies:



- We imagine that each of the copies continues execution independently of the others in the same fashion. The imaginary parallel computation is described by the execution tree.

- To describe the behavior of a NFA on a *string*, we extend δ to apply to a state and a *string* (instead a state and just a *symbol*).
- We define a function $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ so that $\hat{\delta}(q, x)$ *is the set of states* NFA *can be in after reading x starting in q*. So, $\hat{\delta}(q, x)$ is the *set of states to each of which there is a path from q, labeled x*.
- The extended transition function $\hat{\delta}$ is defined as follows:
  - $\hat{\delta}(q, \varepsilon) = \{q\}$
  - $\hat{\delta}(q, wa) = \{p \in Q \,|\, \exists r \in \hat{\delta}(q, w) : p \in \delta(r, a)\}$

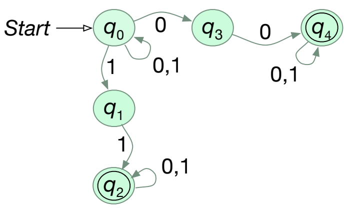


- Since $\hat{\delta}(q, a) = \delta(q, a)$ (prove!), we will for *convenience* write δ instead of $\hat{\delta}$.
- It is useful to extend δ to *sets of states* by $\delta(S, x) = \bigcup_{q \in S} \delta(q, x)$.

- **Definitions.**
  - A string x is said to be accepted by a NFA $M = (Q, \Sigma, \delta, q_0, F)$ if $\delta(q_0, x)$ contains some $p \in F$ (i.e $\delta(q_0, x) \cap F \neq \emptyset$).
  - The language accepted by a NFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set

$$L(M) = \{x \in \Sigma^* \,|\, \delta(q_0, x) \text{ contains a state in } F\}$$

Example (cont'd).



	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

- Suppose $x = 01001$ is input to M. Is x in $L(M)$?
  - We must compute the state $\delta(q_0, x) = \delta(q_0, 01001)$.
  - $\delta(q_0, 01001)$

$$
\begin{aligned}
&= \delta(\{q_0, q_3\},\ 1001) \\
&= \delta(\{q_0, q_1\} \cup \emptyset, 001) = \delta(\{q_0, q_1\}, 001) \\
&= \delta(\{q_0, q_3\} \cup \emptyset, 01) = \delta(\{q_0, q_3\}, 01) \\
&= \delta(\{q_0, q_3\} \cup \{q_4\}, 1) = \delta(\{q_0, q_3, q_4\}, 1) \\
&= \delta(\{q_0, q_1\} \cup \emptyset \cup \{q_4\}, \varepsilon) = \delta(\{q_0, q_1, q_4\}, \varepsilon),\ \text{and } q_4 \in F
\end{aligned}
$$

2.4 Equivalence of DFA's and NFA's

- Every DFA is also NFA.
  - Why? A DFA can be viewed as a *trivial* NFA.
- So the class of languages accepted by NFAs *includes* all the languages accepted by DFAs (*regular* sets).
- But, these are the *only* sets accepted by NFAs !
  - How do we know that? For every NFA there is an *equivalent* DFA (i.e. which accepts the *same* language as the NFA)!
    - The next theorem shows how we construct an equivalent DFA that *simulates* a given NFA.

- **Theorem.** Let L be a set accepted by a NFA M.
  Then there exists a DFA M' that accepts L.
- **Proof idea.**
  - The *states* of the DFA M' correspond to *sets of states* of NFA M.
  - The control unit of the DFA M' keeps track of all states in which NFA M *could have been had it read* the same input as M'.

- **Proof.**
- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA accepting L.
- We define a DFA $M' = (Q', \Sigma, \delta', q_0', F')$ as follows:
  - $Q' = 2^Q$. That is, the *states* of M' correspond *to sets of states* of M.
    How? A state of M' will *encode* all the states in which M *could be* at that moment.
    Specifically, a state of M' will be *denoted* by $[q_{i_1}, \ldots, q_{i_k}]$, where $q_{i_1}, \ldots, q_{i_k} \in Q$.
    So $[q_{i_1}, \ldots, q_{i_k}]$ is a *state* of M' encoding the *set* $\{q_{i_1}, \ldots, q_{i_k}\}$ *of states* of M.
  - $q_0' = [q_0]$
  - $\delta'([q_{i_1}, \ldots, q_{i_k}], a) = [p_{j_1}, \ldots, p_{j_\ell}]$ iff $\delta(\{q_{i_1}, \ldots, q_{i_k}\}, a) = \{p_{j_1}, \ldots, p_{j_\ell}\}$
    That is, δ' applied to a state $[q_{i_1}, \ldots, q_{i_k}]$ of M' is computed by (1) applying δ to each state in $\{q_{i_1}, \ldots, q_{i_k}\}$ and (2) taking the union of the obtained sets.
    The union is a new set of states, $\{p_{j_1}, \ldots, p_{j_\ell}\}$, which is encoded in M' by $[p_{j_1}, \ldots, p_{j_\ell}]$. This is the *value* of $\delta'([q_{i_1}, \ldots, q_{i_k}], a)$.
  - F' is the set of all states in Q' containing a final state of M.

- **Proof (cont'd).**
- Next, we show that

  $\delta'(q_0', x) = [q_{i_1}, \ldots, q_{i_k}]$ iff $\delta(q_0, x) = \{q_{i_1}, \ldots, q_{i_k}\}$, for arbitrary $x \in \Sigma^*$.

  - We prove this by induction on the length $|x|$ of the input string x. (Exercise.)
- Finally, we add that

  $\delta'(q_0', x) \in F'$ iff $\delta(q_0, x)$ contains a state of Q that is in F.

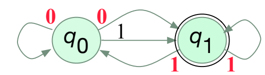
- Thus, $L(M) = L(M')$. $\square$

Example.

- Let $M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$ be an NFA where

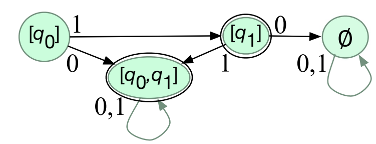
  $\delta(q_0, 0) = \{q_0, q_1\}$ $\quad \delta(q_1, 0) = \emptyset$

  $\delta(q_0, 1) = \{q_1\}$ $\quad \delta(q_1, 1) = \{q_0, q_1\}$



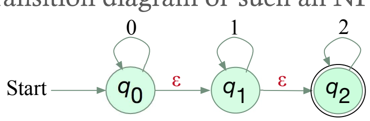
- The DFA $M' = (Q', \{0,1\}, \delta', [q_0], F')$ accepting $L(M)$ has:
  - $Q' = 2^Q = \{\emptyset, [q_0], [q_1], [q_0, q_1]\}$, i.e. all subsets of $Q = \{q_0, q_1\}$
  - $\delta'(\emptyset, 0) = \emptyset$ $\quad \delta'([q_0], 0) = [q_0, q_1]$ $\quad \delta'([q_1], 0) = \emptyset$ $\quad \delta'([q_0, q_1], 0) = [q_0, q_1]$

    $\delta'(\emptyset, 1) = \emptyset$ $\quad \delta'([q_0], 1) = [q_1]$ $\quad \delta'([q_1], 1) = [q_0, q_1]$ $\quad \delta'([q_0, q_1], 1) = [q_0, q_1]$
  - $F = \{[q_1], [q_0, q_1]\}$



2.5 NFA's with ε-Moves

- We may extend the model of the NFA to include spontaneous transitions, that is, *transitions on the empty input* ε .
- **Example.** The transition diagram of such an NFA is:

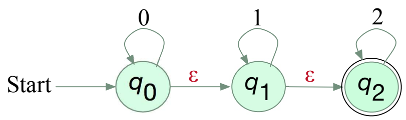


  - The NFA accepts words consisting of any number (including zero) of 0's followed by any number of 1's followed by any number of 2's. Why?
  - The answer is: The NFA accepts a string x if there is a path labeled x from q_0 to q_2. *But edges labeled* ε *may be included in the path, although* ε *does not appear explicitly in* x.
  - For example, $x = 002$ is accepted because there is a path $q_0, q_0, q_0, q_1, q_2, q_2$ with arcs labeled 0, 0, ε , ε , 2.

- **Definition.** A NFA with ε-moves (NFA$_\varepsilon$) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:
  - Q is a finite set of states,
  - Σ is a finite input alphabet,
  - $q_0 \in Q$ is the initial state,
  - $F \subseteq Q$ is the set of final states, and
  - δ is the transition function, i.e. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$

    That is, $\delta(q, a)$ is the *set* of all states p such that there is a transition labeled a from q to p, *where* a *is either* ε *or a symbol in* Σ.
- *Note*: δ can be viewed as a program of NFA$_\varepsilon$. Every NFA$_\varepsilon$ has its own specific δ.

- **Example** (cont'd). The NFA$_\varepsilon$ corresponding to the diagram

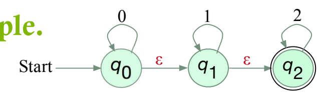


is $(Q, \Sigma, \delta, q_0, F)$, where:

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1, 2\}$
- $F = \{q_2\}$
- $\delta =$

	0	1	2	ϵ
q_0	$\{q_0\}$	$\emptyset$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

- To describe the behavior of such an NFA$_\varepsilon$ on a *string*, we must extend δ to apply to a state and a *string*.
- We will define a function $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ so that $\hat{\delta}(q, x)$ will be *the set of states p such that there is a path labeled x from q to p, perhaps including edges labeled* ε.
  - In the definition of $\hat{\delta}$ we'll need to compute the set of all states *reachable* from the state q with ε-transitions *only* : ε-Closure(q).
  - **Example.**



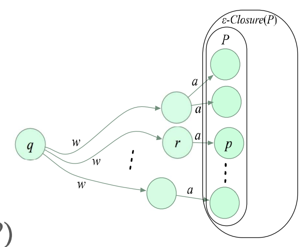
ε-Closure$(q_0) = \{q_0, q_1, q_2\}$,
ε-Closure$(q_1) = \{q_1, q_2\}$,
ε-Closure$(q_2) = \{q_2\}$.

  - We extend the definition to *sets*: ε-Closure$(S) = \bigcup_{q \in S} \varepsilon\text{-Closure}(q)$

- The extended transition function $\hat{\delta}$ is defined inductively:
  - $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-Closure}(q)$
  - For $w \in \Sigma^*$ and $a \in \Sigma$ we have

$$\hat{\delta}(q, wa) = \varepsilon\text{-Closure}(P)$$

where $P = \{p \,|\, \exists r \in \hat{\delta}(q, w) : p \in \delta(r, a)\}$

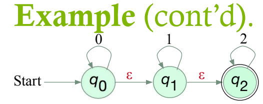


- But now, in general, $\hat{\delta}(q, a) \neq \delta(q, a)$. (Why?)
- We can also extend δ and $\hat{\delta}$ to *sets of states*; if R is a set of states, then

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

$$\hat{\delta}(R, x) = \bigcup_{q \in R} \hat{\delta}(q, x)$$

- **Example** (cont'd). The NFA$_\varepsilon$ with the transition diagram



has $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, $\delta =$

	0	1	2	ϵ
q_0	$\{q_0\}$	$\emptyset$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

Suppose $x = 01$ is the input. What is $\hat{\delta}(q_0, 01)$?

- $\hat{\delta}(q_0, \varepsilon) = \varepsilon\text{-Closure}(q_0) = \{q_0, q_1, q_2\}$
- $\hat{\delta}(q_0, 0) = \varepsilon\text{-Closure}(\delta(\hat{\delta}(q_0, \varepsilon), 0)) = \varepsilon\text{-Closure}(\delta(\{q_0, q_1, q_2\}, 0))$
  $= \varepsilon\text{-Closure}(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)) = \varepsilon\text{-Closure}(\{q_0\} \cup \emptyset \cup \emptyset)$
  $= \varepsilon\text{-Closure}(\{q_0\}) = \{q_0, q_1, q_2\}$
- $\hat{\delta}(q_0, 01) = \varepsilon\text{-Closure}(\delta(\hat{\delta}(q_0, 0), 1) = \varepsilon\text{-Closure}(\delta(\{q_0, q_1, q_2\}, 1))$
  $= \varepsilon\text{-Closure}(\{q_1\}) = \{q_1, q_2\}$

- **Definitions.**
  - A string x is said to be accepted by an NFA$_\varepsilon$ $M = (Q, \Sigma, \delta, q_0, F)$ if $\hat{\delta}(q_0, x)$ contains some $p \in F$.
  - The language accepted by an NFA$_\varepsilon$ $M = (Q, \Sigma, \delta, q_0, F)$ is the set

$$L(M) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \text{ contains a state in } F\}$$

2.6 Equivalence of NFA$_\varepsilon$'s and NFA's

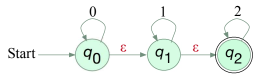
- The ability to make transitions on ε does *not* allow NFA$_\varepsilon$s to accept *non*-regular sets.
- Why? We'll see that NFAs can *simulate* NFA$_\varepsilon$s. That is, for every NFA$_\varepsilon$ there is an *equivalent* NFA (accepting the *same* language as the NFA$_\varepsilon$).

- **Theorem.** Let L be a set accepted by an NFA$_\varepsilon$ M. Then there exists an NFA M' that accepts L.
- **Proof idea.**
  - We want M' to simulate a move of M for each pair of state and input, (q,a). Since M can make also ε-transitions during a move, M' must be able to change to a state p if there is a path in the diagram of M from q to p labeled a, possibly with ε-transitions. Hence, we want $\delta'(q, a) = \hat{\delta}(q, a)$.

- **Proof.**
- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA$_\varepsilon$ accepting L.
- We define a NFA, $M' = (Q, \Sigma, \delta', q_0, F')$ as follows:
  - $\delta' = \hat{\delta}$, that is, $\delta'(q, a) = \hat{\delta}(q, a)$ for every $q \in Q$ and $a \in \Sigma$.
  - $F' = \begin{cases} F \cup \{q_0\} & \text{if } \varepsilon\text{-Closure}(q_0) \text{ contains a state in } F, \\ F & \text{otherwise.} \end{cases}$
- *Note*: M' has no ε-transitions (it is an NFA). So we can use δ' instead of $\hat{\delta}'$. But δ and $\hat{\delta}$ must still be distinguished (as they belong to an NFA$_\varepsilon$).
- *Lemma*. $\delta'(q_0, x) = \hat{\delta}'(q_0, x)$ for $|x| \geqslant 1$.
  Proof: Induction on $|x|$. Exercise. □
- Finally, we prove: $\delta'(q_0, x)$ contains a state of F' *iff* $\delta'(q_0, x)$ contains a state of F.
  Proof. Exercise. □

□

- **Example** (cont'd). The NFA$_\varepsilon$ M with the transition diagram



has: $Q = \{q_0, q_1, q_2\}$ $\quad F = \{q_2\}$ $\quad \delta =$

	0	1	2	ϵ
q_0	$\{q_0\}$	$\emptyset$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

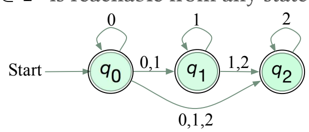
The equivalent NFA is $M' = (Q, \Sigma, \delta', q_0, F')$, where:

- $\delta'(q,a) = \hat{\delta}(q,a) =$

	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$

- $F' = \{q_0, q_1, q_2\}$ (because $q_2 \in F$ is reachable from any state of Q)

The diagram of the NFA M' is:



2.7 Regular Expressions

- The languages accepted by finite automata are easily described by simple expressions called regular expressions.
- In this section we
  - introduce operations of concatenation and closure on languages,
  - define regular expressions, and
  - prove that the class of languages accepted by finite automata is the same as the class of languages describable by regular expressions.

- **Definition.** Let Σ be an alphabet. Let L_1, L_2, L be sets of words from Σ^*. The concatenation of L_1, L_2, denoted L_1L_2, is the set

$$L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

Words in L_1L_2 are formed by taking an x in L_1 and following it by a y in L_2, for all possible x, y.

- **Definition.** Let $L^0 = \{\varepsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The Kleene closure (in short closure) of L, denoted L^*, is the set

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

and the positive closure of L, denoted L^+, is the set

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

L^* is the set of words that are constructed by concatenating *any number* of words from L.
L^+ is the same, but the case of *zero* words (whose concatenation is defined to be ε), is excluded.
Note: L^+ contains ε *iff* L contains ε. (Why? Exercise.)

- **Example.**

  Let $L_1 = \{10, 1\}$ and $L_2 = \{011, 11\}$.
  - Then: $L_1L_2 = \{10, 1\}\{011, 11\} = \{10011, 1011, 111\}$. (*Note:* 1011 = 1011.)
  - Also: $L_1^* = \{10, 1\}^*$
    $= L_1^0 \cup L_1^1 \cup L_1^2 \cup \ldots$
    $= \{10, 1\}^0 \cup \{10, 1\}^1 \cup \{10, 1\}^2 \cup \ldots$
    $= \{\varepsilon\} \cup \{10, 1\} \cup \{1010, 101, 110, 11\} \cup \ldots$
    $= \{\varepsilon, 10, 1, 1010, 101, 110, 11, \ldots\}$
  - And: $L_1^+ = \{10, 1\}^+ = \{10, 1, 1010, 101, 110, 11, \ldots\}$
- **Example.**

  Let Σ be an alphabet. Σ^* is the set of all strings of symbols in Σ.
  - Let $\Sigma = \{1\}$. Then $\Sigma^* = \{\varepsilon, 1, 11, 111, 1111, \ldots\}$
  - Let $\Sigma = \{0,1\}$. Then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}$

- **Definition.** Let Σ be alphabet. The regular expressions (r.e.) over Σ (and the sets that they denote) are defined inductively as follows:
  1) $\emptyset$ is a r.e.; it denotes the *empty set*, $\emptyset$;
  2) ε is a r.e.; it denotes the set $\{\varepsilon\}$;
  3) For each $a \in \Sigma$, $\boldsymbol{a}$ is a r.e.; it denotes the set $\{a\}$;
  4) If r and s are r.e.s denoting languages R and S, respectively, then
     a) $(r+s)$ is a r.e.; it denotes the set $R \cup S$; (union of R and S)
     b) (rs) is a r.e.; it denotes the set RS; (concatenation of R and S)
     c) (r^*) is a r.e.; it denotes the set R^*. (Kleene closure of R)

  Note. The *basic* r.e.s are defined *explicitly* (1,2,3). *All the other* r.e.s are defined *inductively* (4a,b,c). Definitions of this kind are called *inductive*. Properties of the defined objects are often proved by induction.

- **Conventions.**
- We can *omit many parentheses*
  - if we assume that * has *higher precedence* than concatenation and concatenation has higher *higher precedence* than +.

    Example. $((0(1^*)) + 0)$ may be written $01^* + 0$.
  - if we abbreviate the expression rr^* by r^+.
- When
  - *necessary to distinguish* between a regular expression r and the language denoted by r, we use $L(r)$ for the latter;
  - *no confusion is possible* we use r for both the regular expression and the language denoted by the regular expression.

- **Examples.**
  - **00** is a regular expression that denotes the set $\{00\}$.
  - **0*** denotes the set of *strings of any number of* 0s
  - $\mathbf{0^+}$ denotes the set of *strings of at least one* 0
  - **0*1*** denotes the set of *strings of any number of* 0s *followed by any number of* 1s.
  - $\mathbf{0^+1^+}$ denotes the set of *strings with at least one* 0 *followed by at least one* 1.
  - **(0+1)*** denotes the set of *all strings of* 0s *and* 1s.
  - **(0+1)*11** denotes the set of *strings of* 0's *and* 1's *ending in* 11.
  - **(0+1)*00(0+1)*** denotes the set of *strings of* 0s *and* 1s *with at least two consecutive* 0s.
  - **(1+10)*** denotes the set of *strings of* 0s *and* 1s *beginning with* 1 *and not containing* 00. (*Proof*: Induction on i that strings denoted by $\mathbf{(1+10)}^i$ begin with 1 and have no 00.)
  - $\mathbf{(0+\varepsilon)(1+10)^*}$ denotes the set of *all strings of* 0s *and* 1s *whatsoever containing no* 00
  - **0*1*2*** *strings of any num. of* 0's *followed by any num. of* 1s *followed by any num. of* 2s.
  - **00*11*22*** *strings of at least one* 0 *followed by at least one* 1 *followed by at least one* 2. (We may use the shorthand $\mathbf{0^+1^+2^+}$ for **00*11*22***)

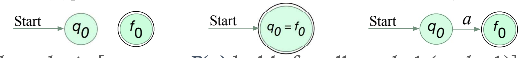
2.8 Equivalence of Finite Automata and Regular Expressions

- We will show that the languages *accepted* by finite automata are precisely the languages *denoted* by regular expressions. (This is why finite automaton languages are called *regular* sets.)
- How? In two steps, by showing that
  - For every regular expression r there is a NFA_ε *accepting* the language $L(r)$. (But NFA_εs are equivalent to NFAs and to DFAs, so they all accept the same class of languages.)
  - For every DFA M there is a regular expression *denoting* the language $L(M)$.
- So, the four language-defining ways (DFA, NFA, NFA_ε, regular expression) define the *same class of languages*, the *regular sets*.

- **Theorem.** Let r be an arbitrary regular expression. Then there exists an NFA$_\varepsilon$ that accepts $L(r)$.
- **Proof idea.** We use *induction on the number of operators in r* to show that, for *any* r.e. r, there exists an NFA$_\varepsilon$ $M = (Q, \Sigma, \delta, q_0, \{f_0\})$ with *one* final state and no transitions out of it, such that $L(M) = L(r)$.

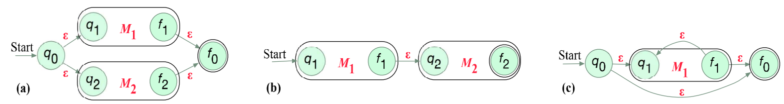
  Note. NFA$_\varepsilon$s with *just one* final state will enable us to easily *combine* them into larger NFA$_\varepsilon$s. No generality will be lost in this way. (Why? Show how an arbitrary general NFA can be transformed into such equivalent NFA$_\varepsilon$.)

- **Proof.**
  - Let $P(n) \equiv$ 'If r is a r.e. with n operators, then there is a NFA$_\varepsilon$ M such that $L(M) = L(r)$.'
  - We prove $P(n)$ by induction on n.
  - *Basis* [check $P(0)$]. If n=0, then r is either $\emptyset$, ε, or $\mathbf{a}$ ($a \in \Sigma$). The associated NFA$_\varepsilon$s are:



  - *Inductive hypothesis* [suppose $P(n)$ holds for all $n \leq k$-1 (so $k \geq 1$)]
  - *Inductive step* [show that then $P(n)$ holds for all $n \leq k$]

    Let r have k operators. There are three cases depending on the form of r:
    - $r = r_1 + r_2$. Each of r_1 ,r_2 has $\leq k$-1 operators. By ind.hyp. there are NFA$_\varepsilon$s M_1, M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The NFA$_\varepsilon$ M corresponding to r is in Fig. (a).
    - $r = r_1 r_2$. Each of r_1, r_2 has $\leq k$-1 operators. By ind.hyp. there are NFA$_\varepsilon$s M_1, M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The NFA$_\varepsilon$ M corresponding to r is in Fig. (b).
    - $r = r_1$*. Here, r_1 has $\leq k$-1 operators. By ind.hyp there is an NFA$_\varepsilon$ M_1 such that $L(M_1) = L(r_1)$. The NFA$_\varepsilon$ M corresponding to r is in Fig. (c).



□

- **Example.**
  - Vaje.

- **Theorem.** Let M be an arbitrary DFA. There exists a regular expression that denotes $L(M)$.
- **Proof idea.**
  - We view $L(M)$ as a *union of sets* (finitely many) .
  - Each of the sets corresponds to a *final state* of M and contains all the words that take M from the initial state to this final state.
  - We then define these sets *inductively* (bottom up, by simpler sets). In parallel we construct to each such set the corresponding regular expression.

- **Proof.**
  - Let be given a DFA $M = (\{q_1, \ldots, q_n\}, \Sigma, \delta, q_1, F)$.
  - By definition: $L(M)$ = 'set of all words that take M from *initial* q_1 to *any final* q_j'
  - **Let** $R^n_{1j} \equiv$ 'the set of all words that take M from q_1 to q_j'. Then $L(M) = \bigcup_{q_j \in F} R^n_{1j}$.
  - So, *if we knew how to construct r.e.s for* R^n_{1j}s, then the *sum* of r.e.s would denote $L(M)$.
  - **Let** $R^k_{ij} \equiv$ 'the set of all words taking M from q_i to q_j and crossing no state indexed $>k$.'
  - Note: R^k_{ij} can be constructed *inductively*: $R^k_{ij} = R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$ (*)

$$R^0_{i,j} = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{a \mid \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & \text{if } i = j \end{cases} \quad (**)$$

  - Can we construct the *r.e.* r^k_{ij} (for R^k_{ij}) *along* the construction of R^k_{ij} ?
  - Yes, we can; this is done in the constructive proof of the following proposition.
  - **Proposition**: $P(k) \equiv$ 'For each i,j,k there is a r.e. r^k_{ij} denoting R^k_{ij}.'

    Proof (induction on k).

    Basis [check $P(0)$]. (**) suggests that R^0_{ij} is denoted by $r^0_{ij} = a_1+\ldots+a_p$ or $r^0_{ij} = a_1+\ldots+a_p + \varepsilon$.

    Ind.hyp. [assume $P(k$-1) holds]. So, for each i,j,k there is a r.e. r^{k-1}_{ij} denoting R^{k-1}_{ij}.

    Ind.step [does $P(k\text{-}1) \Rightarrow P(k)$ hold?]

    (*) and *ind.hyp.* tell us that R^k_{ij} is denoted by the r.e. $r^k_{ij} = r^{k-1}_{ik}(r^{k-1}_{kk})^* r^{k-1}_{kj} + r^{k-1}_{ij}$.

    □

  □

- **Example.**
  - Vaje.

2.9 Applications of Finite Automata

- There are a many *software design problems* that are simplified by automatic conversion of regular expressions to efficient computer implementations of the corresponding DFAs.
- Such software design problems include the design of:
  - *Lexical analyzers*
  - *Text editors*
  - *Data compressors*

- **Lexical analyzers.**
- *Lexical analyzer* is a program that performs lexical analysis. *Lexical analysis* is the process of converting a *sequence of characters* (e.g. program, web page, …) into *a sequence of language tokens*.
  A *language token* is a string with an identified meaning (e.g. keyword, identifier, literal, numeric constant, …).
- Language tokens are usually expressible as regular expressions.
  - Examples.
    - An ALGOL *identifier* is an upper- or lower-case letter followed by any sequence of letters and digits, with no limit on length. Such identifiers are expressed as (letter)(letter+digit)*, where letter = (**A**+**B**+…+**Z**+**a**+**b**+…+**z**) and digit = (**0**+**1**+…+**9**).
    - A FORTRAN *indentifier* has length limit 6 and letters restricted to upper-case and $. These identifiers are expressed as (letter)(ε+letter+digit)^5 where letter = ($+**A**+**B**+…+**Z**).
    - A SNOBOL *arithmetic constant* is expressed as (ε + -)(digit + (· digit* + ε) + · digit$^+$).

- A *lexical-analyzer generator* takes as input a sequence of r.e.s (describing various tokens) and produces a single DFA recognizing any token.
- How?
  - It performs conversions {given r.e.s} → NFA_ε → DFA (rather then via NFA).
  - Each *final state* of the DFA indicates the *particular token* found during lexical analysis.
  - The δ of the DFA is *encoded* (to take less space than a 2D-array).
  - The resulting lexical analyzer is a *fixed program* that *interprets* (*simulates*) the DFA.
  - This lexical analyzer may then be used as a module in a compiler.

- **Text editors.**
- Certain text editors and similar programs offer commands that may accept r.e.s as parameters.
- Examples.
  - In UNIX text editor, the command s/bbb*/b substitutes a single blank b for the *first* string of two or more blanks found in the current line of text.
  - Generally, given a word w and a r.e. r, the command s/r/w substitutes w for the *first* string that *matches* r in the current line of text. (More precisely, the command substitutes w for the first occurrence of *any* word from $L(r)$ in the current text line.)

- **Data compressors.**
- ….
- Examples.
  - …

2.10 Dictionary

token jezikovni simbol finite automaton končni avtomat regular expression regularni izraz finite state system končni sistem state stanje switching circuit preklopno vezje Turing machine Turingov stroj deterministic finite automaton deterministični končni avtomat state transition prehod stanja input symbol vhodni simbol input alphabet vhodna abeceda initial state začetno stanje final state končno stanje accepting state sprejemajoče stanje to accept sprejeti transition diagram diagram prehodov transition function funkcija prehodov control unit nadzorna enota tape trak move poteza window okno extended transition function razširjena funkcija prehodov regular set regularna množica nondeterministic finite automaton nedeterministični končni avtomat execution tree drevo izvajanja ε-move tihi prehod concatenation stik closure zaprtje Kleene closure Kleenovo zaprtje positive closure pozitivno zaprtje lexical analysis leksikalna analiza lexical analyzer leksikalni analizator language token jezikovni simbol lexical-analyzer generator generator leksikalnih analizatorjev text editor urejevalnik data compressor

3
Properties of Regular Sets

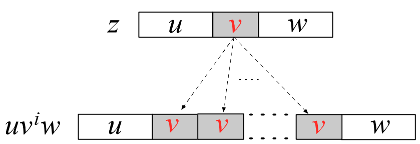
Contents

- **Questions about regular sets.**
- There are many *questions* we can ask about regular sets; for example:
  - Given a language L specified in some way, is L a regular set?
  - Given regular expressions r_1, r_2, are the regular sets $L(r_1)$, $L(r_2)$ equal?
  - Given a FA M, find the minimal equivalent FA (with fewest states).
  - …
- We will provide tools for answering such questions about regular sets. In particular, we will provide:
  - *Pumping lemma* (used to show that certain languages *are not regular*)
  - *Closure properties* (used to show that certain languages *are regular*)
  - *Myhill-Nerode theorem* (used to show that certain languages *are not regular*)
  - An algorithm (used to answer many other questions about r.e.s and FAs)

3.1 The Pumping Lemma for Regular Sets

- The pumping lemma for regular sets is a powerful tool
  - for proving that certain languages are *not* regular
  - for proving that languages of particular FAs are *(in)finite*

- **Pumping Lemma** (for regular sets). Let L be a regular set. Then there is a constant n (depending only on L) such that the following holds:
  if z is any word such that
  $z \in L$ and $|z| \geq n$,
  then there exist words u, v, w such that
  $z = uvw$,
  $|uv| \leq n$,
  $|v| \geq 1$, and
  $\forall i \geq 0$: $uv^iw \in L$.



In addition, n is at most the number of states of the smallest FA accepting L.

- **Informally.** Given any sufficiently long word z accepted by an FA, we can find a subword v near the beginning of z that may be repeated ("pumped") as many times as we like but the resulting word will still be accepted by the FA.

- **Formally.**
  $$L \text{ regular} \Longrightarrow (\exists n)(\forall z)\Big[z\in L \wedge |z|\geq n \Rightarrow (\exists u,v,w)[z=uvw \wedge |uv|\leq n \wedge |v|\geq 1 \wedge (\forall i\geq 0)\, uv^iw\in L]\Big]$$

- **Proof.**
  - Let L be a regular set.
    So there is a DFA $M = (Q,\Sigma,\delta,q_0,F)$ accepting L.
    Let $n := |Q|$.
  - Let $z = a_1\ldots a_m$ ($m \geq n$) be a word in L.
  - *Start M on input z. While reading z, M enters various states.*
    Denote by q_{ℓ_i} the state of M after reading $a_1\ldots a_i$.
    When entire $z = a_1\ldots a_m$ is read, M has entered m+1 states $q_0, q_{\ell_1}, \ldots, q_{\ell_m}$.
  - *Note*: at least two of these states, say q_{ℓ_j}, q_{ℓ_k} ($0 \leq j < k \leq n$), *must be equal* (as $|Q| < m+1$).
    So the path $q_0 \to q_{\ell_1} \to \ldots \to q_{\ell_m}$ has a *loop* $q_{\ell_j} \to \ldots \to q_{\ell_k}$ labeled $a_{j+1}\ldots a_k$.
  - If we take $u := a_1\ldots a_j$, $v := a_{j+1}\ldots a_k$ and $w := a_{k+1}\ldots a_m$, we can prove that
    - $z = uvw$;
    - $|uv| \leq n$;
    - $1 \leq |v|$, and
    - for all $i \geq 0$, $uv^iw \in L$.

□

- **Applications of the pumping lemma.**

- The lemma is useful in *proving* that certain languages are *not* regular. The *method* of proving this is derived from the *formally* written lemma. How?
  - Formally, the pumping lemma is written as
    $$L \text{ regular} \Longrightarrow (\exists n)(\forall z)\Big[\underbrace{z\in L \wedge |z|\geq n}_{P} \Rightarrow (\exists u,v,w)[\underbrace{z=uvw \wedge |uv|\leq n \wedge |v|\geq 1}_{Q} \wedge (\forall i\geq 0)\, uv^iw\in L]\Big]$$
  - Let us *focus* on those z,u,v,w's for which P and Q are *true*; let us *fix* n to the constant whose existence is assured by the lemma. For such 'good' n,z,u,v,w's we can reduce the formula to
    $$L \text{ regular} \Longrightarrow (\forall z)(\exists u,v,w)(\forall i\geq 0)\, uv^iw\in L \qquad (\text{where } n,z,u,v,w \text{ are 'good'})$$
  - Recall from logic: A ⇒ B ≡ ¬B ⇒ ¬A; and $\neg(\forall x)F(x) \equiv (\exists x)\neg F(x)$; and $\neg(\exists x)F(x) \equiv (\forall x)\neg F(x)$.
    If we apply these equivalences to the above formula we obtain
    $$(\exists z)(\forall u,v,w)(\exists i\geq 0)\, uv^iw\notin L \Longrightarrow L \text{ not regular} \quad (\text{where } n,z,u,v,w \text{ are 'good'})$$
  - Notice: *If we prove*, for a given L, that the left-hand side of '⟹' holds, then L is not regular.
    This is the basis of the following method of proving that a language L is not regular.

- **The method.**

- Suppose that we want to prove that a given language L is *not* regular.
  To do this, we try to prove that the following holds for L:
  $$(\exists z)(\forall u,v,w)(\exists i\geq 0)\, uv^iw\notin L \quad (\text{where } n,z,u,v,w \text{ are 'good'})$$
  To prove that we :
  a) Pick an n and declare it to be the constant mentioned in the lemma.
  b) Select a 'good' word z (i.e. such that $z \in L$, $|z| \geq n$)
  c) Find all possible partitions of z into 'good' u,v,w (i.e. such that $z = uvw$, $|uv| \leq n$, $|v| \geq 1$)
  d) Try to prove:
     for every 'good' partition u,v,w
     there exists an $i \geq 0$
     for which $uv^iw \notin L$.

  If d) succeeds, then L is *not* regular.

Example.

- Let $L = \{0^{i^2} \mid i \in \mathbb{N}\}$. We want to prove that L is *not* regular.
- We use the described method.
  - Let n be the constant from the lemma.
  - Select $z = 0^{n^2}$. (z is 'good' because $z \in L$ and $|z| = n^2 \geq n$.)
  - There are many possible partitions of z into 'good' u,v,w (i.e. $z = uvw, |uv| \leq n, |v| \geq 1$).
    Note: for every 'good' partition u,v,w we have $1 \leq |v| \leq n$. (Why?)
  - Let u,v,w be an *arbitrary* 'good' partition of z. We'll show that $uv^2w \notin L$.
    - Compute: $|uv^2w| = |u|+2|v|+|w| = |z|+|v| = n^2+|v|$.
    - But $1 \leq |v| \leq n$.
    - So $n^2+1 \leq |uv^2w| \leq n^2+n$, which is $< (n+1)^2$.
    - Hence $n^2 < |uv^2w| < (n+1)^2$.
      This means that $|uv^2w|$ is *not a perfect square*; consequently $uv^2w \notin L$.
    - We proved that, for *any* 'good' u,v,w, there *exists* an i (=2) such that $uv^iw \notin L$.
  - According to our method, this implies that L is *not* regular.
- There exist *non-regular* languages! For these we will need a model of computation that will be *more powerful than* FA.

Example.

- Let $L = \{0^p \mid p \text{ is a prime}\}$. We want to prove that L is *not* regular.
- We use the described method.
  - Let n be the constant from the lemma.
  - Select $z = 0^p$, where p is the smallest prime $\geq \max\{3,n\}$. (Obviously z is 'good'.)
  - There are many possible partitions of z into 'good' u,v,w (i.e. $z = uvw, |uv| \leq n, |v| \geq 1$).
    For every 'good' partition u,v,w we have $1 \leq |v| \leq n$.
  - Let u,v,w be an *arbitrary* 'good' partition of z. We'll show that $uv^{p+1}w \notin L$.
    - Compute $|uv^{p+1}w| = |u|+(p+1)|v|+|w| = |z|+p|v| = p+p|v| = (p+1)|v|$.
    - This is not a prime (because $p+1$ is *even*).
    - Since $|uv^{p+1}w|$ is *not a prime*, we have $uv^{p+1}w \notin L$!
    - We proved: for *every* 'good' u,v,w, there *exists* an i (=p+1) such that $uv^iw \notin L$.
  - According to our method, this implies that L is *not* regular.
- *There is no FA accepting this L; and L cannot be denoted by a regular expression.*

3.2 Closure Properties of Regular Sets

- Some operations on languages *preserve* regular sets (in the sense that the operations applied to regular sets result in regular sets).
- We say that the class of regular sets is closed under an operation if the operation applied to regular sets is a regular set.
- If the class of regular sets is closed under a particular operation, we call that fact closure property of the class of regular sets.
- We are particularly interested in effective closure properties of the class of regular sets. For such properties, given *descriptors* for regular sets, there is an *algorithm* to construct a *descriptor* for the regular set that results by applying the operation to these regular sets.

- **Closure under union, concatenation, and Kleene closure.**
- **Theorem.** The class of regular sets is closed under union, concatenation and Kleene closure.

Remark. The theorem states that the union $L_1 \cup L_2$ and concatenation L_1L_2 of regular sets L_1, L_2 is a regular set, and the Kleene closure L^* of a regular set L is a regular set.

- **Proof**. The theorem follows directly from the definiton of regular sets.
  - Let L_1 and L_2 be regular sets. Is $L_1 \cup L_2$ a regular set?
    Since L_1,L_2 are regular, there are r.e.'s r_1,r_2 such that $L_1 = L(r_1)$ and $L_2 = L(r_2)$.
    (Recall: r_1,r_2 can be *effectively* constructed from the corresponding FA's M_1,M_2.)
    Now construct r.e. r_1+r_2. But this r.e. denotes $L_1 \cup L_2$. So $L_1 \cup L_2$ is regular.
  - Similarly we prove the effective closure for concatenation and Kleene closure.

□

Closure under complementation and intersection.

Theorem. The class of regular sets is closed under complementation and intersection.

Remark. The theorem states that the complement Σ^*-L of a regular set L is regular, and the intersection $L_1 \cap L_2$ of regular sets L_1, L_2 is a regular set.

Proof.

- (complementation) Let L be a regular set. Is $\Sigma^* - L$ also a regular set?
  Since L is regular, there is a DFA $M = (Q,\Sigma,\delta,q_0,F)$ such that $L = L(M)$. We will construct a new DFA M' for $\Sigma^* - L$. Idea: *M' should have complemented final states*. So, $M' = (Q',\Sigma,\delta',q_0',F')$ where $Q' := Q$, $\Sigma' := \Sigma$, $\delta' := \delta$, $q_0' := q_0$, $F' := Q - F$. Note: *M' accepts x iff M does not accept x.* This means that M' accepts $\Sigma^* - L(M) = \Sigma^* - L$. So $\Sigma^* - L$ is a regular set.
- (intersection) Let L_1,L_2 be regular sets. Is $L_1 \cup L_2$ a regular set too?
  We know that $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, where overbar denotes complementation (with respect to an alphabet that includes the alphabets of L_1, L_2). Now, since the class of r.e. sets is closed under complementation and union, it is also closed under intersection.

□

Closure under substitution and homomorphism.

Definition. Let Σ, Δ be alphabets. A substitution is a function f that maps each symbol of Σ to a language over Δ; i.e. $f(a) \subseteq \Delta^*$ for each $a \in \Sigma$. We extend f to *words* in Σ^* by defining $f(\varepsilon) = \varepsilon$ and $f(wa) = f(w)\,f(a)$; and then to *languages* by defining $f(L) = \bigcup_{x\in L} f(x)$.

- **Question**. The definition of substitution says nothing about the nature of the set L and the sets $f(a)$, $a \in \Sigma$. *What if* we additionally required that L and all $f(a)$, $a \in \Sigma$ be *regular*? Would then $f(L)$ be regular too?
- **Example**. Let $\Sigma=\{0,1\}$, $\Delta=\{a,b\}$ and f a substitution defined by $f(0)=\boldsymbol{a}$, $f(1)=\boldsymbol{b}^*$. *Here both $f(a)$, $f(b)$ are regular.*
  Let $x = 010$. Then $f(x) = f(010) = \ldots = f(0)\,f(1)\,f(0) = \boldsymbol{ab^*a}$.
  Let L be the regular set denoted by $\mathbf{0^*(0+1)1^*}$; then $f(L) = \mathbf{a^*(a+b^*)(b^*)^*}$. This is a regular set too.

Definition. A homomorphism is a substitution h such that $h(a)$ contains a single word for each $a \in \Sigma$. We extend h to words and languages as in the case of the substitution. The inverse homomorphic image of a word w is the set $h^{-1}(w) = \{x\,|\,h(x) = w\}$ and of a language L is the set $h^{-1}(L) = \{x\,|\,h(x) \in L\}$.

- **Example.** Let h be a homomorphism defined by $h(0) = aa$ and $h(1) = aba$.
  Let $x = 010$. Then $h(x) = h(010) = aaabaaa$; and $h^{-1}(aaabaaa) = \{010\}$. (Why? Only 010 maps to $aaabaaa$.)
  Let $L_1 = (\mathbf{01})^*$. Then $h(L_1) = (\boldsymbol{aaaba})^*$. Let $L_2 = (\boldsymbol{ab}+\boldsymbol{ba})^*\boldsymbol{a}$. Then $h^{-1}(L_2) = \{x\,|\,h(x) \in (\boldsymbol{ab}+\boldsymbol{ba})^*\boldsymbol{a}\} = \{1\}$. Why?

Theorem. The class of regular sets is closed under substitution, homomorphism and inverse homomorphism.

Remark. If L and all $f(a)$ are regular, then also $f(L)$ is regular; and if L is regular, $h(L)$ and $h^{-1}(L)$ are regular too.

Proof idea.

- (*substitution*) Let L and all $f(a)$, $a \in \Sigma$ be regular sets. Let L be denoted by r.e. r and $f(a)$ by r_a. *Idea:* replace each occurrence of $\boldsymbol{a}$ in r by r_a. Then prove that the resulting r.e. r' denotes $f(L)$. (Use induction on the number of operators in r'.)
- (*homomorphism*) Closure under homomorphism follows directly from closure under substitution (because every homomorphism is by definition a special substitution).
- (*inverse homomorphism*) Let L be regular and h a homomorphism. We want to prove that $h^{-1}(L)$ is regular. Let M be DFA accepting L. We want to construct a DFA M' such that M' accepts $h^{-1}(L)$ *iff* M accepts L. *Idea:* construct M' so that when M' reads $a \in \Delta$, it simulates M on $h^{-1}(L)$.

□

- Homomorphisms and inverse homomorphisms often simplify proofs.

Closure under quotient.

Definition. The quotient of languages L_1 and L_2 is the set L_1/L_2 defined by $L_1/L_2 = \{x\,|\,\exists y \in L_2 : xy \in L_1\}$.

- *Remark.* So $(L_1/L_2)L_2 = L_1$. (Prove; exercise.)
- **Example**. To do.
- **Question**. The definition tells nothing about the nature of the sets L_1,L_2. *What if* L_1,L_2 were regular? Would L_1/L_2 be regular too? What if L_1 was regular and L_2 arbitrary? Would L_1/L_2 be regular too? Here is the answer.

Theorem. The class of regular sets is closed under quotient with arbitrary sets.

Proof idea. To do or not to do. □

3.3 Decision Algorithms for Regular Sets

- We need algorithms to answer various questions concerning regular sets. The questions we are concerned with include: "Is a given language L *empty* (or *nonempty*)? Is L *finite* (or *infinite*)? Is one FA *equivalent* to another? …"
- The algorithms will answer the questions that ask for the answer YES or NO. Problems that ask for YES/NO answers are called decision problems, and algorithms that solve such problems are decision algorithms.
- We must decide on a representation of regular sets. We'll assume regular sets are represented by FA. (Alternatively, we could assume that regular sets are represented by r.e.'s, as there are mechanical translations from r.e.'s into FA.)

- **Emptiness and finiteness of regular sets.**

  Algorithms to determine whether a regular set is *empty* (or *nonempty*), and *finite* (or *infinite*) may be based on the following theorem.

- **Theorem.** The set $L(M)$ accepted by a FA M with n states is:
  1) *nonempty* *iff* M accepts a word of length ℓ, where $\ell < n$.
  2) *infinite* *iff* M accepts a word of length ℓ, where $n \leq \ell < 2n$.
- **Algorithms** (naïve). The obvious procedure to decide
  - *"Is $L(M)$ nonempty?"* is: *"See if any word of length $\ell < n$ is in $L(M)$."*
  - *"Is $L(M)$ infinite?"* is: *"See if any word of length $n \leq \ell < 2n$ is in $L(M)$."*

  We can generate, for any ℓ, all the words of length ℓ and, for each generated word, check whether M accepts it. So, both procedures *halt* and return a YES or NO.

- **Equivalence of finite automata.**
- **Definition.** Two finite automata M_1 and M_2 are said to be equivalent if they accept the same language, i.e. if $L(M_1) = L(M_2)$.
- **Theorem.** There is an algorithm to determine if two FA are *equivalent*.
  - **Proof.** Let M_1 and M_2 be FA and $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Let $L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$. L_3 is regular (due to closure properties) and accepted by some FA M_3. But we can show that M_3 accepts a word *iff* $L_1 \neq L_2$. (Exercise.) $\square$

3.4 The Myhill-Nerode Theorem and Minimization of FA

- Let L be a *regular set* accepted by a DFA M. There are infinitely many other FA that also accept L. Although equivalent, these FA may greatly differ in the components Q, δ, F.
- Question: *Is there* a minimum state DFA, i.e. a DFA that has, among all DFA accepting L, the *smallest number of states*? If there is, can we algorithmically find (construct) it?
- The answer is YES. But to see this we will need the so called Myhill-Nerode Theorem.

- Before we state the *Myhill-Nerode Theorem* we need some definitions.
- **Definition.** Let $L \subseteq \Sigma^*$ be an *arbitrary* language. Define a relation R_L on Σ^* by

$$xR_Ly \quad \textit{iff} \quad \forall z\in\Sigma^*: xz \in L \Leftrightarrow yz \in L.$$

  - ***Remarks***. Two words $x,y \in \Sigma^*$ are in relation R_L *iff* their *arbitrary* extensions *xz,yz* are *either* both in L *or* both outside L. Now, R_L is an *equivalence relation* (prove). So, R_L divides L into *equivalence classes*. The number of these is the index of R_L. Generally, the index of R_L is *finite* or *infinite*. (Example*:* If each $x \in \Sigma^*$ is in relation R_L with *no other y*, then the index of R_L is infinite.)
- **Definition.** Let $M = (Q,\Sigma,\delta,q_0,F)$ be a DFA. Define a relation R_M on Σ^* by

$$xR_My \quad \textit{iff} \quad \delta(q_0,x) = \delta(q_0,y).$$

  - ***Remarks***. Two words $x,y\in\Sigma^*$ are in relation R_M *iff* they take M from q_0 to the *same* $q \in Q$. R_M is *equivalence relation* (why). It divides Σ^* into *equivalence classes, one for each state q reachable from* q_0. The number of the classes is the index of R_M. The index of R_M is *finite* (because $|Q|$ is finite). Note that $L(M)$ is the union of those equivalence classes which correspond to final states $q \in F$. We can easily prove that $xR_My \Rightarrow \forall z\in\Sigma^*: xzR_Myz$; we say that R_M is right invariant.

- The next theorem tells us how these notions are related when L is a *regular* set.
- **Theorem.** (Myhill-Nerode) The following statements are *equivalent*:
  1) $L \subseteq \Sigma^*$ is a *regular set*;
  2) R_L is of *finite* index;
  3) L is the union of some of the equivalence classes of a *right invariant equivalence relation of finite index.*
  - ***Remarks***. The theorem is useful when, for a given L, we have proved one of 1,2,3. Then, the other two hold and thus tell us even more about L.
    - **Example**. If we have proved (3) for *some* 'right invariant equivalence relation of finite index', then (1) tells us that L is regular.
    - **Example**. If have proved (1) that *some* DFA M accepts L, then (3) tells us that there is a 'right invariant equivalence relation of finite index' such that L is the union of some of its equivalence classes. (Moreover, we know that this relation is R_M).

- A consequence of the Myhill-Nerode Theorem is that there is an essentially unique *minimum state DFA* for every regular set.
- **Theorem.** (minimum state DFA) The minimum state DFA accepting a regular set L is *unique up to an isomorphism* (renaming of the states).
- **Proof idea**.
  - Let L be regular. By *Myhill-Nerode Theorem* there are *finite* number of equivalence classes of R_L. Denote by $[x]$ the eq.class containing $x \in \Sigma^*$. Then $\{[x] \mid x \in \Sigma^*\}$ is the set of all eq.classes of R_L.
  - Construct a DFA $M' = (Q',\Sigma,\delta',q'_0,F')$ as follows:
    - $Q' := \{[x] \mid x \in \Sigma^*\}$; (each state corresponds to an eq.class of R_L)
    - $\delta'([x],a) := [xa]$, for $a \in \Sigma$;
    - $q'_0 := [\varepsilon]$;
    - $F' := \{[x] \mid x \in L\}$.
  - Note: $\delta'(q'_0, w) = \delta'(q'_0, a_1a_2\ldots a_n) = [a_1a_2\ldots a_n] = [w]$. Thus, M' accepts w *iff* $[w] \in F'$. This means that M' accepts L.
  - It follows from the *proof* of Myhill-Nerode Theorem that this M' is the *minimum state DFA* for L.

□

3.5 Dictionary

regular set regularna množica pumping lemma lema o napihovanju closure property zaprtost closed under an operation zaprt za operacijo effective efektiven substitution substitucija homomorphism homomorfizem inverse homomorphic image inverzna homomorfna slika quotient kvocient decision problem odločitveni problem decision algorithm odločitveni algoritem, odločevalnik representation predstavitev minimum state FA najmanjši končni avtomat right invariant relation desno invariantna relacija

4
Context-Free Grammars

Contents

4.1 Introduction

- We will introduce context-free grammars (CFG) and the languages they describe—the context-free languages (CFL).
- The CFLs are of great practical importance, notably in
  - *defining* programming languages,
  - formalizing the notion of *parsing*,
  - simplifying *translation* of programming languages, and in
  - other string-processing applications.
- **Example.** CFGs are useful for describing
  - *arithmetic expressions*, with arbitrary nesting of balanced parentheses,
  - *block structure* in programming languages (e.g. matching of {'s and }'s in Java).

  Neither of these aspects of programming languages can be represented by regular expressions.

- A CFG is a finite set variables, also called nonterminals, each of which *represents* a language. The languages represented by the variables are described *recursively* in terms of each other and *primitive symbols* called terminals. The rules relating the variables are called productions.
- **Example.** Below is a CFG which defines *arithmetic expressions* with *operators* +, ∗ and *operands* which which are represented by the symbol **id** :
  - 1) $\langle\text{expression}\rangle \to \langle\text{expression}\rangle + \langle\text{expression}\rangle$
  - 2) $\langle\text{expression}\rangle \to \langle\text{expression}\rangle * \langle\text{expression}\rangle$
  - 3) $\langle\text{expression}\rangle \to (\langle\text{expression}\rangle)$
  - 4) $\langle\text{expression}\rangle \to \mathbf{id}$
  - There is one *variable*, ⟨expression⟩; the *terminals* are +, ∗, (,) and **id.** There are four productions:
    - (1) and (2) say that an expression can be composed of two expressions connected by + or ∗
    - (3) says that an expression may be another expression surrounded by parentheses
    - (4) says that a single operand is an expression.

- By applying productions repeatedly we can derive more and more complicated expressions. The symbol $\Rightarrow$ denotes the act of deriving, that is, replacing a variable by the right-hand side of a production for that variable.
- **Example (cont'd).** Here is the derivation of the expression (**id** + **id**) $*$ **id** in the example CFG:

$$\begin{aligned}\langle\text{expression}\rangle &\Rightarrow \langle\text{expression}\rangle * \langle\text{expression}\rangle\\ &\Rightarrow (\langle\text{expression}\rangle) * \langle\text{expression}\rangle\\ &\Rightarrow (\langle\text{expression}\rangle) * \mathbf{id}\\ &\Rightarrow (\langle\text{expression}\rangle + \langle\text{expression}\rangle) * \mathbf{id}\\ &\Rightarrow (\langle\text{expression}\rangle + \mathbf{id}) * \mathbf{id}\\ &\Rightarrow (\mathbf{id} + \mathbf{id}) * \mathbf{id}\end{aligned}$$

4.2 Context-Free Grammars

- First we give a formal definition of the *context-free grammar*, CFG.
- **Definition.** A context-free grammar (CFG) is a 4-tuple $G = (V, T, P, S)$ where:
  - V is a finite set of variables,
  - T is a finite set of terminals,
  - P is a finite set of productions, each of which is of the form $A \rightarrow \alpha$, where $A \in V$ and α is a word from the language $(V \cup T)^*$;
  - S is a special variable called the start symbol.

- **Conventions.** To make things more readable, we will use symbols:
  - A,B,C,D,E,S ...for *variables*;
  - a,b,c,d,e,0,1,2,3,4,5,6,7,8,9,**boldstrings** ...for *terminals*;
  - X,Y,Z ... for symbols that may be *either variables or terminals;*
  - u,v,w,x,y,z ...for *strings of terminals*; and
  - α,β,γ ...for *strings of variables and terminals.*
  - If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_k$ are the productions for the variable A, then we may express them by the notation $A \rightarrow |\alpha_1|\alpha_2|\ldots|\alpha_k$, where the vertical bar is read 'or'.
- **Example.** The grammar from the previous example is now $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

- We now want to define the *language generated* by a CFG $G = (V,T,P,S)$.
  - **Definitions.** Let $A \rightarrow \beta$ be a production of P and $\alpha,\gamma \in (V \cup T)^*$ any strings. We say that we apply the production $A \rightarrow \beta$ to the string $\alpha A\gamma$ to obtain the string $\alpha\beta\gamma$ *if* we substitute A by β in $\alpha A\gamma$. (We also say that $\alpha A\gamma$ directly derives $\alpha\beta\gamma$ by $A \rightarrow \beta$.) Two strings are related by relation ${}_G\Rightarrow$ if the first directly derives the second by one application of some production of G. Let $\alpha_1, \alpha_2, \ldots, \alpha_m \in (V \cup T)^*$, $m \geq 1$, and suppose that $\alpha_1 \,{}_G\Rightarrow \alpha_2$, $\alpha_2 \,{}_G\Rightarrow \alpha_3, \ldots, \alpha_{m-1} \,{}_G\Rightarrow \alpha_m$. Then α_1 derives α_m in G; that is, $\alpha_1 \,{}_G\Rightarrow^* \alpha_m$.

    The relation ${}_G\Rightarrow^*$ is the *reflexive and transitive closure* of ${}_G\Rightarrow$.
- **Definition.** The language generated by a CFG $G = (V,T,P,S)$ is the set

$$L(G) = \{w \mid w \in T^* \text{ and } S \,{}_G\Rightarrow^* w\}.$$

  - **Definitions.** A language L is called context-free (CFL) if it is $L(G)$ for some CFG G. A string $\alpha \in (V \cup T)^*$ is called a sentential form if $S \,{}_G\Rightarrow^* \alpha$. Two grammars G_1 and G_2 are said to be equivalent if $L(G_1) = L(G_2)$.

- **Example.** Consider a CFG $G = (V, T, P, S)$, where
  - $V = \{S\}$,
  - $T = \{a, b\}$,
  - $P = \{S \to aSb,\ S \to ab\}$.

So, S is the only variable and a, b are terminals. There are two productions, $S \to aSb$ and $S \to ab$.

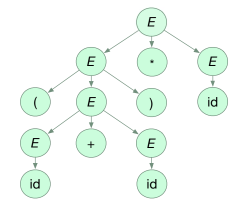
What is $L(G)$, the language generated by this G?

- By applying the production $S \to aSb$ n-1 times, and then the production $S \to ab$, we have $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow a^3Sb^3 \Rightarrow \ldots \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^nb^n$.
  We have proved that S derives in G any a^nb^n, $n \geq 1$; that is, $S_G \Rightarrow^* a^nb^n$, for $n \geq 1$.
- *But, can S derive anything else?* No. We can show that the *only* strings in $L(G)$ are a^nb^n, $n \geq 1$. How? Each time $S \to aSb$ is applied, the number of S's remains the same. After applying $S \to ab$, the number of S's in the sentential form decreases by one. So after applying $S \to ab$, no S's remain. Since both productions have an S on the left, the only order in which the productions can be applied is $S \to aSb$ some number of times followed by one application of $S \to ab$. Thus, $L(G) = \{a^nb^n \mid n \geq 1\}$.

4.3 Derivation Trees

- Derivations can be displayed as derivation (or parse) trees. These are used in applications such as the compilation of programming languages. The *vertices* of such a tree are labeled with *terminal* or *variable* symbols of the grammar or possibly with ε. If an interior vertex is labeled A, and its *sons* are labeled $X_1, X_2, \ldots, X_k$ from the left, then $A \to X_1X_2\ldots X_k$ must be a *production*.
- **Example** (cont'd). The derivation $E \Rightarrow^* (\mathbf{id} + \mathbf{id}) * \mathbf{id}$ is displayed by the following tree:

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow (E) * E \\
&\Rightarrow (E) * \mathbf{id} \\
&\Rightarrow (E + E) * \mathbf{id} \\
&\Rightarrow (E + \mathbf{id}) * \mathbf{id} \\
&\Rightarrow (\mathbf{id} + \mathbf{id}) * \mathbf{id}
\end{aligned}
$$

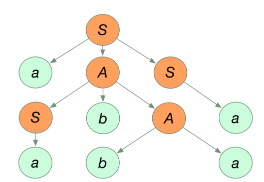


- We now define the notion of a *derivation tree* formally.
- **Definition.** Let $G = (V, T, P, S)$ be a CFG. A tree is called a derivation (or parse) tree for G if:
  1) Every vertex has a label, which is a symbol of $V \cup T \cup \{\varepsilon\}$.
  2) The label of the root is S.
  3) If a vertex is interior and has label A, then A must be in V.
  4) If n has label A and vertices $n_1, n_2, \ldots, n_k$ are the sons of vertex n, in order from the left, with labels $X_1, X_2, \ldots, X_k$, respectively, then $A \to X_1X_2\ldots X_k$ must be a production in P.
  5) If vertex n has label ε, then n is a leaf and is the only son of its father.

- **Example.** Consider the grammar $G = (\{S,A\}, \{a,b\}, P, S)$, where P consists of

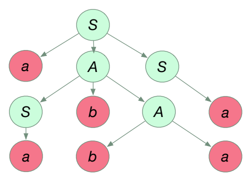
$$
\begin{aligned}
S &\to aAS \mid a \\
A &\to SbA \mid SS \mid ba
\end{aligned}
$$

Is the following tree a derivation tree for G?



To answer this, we check whether the tree meets all the conditions of the previous definition. The *interior* vertices are *orange*. The root is labeled S; its sons, from the left, are labeled a,A,S; and we see $S \to aAS$ is a production of G. Similarly we check every internal vertex whether it and its sons correspond to a production in G. Hence, all the conditions are met; the tree is a derivation tree G.

- A derivation tree is a natural description of the derivation of a *particular sentential form* of the grammar G. Why?
  - **Definition.** If we read the labels of the *leaves* in the *left-to-right ordering* (= preorder search through the tree that retains only leaves), we obtain a string which is called the yield *of the derivation tree*.
  - Soon we will prove the following:
    α is the *yield of some derivation tree for* $G = (V, T, P, S)$ *iff* $S \,_G\!\Rightarrow^* \alpha$.
- **Example(cont'd).** The yield of the derivation tree below is *aabbaa* (red)

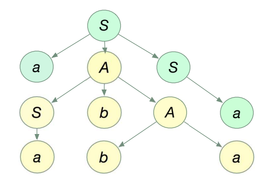


.

In this example, the yield consists of terminals only *but it is not always so.*

- We will need one more new notion.
  - **Definition.** A subtree of a derivation tree is a particular vertex of the tree together with all its descendants, the edges connecting them, and their labels. If variable A labels the root, then we call the subtree an A-tree.

    A subtree is like a derivation tree, but the label of the root of the subtree may not be the start symbol of the grammar.
- **Example(cont'd).** Below is a derivation tree and one of its A-trees (yellow). The yield of this A-tree is *abba.*

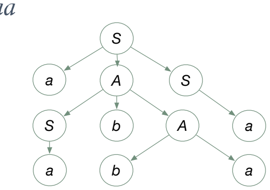


- **The relationship between derivation trees and derivations.**
- **Theorem.** Let $G = (V, T, P, S)$ be a CFG.
  Then $S \,_G\!\Rightarrow^* \alpha$ *iff* there is a derivation tree for G with yield α.
- **Proof idea.** Induction on the number of interior vertices of the tree. □

- **Leftmost and rightmost derivations.**
- **Definition**. A derivation is said to be leftmost if at *each step* in the derivation a production is applied to the *leftmost variable*. A derivation is rightmost if the *rightmost variable* is replaced at *each step*.
- **Example.** The leftmost derivation corresponding to the tree below is
  $S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$

  and the rightmost derivation is

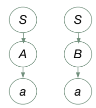
  $S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$



- **Ambiguity.**
- If $w \in L(G)$ for a CFG G, then w has *at least one* derivation tree. Corresponding to a *particular* derivation tree, w has a *unique leftmost* and a *unique rightmost* derivation.
- **Definition.** A CFG G is said to be ambiguous if some word has more than one derivation tree.

  An *equivalent definition* is: A CFG is ambiguous if some word has more than one leftmost derivation (or more than one rightmost derivation).

  - **Example**. $G = (\{S,A,B\},\{a\},\{S \to A \mid B,\ A \to a,\ B \to a\})$. Note, that a has two derivation trees corresponding to two derivations: $S \underset{G}{\Rightarrow} A \underset{G}{\Rightarrow} a$ and $S \underset{G}{\Rightarrow} B \underset{G}{\Rightarrow} a$.



- **Definition.** A CFL L is said to be inherently ambiguous if *every* CFG for L is ambiguous.

  We will see later that such CFL's *do exist*.

4.4 Simplification of Context-Free Grammars

- There are several ways to *restrict the* format of productions *without reducing the power* of CFG's. If L is a nonempty CFL then it can be generated by a CFG G with the following properties:
  - Each variable and terminal of G appears in the derivation of some word in L.
  - There are no productions of the form $A \to B$ where A and B are variables.
  - If ε is not in L, there need be no productions of the form $A \to \varepsilon$.
  - If ε is not in L, we can require that
    - every production of G be of one of the forms $A \to BC$ and $A \to a$, where A, B, C are arbitrary variables and a is an arbitrary terminal;
    - or, alternatively, every production of G be of the form $A \to a\alpha$, where α is a string of variables (perhaps empty).

  These two special forms are called *Chomsky normal form* and *Greibach normal form*, respectively.

- **Elimination of useless symbols.**
- Of course, we want to eliminate all *useless* symbols from a grammar.
- **Definition.** Let $G = (V, T, P, S)$ be a grammar. A symbol X is useful if there exists a derivation $S \Rightarrow^* \alpha X\beta \Rightarrow^* w$ for some α,β, and $w \in T^*$. Otherwise X is useless.

  - **Lemma.** Given a CFG $G = (V, T, P, S)$ with $L(G) \neq \emptyset$, we can *effectively* find an *equivalent* CFG $G' = (V', T, P', S)$ such that for each $A \in V'$ there is a $w \in T^*$ so that $A \Rightarrow^* w$.
  - **Lemma.** Given a CFG $G = (V, T, P, S)$, we can *effectively* find an *equivalent* CFG $G' = (V', T, P', S)$ such that for each $X \in V' \cup T'$ there are $\alpha,\beta \in (V' \cup T')^*$ so that $A \Rightarrow^* \alpha X\beta$.

- By applying the lemmas in *this order*, we can convert a CFG to an equivalent one with no useless symbols. (Interestingly, applying them in *the reverse order may fail* to eliminate all useless symbols.)
- **Theorem.** Every nonempty CFL is generated by a CFG with no useless symbols.
- From here on we assume that no grammar has useless symbols.

- **Proof idea** (1st lemma).

  … *To do.*

  □

- **Proof idea** (2nd lemma).

  … *To do.*

  □

- **Elimination of ε-productions.**
- **Definition.** An ε-production is a production of the form $A \to \varepsilon$.
- Clearly, if ε is in $L(G)$, we cannot eliminate *all* ε-productions from G. (Otherwise, ε would no longer be in the generated language.) But if ε is *not* in $L(G)$, we *can* eliminate all ε-productions from G.
- **Theorem.** If $L = L(G)$ for some CFG $G = (V, T, P, S)$, then $L - \{\varepsilon\}$ is $L(G')$ for some CFG G' with no useless symbols or ε-productions.
- **Proof idea.**
  - Determine for each $A \in V$ whether $A \Rightarrow^* \varepsilon$. If so, call A nullable.
  - Then replace each production $B \to X_1X_2\ldots X_n$ by all productions formed by striking out some subset of those X_i's that are nullable, but do not include $B \to \varepsilon$, even if all X_i's are nullable.

  □

- **Elimination of unit productions.**
- **Definition.** A unit production is a production of the form $A \to B$.

  The right-hand side must be a *single variable*; all other productions, including $A \to a$ and ε-productions, are non-unit.
- **Theorem.** Every CFL without ε is defined by a grammar with no useless symbols, ε-productions, or unit productions.
- **Proof idea.**
  - *To do.*

  □

4.5 Chomsky Normal Form

- Normal-form theorems state that all CFG's are equivalent to grammars with *certain restrictions* on the *forms of productions*. The first such theorem is due to *Noam Chomsky*.
- **Theorem (Chomsky normal form).** Every CFL without ε can be generated by a grammar in which every production is of the form

$$A \to BC \quad \text{or}$$
$$A \to a$$

where A, B, C are variables and a is a terminal.

- **Proof** (constructive).
  - Let $L(G)$ be a CFL without ε.
  - Find an equivalent CFG $G_1=(V,T,P,S)$ without useless variables, unit productions and ε-productions.
  - *If a production of P has a single symbol on the right*, that symbol must be a terminal, and so the production is already in an acceptable form.
  - *If a production of P does not have a single symbol on the right*, it must be of the form $A \to X_1X_2\ldots X_m$, where $m \geq 2$. (Here X_i may be a variable or a terminal.)
    - *If X_i is a terminal*, say a, *then*
      - introduce a new variable C_a
      - introduce a new production $X_i \to a$ (which is in allowable form), and
      - replace X_i with C_a.

      When this is done for all X_i that are terminals, we have a new set V' of variables and a new set P' of productions. Let $G_2 = (V',T,P',S)$. We can show that $L(G_1) = L(G_2)$. (Exercise.)
  - So, $L(G)$ is generated by a CFG G_2 whose every production is either of the form $A \to a$ or $A \to B_1B_2\ldots B_m$, $m \geq 2$. (Here Bi are variables and a is a tereminal.)
    - *If a production is $A \to B_1B_2\ldots B_m$, where $m \geq 3$, then*
      - create new variables $D_1, D_2, \ldots, D_{m-2}$
      - replace the production by the productions

$$\begin{aligned} A &\to B_1D_1, \\ D_1 &\to B_2D_2, \\ &\ddots \\ D_{m-3} &\to B_{m-2}D_{m-2}, \\ D_{m-2} &\to B_{m-1}B_m. \end{aligned}$$

      When done for all productions $A \to B_1B_2\ldots B_m$, $m \geq 3$, we have a set V'' and a set P'' of productions of the form $A \to a$ or $A \to BC$. Let $G_3 = (V'',T,P'',S)$. We can show that $L(G_2) = L(G_3)$; so $L(G) = L(G_3)$.(Exercise.)

  □

- **Example.**

  ... To do.

  □

4.6 Greibach Normal Form

- There is another normal-form theorem that uses productions whose right-hand sides each *start with a terminal symbol* perhaps *followed by some variables*. The theorem is due to *Sheila Greibach*.
- **Theorem** (Greibach normal form)**.** Every CFL without ε can be generated by a grammar in which every productions is of the form

$$A \rightarrow a\alpha$$

  where A is a variable, a is a terminal, and α is a (possibly empty) string of variables.

- **Proof idea** (constructive).
  - Let $L(G)$ be a CFL without ε where $G = (V,T,P,S)$ is in Chomsky normal form and $V = \{A_1, A_2, \ldots, A_m\}$.
  - Construct an equivalent CFG G_1=(V,T,P,S) without useless variables, unit productions and ε-productions.
  - Modify the productions so that the following will be fulfilled: *if $A_i \rightarrow A_j\gamma$ is a production, then $j > i$.*
    This modification introduces new variables $B_1, B_2, \ldots, B_m$ and returns only productions of the forms

$$A_i \rightarrow A_j\gamma, \text{ where } j > i$$
$$A_i \rightarrow a\gamma, \text{ where } a \in T$$
$$A_i \rightarrow a\gamma, \text{ where } \gamma \in (V \cup \{B_1, B_2, \ldots, B_{i-1}\})^*.$$

  - Modify all A_m-productions, then all A_{m-1}-productions, then all A_{m-2}-productions, and so on.
    The A_k-productions, $m \geq k \geq 1$, modify as follows:
      For each A_k-production do the following:
        locate in the right-hand side of the production the *leftmost variable,* say X;
        replace X by the right-hand sides of all X-productions.
    Now all A-productions have right sides beginning with terminal.
    But B-productions may still have right-hand sides beginning with *variables* A_i. This corrects the next step.
  - Modify the productions for the new variables $B_1, B_2, \ldots, B_m$.
    For each B-production whose right-hand size begins with a *variable*, say A_i , do the following:
      replace A_i by the right-hand sides of all A_i-productions.

  □

4.7 Inherently Ambiguous Context-Free Languages

- It is easy to exhibit ambiguous CFG's. For example, the CFG with productions $S \rightarrow A \mid B$, $A \rightarrow a$, $B \rightarrow a$ is ambiguous. Why?
- Not so easy to do is to find a CFL for which *every* CFG is *ambiguous*. Such a CFL is said to be inherently ambiguous.
  But, do such CFL's exist? Yes.
- **Theorem.** The CFL $L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$ is inherently ambiguous.
- **Proof**. By contradiction. (Long and tedious. We omit it.) □

4.8 Dictionary

context-free grammar kontekstno neodvisna gramatika context-free language kontekstno neodvisen jezik terminal terminal production produkcija to derive izpeljati start symbol začetni simbol to apply (a production) uporabiti (produkcijo) to directly derive neposredno izpeljati language generated generiran (izpeljan) jezik sentential form stavčna oblika derivation tree drevo izpeljave yield (of a derivation tree) krošnja (drevesa izpeljave) subtree poddrevo leftmost/rightmost derivation leva/desna izpeljava ambiguous dvoumen inherently ambiguous bistveno dvoumen format of a production oblika produkcije useful/useless symbol potreben/nepotreben simbol ε-production ε-produkcija nullable variable uničljiva spremenljivka unit production enotska produkcija Chomsky normal form normalna oblika Chomskega Greibach normal form normalna oblika Greibachove

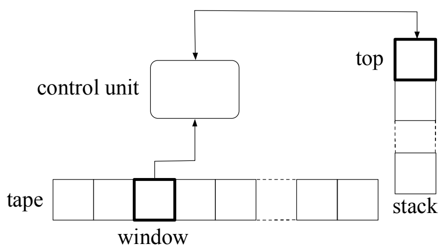
5 Pushdown Automata

Contents

5.1 Introduction

- Just as the r.e.'s have an equivalent automaton—the FA, the CFG's have their machine counterpart—the pushdown automaton (PDA).
- The PDA is essentially an FA with control of an input tape and a stack.
- But, the PDA is a *nondeterministic* device, and the *deterministic version,* DPDA, accepts only a *subset* of all CFL's.
- Happily, this subset includes the syntax of *most programming languages*.

- The PDA will have an input tape, a control unit, and a stack.
- The *stack* will be a string of symbols from some alphabet. The *leftmost* symbol of the string will be considered to be at the *top* of the stack.



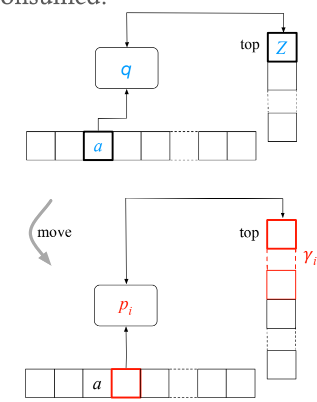
- The device will be by definition nondeterministic, having some finite number of choices of moves in each situation.

- The moves will be of two types, *regular moves* and *ε-moves*.
  - In the regular move, an input symbol *will* be consumed.

    Depending on the
    - *state* q of the finite control,
    - *input symbol* a, and
    - *top symbol* Z on the stack,

    a finite number of choices will be possible.
    The i-th choice will consist of a
    - *next state* p_i for the finite control,
    - (possibly empty) *string* γ_i of symbols *to replace the top* stack symbol.

    After selecting a choice,
    - the window will *advance* one symbol.

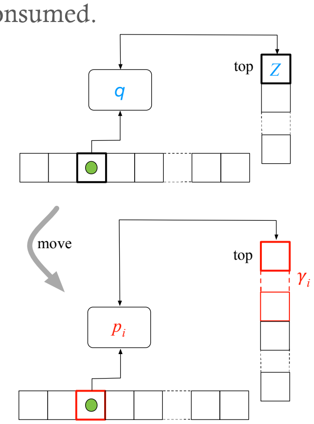


- (cont'd)
  - In the ε-move, an input symbol *will not* be consumed.

    Depending on the
    - *state* q of the finite control,
    - *independent of the input symbol*, and
    - *top symbol* Z on the stack,

    a finite number of choices will be possible.
    The i-th choice will consist of a
    - *next state* p_i for the finite control,
    - (possibly empty) *string* γ_i of symbols *to replace the top* stack symbol.

    After selecting a choice,
    - the window will *not advance*.



  - This type of move will allow PDA to manipulate the stack without reading input symbols.

- We will also define the *language accepted by a* PDA. The language can be defined in two ways, namely as the set of all inputs for which
  1. *some* sequence of moves causes the PDA to *empty its stack*.
     This is the language accepted by empty stack; or
  2. *some* sequence od moves causes the PDA to *enter a final state*.
     This is the language accepted by final state.
- We'll see that the definitions are *equivalent* in the sense that
  a set is accepted by empty stack by some PDA iff it is accepted by final state by some other PDA.
- The 2nd definition is more common. But by using the 1st definition it will be easier to prove the basic theorem of PDA. This will state that *a language is accepted by a PDA iff it is a CFL.*

5.2 Definitions

- **Definition**. A pushdown automaton (PDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:
  - Q is a finite set of states,
  - Σ is the input alphabet,
  - Γ is the stack alphabet,
  - $q_0 \in Q$ is the initial state,
  - $Z_0 \in \Gamma$ is the start symbol,
  - $F \subseteq Q$ is the set of final states, and
  - δ is the transition function, i.e. a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.
- *Note*: δ can be viewed as a program of PDA. Every PDA has its own specific δ.

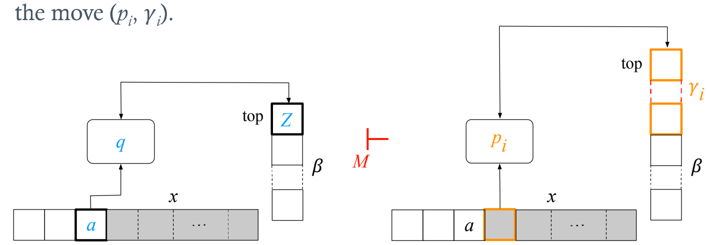
- **Moves of the PDA.**
- The interpretation of the move
  - $\delta(q,a,Z) = \{(p_1,\gamma_1), (p_2,\gamma_2), \dots, (p_m,\gamma_m)\}$ is that the PDA in state q, with *input symbol* a and Z the top symbol on the stack can, for any i, $1 \le i \le m$, enter state p_i, replace symbol Z by string γ_i, and advance the window one symbol. We call this the regular move.
  - $\delta(q,\varepsilon,Z) = \{(p_1,\gamma_1), (p_2,\gamma_2), \dots, (p_m,\gamma_m)\}$ is that the PDA in state q, *independent of the input symbol being scanned* and with Z the top symbol on the stack, can enter state p_i, and replace Z by γ_i, for any i, $1 \le i \le m$. In this case, the window is not advanced. We call this the ε-move.

Conventions: the *leftmost* symbol of γ_i will be placed *highest* on the stack and the *rightmost* symbol of γ_i *lowest* on the stack. We will use $a,b,c,\dots$ for input symbols, $u,v,w,\dots$ for strings of input symbols, *capital letters* for stack symbols, and *Greek letters* for strings of stack symbols.

- **Instantaneous descriptions of the PDA.**
- We want to describe the *configuration* of a PDA at a given *instant*. These "snapshots" of PDA's execution are formalized by instantaneous descriptions.
- **Definitions.** An instantaneous description (ID) is a tripple (q,w,γ), where q is a state, w a string of input symbols, and γ a string of stack symbols.
  - If $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, we say that ID $(q, ax, Z\beta)$ can directly become ID $(p_i, x, \gamma_i\beta)$, --- written $(q, ax, Z\beta) \vdash_M (p_i, x, \gamma_i\beta)$, --- *if* $\delta(q, a, Z)$ contains (p_i, γ_i). Here, a may be an input symbol or ε.
  - We write $\vdash_M^*$ for the *reflexive and transitive closure* of $\vdash_M$ and say that an ID I can become ID J *if* $I \vdash_M^* J$. We write $I \vdash_M^k J$ *if* $I \vdash_M^* J$ in exactly k moves.

The subscript M can be dropped whenever the particular PDA M is understood.

- (cont'd)
  - Informally, the situation on the left *can* directly change to the situation on the right *only if* the PDA M contains the instruction $\delta(q, a, Z) = \{\dots, (p_i, \gamma_i), \dots\}$. Whether or not the change will actually take place depends on whether or not PDA will choose the move (p_i, γ_i).



ID $(q, ax, Z\beta)$ directly becomes ID $(p_i, x, \gamma_i\beta)$

if $\delta(q,a,Z)$ contains (p_i, γ_i)

Accepted languages of the PDA.

- **Definitions.** For PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ we define two languages:
  - $L(M)$, the language accepted by final state, to be
    $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \varepsilon, \gamma)$ for some $p \in F$ and $\gamma \in \Gamma^*\}$
  - $N(M)$, the language accepted by empty stack, to be
    $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon)$ for some $p \in Q\}$.

$L(M)$ contains a word w if after reading w, M *can* be (nondeterminism!) in some final state.
$L(M)$ contains a word w if after reading w, M *can* have (nondeterminism!) its stack empty.

If acceptance is by empty stack, final states are irrelevant; in this case, we usually let $F = \emptyset$.

Example. Here is a PDA M accepting $\{wcw^R \mid w \in (\mathbf{0}+\mathbf{1})^*\}$ by empty stack.

- **Idea**. Read input and, for each symbol read, *push* its representative (B for 0, G for 1) on the stack. When c is read, *change* the state. Continue reading the input and, for each symbol read, *pop* the stack symbol. If there are no more input symbols and R (bottom of the stack) has just been popped, the input must have been of the form wcw^R. So the stack is emptied to signal the acceptance of the input.

- $M = (\{q_1, q_2\}, \{0,1,c\}, \{R,B,G\}, \delta, q_1, R, \emptyset)$, where δ is defined as follows:

1. $\delta(q_1, 0, R) = \{(q_1, BR)\}$ 2. $\delta(q_1, 1, R) = \{(q_1, GR)\}$
3. $\delta(q_1, 0, B) = \{(q_1, BB)\}$ 4. $\delta(q_1, 1, B) = \{(q_1, GB)\}$
5. $\delta(q_1, 0, G) = \{(q_1, BG)\}$ 6. $\delta(q_1, 1, G) = \{(q_1, GG)\}$
7. $\delta(q_1, c, R) = \{(q_2, R)\}$
8. $\delta(q_1, c, B) = \{(q_2, B)\}$
9. $\delta(q_1, c, G) = \{(q_2, G)\}$
10. $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$ 11. $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
12. $\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$

Note. Although PDA's are nondeterministic by definition, the above M has just *one choice* of move in each situation.

Example. Here is a PDA M' accepting $\{ww^R \mid w \in (\mathbf{0}+\mathbf{1})^*\}$ by empty stack.

- **Note.** Now there is *no symbol c indicating the middle* of the input word (as in previous example). So, the M' will have to *guess* that the middle of the word has been reached. How? Recall that PDA is by definition *non*-deterministic, always choosing the right move when there is one. We'll have to add the possibility of *choosing* to the program of M'.

- **Idea**. Read input and, for each symbol read, *push* its representative (B for 0, G for 1) on the stack. Whenever the input symbol "equals" the top stack symbol, the middle of the input word *may* have been reached. *Non*-deterministically decide if this is so and, in this case, *change* the state (otherwise push the representative of the input symbol on the stack). After the middle of the word has been guessed, continue reading the input and, for each symbol read, *pop* the stack symbol *if* it *represents* the input symbol (if it *doesn't*, the input word is *not* of the form ww^R, so halt as there is no instruction for this situation). If there are no more input symbols and R (bottom of the stack) has just been popped, the input must have been of the form ww^R. So empty the stack to signal the acceptance of the word.
If M' never detected the middle of the input word, the word must have been ε or a single symbol, so accept the word.

- $M' = (\{q_1, q_2\}, \{0,1\}, \{R,B,G\}, \delta, q_1, R, \emptyset)$, where δ is defined as follows:

1. $\delta(q_1, 0, R) = \{(q_1, BR)\}$ 2. $\delta(q_1, 0, G) = \{(q_1, BG)\}$
3. $\delta(q_1, 1, R) = \{(q_1, GR)\}$ 4. $\delta(q_1, 1, B) = \{(q_1, GB)\}$
5. $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$ 6. $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$
7. $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$ 8. $\delta(q_2, 1, G) = \{(q_2, \varepsilon)$
9. $\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$ 10. $\delta(q_1, \varepsilon, R) = \{(q_2, \varepsilon)\}$

- Informally, the example PDA that accepted $\{wcw^R \mid w \in (\mathbf{0}+\mathbf{1})^*\}$ was "deterministic" because at most one move was possible from any ID. But, the *formal* definition of the *deterministic* PDA is more precise.

- **Definition**. A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is called deterministic if δ fulfills two additional conditions for every $q \in Q$ and $Z \in \Gamma$:
  1. $\delta(q, \varepsilon, Z) \neq \emptyset \implies \forall a \in \Sigma : \delta(q, a, Z) = \emptyset$
  2. $\forall a \in \Sigma \cup \{\varepsilon\} : |\delta(q, a, Z)| \leq 1$

  Now what does that mean? Condition 1 prevents the possibility of a *choice between* an *ε-move* and a *regular* move. Condition 2 prevents the possibility of a *choice* in the case of an *ε-move* and the possibility of a *choice* in the case of a *regular* move.

- **Note**. Unlike FA, a PDA is assumed to be nondeterministic unless we state otherwise. In this case, we denote it by DPDA (for deterministic PDA).

5.3 Pushdown Automata and Context-Free Languages

- We saw that the *deterministic* FA's accept the same class of languages as the *nondeterministic* FA's (i.e. regular sets).
- **Question**: Do *deterministic* PDA's accept the same class of languages as the *nondeterministic* PDA's?
  - But PDA's can accept in two different ways, by *empty stack* and *final state*, so there are two kinds of accepted languages, $L(M)$'s and $N(M)$'s.
  - **Question**: Which of the two are meant in the above question?
  - **Answer**: It doesn't matter; we'll see that the class of all $L(M)$'s and the class of all $N(M)$'s are the *same*.
  - **Question**: Does this class contain any languages that we already know?
  - **Answer**: Yes; we'll see that this class and the class of all CFL's are the *same*.
- **Answer**: No; ww^R is accepted by a nondeterministic PDA, but by no DPDA.

Equivalence of acceptance by final state and empty stack

- Do acceptance by *final state* and acceptance by *empty stack* differ in their power? We suspect the answer is *no*. To prove that, must prove that the *class* of languages accepted by PDA's by *final state* is the *same* as the *class* of languages accepted by PDA's by *empty stack*. Hence, we must show that *if* a language L is accepted by some PDA by *final state*, *then* L is accepted by some PDA by *empty stack---and vice versa*. We can prove both.
- **Theorem.** If $L=L(M_2)$ for some PDA M_2, then $L=N(M_1)$ for some PDA M_1.
  - **Proof idea.** Given an arbitrary $L = L(M_2)$, construct a PDA M_1 that simulates M_2 but erases the stack whenever M_2 enters a final state. So we have $L = N(M_1)$ too. □
- **Theorem.** If $L=N(M_1)$ for some PDA M_1, then $L=L(M_2)$ for some PDA M_2.
  - **Proof idea.** Given an arbitrary $L = N(M_1)$, construct a PDA M_2 that simulates M_1 but enters a final state whenever M_1 erases the stack. So we also have $L = L(M_2)$. □
- **Summary:** The class of languages accepted by PDA's by final state is the same as the class of languages accepted by PDA's by empty stack.

Equivalence of PDA's and CFL's

- Is there any link between the languages accepted by PDA's and the regular or context-free languages? We suspect that PDA's can accept *more* than just regular sets. (Why?) So, can PDA's accept CFL's? To prove that, we must show that *if* L is a CFL, *then* L is accepted by some PDA. If so, can PDA's accept *more than* CFL's? To prove that they *can't*, we must show that *if* L is accepted by a PDA, *then* L is CFL. Both can be proved.
- **Theorem.** If L is a CFL, then there exists a PDA M such that $L=N(M)$.
  - **Proof idea.** Let L be an arbitrary CFL. L can be generated by a CFG G in Greibach normal form. Construct a PDA M that simulates leftmost derivations of G. (It is easier to have M accept by empty stack.) So $L = N(M)$.□
- **Theorem.** If $L=N(M)$ for some PDA M, then L is a CFL.
  - **Proof idea.** Let M be an arbitrary PDA. Construct a CFG G in such a way that a leftmost derivation in G of a sentence x is a simulation of the PDA M when given the input x. So $L = L(G)$, a CFG. □
- **Summary:** The class of languages accepted by PDA's is exactly the class of CFL's.

Deterministic vs. nondeterministic PDA's

- We now know that (nondeterministic) PDA's accept exactly CFL's. What about *deterministic* PDA's? These are obtained by restricting PDA's, so it is natural to ask whether they are powerful enough to accept all CFL's?

  Question: Is the class of languages accepted by DPDA's the same as the class of CFL's?

  Answer: No; there exist CFL's that are not accepted by any deterministic PDA.
- **Theorem.** $\{ww^R \mid w \in (\mathbf{0}+\mathbf{1})^*\}$ is accepted by a PDA but not by any DPDA.
  - **Proof idea.** Omitted. □
- **Summary:** Deterministic PDA's are *less powerful* than nondeterministic PDA's.

5.4 Dictionary

pushdown automaton skladovni avtomat stack sklad regular move običajen prehod, običajna poteza ε-move tihi prehod, tiha poteza language accepted by empty stack (final state) jezik sprejet s praznim skladom (končnim stanje) basic theorem of PDA osnovni izrek skladovnih avtomatov stack alphabet skladovna abeceda instantaneous description trenutni opis directly becomes neposredno preide v becomes preide v

6 Properties of Context-Free Languages

Contents

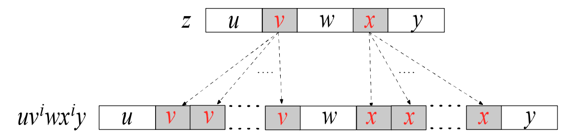
6.1 Introduction

- This chapter parallels Chapter 3 (Properties of Regular Sets). In this chapter, we shall:
  - give a pumping lemma for CFL’s. We will use the lemma to show that certain languages are *not* context-free.
  - consider some operations that preserve CFL’s. These closure properties are useful not only for constructing and proving that certain languages *are* context-free, but also in proving languages *not to be* context-free.
  - give decision algorithms to answer certain questions about CFL's. These questions include whether a given CFL is *empty*, *finite*, or *infinite* and whether a given word is a *member* of a given CFL. We will also see, that certain questions about CFL's *no algorithm* can answer.

6.2 The Pumping Lemma for CFL's

- **Recall:** The pumping lemma for *regular sets* states that every *sufficiently long* string in a regular set contains a *short substring* that can be pumped. (That is, inserting as many copies of the substring as we like always yields a string which is in the same regular set.)
- The pumping lemma for CFL's states that there are always *two short substrings close together* that can be repeated, *both* the *same number of times*, as often as we like, and the obtained string is in the same CFL.
- The formal statement of the pumping lemma is as follows.

- **Pumping Lemma** (for CFL's). Let L be a CFL. Then there is a constant n (depending only on L) such that the following holds: if z is any word such that
  $z \in L$ and $|z| \geq n$,
  then there are words u, v, w, x, y such that
  $z = uvwxy$,
  $|vx| \geq 1$,
  $|vwx| \leq n$, and
  $\forall i \geq 0$: $uv^iwx^iy \in L$.



- **Informally.** Given any sufficiently long word z in a CFL L, we can find two short subwords v and x close together that may be repeated ("pumped"), both the same number of times as we like, but the resulting word will still be in L.
- **Proof**. Omitted. □

- **Example.**
  - Let $L = \{a^ib^ic^i \mid i \geq 1\}$. We want to prove that L is *not* context-free.
  - The method is similar to that for regular sets.
    - Let n be the constant from the lemma.
    - Select $z = a^nb^nc^n$. (z is 'good' because $z \in L$ and $|z| = 3n \geq n$.)
    - There are many 'good' partitions of z into u,v,w,x,y ($z = uvwxy, |vwx| \leq n, |vx| \geq 1$).
    - Where can v and x lie in $a^nb^nc^n$?
      - Since $|vwx| \leq n$, it is not possible for vx to contain both a's and c's.
      - So vx can consist of
        1) a's only *or* a's and b's; or
        2) b's only *or* b's and c's; or
        3) c's only.
      - We analyze the above alternatives:
        - If v and x consist of a's only, then $uv^0wx^0y = uwy$ contains n b's and n c's but fewer than n a's (because $|vx| \geq 1$). Thus uv^0wx^0y is *not* in L.
        - If v and x consist of a's and b's, then $uv^0wx^0y = uwy$ has more c's than a's or b's, so it is *not* in L.
        - *The other three alternatives are analyzed similarly. Each leads to the conclusion that uv^0wx^0y is not in L.*
    - According to our method, this implies that L is *not* context-free.
  - There exist languages that are *not* context-free. For these we will need a model of computation that will be more powerful than PDA.

- ***Ogden's Lemma.**
- There are certain non-CFL's for which the pumping lemma is of *no help*. (E.g. $L = \{a^ib^jc^id^j \mid i,j \geq 1\}$.) We need is a stronger version of the pumping lemma for CFL's that will allow us to *focus on some small number of positions* in the string and pump them. (Such an extension is easy for regular sets. The result for CFL's is much harder to obtain.) Here is a weak version of the so-called *Ogden's lemma*. Using this lemma we can prove that the above L is *not* CFL.
- **Ogden's Lemma**. Let L be a CFL. Then there is a constant n (which may be the same as for the pumping lemma) such that the following holds: if z is any word such that
  $z \in L$ and we mark any n or more positions of z 'distinguished',
  then there are words u, v, w, x, y such that
  $z = uvwxy$,
  vx has *at least one* distinguished place,
  vwx has *at most* n distinguished places, and
  $\forall i \geq 0$: $uv^iwx^iy \in L$.
- **Proof**. Omitted. □

6.3 Closure Properties for CFL's

- Some operations on languages *preserve* CFL's (in the sense that the operations applied to CFL's result in CFL's).
- We say that the class of CFL's closed under an operation if the operation applied to its members is a member of the class.
- If the class of CFL's is closed under a particular operation, we call that fact closure property of the class of CFL's.
- We are particularly interested in effective closure properties of the class of CFL's. For such properties, given *descriptors* for CFL's, there is an *algorithm* to construct a *descriptor* for the CFL that results by applying the operation to these CFL's.

- **Theorem.** The class of CFL's is closed under
  - union,
  - concatenation,
  - Kleene closure,
  - substitution (and hence homomorphism),
  - inverse homomorhism.

Proof. Omitted. □

- **Theorem.** The class of CFL's is *not* closed under
  - intersection,
  - complementation.

Proof. Omitted. □

- **But**: the class of CFL's is closed under intersection with a *regular set*:
  Theorem. If L is a CFL and R a regular set, then $L \cap R$ is a CFL.

Proof. Omitted. □

6.4 Decision Algorithms for CFL's

- We are now interested in decision algorithms for various decision problems about CFL's; e.g. "Is a given CFL L *empty* (or *nonempty*)? Is L *finite* (or *infinite*)? Is a given word in L?" For these problems, we will find decision algorithms.
- There are other decision problems about CFL's: "Is the *complement* of L a CFL? Is L *cofinite*? Are two CFG's *equivalent*? Is a CFG *ambiguous*?" We'll find *tools* for showing that *no algorithm* can do a particular job. Only later (Chap. 8) we will *actually prove* that the above problems have no decision algorithms !!!
- CFL's can be represented by *CFG's*, *PDA's* (*empty stack*) and *PDA's* (*final state*). But we can algorithmically transform one representation into another, so our results will not depend on the representation we choose. So we choose CFG's.

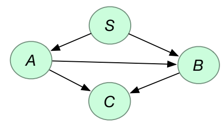
- **Emptiness and finiteness of CFL's.**
- **Theorem.** There exist decision algorithms to determine if a CFL is:
  1) empty;
  2) finite;
  3) infinite.

- **Proof idea.** Let $G = (V, T, P, S)$ be a CFG.
  - To test whether $L(G)$ is (*non*)*empty*, use the test to determine if a variable generates any string of terminals. In particular: $L(G)$ is nonempty *iff* the start symbol S generates some string of terminals.
  - To test if $L(G)$ is (*in*)*finite*, find a CFG $G' = (V', T, P', S)$ in Chomsky Normal Form with no useless symbols, generating $L(G) - \{\varepsilon\}$. (Note: $L(G')$ is finite *iff* $L(G)$ is finite.) Draw a directed graph with a vertex for each variable in V' and an arc from A to B if there is a production in P' of the form $A \rightarrow BC$ or $A \rightarrow CB$ (for any C). Then $L(G')$ is finite *iff* the graph has no cycles.

  □

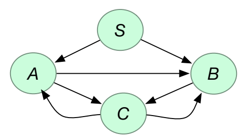
- **Example.** Consider the grammar $G_1 = (V, T, P, S) = (\{A,B,C,S\}, \{a,b\}, P, S)$, where P consists of the following productions:

$$S \to AB$$
$$A \to BC \mid a$$
$$B \to CC \mid b$$
$$C \to a$$



G_1 is in Chomsky Normal Form and has no useless variables. The corresponding graph (see above) has *no* cycles, so $L(G_1)$ is *finite*.

- **Example.** Let us add the production $C \to AB$ to the above grammar. The new grammar G_2 is still in Chomsky Normal Form and has no useless variables. The corresponding graph (see below) *has* cycles, so $L(G_2)$ is *infinite*.



□

- **Membership.**

- Definition. The membership problem for CFG's is the question ``Given a CFG $G = (V, T, P, S)$ and a string $x \in T^*$, is $x \in L(G)$?''

- Question. Does there *exist* a decision algorithm such that, given an *arbitrary* CFG G and an *arbitrary* word x over the alphabet T, answers the question `*Is x a member of L(G)*?'

- Answer. The answer is YES; there is the following *naïve* algorithm:

  1. Convert G to *Greibach normal form* (GNF) G'. /* Recall: $L(G') = L(G) - \{\varepsilon\}$ */
  2. *If* $x = \varepsilon$ *then* Test whether $S \,_G\!\Rightarrow^* \varepsilon$ *else*

     /* Now $x \in L(G')$ *iff* $x \in L(G)$, so focus on GNF G'. *Note*: every production of a GNF grammar adds exactly *one* terminal to the string being generated. So, if x has a derivation in G', then the derivation has *exactly* $|x|$ steps. Next, if *every variable* of G' has $\leq k$ *productions*, then there are $\leq k^{|x|}$ leftmost *derivations* of *words of length* $|x|$. So, is x among them? */

     Try all such derivations systematically to see if x is among them.

- (cont'd)
  - The naïve decision algorithm is *inefficient* because it may check *exponential* number of derivations.
  - However, there is a better, more efficient decision algorithm, called the CYK algorithm (for Cocke-Younger-Kasami). This algorithm
    - is based on the *dynamic programming technique*, and
    - runs in $O(n^3)$ *time*, where $n = |x|$.

- *The CYK algorithm (for Cocke-Younger-Kasami)

- Let x be an arbitrary word of length $n \geq 1$, and G an arbitrary CFG in *Chomsky normal form* (CNF).

- Let x_{ij} be the subword of x of length j beginning at position i. Note: $1 \leq i \leq n$ and $1 \leq j \leq n-i+1$.

- We want to determine for each i and j and for each variable A, whether $A \,_G\!\Rightarrow^* x_{ij}$.
  To achieve that, we make the following key observations :
  - [Case $j=1$] x_{ij} is just one symbol (terminal). *Note*: $A \,_G\!\Rightarrow^* x_{ij}$ *iff* $A \to x_{ij}$ is a production.
  - [Case $j>1$] x_{ij} has at least 2 symbols (terminals). *Note*: $A \,_G\!\Rightarrow^* x_{ij}$ *iff* there is some production $A \to BC$ and some k $(1 \leq k \leq j)$ such that B derives the first k symbols of x_{ij} (i.e. $B \,_G\!\Rightarrow^* x_{ik}$) and C derives the last j-k symbols of x_{ij} (i.e. $C \,_G\!\Rightarrow^* x_{i+k,j-k}$).
  - [Case $j = n$] There is just one subword, x_{1n}, i.e. the whole x. *Note*: We must determine whether $S \,_G\!\Rightarrow^* x_{1n}$.

  Of course, *several* variables may generate x_{ij}; let us collect them in the set $V_{ij} = \{A \mid A \,_G\!\Rightarrow^* x_{ij}\}$.
  Note: given j, the variable i can vary from 1 to n-j+1.

- **Algorithm idea.** Compute the sets V_{ij} by increasing $j = 1, \ldots, n$ while applying the notes in the above cases [j=1], [j>1], and [j=n].

- (cont'd)

begin /* CYK Algorithm

1) **for** $i := 1$ to n do
2) $V_{i1} := \{A \mid A \rightarrow a$ is a production $\wedge$ the ith symbol of x is $a\}$;
3) **for** $j := 2$ to n do
4) **for** $i := 1$ to $n - j + 1$ **do**
5) $V_{ij} := \emptyset$;
6) **for** $k := 1$ to $j - 1$ **do**
7) $V_{ij} := V_{ij} \cup \{A \mid A \rightarrow BC$ is a production $\wedge$ $B \in V_{ik}$ $\wedge$ $C \in V_{i+k,j-k}\}$

endfor

endfor

end

6.5 Dictionary

pumping lemma for CFL lema o napihovanju za KNJ Ogden's lemma Ogdenova lemma cofinite kofiniten CYK algorithm algoritem CYK

7 Turing Machines

Contents

7.1 Introduction

- **What is algorithm? What is computation?**
- The algorithm was traditionally *intuitively* understood as a *recipe*, i.e., a *finite* list of *directives* written in *some language* that tells us how to solve a problem *mechanically*. In other words, the algorithm is a *precisely described routine procedure* that can be applied and systematically followed through to a solution of a problem.
- **Definition** (algorithm intuitively) An ''algorithm'' for solving a problem is a *finite* set of *instructions* that lead the *processor*, in a *finite* number of steps, from the *input data* of the problem to the corresponding *solution*.
- Because there was no need to define formally the concept of the algorithm, it remained firmly at the *intuitive*, informal level.

- The need for a *formal* definition of the concept of algorithm was made clear during the *first few decades* of the *twentieth century* as a result of events taking place in math. **What happened?**
  - At the beginning of the century, Cantor's naive set theory was born. This theory was very *promising* because it offered a *common foundation* to all the fields of mathematics. However, it treated *infinity incautiously* and boldly. This called for a response, which soon came in the form of logical paradoxes.
  - Since Cantor's set theory was *unable* to eliminate them, formal logic was engaged. As a result, three *schools* of mathematical thought—intuitionism, logicism, and formalism—contributed important ideas and tools that enabled an exact and concise mathematical expression and brought rigor to mathematical research.
  - Hilbert's Program was a promising formalistic attempt to *recover* mathematics. Unfortunately, the program was severely *shaken* by *Gödel's* astonishing and far-reaching discoveries about the general properties of formal axiomatic systems and their theories. Thus Hilbert's attempt fell short of formalists' expectations.
  - Nevertheless, the program left *open* an important question about the existence of a certain algorithm—a question that led to the *birth* of *Computability Theory*.

- **The 'big' question**. Now the question was: How can we answer the question "*Is there an algorithm that solves a given problem?*" if it is not clear what algorithm is?
  - Namely, to prove that there *exists* an algorithm that solves the problem, it would suffice to *construct some recipe* and show that the recipe meets all the conditions.
  - But to prove that such an algorithm *does not exist* we should *reject every possible recipe* by showing that it does not meet all the conditions necessary for an algorithm to solve the problem. But there are *infinitely many* possible recipes!
    - To accomplish such a proof, a *characterization* of the *concept of the algorithm* was needed. That is, (1) a *property* had to be found such that *all algorithms* and *algorithms only* have this property; and (2) a precise and rigorous definition of the the *environment* capable of executing algorithms, had to be found. All of this is called the *model of computation*. (Only then we could systematically eliminate all the infinitely many possible recipes.)
    - So, the need to find a model of *computation became* apparent. Here is the definition of what was needed.
  - **Definition.** (model of computation) A model of computation is a definition that formally characterizes the basic notions of algorithmic computation (that is, the algorithm, its environment, and the computation).

- In the 1930 the *search* for a model of computation started and proceeded into *different directions*. Eventually, several models of computation were *proposed*. Each direction proposed its own models of computation. The models are:
  - μ-recursive functions (Kurt Gödel, Stephen Kleene)
  - general recursive function (Jacques Herbrand, Kurt Gödel)
  - λ-calculus (Alonzo Church)
  - Turing machine (Alan Turing)
  - Post machine (Emil Post)
  - Markov algorithms (Andrej Markov)
- These models were *completely different*. Naturally, the following question arose:

  Which model (if any) is the ''best'', i.e. the ''right'' one?

  The majority of researchers accepted the Turing machine as the model which *most adequately* describes (captures) the basic concepts of computation.
- Morover, surprisingly, it was soon *proved* that the models are equivalent in the sense:

  What can be computed by one can also be computed by the others.

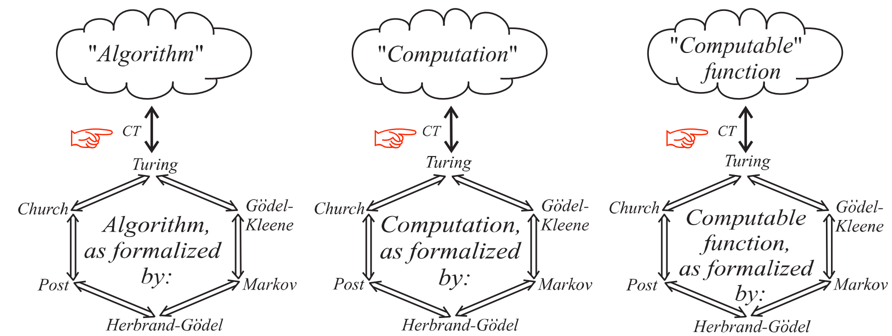
- What about the intuitive understanding of the basic concepts of computation?

  Is there any connection between the *intuitively understood concepts of computation* (i.e. ''algorithm'', ''computation'' and ''computable function'') on the one hand, and the *formal models of computation* on the other?

- The answer is YES. Since all the known models of computation were proved to be *equivalent*, although completely *different*, the following *thesis* was proposed:

- **Computability Thesis** (also Church-Turing thesis). The basic intuitive concepts of computing are perfectly *formalized* as follows:
  - ''*algorithm*'' is formalized by *Turing program*
  - ''*computation*'' is formalized by *execution of a Turing program* in a Turing machine
  - ''*computable function*'' is formalized by *Turing-computable function*

- The thesis was *accepted* by the *majority* of researchers. Nowadays we still accept it (since no one succeeded to refute it).



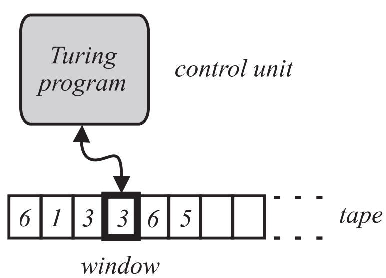
The **Computability Thesis** established a *bridge* between the *intuitive concepts* of "algorithm," "computation," and "computability" on the one hand, and their *formal counterparts* defined by models of computation on the other. In this way it finally opened the door to a *mathematical treatment* of these intuitive concepts.

7.2 The Turing Machine Model

- The FA's and PDA's are somewhat limited: they can only *read* symbols *in succession* from *left to right* from *bounded* input tapes.

- The Turing machine (TM) also has a tape with a window and a control unit with a program.

- But Turing machine can read *and* write symbols anywhere on the *potentially* infinite tape.



- **Definition**. (Turing machine) The basic variant of the Turing machine has the following components: a *control unit* containing a *Turing program*; a *tape* consisting of *cells*; and a movable *window* over the tape, which is connected to the control unit. The details are:
  - The tape is used for *writing* and *reading* the input data, intermediate data, and output data (results). It is divided into equally sized cells, and is *potentially infinite* in one direction (i.e., it can be *extended* in that direction with a *finite number of cells*).

    Each cell contains a tape symbol belonging to a tape alphabet $\Gamma=\{z_1, \ldots, z_t\}$, $t \geq 3$. The symbol z_t is special, for it indicates that a cell is *empty*; for this reason it is denoted by $\sqcup$ and called the empty space. In addition to $\sqcup$ there are at least two additional symbols: 0 and 1. We will assume that $z_1 = 0$ and $z_2 = 1$.

    The *input data* are contained in the input word. This is a word over an input alphabet Σ, such that $\{0,1\} \subseteq \Sigma \subseteq \Gamma\text{-}\{\sqcup\}$. Initially, all the cells are empty (each contains $\sqcup$) except for the *leftmost* cells, which contain the input word.

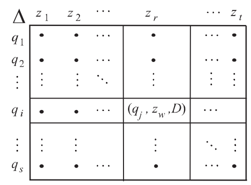
- The control unit is always in some state from a finite set of states $Q = \{q_1, \ldots, q_s\}$, $s \geq 1$. We call q_1 the initial state. Some states are called final; they are gathered in the set $F \subseteq Q$. All the other states are *non*-final. If the index of a state is of no importance, we use q_{yes} and q_{no} to refer to any final and non-final state, respectively.

  There is a Turing program (TP) in the control unit. TP directs TM's components. TP is *characteristic* of the *particular* TM, i.e., different TMs have different TPs. A TP is a *partial* function δ: $Q\times\Gamma \to Q\times\Gamma\times\{L,R,S\}$, called the transition function.

  Note. The TM is by definition deterministic, having at most one choice for a move in each situation.

  We can view δ as a *table* $\Delta = Q\times\Gamma$, where
  - $\Delta[q_i, z_r] = (q_j, z_w, D)$ if $\delta(q_i, z_r) = (q_j, z_w, D)$ is an instr. of δ,
  - $\Delta[q_i, z_r] = 0$ if $\delta(q_i, z_r)\uparrow$ (*undefined*).

  Without loss of generality, we can assume that there is always a transition from a q_{no}, and none from q_{yes}.



- The window can move over *any* cell. Then, the control unit can *read* a symbol through the window, and *write* a symbol through the window, *substituting* the previous symbol. In *one* step, the window can only move to the *neighboring* cell.

- **Before the TM is started**, the following must take place:
  - a. an *input word* is written to the *beginning of the tape*;
  - b. the *window* is shifted to the *beginning of the tape*;
  - c. the *control unit* is set to the *initial state*.
- **From now on** the TM operates independently, in a mechanical stepwise fashion as instructed by its TP. If the TM is in a state $q_i \in Q$ and it reads a symbol $z_r \in \Gamma$, then:

  if q_i is a final state, **then** TM *halts;*

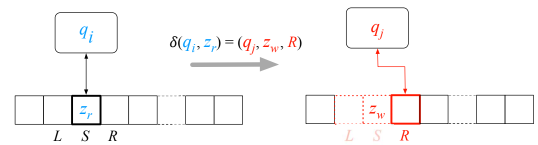
  else, if $\delta(q_i, z_r)\uparrow$ (i.e. TP has no next instruction), **then** the TM *halts*;

  else, if $\delta(q_i, z_r)\downarrow = (q_j, z_w, D)$, **then** the TM does the following:

  a) changes the state to q_j;

  b) writes z_w through the window;

  c) moves the window to the next cell in direction $D \in \{L,R\}$ (for *left* and *right*), or leaves the window where it is ($D = S$, for *stay*).

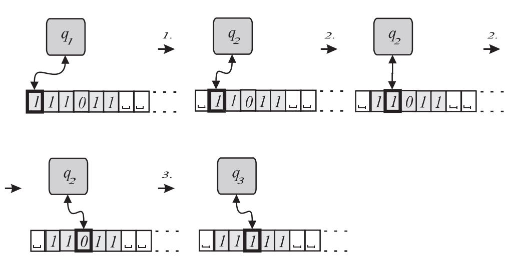


- Formally, a TM is a seven-tuple $T = (Q,\Sigma,\Gamma,\delta,q_1,\sqcup,F)$. To fix a *particular* TM, we must *fix* Q,Σ,Γ,δ,F. (**end of definition**)

- **Example**. Here is a TM T that computes the sum $m+n$ of natural numbers. The input data m,n are in the *input word* 1^m01^n; their sum is returned on the tape in the word 1^{m+n} after T halts. E.g., given input word 111011, T returns the word 11111.
  - **Algorithm idea**. If the first symbol of the input word is 1, then TM deletes it (instr.1), and then moves the window to the right over all the symbols 1 (instr.2) until the symbol 0 is read. TM then substitutes this symbol with 1 and halts (instr.3). But, if the first symbol of the input word is 0, then TM deletes it and halts (instr.4).
  - **Turing machine T.**

    $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$, where:
    - $Q = \{q_1, q_2, q_3\}$
    - $\Sigma = \{0,1\}$
    - $\Gamma = \{0,1,\sqcup\}$
    - $F = \{q_3\}$
    - δ has the following instructions:
      1. $\delta(q_1, 1) = (q_2, \sqcup, R)$
      2. $\delta(q_2, 1) = (q_2, 1, R)$
      3. $\delta(q_2, 0) = (q_3, 1, S)$
      4. $\delta(q_1, 0) = (q_3, \sqcup, S)$

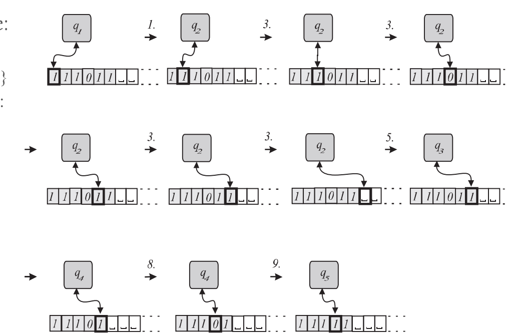
  T's computation.



- **Example**. Here is *another* TM T' that computes the sum $m+n$ of natural numbers.
  - **Algorithm idea**. First, the window is moved to the right until $\sqcup$ is reached. Then the window is moved to the left (i.e., to the last symbol of the input word) and the symbol is deleted. If the deleted symbol is 0, the machine halts. Otherwise, the window keeps moving to the left and upon reading 0 the symbol 1 is written and the machine halts.
  - **Turing machine T'.**

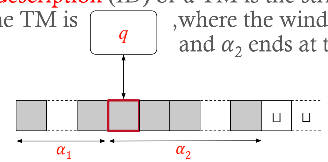
    $T' = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$, where:
    - $Q = \{q_1, q_2, q_3, q_4, q_5\}$
    - $\Sigma = \{0,1\}$ $\Gamma = \{0,1,\sqcup\}$ $F = \{q_5\}$
    - δ has the following instructions:
      1. $\delta(q_1, 1) = (q_2, 1, R)$
      2. $\delta(q_1, 0) = (q_2, 0, R)$
      3. $\delta(q_2, 1) = (q_2, 1, R)$
      4. $\delta(q_2, 0) = (q_2, 0, R)$
      5. $\delta(q_2, \sqcup) = (q_3, \sqcup, L)$
      6. $\delta(q_3, 0) = (q_5, \sqcup, S)$
      7. $\delta(q_3, 1) = (q_4, \sqcup, L)$
      8. $\delta(q_4, 1) = (q_4, 1, L)$
      9. $\delta(q_4, 0) = (q_5, 1, S)$

  T''s computation.



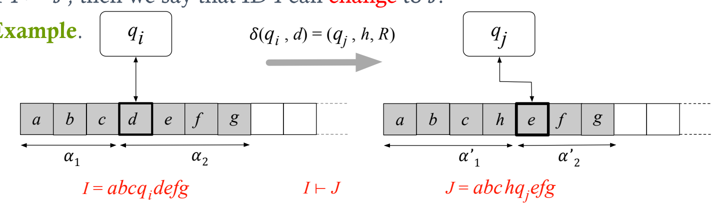
Definitions.

- An instantaneous description (ID) of a TM is the string $I = \alpha_1 q \alpha_2$, if the current configuration of the TM is (diagram) ,where the window is over the first symbol of α_2 and α_2 ends at the rightmost non-blank symbol.

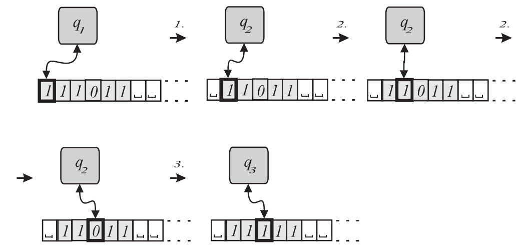


An ID is the ``snapshot'' of a current configuration (status) of TM's components between successive instructions.

- An ID I can directly change to J -- written $I \vdash J$ -- *if* there is an instruction in TM's program whose execution changes I to J. The *reflexive and transitive closure* of $\vdash$ is $\vdash^*$; if $I \vdash^* J$, then we say that ID I can change to J.
- **Example**.



- **Example.** Let us be given the following sequence of `snapshots' (situations) of some Turing machine while the machine executes its program δ:



- The computation is described by the following sequence of ID's (snapshots):

$$q_1 111011 \vdash \sqcup q_2 11011 \vdash \sqcup 1 q_2 1011 \vdash \sqcup 11 q_2 011 \vdash \sqcup 11 q_3 111$$

7.3 Use of a Turing Machine

There are three elementary tasks where a TM can be used. These are:

- Function computation, i.e. the task
  ``*Given a function φ and arguments $a_1, \dots, a_k$, compute $\varphi(a_1, \dots, a_k)$.*''
- Set recognition, i.e. the task
  ``*Given an object x and a set S, decide whether or not $x \in S$.*''
- Set generation, i.e. the task
  ``*Given a set S, generate a list $x_1, x_2, x_3, \dots$ of exactly the members of S.*''

- **Function computation on TM's.**
- Each TM T is implicitly associated, for each $k \geq 1$, with a function φ_T that maps k words into one word. We arrive at φ_T as follows.

  Definition. Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a TM and $k \geq 1$. The (k-ary) proper function of T is a partial function $\varphi_T : (\Sigma^*)^k \to \Sigma^*$, defined as follows:

  If the input to T is k words $u_1, \dots, u_k \in \Sigma^*$, then the value of φ_T at $u_1, \dots, u_k$ is *defined* to be

$$\varphi_T(u_1, \dots, u_k) := \begin{cases} v, & \text{if } T \text{ halts} \wedge \text{returns on the tape the word } v \wedge v \in \Sigma^*; \\ \uparrow, & \text{if } T \text{ doesn't halt} \vee \text{the tape doesn't have a word in } \Sigma^*. \end{cases}$$

- The *interpretation* of $u_1, \dots, u_k$ and v is *arbitrary.* For example, we can regard $u_1, \dots, u_k$ as encodings of natural numbers $a_1, \dots, a_k$; then φ_T can be viewed as an arithmetical function (from $\mathbb{N}^k$ to $\mathbb{N}$), and v as encoding of $\varphi_T(a_1, \dots, a_k)$.

- In practice, however, we usually face the opposite task:

  ``Given a k-ary function $\varphi : (\Sigma^*)^k \to \Sigma^*$, find a TM T that can compute φ's values.''

  In other words, given φ, we must find a Turing machine T such that $\varphi = \varphi_T$ will hold.

- The *capacity* of such a T (the extent to which T can possibly compute φ's values) *depends on* φ. We distinguish between three kinds of φ's, depending on *how powerful*, *if at all*, *T can be*. Informally, we say that a function φ is:
  - **computable** if there is a T that *can compute* φ's value *for any arguments*;
  - **partial computable** if there is a T that *can compute* φ's value *whenever* φ *is defined*;
  - **incomputable** if there is *no* T that can compute φ's value whenever φ is defined.

  Definition. Let $\varphi : (\Sigma^*)^k \to \Sigma^*$ be a function. Then:
  - φ is computable if $\exists$ TM that can compute φ anywhere on dom(φ) $\wedge$ dom(φ) = $(\Sigma^*)^k$;
  - φ is partial computable (p.c.) if $\exists$ TM that can compute φ anywhere on dom(φ) ;
  - φ is incomputable if there *is no* TM that can compute φ anywhere on dom(φ) .

- **Set recognition on TM's.**

- Each TM T is implicitly associated with a language L(T) , the *language accepted* by T. Here is the definition.

  Definitions. Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a TM and $w \in \Sigma^*$ a string.
  We say that w is accepted by T if $q_1 w \vdash^* \alpha_1\, p\, \alpha_2$, for some $p \in F$ and $\alpha_1, \alpha_2 \in \Gamma^*$.
  The language accepted by T is the set $L(T) = \{w \mid w \in \Sigma^* \wedge w$ is accepted by $T\}$.

  So, a word is *accepted* by T if it causes T to enter a *final state* (if submitted as input).
  The *language accepted* by T consists of all such words.

- Again, the *interpretation* of w is *arbitrary*. For example, we can view w as encoding of a natural number; then we can view L(T) as the set of all natural numbers that are accepted by T.

- In reality, however, we usually face the opposite task:

  ``Given a set $S \subseteq \Sigma^*$, find a TM T that accepts S.''

  In other words, given a language (set) S, we must find a TM T such that $S = L(T)$ will hold.

- Again, the *ability* of such a T (the extent to which T can recognize members and non-members of S) *depends on* S. We distinguish between three kinds of sets S, depending on *how powerful*, *if at all*, *T can be*. Informally, we say that a set S is:
  - **decidable** if there is a T that can decide the question ``Is $x \in S$?'' **for any** x;
  - **semi-decidable** if there is a T that answers **YES** to ``Is $x \in S$?'' **if** x **is in** S;
  - **undecidable** if there is *no* such T.

  Definition. Let $S \subseteq \Sigma^*$ be a language (set). Then:
  - S is decidable if $\exists$ TM that answers YES/NO to ``Is $x \in S$?'', for *any* $x \in \Sigma^*$.
  - S is semi-decidable if $\exists$ TM that answers YES to ``Is $x \in S$?'' *whenever* $x \in S$.
  - S is undecidable if there is *no* TM that answers YES to ``Is $x \in S$?'' whenever $x \in S$.

- **Comment.** This means that, for *some* sets S, we *cannot algorithmically answer* the question ``Is $x \in S$?'' Why?

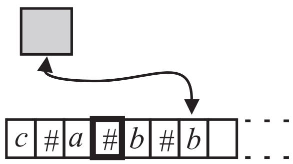
  If $S = L(T)$ is such a language, and $x \in \Sigma^*$ an arbitrary input word, then:
  - If x **is** in S, then T will eventually halt on input x (and *accept* x).
  - *But*, T fails to halt on at least one $w \in \Sigma^*$ which is **not in** L.

    (So, while T is running, it may be so because x is one of such unfortunate w's.)

  For such an S, as long as T is *still running* on input x, we cannot tell whether
  - T will eventually halt (and accept or reject x) if we let T run long enough, or
  - T will run forever.

  In other words:
  - If, in truth, $x \in S$, then T will *halt and accept* x (T will answer YES to ``Is $x \in S$?'')
  - If, however, in truth, $x \notin S$, then
    - T may *halt and reject* x (T answers NO to the question ``Is $x \in S$?''); or
    - T may *never halt* (T *never answers* NO *to the question* ``Is $x \in S$?'')

- **Set generation on TM's**
- A TM T (not every) may be implicitly associated with a language $G(T)$, the *language generated* by T. Here is the definition.
- **Definition**. Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a TM. T is called a generator if it writes to its tape, in succession and delimited by #, (some) words from Σ^*. We assume that # is not in Σ. The language generated by T is defined to be the set $G(T) = \{w \mid w \in \Sigma^* \wedge T \text{ eventually writes } w \text{ to the tape}\}$.
- Example. The elements c,a,b are in the generated language $G(T) = \{c, a, b, \ldots\}$



- In practice, we usually face the opposite task:

  ``Given a set S, generate a list $x_1, x_2, x_3, \ldots$ of exactly the members of S.''

  In other words, given a language (set) S, we must find a TM T such that $S = G(T)$ will hold.
- **Observation.** Some sets can be generated and others can't.

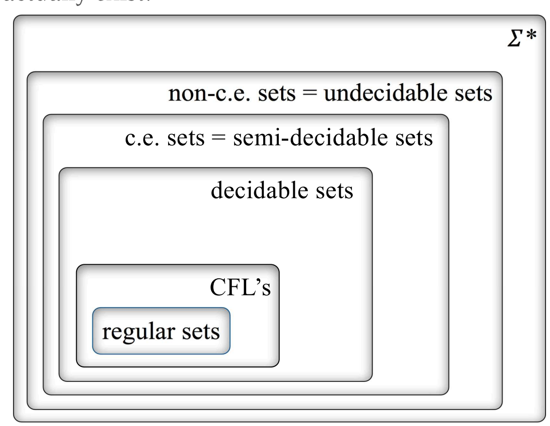
  Examples. The set **N** of natural numbers *can* be generated by an obvious algorithm: 1, 2, 3, ... Also the set **Z** of integers can be generated by an obvious algorithm: 0, 1, -1, 2, -2, 3, -3, ... And also the set **Q** of rational numbers can be generated. How? And the set **P** of all primes too. How? But, the sets **R** of all reals and the set of reals in the interval [0,1] *cannot*.
- Questions: *When* can a set S be generated? When can elements of S be *listed* in a finite or infinite *sequence* $x_1, x_2, x_3, \ldots$ so that *each and every* element of S *eventually* appears in the sequence? When can the sequence be generated by a TM, i.e *algorithm*? Can every *countable* set be *algorithmically* generated?

- **C.e. languages (sets)**
- Suppose that a set S can be *generated* by some TM T. Then, T can write the elements of S in succession to the tape. If x is and *arbitrary* element of S, then x will *eventually* appear in the generated list $x_1, x_2, x_3 \ldots$ as x_n, for some $n \in$ **N**. So we can speak of the 1st, 2nd, 3rd, ... nth element of S. Since the elements of S can be *enumerated*, we say that the set S is enumerable.
- But we are interested in enumerable sets can be *algorithmically generated*, i.e. generated by a TM? Such sets (languages) will be called *computably enumerable*.

  Definition. A set S is computably enumerable (c.e.) if $S = G(T)$ for some TM T, that is, if S can be generated by a Turing machine.
- **Theorem**. A set S is c.e. *iff* S is semi-decidable.

  Proof. Omitted. □

- **Summary**. The relation between the *classes* of languages that we have met until now is below. We will prove later that semi-decidable and undecidable languages actually exist.

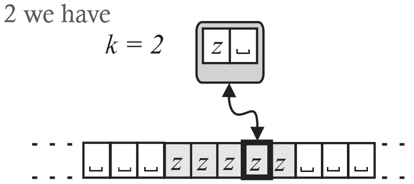


7.4 Modifications of the Turing Machine

- One reason for the acceptance of the TM as a *general model of computation* is that the *basic* model of the TM is equivalent to many *modified versions* (that *seem* to have *increased computing power*). We'll give *informal proofs* of these equivalences. Each version has one or several of the following modifications:
  - Finite storage
  - Multiple-track tape
  - Two-way infinite tape
  - Multiple tapes
  - Multidimensional tape
  - Nondeterministic program

TM with finite storage.

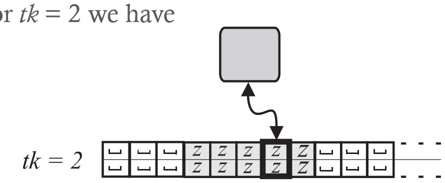
- This variant V has in its control unit a finite storage capable of memorizing $k \geqslant 1$ tape symbols and using them during the computation. The Turing program (TP) is the function $\delta_V : Q \times \Gamma \times \Gamma^k \to Q \times \Gamma \times \{L, R, S\} \times \Gamma^k$.
- Example. For $k = 2$ we have



- Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

TM with multiple-track tape.

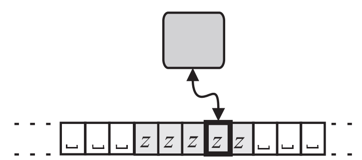
- This variant V has the tape divided into $tk \geqslant 2$ *tracks*. On each track there are symbols from the alphabet Γ. The window displays tk-tuples of symbols, one symbol for each track. The TP is $\delta_V : Q \times \Gamma^{tk} \to Q \times \Gamma^{tk} \times \{L, R, S\}$.
- Example. For $tk = 2$ we have



- Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

TM with two-way infinite tape.

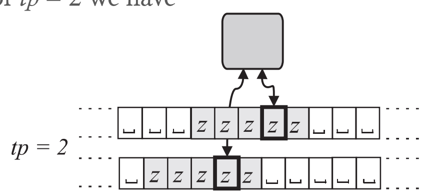
- This variant V has the tape unbounded in both directions. Formally, the TP is the function $\delta_V : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$.



- Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

TM with **multiple tapes**.

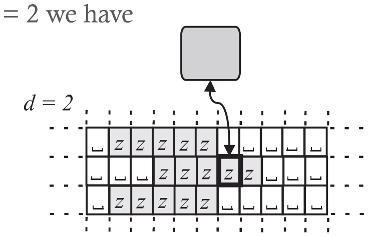
- This variant V has $tp \geqslant 2$ unbounded tapes. Each tape has its own window that is independent of other windows. TP is $\delta_V : Q\times\Gamma^{tp} \to Q\times(\Gamma^{tk}\times\{L,R,S\})^{tp}$.
- Example. For $tp = 2$ we have



- Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

TM with **multidimensional tape**.

- This variant V has a d-dimensional tape, $d \geqslant 2$. The window can move in d dimensions, i.e., $2d$ *directions* $L_1, R_1, L_2, R_2, \ldots, L_d, R_d$. The Turing program is $\delta_V : Q\times\Gamma \to Q\times\Gamma\times\{L_1,R_1,L_2,R_2,\ldots,L_d,R_d,S\}$.
- Example. For $d = 2$ we have



- Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

TM with **nondeterministic program**.

- This variant V has a Turing program δ that assigns to each (q_i, z_r) a *finite* set of *alternative transitions* $\{(q_{j_1}, z_{w_1}, D_1), (q_{j_2}, z_{w_2}, D_2), \ldots\}$. The machine *nondeterministically* chooses a transition from the set and makes it.
- How does V choose a transition out of the current alternatives?

  The following assumed: *the machine chooses one of the transitions that lead it to a solution* (e.g., to a state q_{yes}), *if such transitions exist; otherwise, the machine halts.*

  The nondeterministic TM is not a reasonable model of computation because it can foretell the future when choosing from alternative transitions. Nevertheless, it is a very useful tool, which makes it possible to define the *minimum number of steps* needed to compute the solution (when a solution exists). This is important when we investigate the computational complexity *of problem solving*. We will see that in the following chapters.
- Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

The importance of the modifications of the TM.

- Are the modifications of TM of *any* use in Computability Theory ?
  The answer is yes. The modifications are useful when we try to *prove the existence of a TM for solving a given problem P*. Usually, the construction of such a TM is *easier* if we choose a *more versatile* modification of TM.

  Sometimes, we can even *avoid* the complicated constructing of the *actual* TM for solving P. How? We do as follows:
  1. We devise an *intuitive algorithm* A (a ``recipe'', finite list of instr.) for solving P.
  2. Then we say: ``By the *Computability Thesis*, there is a TM T that does the same as A."

  Then, we can refer to this T (as the true algorithm for solving P) and treat it mathematically.
- Since *none* of the modifications is more powerful than the basic TM, this additionally supports our belief in the correctness of the Computability Thesis.
- The computations on the modifications of TM can considerably differ in *time* (number of steps) and *space* (number of visited cells). But this will become important only in Computational Complexity Theory (where we will investigate the time and/or space complexity *of problem solving*).

7.5 Universal Turing Machine

- In this section we will describe how Turing discovered a seminal fact about Turing machines.
  We will:
  - explain how TM's can be encoded (represented by words over an alphabet);
  - realize that TM's can read codes of other TM's and compute with them;
  - explain how Turing applied this to construct the Universal Turing Machine (UTM), *a special TM that can compute whatever is computable by any other TM.*
  - explain how Turing's discovery triggered the search for a *physical* realization of the UTM, which in 1940s resulted in the first general-purpose computers.

Coding of TM's

- Given a TM $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$, we want to encode T, i.e. represent T by a word over some coding alphabet.
- How will we encode TM T?
  - The *coding alphabet* will be $\{0,1\}$.
  - We'll only encode δ, but in such way that Q,Σ,Γ,F, which determine the particular T, can be restored from the encoded δ. How will we encode TP δ?
    1. If $\boldsymbol{\delta(q_i, z_j) = (q_k, z_\ell, D_m)}$ is an instruction of δ,
       then we encode the instruction by the word
       $K = \mathbf{0^i10^j10^k10^\ell10^m}$ where $D_1=L$, $D_2=R$, $D_3=S$.
    2. In this way, we encode each instruction of δ.
    3. From the obtained codes $K_1, K_2, \ldots, K_r$ we construct the code $\langle\delta\rangle$ of δ:
       $\langle\delta\rangle = \mathbf{111}K_1\mathbf{11}K_2\mathbf{11}\ \mathbf{\ldots}\ \mathbf{11}K_r\mathbf{111}$
  - The code $\langle T\rangle$ of the TM T can now be identified with $\langle\delta\rangle$ (i.e. $\langle T\rangle := \langle\delta\rangle$).

- **Example.**
- What is the code $\langle T\rangle$ of the first TM T that computes $m+n$ (see Sect.7.2)?
- The components of T were:
  - $Q = \{q_1, q_2, q_3\}$ or encoded: $Q = \{\mathbf{0,00,000}\}$
  - $\Sigma = \{0,1\}$
  - $\Gamma = \{0,1,\sqcup\}$ or encoded: $\Gamma = \{\mathbf{0,00,000}\}$ (note: $0=z_1$, $1=z_2$, $\sqcup=z_3$)
  - $F = \{q_3\}$

The Turing program δ of T had four instructions:

1. $\delta(q_1, 1) = (q_2, \sqcup, R)$ or encoded: $K_1 = \mathbf{01001001000100}$
2. $\delta(q_2, 1) = (q_2, 1, R)$ or encoded: $K_2 = \mathbf{00100100100100}$
3. $\delta(q_2, 0) = (q_3, 1, S)$ or encoded: $K_3 = \mathbf{001010001001000}$
4. $\delta(q_1, 0) = (q_3, \sqcup, S)$ or encoded: $K_4 = \mathbf{010100010001000}$

Then the code of δ is

$\langle\delta\rangle = \mathbf{111010010010001001100100100100100110010100010010001101010001000100011 1}$

Enumeration of TM's

- We can interpret $\langle T\rangle$ to be the *binary code* of some *natural number*. We call this number the index of T.
  - Example. The index of the TM T for computing $m+n$ (see previous example) is 13310153014029126947161548189999893572326199465 67. So, indexes are *huge* numbers. This will *not* be an obstacle, because we will *not* use them in arithmetic operations.
- But, some natural numbers are *not* indexes, because their binary codes do not have the required form, which follows from the encoding method.
  - To avoid this, we make the following convention:
    Any natural number whose binary code is not of the required form is an index of the empty TM.
    The δ of this TM is *everywhere undefined*; hence, for every input, this TM *immediately halts*, in 0 steps.
- Now, we may say: Every natural number is the index of exactly one TM.

- Given an arbitrary natural number $n \in \mathbf{N}$, we can *restore* from the number n the components Q, Σ, Γ, F, which determine the particular TM.
  - How? We (1) inspect the binary code of n to check if it is of the required form $111K_1 11K_2 11 \ldots 11K_r 111$. If it is, we (2) partition this code into strings $K_1, K_2, \ldots, K_r$ and by analyzing these we can collect all the information needed to restore all the components δ, Q, Σ, Γ, F of the TM T.
- The restored TM can be viewed as the nth basic TM and be denoted by T_n .
- By letting n run through 0,1,2, … we obtain the sequence

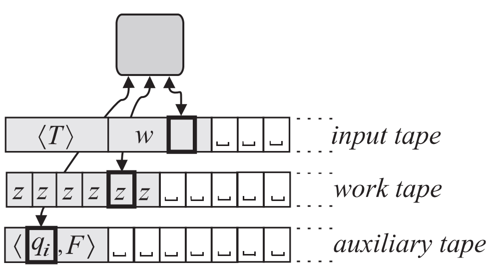
$$T_0, T_1, T_2, \ldots$$

of Turing machines. This is an enumeration of all basic TMs.

- **The existence of a Universal Turing Machine**
- In 1936, using the enumeration of TM's, Turing discovered a seminal fact about TM's. We state the discovery in the following proposition.
- **Proposition**. There is a Turing machine that can compute whatever is computable by any other Turing machine.
- **Proof idea.** The idea is to construct a Turing machine U that will be capable of *simulating any other* TM T. To achieve this, we use the method of proving by *Computability Thesis* (CT):
  a) first, we describe the *concept* of the machine U and describe the *intuitive algorithm* (that should be) executed by U's Turing program, and
  b) then we refer to CT to prove that U *exists*.

  (After this we can, of course, try to construct U in full detail.)

- **Proof**.
  (a) The concept of the machine U is depicted below.



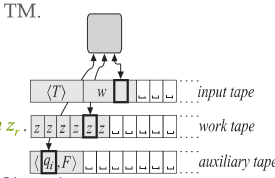
The input tape contains an input word consisting of two parts: the code $\langle T \rangle$ of an *arbitrary Turing machine* $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$, and an *arbitrary word* w.

The work tape is initially empty. The machine U will use it in *exactly the same way* as T would use its own tape when given the input w.

The auxiliary tape is initially empty. The machine U will use it to record the *current state* in which T would be at that time, and for *comparing* this state with the *final states* of T.

The control unit contains a Turing program that executes an algorithm, which is *intuitively* described (on the next slide).

The Turing program of U should execute the following *intuitive algorithm*:



1. Check if the input word is $\langle T, w \rangle$, where $\langle T \rangle$ is a code of some TM. If it is not, halt.
2. From $\langle T \rangle$ restore F and write $\langle q_1, F \rangle$ to the auxiliary tape.
3. Copy w to the work tape and shift its window to the left.
4. *// Let the auxiliary tape have $\langle q_i, F \rangle$ and the work tape window scan z_r .*
   If $q_i \in F$, halt. *// T would halt in the final state q_i.*
5. On the input tape, search in $\langle T \rangle$ for the code of the instruction $\delta(q_i, z_r) = \ldots$
6. If not found, halt. *// T would halt in the non-final state q_i.*
7. *// Suppose that the found instruction is $\delta(q_i, z_r) = (q_j, z_w, D)$.*
   On the work tape, write the symbol z_w and move the window in direction D.
8. On the auxilliary tape, replace $\langle q_i, F \rangle$ by $\langle q_j, F \rangle$.
9. Return to step 4.

- **(b)** This algorithm can be executed by a *human*. So, by the *Computability Thesis*, there is a TM $U = (Q_U, \Sigma_U, \Gamma_U, \delta_U, q_1, \sqcup, F_U)$ whose program δ_U executes this algorithm. We call U the Universal Turing Machine (UTM).

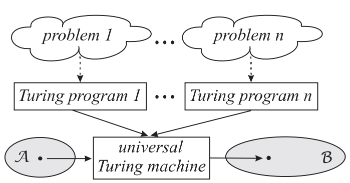
□

- **Construction of an UTM**
- The universal TM U was actually constructed (i.e. described in detail).
- It was to be expected that $\langle U \rangle$ would be a *huge* sequence of 0s and 1s.
  - Indeed, the code of U constructed by Penrose and Deutsch in 1989 had ≈ 5,500 bits.
- But there are *other* TM's that are *equivalent* to U (e.g. the *basic model* simulating U).
  - So, there are other *universal* TM (each differs from U but has the same power as U).
- What is the *simplest* universal TM?
  - We focus on universal TM's with *no storage* in the control unit, and a *single two*-way infinite tape with *one track*.
  - How shall we measure the `simplicity' of such universal TM's?
    - Shannon proposed the *product* $|Q_U|\cdot|\Gamma_U|$ (= the *maximum* number of instructions in δ_U);
    - Alternatively, a more realistic would be the number of *actual* instructions in δ_U.

- Soon it became clear that there is a trade-off between $|Q_U|$ and $|\Gamma_U|$; that is, $|Q_U|$ can be *decreased* if $|\Gamma_U|$ is *increased*, and vice versa.
- So the researchers focused on different *classes* of universal TM's.
  - Such a class is denoted by UTM(s,t), for some $s,t \geq 2$, and by definition contains all the universal TM's with s states and t tape symbols.
  - In 1996, Rogozhin constructed universal TM's in the classes
    - UTM(2,18), i.e., the machine has 2 states and 18 tape symbols
    - UTM(3,10),
    - UTM(4,6),
    - UTM(5,5),
    - UTM(7,4),
    - UTM(10,3),
    - UTM(24,2). i.e., the machine has 24 states and 2 tape symbols

    Of these, the machine $U \in$ UTM(4,6) has the smallest number of instructions: 22.
  - The search for better U's continues.

- **The importance of the Universal Turing Machine**
- Turing's proof of the existence of a universal TM was a *theoretical proof* that a general-purpose computing machine is *possible*, *at least in principle*.



- Turing was certain that such a machine is *possible in reality* too:

  It is possible to construct a *physical* computing machine that can compute whatever is computable by *any other physical* computing machine.

  He envisaged something that is today called the general-purpose computer.

- **Practical consequences: General-purpose computer**
- The construction of a *general-purpose computing machine* started in the1940s. Researchers developed the first such machines, now called computers.
  - For example, ENIAC, EDVAC, IAS. By the mid-1950s, a dozen other computers emerged.
- But, the development of early computers did *not* closely follow the structure of the universal TM. The reasons for this were
  - the desire for the *efficiency* of the computing machine and
  - the *technological conditions* of the time.

- Abstracting the essential differences between these computers and the UTM, and describing the differences in terms of TM's, we find the following:
  - *Cells are now enumerated.*
  - The *program is written on the tape* (instead of the control unit)
  - The *control unit has:*
    - *direct access* to any cell in constant time (so there is no window).
    - *different duties*. In each step, it typically does the following:
      1. reads an instruction from a cell;
      2. reads operands from cells;
      3. executes the operation on the operands;
      4. writes the result to a cell.
    - *new components*: *program counter* (to point the cell with the next instruction to be read), *registers* (for the operands of the operation), *accumulator* (for the result of the operation).
- Due to these differences, terminological differences also arose: *main memory* (≈tape), *program* (≈Turing program), *processor* (≈control unit), *memory location* (≈cell), and *memory address* (≈cell number). The general structure of these computers was called the *von Neumann architecture*.

7.6 The First Basic Results

- In the previous chapters we have defined the *basic notions* and *concepts* of a theory that we are interested in *Computability Theory*.
- In particular, we have formally defined the notions of *algorithm*, *computation*, and *computable function*. We have also defined a few new notions, such as the *decidability* and *semi-decidability* of a set, that will play key roles in the next chapter.
- We now start using this apparatus and deduce the first *theorems* of *Computability Theory*. In this short section we will list several simple but useful theorems about decidable and semi-decidable sets.

- Theorems.

  a) S *is decidable* $\Longrightarrow$ S *is semi-decidable*

  b) S *is decidable* $\Longrightarrow$ $\overline{S}$ *is decidable*

  c) S *and* $\overline{S}$ *are semi-decidable* $\Longrightarrow$ S *is decidable*

  d) A *and* B *are semi-decidable* $\Longrightarrow$ $A \cap B$ *and* $A \cup B$ *are semi-decidable*

  e) A *and* B *are decidable* $\Longrightarrow$ $A \cap B$ *and* $A \cup B$ *are decidable*

  Proofs. Omitted. □

- Here, we also omit the following important theorems:
  - the *Padding Lemma*,
  - the *Parametrization* (*s-m-n*) *Theorem*, and
  - the *Recursion* (*Fixed-Point*) *Theorem*.

7.7 Dictionary

Turing machine Turingov stroj naive set theory naivna teorija množic paradox paradoks, protislovje intuitionism intuicionizem logicism logicizem formalism formalizem Hilbert's program Hilbertov program model of computation računski model μ-recursive function μ-rekurzivna funkcija general recursive functions splošno rekurzivna funkcija λ-calculus λ-račun Post machine Postov stroj Markov algorithms algoritmi Markova, Markovski algoritmi computability thesis teza o izračunljivosti tape trak cell celica tape alphabet tračna abeceda empty space presledek input word vhodna beseda input alphabet vhodna abeceda control unit nadzorna enota state (initial, final) stanje(začetno, končno) Turing program Turingov program transition function funkcija prehodov window okno instantaneous description trenutni opis directly changes neposredno preide changes preide elementary task osnovna naloga function computation računanje (vrednosti) funkcij set recognition razpoznavanje množic set generation generiranje množic k-ary proper function k-mestna lastna funkcija computable function izračunljiva funkcija partial computable function parcialna izračunljiva funkcija incomputable function neizračunljiva funkcija language accepted by jezik, ki ga sprejme decidable odločljiv semi-decidable polodločljiv undecidable neodločljiv to halt ustaviti se language generated by jezik, ki ga generira enumerable prešteven computably enumerable (c.e.) izračunljivo prešteven (c.e.) finite storage končni pomnilnik multiple-track večsledni two-way infinite dvosmerni multiple-tape večtračni multidimensional večdimenzionalni universal TM univerzalni TS coding kodiranje index indeks enumeration oštevilčenje general-purpose splošno namenski

8
Undecidability

Contents

8.1 Computational Problems

- In previous chapter we discussed:
  - how the values of *functions can be computed*,
  - how *sets can be generated*, and
  - how *sets can be recognized*.

  All of these are elementary computational tasks in the sense that they are all closely connected with the *Turing machine*.
- However, in practice we are confronted with *other kinds of problems* that require certain computations to yield their solutions. All such problems we call computational problems.

- **Decision problems and other kinds of computational problems**
- We define the following four kinds (classes) of computational problems:
  - Decision problems (also called yes/no problems). The solution of a decision problem is the answer YES or NO. (The solution is a single bit.)
    - Examples: *Is there* a prime number in a given set of natural numbers?
      Is there a Hamiltonian cycle in a given graph?
  - Search problems. The solution of a search problem is an element of a given set S such that the element has a given property P. (The solution is an element of a set.)
    - Examples: *Find* the largest prime number in a given set of natural numbers.
      Find the shortest Hamiltonian cycle in a given weighted graph.

(cont'd)

- Counting problems. The solution of a counting problem is the number of elements of a given set S that have a given property P. (The solution is a natural number.)
  - Examples: *How many* prime numbers are in a given set of natural numbers? *How many* Hamiltonian cycles are in a given graph?
- Generation problems (also called enumeration problems). The solution of a generation problem is a list of elements of a given set S that have a given property P. (The solution is a sequence of elements of a set.)
  - Examples: *List* all the prime numbers in a given set of natural numbers. *List* all the Hamiltonian cycles of a given graph.

- **Which of these kinds of problems should we focus on**?
- **Answer**: We will focus on the decision problems.

  Why? Because the decision problems ask for the simplest possible solutions, i.e. solutions representable by a single bit. We are pragmatic and expect that this will make our study of computational problems simpler.
- **But**, our choice does not imply that other kinds of computational problems are not interesting—we only want to postpone their treatment until the decision problems are better understood.

8.2 Problem Solving

- Now the following question immediately arises:

  Can we use the accumulated knowledge
  about how to solve the three elementary computational tasks
  to solve other kinds of computational problems?
- In this section we will explain how this can be done for decision problems. In particular, we will
  1. establish a link between *sets* (*formal languages*) and *decision problems*
  2. and *apply our knowledge about sets* (*formal languages*) to decision problems.

- **Language of a decision problem.**
- There is a link between *decision problems* and *sets*, which enables us to *reduce* the *questions about decision problems* to *questions about sets*. We uncover it in 4 steps.

  1. **Let D be a decision problem.**
  2. We are usually faced with a particular instance d of the problem D. The instance d is obtained from D by replacing the variables in the definition of D with actual data. The problem D can be viewed as the *set of all the possible instances* of D. We will say that an instance $d \in D$ is positive/negative if the answer to d is YES/NO, respectively.

  So: **Let d be an instance of D.**

  Example. Let D_{Prime} = "Is n a prime number?" be a decision problem. If we replace the *variable* n by *actual data*, say 4, we obtain the *instance* d_1 = "Is 4 a prime number?" of D_{Prime}. This instance is *negative* because its solution is the answer NO. In contrast, since 2009 we know that the solution to d_2 = "Is $2^{43112609}$-1 a prime number?" is YES, so d_2 is *positive*. □

3. But how can we *compute* the answer to the instance d, say on the TM?

In the natural-language description of d there can be various actual data (numbers, matrices, graphs, …). However, to compute the answer on a *machine*—be it an abstract model such as the TM or a modern computer—*we must rewrite these actual data in a form that is understandable to the machine*. How?

Since any machine uses its own alphabet Σ (e.g. $\Sigma = \{0, 1\}$), we must choose a function that will *transform* (*encode*) *every instance of* D *into a word in* Σ^*. We call such a function the coding function and denote it by 'code'. Thus, code : $D \rightarrow \Sigma^*$, and code(D) is the *set of codes of all instances of* D. We will write $\langle d \rangle$ instead of code(d).

Example. The instances of the problem D_{Prime}= "Is n a prime number?" can be encoded by the function code : $D_{\text{Prime}} \rightarrow \{0,1\}^*$ that maps a number n to its binary representation. For example, ⟨"Is 4 a prime number?"⟩ = 100 and ⟨"Is 5 a prime number?"⟩ = 101. □

So: **Let code : $D \rightarrow \Sigma^*$ be a coding function.**

4. We *could* now search for a TM that will compute the answer to d when given $\langle d \rangle$. But, we proceed differently! How?

Gather the codes of all the positive instances of D in a set $L(D)$.

$L(D)$ is a subset of Σ^*, so it's a formal language. It is associated with the problem D. Here is it's official definition.

Definition. The language of a decision problem D is the set $L(D)$ which is defined by $L(D) = \{\langle d \rangle \in \Sigma^* \mid d \text{ is a positive instance of } D\}$.

Example. The language of the decision problem D_{prime} = "Is n a prime number?" is the set $L(D_{\text{Prime}}) = \{10, 11, 101, 111, 1011, 1101, 10001, 10011, 10111, 110101, \ldots\}$. □

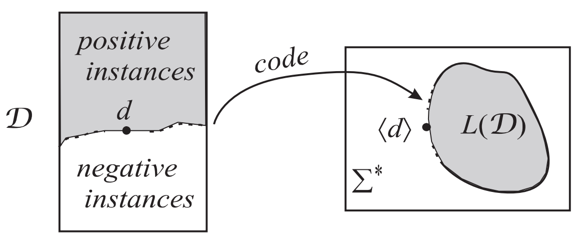
- Now the following should be obvious:

$$d \in D \text{ is positive} \iff \langle d \rangle \in L(D) \qquad (\text{✻})$$

This is *the link* between *decision problems* and *sets*.

- **What did we gain by this?** The equivalence ✻ tells us that computing the answer to d can be substituted with deciding whether or not $\langle d \rangle$ is in $L(D)$. That is:

Solving a decision problem D can be reduced to recognizing the set L(D) in Σ^*.



The answer to the instance d of D *can be found if we* determine where $\langle d \rangle$ is relative to $L(D)$

- The link ✻ is important because it enables us to apply---when solving *decision problems*---all the theory developed to *recognize sets*.
- What does the *recognizability of* $L(D)$ tell us about the *solvability of* D?
  - $L(D)$ is decidable ⇒ There is an algorithm that, for *any* $d \in D$, answers YES or NO.

    Proof. There is a TM that, for <u>*any*</u> $\langle d \rangle \in \Sigma^*$, *decides* whether or not $\langle d \rangle \in L(D)$. Then apply ✻. □
  - $L(D)$ is semi-decidable ⇒ Then there is an algorithm that,
    - for *any positive* $d \in D$, answers YES;
    - for *a negative* $d \in D$, *may* or *may not* answer NO in *finite time.*

    Proof. There is a TM that, for <u>*any*</u> $\langle d \rangle \in L(D)$, accepts $\langle d \rangle$. However, if $\langle d \rangle \notin L(D)$, the algorithm *may or may not reject* $\langle d \rangle$ *in finite time*. Then apply ✻. □
  - $L(D)$ is undecidable ⇒ There is *no algorithm* that, for *any* $d \in D$, *answers* YES or NO.

    Proof. There is *no* TM capable deciding, for <u>*any*</u> $\langle d \rangle \in \Sigma^*$, whether or not $\langle d \rangle \in L(D)$. Then apply ✻. □

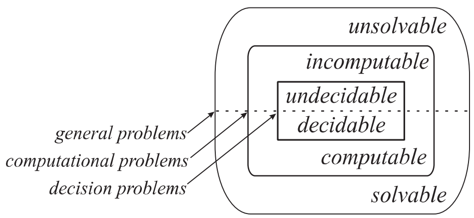
- We can now *extend* our *terminology* about sets to *decision problems*.

Definition. Let D be a decision problem. We say that the problem

- D is decidable (or *computable*) if $L(D)$ is decidable set;
- D is semi-decidable if $L(D)$ is semi-decidable set;
- D is undecidable (or *incomputable*) if $L(D)$ is undecidable set.

- Terminology.

Instead of a *decidable*/*undecidable* problem we can say *computable*/*incomputable* problem. But the latter notion is more general: it can be used with *all kinds of computational problems*, not only decision problems. The terms *solvable*/*unsolvable* is even more general: it addresses all kinds of *computational* and *non-computational* problems.



8.3 There is an Incomputable Problem Halting Problem

- We now know what is a decidable, semi-decidable, and undecidable decision problem. But, at this point we actually *do not know whether there exists any semi-decidable* (but not decidable) or *any undecidable problem*. How can we find such a D (if there is one at all)?

- In 1936, Turing succeeded in this. He was aware of the fact that difficulties in obtaining computational results are caused by those TM's that may not halt on some input data. *It would be beneficial*, he reckoned, *if we could check, for any TM T and any input word w, whether or not T halts on w.*
  - If such a checking were possible, then, given an arbitrary pair $\langle T,w\rangle$, we would first check $\langle T,w\rangle$ and then, depending on the outcome of the checking, we would either start T on w, or try to improve T so that it would halt on w, too.

- **Halting Problem**

- This led Turing to define a decision problem, called the *Halting Problem.*

Definition. The Halting Problem D_{Halt} is defined by

$$D_{Halt} = \text{“Given a TM } T \text{ and } w \in \Sigma^*\text{, does } T \text{ halt on } w\text{?”}$$

- Then Turing proved the following theorem.

Theorem. *The Halting Problem D_{Halt} is undecidable.*

Comment. This means that *there exists no algorithm* capable of answering, for *arbitrary* T and w, the question "*Does T halt on w?*"
So, *any algorithm* whatsoever, which we might design now or in the future for answering this question, *will fail* to give the answer *for at least one pair T, w.*

- **Proof.**

- Before we go to the proof, we introduce two sets that play an important role in the proof. These are called the *universal* and *diagonal languages*, respectively.

Definition. The universal language, dented by K_0, is the language of the *Halting Problem,* that is, $K_0 = L(D_{Halt}) = \{ \langle T, w\rangle \mid T \text{ halts on } w \}$.

The second language is obtained from K_0 by imposing the restriction $w := \langle T\rangle$

Definition. The diagonal language, dented by K, is defined by

$$K = \{ \langle T, T\rangle \mid T \text{ halts on } \langle T\rangle \}.$$

- **Note**:
  - K is the language of the problem D_H = "Given a TM T, does T halt on $\langle T\rangle$?"
  - D_H is a *subproblem* of D_{Halt} (it is obtained from D_{halt} by restricting w to $w = \langle T\rangle$).

- We now proceed to the proof.

  The plan is:
  - prove (in a lemma) that K is an *undecidable set*;
  - this will imply that K_0 is *undecidable* and, hence, D_{halt} is an *undecidable problem*.

- The lemma is instructive; it *applies* a *cunningly* defined TM S to its *own* code $\langle S\rangle$.

Lemma. *The set K is undecidable.*

Proof of the lemma. (proof by contradiction).

(★) *Suppose* that K is a *decidable* set.

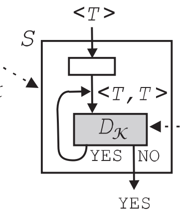
- Then there must exist a TM D_K that, for *any* T, answers $\langle T,T\rangle \in ? K$ with

$$D_K(\langle T,T\rangle) = \begin{cases} \text{YES}, & \text{if } T \text{ halts on } \langle T\rangle; \\ \text{NO}, & \text{if } T \text{ doesn't halt on } \langle T\rangle. \end{cases}$$

- Now we construct a new TM S.
  The *intention* is to construct S in such a way that, *when given as input its own code $\langle S\rangle$, S will expose the incapability of D_K in predicting whether or not S will halt on $\langle S\rangle$.*

The TM S is:

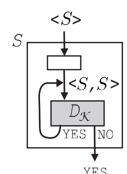
The shrewd Turing machine S uncovers the incapability of D_K to answer whether S halts on its own code <S>.



The supposed machine D_K answers, for arbitrary T, whether T halts on its own code <T>.

- *S operates as follows.* The input to S is the code $\langle T\rangle$ of an arbitrary TM T. S doubles $\langle T\rangle$ into $\langle T,T\rangle$, sends this to D_K, and starts it. D_K eventually halts on $\langle T,T\rangle$ and answers either YES or NO to the question $\langle T,T\rangle \in ? K$. If D_K has answered YES, then S asks D_K again the same question. If, however, D_K has answered NO, then S outputs its own answer YES and halts.
- *But S is shrewd*: if given as input its *own code* $\langle S\rangle$, it puts the supposed D_K in insurmountable trouble. Let us see why.

  Given the input $\langle S\rangle$, S doubles it into $\langle S,S\rangle$ and hands it over to D_K, which in finite time answers the question $\langle S,S\rangle \in ? K$ with either YES or NO.



The consequences of the answers to $\langle S,S\rangle \in ? K$ are:

a) *Suppose that D_K has answered* YES. Then S repeats the question $\langle S,S\rangle \in ?K$ to D_K, which in turn repeats its answer $D_K(\langle S,S\rangle)$ = YES. So S cycles and will *not halt*. But, at the same time, D_K repeatedly predicts just the opposite (that S *will halt* on $\langle S\rangle$). So in (a) D_K *fails* to compute the correct answer.

b) *Suppose that D_K has answered* NO. Then S returns to the environment its own answer and *halts*. But just before that D_K has computed $D_K(\langle S,S\rangle)$ = NO and thus predicted that S will *not* halt on $\langle S\rangle$. So, in the case (b) D_K *fails* to compute the correct answer.

Thus, D_K is *unable* to correctly decide the question $\langle S,S\rangle \in ? K$. This contradicts our supposition (★) that K is a *decidable* set and D_K its decider. So K is *not decidable.*

The lemma is proved.

- Because K is undecidable, so is the problem D_H. But D_H is a subproblem of D_{Halt}. So the *Halting Problem* D_{halt} is undecidable too. □

8.4 The Basic Kinds of Decision Problems

- Now we know that besides *decidable* problems there also exist *undecidable* problems.
- What about *semi*-decidable problems? Do they exist? Are there *undecidable* problems that are *semi*-decidable? That is, are there decision problems such that only their positive instances are guaranteed to be solvable?
- The answer is yes. In this section we explain why this is so.

The are undecidable sets that are still semi-decidable

- **Theorem.** *K_0 is a semi-decidable set.*

  Proof. We must find a TM that accepts K_0 . Here is an idea. Given an arbitrary input $\langle T, w\rangle$, the machine must simulate T on w, and *if* the simulation *halts*, the machine must return YES and halt. So, if such a machine exists, it will answer YES *iff* $\langle T,w \rangle \in? K_0$. But we already know that such a machine exists: it is the *Universal Turing Machine U*. Hence, K_0 is semi-decidable. □

  Comment. This is why K_0 is called the *universal* language.

  The last two theorems imply the following consequence:

  Corollary. *K_0 is an undecidable semi-decidable set.*

- Similarly we prove that K is an undecidable semi-decidable set.

The are undecidable sets that are not even semi-decidable

- What about the set $\overline{K_0}$, the complement of K_0 ?

  Theorem. *$\overline{K_0}$ is not a semi-decidable set.*

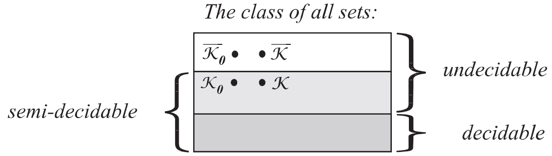
  Proof. If $\overline{K_0}$ were semi-decidable, then both K_0 and $\overline{K_0}$ would be semi-decidable. But then K_0 would be decidable (due to Post's theorem). This would be a contradiciton. So $\overline{K_0}$ is *not* semi-decidable. □

- Similarly we prove that $\overline{K}$ is not a semi-decidable set.

The basic kinds of decision problems

- We have proved the existence of *undecidable semi-decidable* sets (e.g. K_0 and K) and the existence of *undecidable* sets that are *not even semi-decidable* ($\overline{K_0}$ and $\overline{K}$). Consequently, the class of all the sets partitions into *three non-empty subclasses*:

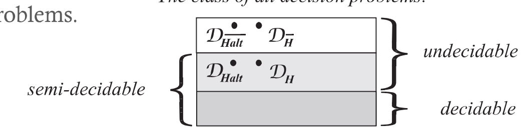
The class of all sets:



(cont'd)

- We can view sets as languages of decision problems. Then, K_0 and K are the languages of decision problems D_{Halt} and D_H , respectively.
  What about the sets $\overline{K_0}$ and $\overline{K}$? These are the languages of decision problems
  - $D_{\overline{Halt}}$ = "Given a TM T and a word w, does T *never halt* on w ?" and
  - $D_{\overline{H}}$ = "Given a TM T, does T *never halt* on $\langle T \rangle$?"
- The class of all the *decision problems* partitions into two *non-empty* subclasses:
  - the class of decidable problems and
  - the class of undecidable problems.

The class of all decision problems:



There is also a third class, the class of semi-decidable problems (which contains all the decidable and some, but not all, of the undecidable problems).

(cont'd)

- In other words, a decision problem D can be of one of the 3 kinds:
  - D is decidable.

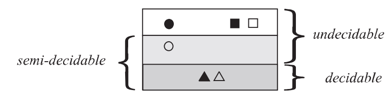
    This means that there is an algorithm that can solve an arbitrary instance $d \in D$. Such an algorithm is called the *decider* of the problem D.
  - D is semi-decidable undecidable.

    This means that no algorithm can solve an *arbitrary* instance $d \in D$, but that there is an algorithm that can solve an *arbitrary positive* $d \in D$. Such an algorithm is called the *recognizer* of the problem D.
  - D is not semi-decidable.

    This means that for *any* algorithm there exists a positive instance and a negative instance of D such that the algorithm cannot solve either of them.

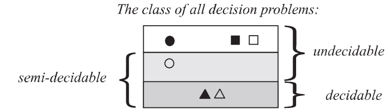
- **Complementary sets and decision problems**.
- From the previous theorems it follows that there are only three possibilities for the decidability of a set S and its complement $\overline{S}$:
  - S and $\overline{S}$ are *decidable* (△▲, see figure below);
  - S and $\overline{S}$ are *undecidable*; *one is semi-decidable* and *the other is not* (○●);
  - S and $\overline{S}$ are *undecidable* and *neither is semi-decidable* (□■).

The class of all sets:



- The same holds for the decidability of the corresponding *decision problems:*

The class of all decision problems:



8.5 Some Other Incomputable Problems

- Are there any other incomputable problems? The answer is yes.
- Since the 1940s many other incomputable problems were discovered. The first of these problems referred to the properties and the operations of models of computation. After 1944, more realistic incomputable problems were (and are still being) discovered in different fields of science and in other nonscientific fields.
- In this section we list some of the known incomputable problems, grouped by the fields in which they occur.
  No algorithm can solve any of the following problems in general.

- **Problems about algorithms and computer programs**.
- TERMINATION OF ALGORITHMS (PROGRAMS)

  Let A be an arbitrary algorithm and d be arbitrary input data. Questions:
  - D_{Term} = "Does A terminate on every input data?"
  - "Does A terminate on input data d?"
- CORRECTNESS OF ALGORITHMS (PROGRAMS)

  Let P be an arbitrary computational problem and A an arbitrary algorithm. Question:
  - D_{Corr} = "Does the algorithm code(A) correctly solve the problem code(P)?"
- EXISTENCE OF SHORTER EQUIVALENT PROGRAMS

  Let code(A) be a program describing an algorithm A. Question:
  - "Given a program code(A), is there a shorter equivalent program?"

Problems about programming languages and grammars

- AMBIGUITY OF CFG GRAMMARS

  Let G be a context-free grammar. Question:

  - "Is there a word that can be generated by G in two different ways?"

- EQUIVALENCE OF CFG GRAMMARS

  Let G_1 and G_2 be CFGs. Question:

  - "Do G_1 and G_2 generate the same language?"

- OTHER PROPERTIES OF CFG s AND CFL s

  Let G and G' be arbitrary CFGs, and let C and R be an arbitrary CFL and a regular language, respectively. As usual, Σ is the alphabet. Questions:

  - "Is $L(G) = \Sigma^*$?" "Is $L(G)$ regular?" "Is $R \subseteq L(G)$?"
  - "Is $L(G) = R$?" "Is $L(G) \subseteq L(G')$?" "Is $L(G) \cap L(G') = 0$?"
  - "Is $L(G) \cap L(G')$ CFL?" "Is C ambiguous CFL?" "Is there a palindrome in $L(G)$?"

Problems about computable functions

- INTRINSIC PROPERTIES OF COMPUTABLE FUNTIONS

  Let $\varphi: A \to B$ and $\psi: A \to B$ be arbitrary computable functions. Questions:

  - "Is dom(φ) empty?"
  - "Is dom(φ) finite?"
  - "Is dom(φ) infinite?"
  - "Is A − dom(φ) finite?"
  - "Is φ total?"
  - "Can φ be extended to a total computable function?"
  - "Is φ surjective?"
  - "Is φ defined at x?"
  - "Is φ defined at x?"
  - "Is $\varphi(x) = y$ for at least one x?"
  - "Is dom(φ) = dom(ψ)?"
  - "Is $\varphi = \psi$?"

Problems from number theory, algebra, and analysis

- SOLVABILITY OF DIOPHANTINE EQUATUINS

  Let $p(x_1, \ldots, x_n)$ be an arbitrary polynomial with unknowns $x_1,\ldots,x_n$ and rational integer coefficients. Question:

  - "Does a Diophantine equation $p(x_1, \ldots, x_n) = 0$ have a solution?"

- MORTAL MATRIX PROBLEM

  Let M be a finite set of $n \times n$ matrices with integer entries. Question:

  - "Can the matrices of M be multiplied in some order, possibly with repetition, so that the product is zero matrix O ?"

- EXISTENCE OF ZEROS OF FUNCTIONS

  Let $f: \mathbf{R} \to \mathbf{R}$ be an arbitrary elementary function. Question:

  - "Is there a real solution to the equation $f(x) = 0$?"

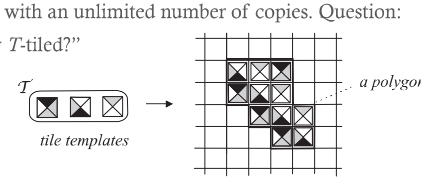
A function $f(x)$ is *elementary* if it can be constructed from a finite number of exponentials, logarithms, roots, real constants, and the variable x by using function composition and the four basic operations +, −, ×, and ÷.

Problems about games

- DOMINO TILING PROBLEM

  Let T be a finite set of tile templates, each with an unlimited number of copies. Question:

  - "Can every finite polygon be regularly T-tiled?"

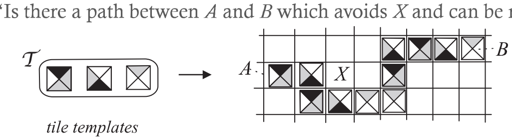


a regular $\mathcal{T}$-tiling of the polygon

- DOMINO SNAKE PROBLEM

  Let T be a finite set of tile templates and A, B, X arbitrary 1×1 squares in $\mathbb{Z}^2$. Question:

  - "Is there a path between A and B which avoids X and can be regularly T-tiled?"



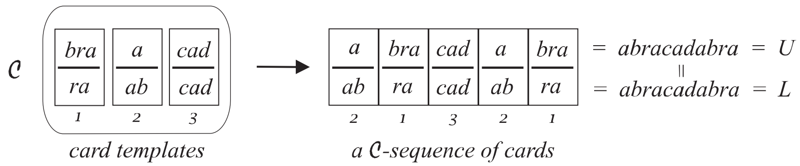
a regular $\mathcal{T}$-tiling of a path $p(A,B,X)$

Post's correspondence problem

- POST'S CORRESPONDENCE PROBLEM

  Let C be a finite set of card templates, each with an unlimited number of copies. Question:

  - "Is there a finite C-sequence such that $U=L$?"



Busy beaver problem

- Informally, a busy beaver is the most 'productive' TM of its kind.
- What *kind* of TMs do we mean?

  We mean TM's that *do not waste time* with writing symbols other than 1 or not moving the window. Let us group such TM's into classes $\mathcal{T}_n$, $n = 1,2,\ldots$ where $\mathcal{T}_n$ contains TM's with the same number of states.

  Definition. Define $\mathcal{T}_n$ (for $n \geq 1$) to be the class of all TM's that have:
  - the tape unbounded in both ways;
  - n non-final states (including q_1) and one final state q_{n+1} ;
  - $\Sigma = \{0,1\}$ and $\Gamma = \{0,1,\sqcup\}$;
  - δ that writes only the symbol 1 and moves the window either to L or R.

  Theorem. (Radó) For any $n \geq 1$, there are *finitely many* TM's in $\mathcal{T}_n$.

- **Definition**. We say that a TM $T \in \mathcal{T}_n$ is a stopper if T *halts* on an *empty input*.

  Theorem. (Radó) For *every* $n \geq 1$, there *exists* a stopper in $\mathcal{T}_n$.
  Hence there is *at least one* and *at most finitely many* (i.e. $|\mathcal{T}_n|$) stoppers in $\mathcal{T}_n$.

- So, there must exist in $\mathcal{T}_n$ a stopper that attains, among all the stoppers in $\mathcal{T}_n$, the *maximum number of* 1's, which are left on the tape *after halting*.

  Definition. Such a stopper is called the *n*-state *Busy Beaver* and denoted *n*-BB.

- BUSY BEAVER PROBLEM

  Let $T \in \bigcup_{i\geq 1} \mathcal{T}_i$ be an arbitrary TM. Question:

  - "Is T a Busy Beaver?" (i.e. ``Is there $n \geq 1$, such that $T = n$-BB?'')

- **Definition** The *Busy Beaver function* is $s(n)$ = 'the number of 1's attained by n-BB'

  Theorem. The Busy Beaver function is *incomputable*.

8.6 Dictionary

undecidability neodločljivost computational problems računski problemi decision problem odločitveni problem search problem problem iskanja, iskalni problem counting problem problem preštevanja generation problem problem generiranja language of a decision problem jezik odločitvenega problema instance primerek problema, naloga coding function kodirna funkcija code koda decidable, semi-decidable, undecidable decision problem odločljiv, polodločljiv, neodločljiv odločitveni problem computable/incomputable problem izračunljiv/neizračunljiv problem solvable/unsolvable problem rešljiv/nerešljiv problem halting problem problem ustavitve universal language univerzalni jezik diagonal language diagonalni jezik termination of ustavljivost correctness pravilnost ambiguity dvoumnost intrinsic property vsebovana (naravna, bistvena) lastnost solvability of Diophantine equations rešljivost Diofantskih enačb tiling problem problem tlakovanja Post's correspondence problem Postov korespondenčni problem busy beaver problem problem garača stopper stroj, ki se ustavi (uspešnež?)

9
The Chomsky Hierarchy

Contents

- THIS YEAR LEFT OUT

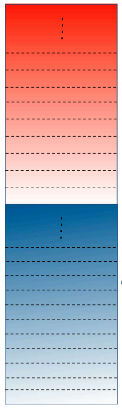
10
Computational Complexity Theory

Contents

10.1 Introduction

- We were interested in *what can be computed and what cannot, regardless of the amount of computational resources (time, space) needed for that.*
- We discovered that there are *computable* and *incomputable* problems.

incomputable problems



No algorithm can solve any of the incomputable problems *in general*. There are *infinitely many* different *degrees* of incomputability. There is *no* most incomputable problem!

computable problems

Each computable problem has *an algorithm that solves it*. Intuition tells us: *given more time/space, larger instances* or *more difficult problems* can be solved.

How much time/space can we afford?

10.2 Deterministic time and space (classes DTIME, DSPACE)

- **Question**: How much *time* or *space* needs an algorithm to *solve* a (decidable) *decision problem D*?
- Due to the close link ❇ between decision problems and languages we can express this question in terms of formal languages:

  Question: How many *steps* or *tape cells* needs a TM to *recognize* the language $L(D)$ of a decision problem D?
- In this section we make these questions more exact.

- **Deterministic time complexity & complexity classes DTIME**
- **Definition**. Let $M = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a deterministic TM with $k \geq 1$ two-way infinite tapes. We say that M is of (deterministic) time complexity $T(n)$ if, for every input $w \in \Sigma^*$ of length n, M makes $\leq T(n)$ steps before halting.
  - It is assumed that M reads all of w; thus $T(|w|) \geq |w| + 1$, so $T(n) \geq n + 1$. If $n = a$, we understand that $T(a)$ means $\max\{a+1, \lceil T(a) \rceil\}$.
- A TM M of time complexity $T(n)$ can decide $w \in? L(M)$ in $\leq T(|w|)$ steps.

  This motivates the next definition.

(con't)

- **Definitions**. A *language L* is of (deterministic) time complexity $T(n)$ if there exists a deterministic TM M of (det.) time complexity $T(n)$ such that $L = L(M)$. We define the *class* of all such languages by

  $\mathrm{DTIME}(T(n)) = \{L \mid L \text{ is a language} \wedge L \text{ is of (det.) time complexity } T(n)\}$

  DTIME($T(n)$) contains all L's for which the problem $w \in? L$ can be det. solved in $\leq T(|w|)$ time.
- Using the link ❇, we can restate both definitions in terms of *decision problems*:

  Definitions. A *decision problem D* is **of (deterministic) time complexity** $T(n)$ if its language $L(D)$ is of (det.) time complexity $T(n)$. We define the *class* of all such decision problems by

  $\mathbf{DTIME}(T(n)) = \{D \mid D \text{ is a decis. probl.} \wedge D \text{ is of (det.) time complexity } T(n)\}$

  DTIME($T(n)$) has all D's whose instances d can be deterministically solved in $\leq T(|\langle d \rangle|)$ time.

- **Deterministic space complexity & complexity classes DSPACE**
- **Definition**. Let $M = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a deterministic TM with one *input tape* and $k \geq 1$ *work tapes*. Then M is of (deterministic) space complexity $S(n)$ if, for every input $w \in \Sigma^*$ of length n, M uses, before halting, $\leq S(n)$ cells on each work tape.
  - Note that the input-tape cells don't matter.
  - It is assumed that M uses at least the cell under the initial position of the window. Thus, $S(|w|) \geq 1$, so $S(n) \geq 1$. If $n = a$, we understand that $S(a)$ is $\max\{1, \lceil S(a) \rceil\}$.
- A TM M of space complexity $S(n)$ can decide $w \in ? L(M)$ on space $\leq S(n)$.

  This motivates the next definition.

(con't)

- **Definitions**. A *language* L is of (deterministic) space complexity $S(n)$ if there exists a deterministic TM M of (det.) space complexity $S(n)$ such that $L = L(M)$. We define the *class* of all such languages by

  DSPACE($S(n)$) = $\{L \mid L$ is a language $\wedge$ L is of (det.) space complexity $S(n)\}$

  DSPACE($S(n)$) has all L's for which the problem $w \in ? L$ can be det. solved on $\leq S(|w|)$ space.
- Again, using ❇ we can restate both definitions in terms of *decision problems*:

  Definitions. A *decision problem* D is **of (deterministic) space complexity** $S(n)$ if its language $L(D)$ is of (det.) space complexity $S(n)$. We define the *class* of all such decision problems by

  DSPACE($S(n)$) = $\{D \mid D$ is a decis. probl. $\wedge$ D is of (det.) space complexity $S(n)\}$

  DSPACE($S(n)$) has all D's whose instances d can be deterministically solved on $\leq S(|\langle d \rangle|)$ space.

10.3 Nondeterministic time and space (classes NTIME, NSPACE)

- Now suppose that we could use *non*-deterministic TM's.

  Question: How many *steps* or *tape cells* would require a *non*deterministic TM to *recognize* the language $L(D)$ of a decision problem D?
- Stated in terms of algorithms and decision problems:

  Question: How much *time* or *space* would require a *non*deterministic algorithm to *solve* a *decision problem* D?
- We now make these questions more precise.

- **Nondeterministic time complexity & complexity classes NTIME**
- **Definition**. Let $N = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a *nondeterministic* TM. Then N is of nondeterministic time complexity $T(n)$ if, for every input $w \in \Sigma^*$ of length n, there *exists* a computation in which N makes $\leq T(n)$ steps before halting.
  - Again, it is assumed that N reads all of w; thus $T(|w|) \geq |w| + 1$, so $T(n) \geq n + 1$. If $n = a$, we understand that $T(a)$ means $\max\{a+1, \lceil T(a) \rceil\}$.
- A nondeterministic TM N of time complexity $T(n)$ can decide any question $w \in ? L(M)$ in $\leq T(|w|)$ steps.

  This motivates the next definition.

(con't)

- **Definitions**. A *language* L is of nondeterministic time complexity $T(n)$ if there is a nondeterministic TM N of nondet. time complexity $T(n)$ such that $L = L(M)$. The *class* of *all such languages* is

  NTIME($T(n)$) = $\{L \mid L$ is a language $\wedge$ L has nondet. time complexity $T(n)\}$

  NTIME($T(n)$) contains all L's for which the *problem* $w \in ? L$ can be *non*det. solved in $\leq T(|w|)$ time.

- Restating both definitions in terms of *decision problems* we obtain:

  Definitions. A *decision problem* D is **of nondeterministic time complexity** $T(n)$ if its language $L(D)$ is of nondet. time complexity $T(n)$. We define the *class* of all such decision problems by

  NTIME($T(n)$) = $\{D \mid D$ is a decision probl. $\wedge$ D is of nondet. time complexity $T(n)\}$

  NTIME($T(n)$) has all D's whose instances d can be nondet. solved in $\leq T(|\langle d\rangle|)$ time.

- **Nondeterministic space complexity & complexity classes NSPACE**

- **Definition**. Let $N = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a *nondeterministic* TM with one *input tape* and $k \geq 1$ *work tapes*. Then N is of nondeterministic space complexity $S(n)$ if, for every input $w \in \Sigma^*$ of length n, there *exists* a computation in which N uses, before halting, $\leq S(n)$ cells on each work tape.
  - Again, the input-tape cells don't count.
  - It is assumed that M uses at least the cell under the initial position of the window. Thus, $S(|w|) \geq 1$, so $S(n) \geq 1$. If $n = a$, we understand that $S(a)$ is $\max\{1, \lceil S(a)\rceil\}$.

- A nondeterministic TM N of space complexity $S(n)$ can decide any question $w \in ? L(M)$ on $\leq S(|w|)$ space.

  This motivates the next definition.

(con't)

- **Definitions**. A *language* L is of nondeterministic space complexity $S(n)$ if there is a nondeterministic TM N of nondet. space complexity $S(n)$ such that $L = L(M)$. The *class* of all languages is

  NSPACE($S(n)$) = $\{L \mid L$ is a language $\wedge$ L is of nondet. space complexity $S(n)\}$

  NSPACE($S(n)$) has all L's for which $w \in ? L$ can be *non*deterministically solved on $\leq S(|w|)$ space

- In terms of *decision problems:*

  Definitions. A *decision problem* D is **of nondeterministic space complexity** $S(n)$ if its language $L(D)$ is of nondet. space complexity $S(n)$. We define the *class* of all such decision problems by

  NSPACE($S(n)$) = $\{D \mid D$ is a decision probl. $\wedge$ D is of nondet. space complexity $S(n)\}$

  NSPACE($S(n)$) has all D's whose instances d can be nondet. solved on $\leq S(|\langle d\rangle|)$ space.

- **Summary of complexity classes**

- **In terms of formal languages:**

  DTIME($T(n)$) = $\{L \mid L$ is a *language* $\wedge$ L of *time* complexity $T(n)\}$
  DSPACE($S(n)$) = $\{L \mid L$ is a *language* $\wedge$ L of *space* complexity $S(n)\}$
  NTIME($T(n)$) = $\{L \mid L$ is a *language* $\wedge$ L of *nondet. time* complexity $T(n)\}$
  NSPACE($S(n)$) = $\{L \mid L$ is a *language* $\wedge$ L of *nondet. space* complexity $S(n)\}$

- **In terms of decision problems:**

  DTIME($T(n)$) = $\{D \mid D$ is a *decision problem* $\wedge$ $L(D)$ of *time* complexity $T(n)\}$
  DSPACE($S(n)$) = $\{D \mid D$ is a *decision problem* $\wedge$ $L(D)$ of *space* complexity $S(n)\}$
  NTIME($T(n)$) = $\{D \mid D$ is a *decision problem* $\wedge$ $L(D)$ of *nondet. time* complexity $T(n)\}$
  NSPACE($S(n)$) = $\{D \mid D$ is a *decision problem* $\wedge$ $L(D)$ of *nondet. space* complexity $S(n)\}$

- **Informally:** DTIME($T(n)$) = {decision problems solvable deterministically in time $T(n)$}
  DSPACE($S(n)$) = {decision problems solvable deterministically on space $S(n)$}
  NTIME($T(n)$) = {decision problems solvable nondeterministically in time $T(n)$}
  NSPACE($S(n)$) = {decision problems solvable nondeterministically on space $S(n)$}

10.4 Tape compression, linear speedup, and reductions in the number of tapes

- In this section we show that
  - space complexity can always be *reduced by a constant factor* (by encoding several tape symbols into one); and
  - time complexity can always be *reduced by a constant factor* (by grouping several steps into one)
- So, when we study computational complexity, we can *ignore constant factors* and focus on the *rate of growth* of complexity functions.

- **Tape compression**
- **Motivation**. We defined the space needed by a computation of a TM to be the maximum number of cells that are used on any work tape. **Idea**: Let is *encode several symbols by one symbol from a larger alphabet.*
  - Example. Group 00110110 into 00 11 01 10, and encode each pair by a symbol from {0,1,2,3}, say by 00→0, 01→1, 10→2, 1→3. The result is a word 0312 with length 4.

  By expanding alphabets, we reduced the space. This holds in general.
- **Theorem**. If L is of space complexity $S(n)$, then for any $c > 0$, L is of space complexity $c\,S(n)$. This holds for the nondeterministic space complexity too.

  Proof. Along the example in the motivation. □
- **Corollary**: For *any* $c > 0$ is DSPACE($S(n)$) = DSPACE($cS(n)$) and NSPACE($S(n)$) = NSPACE($cS(n)$)

- **Linear speedup**
- Can we do similarly with time? **Idea**. Since the time needed by a computation is the *number of steps* made before halting, *we group several steps into a larger step.* To do similarly as with space, it turns out that two conditions must be fulfilled:
  - TM must have at least 2 tapes (i.e. k>1),
  - $\inf_{n\to\infty} T(n)/n = \infty$ must hold. (Definition: $\inf_{n\to\infty} f(n) = \lim_{n\to\infty} \mathrm{glb}\{f(n), f(n+1), \ldots\}$)

    This means that $T(n)$ must increase (at least slightly) faster than n. Then we can be sure that after reading the input there will remain some time available for computation.
- **Theorem**. Let $\inf_{n\to\infty} T(n)/n = \infty$ and k>1. Then:
  If L is of time complexity $T(n)$, then for any $c > 0$, L is of time complexity $cT(n)$. This holds for the nondeterministic space complexity too.
- **Corollary**: If $\inf\ T(n)/n = \infty$, then for *any* $c > 0$ DTIME($T(n)$) = DTIME($cT(n)$) and NTIME($T(n)$) = NTIME($cT(n)$)

- **Summary**
- **Under certain (but reasonable) conditions:**

  DTIME($T(n)$) = DTIME($cT(n)$)
  NTIME($T(n)$) = NTIME($cT(n)$)
  DSPACE($S(n)$) = DSPACE($cS(n)$)
  NSPACE($S(n)$) = NSPACE($cS(n)$)
- Positive constants *c have no impact* on the contents of the class.
  - Example: DTIME(n^2) = DTIME($0.33\,n^2$) = DTIME($4n^2$) = DTIME($7n^2$) = …
- Instead of writing that a decision problem D is in DTIME(n^2), we can say that D is of (deterministic) time complexity which is of the order O(n^2).

Reductions in the number of tapes

- To study *time complexity* we use TMs with $k \geqslant 1$ tapes.
  Question: How does *reduction* of the number k affect the *time complexity*?
  Answer: if we restrict TMs to 1 tape, the time complexity *may become squared*, but if we restrict them to 2 tapes, the loss of time is smaller.
  Theorem.
  - If $L \in \mathrm{DTIME}(T(n))$, then L is accepted in time $T^2(n)$ by a 1-tape TM.
    If $L \in \mathrm{NTIME}(T(n))$, then L is accepted in time $T^2(n)$ by a 1-tape nondet. TM.
    If $L \in \mathrm{DTIME}(T(n))$, then L is accepted in time $T(n)\log T(n)$ by a 2-tape TM.
    If $L \in \mathrm{NTIME}(T(n))$, then L is accepted in time $T(n)\log T(n)$ by a 2-tape nondet. TM.
- To study *space complexity* we use TMs with $k \geqslant 1$ work tapes and 1 input tape.
  Question: How does *reduction* in k affect the *space complexity?*
  Answer: The reduction of tapes does not affect space complexity.
  Theorem. If L is accepted by a k-work-tape TM of space complexity $S(n)$, then L is accepted by a 1-work-tape TM of space complexity $S(n)$.

10.5 Relations between DTIME, DSPACE, NTIME, NSPACE

- What are *inclusions* (i.e. $\subseteq$) between the introduced complexity classes? We are interested in the inclusions
  - between the *classes of the same kind* (e.g. DTIME($T(n)$) with various $T(n)$)
  - between *different classes* (e.g. DTIME and NTIME with unspecified $T(n)$)
- We will see that
  - for each CLASS (of the introduced classes) there is an *infinite hierarchy*
    $\mathrm{CLASS}(f_1(n)) \subsetneq \mathrm{CLASS}(f_2(n)) \subsetneq \ldots$ for some functions $f_i(n)$, $i = 1, 2, \ldots$
  - *replacement* of a *nondeterministic* algorithm *by a deterministic* one causes at most *exponential* increase in *time complexity* and at most *quadratic* increase in *space complexity*.

Relations between complexity classes of the same kind

- Hierarchies, …

THIS YEAR LEFT OUT.

Relations between different complexity classes

- The next theorem states the main inclusions between different classes.

Theorem.

- $\mathrm{DTIME}(T(n)) \subseteq \mathrm{DSPACE}(T(n))$
  i.e. What can be solved *in time* $O(T(n))$, can also be solved *on space* $O(T(n))$.
- $L \in \mathrm{DSPACE}(S(n)) \wedge S(n) \geqslant \log_2 n \Rightarrow \exists c: L \in \mathrm{DTIME}(c^{S(n)})$
  i.e. What can be solved *on space* $O(S(n))$, can also be solved *in (at most) time* $O(c^{S(n)})$. (Here c depends on L.)
- $L \in \mathrm{NTIME}\ (T(n)) \Rightarrow \exists c: L \in \mathrm{DTIME}(c^{T(n)})$
  i.e. What can be solved *nondeterministically in time* $O(T(n))$, can be solved *deterministically in (at most) time* $O(c^{T(n)})$. Consequently, the substitution of a nondeterministic algorithm with a deterministic one causes at most exponential increase in the time required to solve a problem.
- $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DSPACE}(S^2(n))$, *if* $S(n) \geqslant \log_2 n \wedge S(n)$ is ``well-behaved.''
  i.e. What can be solved *nondeterministically on space* $O(S(n))$, can also be solved *deterministically on space* $O(S^2(n))$. Consequently, the substitution of a nondeterministic algorithm with a deterministic one causes at most quadratic increase in the space required to solve a problem.

- **``Well-behaved'' complexity functions**
- To avoid some pathological cases, we often use complexity functions $S(n), T(n)$ that are ``well-behaved.'' Below we define what ``well-behaved'' means.
- **Definition**. A function $S(n)$ is space constructible if there is a TM M of space complexity $S(n)$, such that for each n, there *exists an input of length* n on which M uses exactly $S(n)$ tape cells. If for each n, M uses exactly $S(n)$ cells on *any input of length* n, then we say that $S(n)$ is fully space constructible.
- **Definition**. A function $T(n)$ is time constructible if there is a TM M of time complexity $T(n)$, such that for each n, there *exists an input of length* n on which M makes exactly $T(n)$ moves. If *for all* n, M makes exactly $T(n)$ moves on *any input of length* n, then we say that $T(n)$ is fully time constructible.

The sets of space and time constructible functions are very rich and include all common functions. Moreover, most common functions are also fully space and fully time constructible.

- **Proofs.**

THIS YEAR LEFT OUT.

□

10.6 The classes P, NP, PSPACE, NPSPACE

- Of practical interest are the complexity classes
  - DTIME($T(n)$),
  - NTIME($T(n)$),
  - DSPACE($S(n)$),
  - NSPACE($S(n)$),

  whose complexity functions $T(n)$ and $S(n)$ are *polynomials*.

- **Why polynomials?**
- The requirements of a computation for a computational resource (e.g. time, space) are considered to be *reasonable* if they are *bounded by some polynomial*.
  - The following table shows how *exponential* time complexity, such as $T(n) = 2^n$ or $T(n) = 3^n$, becomes unacceptably large even for modest values of n (e.g. n>20).

$T(n)$						
3^n	0.059 sec	58 min	6.5 years	3855 centuries	$2 \cdot 10^8$ centuries	$1.3 \cdot 10^{13}$ centuries
2^n	0.001 sec	1.0 sec	17.9 min	12.7 days	35.7 years	366 centuries
n^5	1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec	0.125 sec	0.216 sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.0016 sec	0.0025 sec	0.0036 sec
n	0,00001 sec	0.00002 sec	0.00003 sec	0.00004 sec	0.00005 sec	0.00006 sec
	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$

- **P, NP, PSPACE, NPSPACE**
- **Definition**. Define the complexity classes P, NP, PSPACE and NPSPACE as:
  - P = $\bigcup_{i\geq 1}\mathrm{DTIME}(n^i)$
    is the class of all decision problems solvable *in polynomial time*
  - NP = $\bigcup_{i\geq 1}\mathrm{NTIME}(n^i)$
    is the class of all decision problems *nondetermistically* solvable *in polynomial time*
  - PSPACE = $\bigcup_{i\geq 1}\mathrm{DSPACE}(n^i)$
    is the class of all decision problems solvable *on polynomial space*
  - NPSPACE = $\bigcup_{i\geq 1}\mathrm{NSPACE}(n^i)$
    is the class of all decision problems *nondeterministically* solvable *on polynomial space*

- **The basic relations** (between P, NP, PSPACE, NPSPACE)
- **Theorem**. The following inclusions hold: P ⊆ NP ⊆ PSPACE = NPSPACE

Proof.
- (**P ⊆ NP**) Every deterministic TM of polynomial time complexity can be viewed as a (trivial) *non*deterministic TM of the same time complexity.
- (**NP ⊆PSPACE**) If $L \in$ NP, then $\exists\, k$ such that $L \in \mathrm{NTIME}(n^k)$. So (by theorem) $L \in \mathrm{NSPACE}(n^k)$, and hence (by Savitch) $L \in \mathrm{DSPACE}(n^{2k})$. Therefore $L \in$ PSPACE.
- (**PSPACE = NPSPACE**) Clearly, PSPACE ⊆ NPSPACE. Now the other direction:
  NSPACE =(def)= $\bigcup \mathrm{NSPACE}(n^i)$ ⊆(by Savitch)⊆ $\bigcup \mathrm{DSPACE}(n^i)$ ⊆ PSPACE.

□

10.7 The question P =? NP

- We've just proved: PSPACE = NPSPACE.
  We can interpret this as follows:

  When space complexity is polynomial,
  nondeterminism adds nothing to the computational power.
- Does similar hold for *time* too? (Recall: P ⊆NP.)

  So, is it: P = NP?

  Or is it: P ⊊ NP?
- In spite of intense research of world's most eminent researchers in the last decades, P =? NP remains open; it's the main question of TCS.

Why is P =? NP **so important ?**

- Many *practically important decision problems* are known to be in NP. So each such problem D has a *nondeterministic algorithm* N_D solving D in *nondetermistic polynomial time*.
- But nondeterministic algorithms are considered to be *unrealistic* because *no real* computer can directly execute any of them. Indeed, how could a real computer *always unmistakably make the right choice from several possible alternatives*?
- So, we must replace N_D by an *equivalent deterministic algorithm* A_D, which computes the same result as N_D by *simulating each right choice* of N_D by a *sequence of deterministic steps*.
- Clearly, A_D requires *additional time* (compared with N_D) to obtain the same result. But, how much time does A_D need in total to solve D?

- Recall the theorem: $L \in$ NTIME $(T(n)) \Rightarrow \exists c : L \in$ DTIME$(c^{T(n)})$. It tells us that the substitution of a nondeterministic algorithm with a deterministic one may cause *at most exponential* increase in the time required to solve a problem.
- In our case, $D \in$ NP, i.e. $D \in$ NTIME(n^k) for some $k \geq 1$. So $D \in$ DTIME(c^{n^k}). Hence, D is deterministically solvable in c^{n^k} *time*. In other words, A_D requires *at most* $O(c^{n^k})$ time to solve D.
- Can A_D, *in spite of the upper bound* $O(c^{n^k})$, solve D in *deterministic polynomial time*?
- *If so,* is this true for an *arbitrary* $D \in$ NP? The question is equivalent to ``Is P=NP?''
- *If* P=NP, then *every* $D \in$ NP is also *deterministically* solvable in polynomial time. So the question ``Is P=NP?'' can also be stated as follows:

Is it true that when time complexity is polynomial,
nondeterminism adds nothing to the computational power.

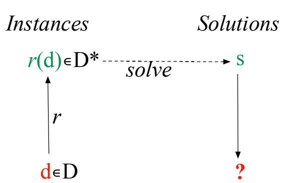
- **How to approach the question** $P =^? NP$ **?**
- The prevalent *belief* is that P ≠ NP (i.e. P ⊊ NP).

  Why? Some consequences of P = NP would be just *too surprising*.
- So we try to prove that P ⊊ NP.
- How? An important method is:
  1. find the ``most difficult'' (i.e. ``hardest'') problem in NP
  2. prove that this problem is not in P.

  The method is based on our *intuition* which suggests that
  - if there are *any* problems in NP−P, then the ``most difficult'' problem in NP should be among them;
  - it should be easier to prove that the ``most difficult'' problem in NP is not in P (than to prove that some other problem in NP is not in P).

- **Problem reductions**
- When is a problem the ``most difficult'' in NP? How do we define that?

  A trivial observation: D^* is the ``*most difficult*'' in NP *if every* $D \in$ NP is ``*at most as difficult as*'' D^*.
- **Idea**: *Suppose* that there *existed* a $D^* \in$ NP, such that we could ``easily'' reduce *every* $D \in$ NP to D^* in the following sense:
  - there would exist a function $r : D \rightarrow D^*$
  - which could ``easily'' transform any instance $d \in D$ into an instance $r(d) \in D^*$,
  - such that the solution s to $r(d)$ could be ``easily'' transformed into the solution ``?'' to d.



  So, for *every* problem D, solving of D could be ``easily'' *replaced* by solving of D^*.
- *If this were possible,* then *every* D could be regarded as ``*at most as difficult as*'' $D^*$. Consequently, $D^*$ could be regarded as ``*the most difficult*'' problem in NP.

- **Polynomial-time reductions**
- But, we must still define what the term ``easily'' should mean. Let us define ``easily'' = ``*in deterministic polynomial time*''
- We are now ready to state the following

  Definition. A problem $D \in$ NP is polynomial-time reducible to a problem D', i.e. $D \leq^p D'$, if there is a deterministic TM M of polynomial time complexity which, for any $d \in D$, returns a $d' \in D'$, such that d *is positive* ⟺ d' *is positive*. The relation $\leq^p$ is called polynomial-time reduction.

  So, M *replaces*, in polynomial time, $d \in D$ with $d' \in D'$ that has the same answer as d.
  (M takes $<d>$ and in poly. time returns a word $M(<d>)$, where $<d> \in L(D) \Leftrightarrow M(<d>) \in L(D')$.)
- So, the ``most difficult'' problem in NP can be a problem with the properties:
  - $D^* \in$ NP
  - $D \leq^p D^*$, for every $D \in$ NP

  In the next section we will call such a problem *NP-complete*.

10.8 NP-complete and NP-hard problems

- In this section we define the notion of the NP-complete problem. Informally, this is just another naming for the ``the most difficult'' problems in NP.
- We then show that there actually *exists* an NP-complete problem.
- Finally we describe, how NP-completeness of other problems can be proved.

- **NP-complete and NP-hard problems**
- We have seen that the ``most difficult'' problem in NP could be defined as the problem D^* that has the following property:
  - $D^* \in$ NP
  - $D \leq^p D^*$, for every $D \in$ NP
- We now give the official definition of such problems.

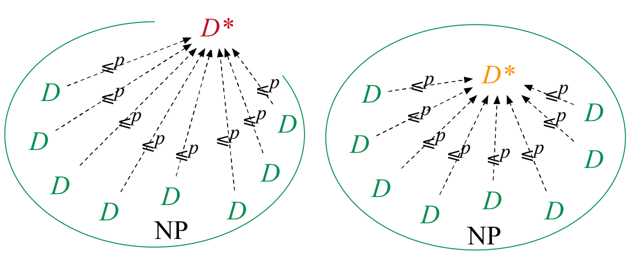
  Definitions. A problem D^* is said to be NP-hard if $D \leq^p D^*$, for every $D \in$ NP.

  A problem D^* is said to be NP-complete if
  - $D^* \in$ NP
  - $D \leq^p D^*$, for every $D \in$ NP.

  Hence, D^* is NP-complete *if* D^* is in NP *and* D^* is NP-hard.

(cont'd)

- We can depict the NP-completeness and NP-hardness of D^* as follows:



D^* is NP-*hard* D^* is NP-*complete*

The dotted arrows represent polynomial-time reductions $D \leq^p D^*$.
Note: an NP-*hard* D^* *may or may not* be in NP.

- **An NP-complete problem, SAT**
- But, *does there exist* an NP-complete problem? That is, does D^* exist?
  Yes! The first such problem was discovered independently by Cook and Levin.
- **Definition.** A Boolean expression is inductively defined as follows:
  - Boolean variables $x_1, x_2, \ldots$ are Boolean expressions.
  - If E, F are Boolean expressions then so are $\neg E$, $E \vee F$, and $E \wedge F$.
- **Definition.** A Boolean expression E is satisfiable if the variables of E can be consistently replaced with values TRUE/FALSE so that E evaluates to TRUE.
- **Definition.** The problem SAT = ``Is a Boolean expression E satisfiable?''

  SAT is called the *Satisfiability Problem.*
- **Theorem (Cook-Levin).** SAT is NP-complete.

  So, for D^* we can take SAT.

- **Proof idea.**
- THIS YEAR LEFT OUT
- □

Proving NP-completeness of problems

- Here is the first theorem.

Theorem. Let $D \leqslant^p D'$. Then
- $D' \in \mathrm{P} \Rightarrow D \in \mathrm{P}$
- $D' \in \mathrm{NP} \Rightarrow D \in \mathrm{NP}$.

So, any problem D that is $\leqslant^p$-reduced to a problem in P (or in NP), is also in P (or NP).

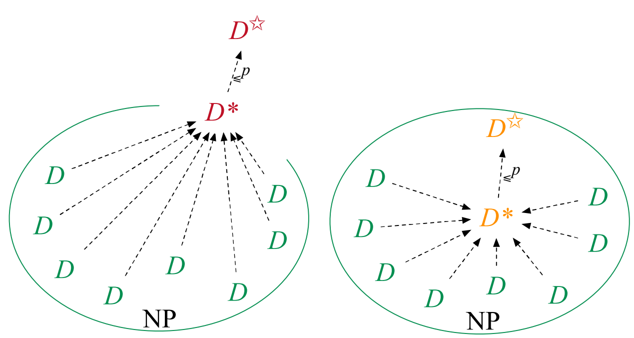
- **Theorem**. The relation $\leqslant^p$ is transitive.

  In other words: $D \leqslant^p D' \wedge D' \leqslant^p D'' \Rightarrow D \leqslant^p D''$.

- **Corollary**. The following holds:
  - D^* is NP-complete $\wedge$ $D^* \leqslant^p D^☆$ $\Rightarrow D^☆$ is NP-hard
  - D^* is NP-complete $\wedge$ $D^* \leqslant^p D^☆$ $\wedge$ $D^☆ \in \mathrm{NP}$ $\Rightarrow D^☆$ is NP-complete

(cont'd)

- Below we depict *the method of proving* NP-completeness or NP-hardness of $D^☆$:



$D^☆$ is NP-*hard* $D^☆$ is NP-*complete*

Examples of NP-complete problems

- In this way, *several thousands of problems* have been proved NP-complete. Here are just three of them.

- PARTITION

  Instance: A finite set A of natural numbers.

  Question: Is there a subset $B \subseteq A$ such that $\sum_{a\in B} a = \sum_{a\in A-B} a$

- HAMILTONIAN CYCLE

  Instance: A graph $G(V,E)$.

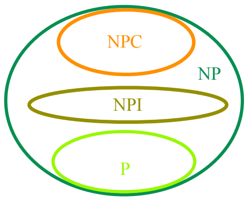
  Question: Is there a Hamiltonian cycle in G?

- BIN PACKING

  Instance: A finite set A of natural numbers, and natural numbers c and k.

  Question: Is there a partition of U into disjoint sets $U_1, U_2, \ldots, U_k$ such that the sum of numbers in each U_i is at most c?

- **Summary.**
- *If* P ≠ NP, *then* the situation in the class NP is depicted below:



If P ≠ NP

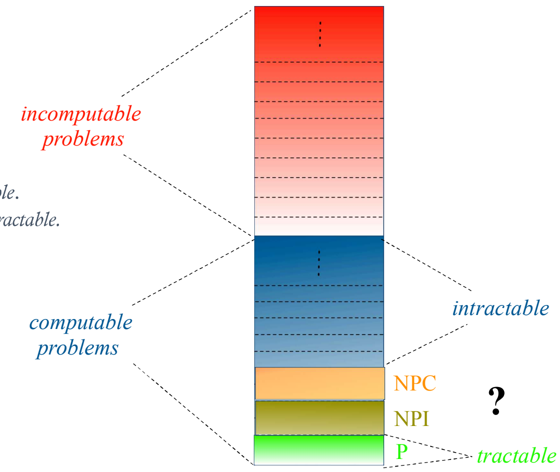
Here:

- NPC is the class of all NP-*complete* problems.
- NPI is the class of all NP-*intermediate* problems. What????

  Ladner has proved: If P≠NP, then there exists a problem in NP that is neither in P nor in NPC.
  Such a problem is called NP-intermediate.
  A candidate problem for NPI: Is a given natural number composite?

If P≠NP, *then* no problem in NPC or NPI has polynomial time complexity.

- (cont'd)
- The problems in P are called *tractable*. Other computable problems are *intractable*.
- The exception are NPC and NPI, because it is still not clear whether they are tractable or intractable.



10.9 Dictionary

computational complexity računska zahtevnost computational resource računski vir (non)deterministic time complexity (ne)deterministična časovna zahtevnost complexity class razred zahtevnosti (non)deterministic space complexity (ne)deterministična prostorska zahtevnost tape compression stiskanje trakov linear speedup pohitritev reduction in the number of tapes zmanjšanje števila trakov "well-behaved" function „lepa, pohlevna" funkcija space/time constructible function prostorsko/časovno predstavljiva (ali verna) funkcija fully space/time constructible function popolnoma prostorsko/časovno predstavljiva (ali verna) funkcija polynomial polinom (non)deterministic polynomial time/space complexity (ne)deterministična polinomska časovna/prostorska zahtevnost hard/difficult problem težek problem problem reduction prevedba problema easy problem lahek problem polynomial-time reduction polinomska časovna prevedba logarithmic-space reduction logaritmična prostorska prevedba NP-complete problem NP-poln problem NP-hard problem NP-težek problem Boolean expression Boolov izraz satisfiable izpolnljiv satisfiability problem problem izpolnljivosti NP-intermediate problem NP-vmesni problem (in)tractable (ne)obvladljiv

11 Intractable Problems

Contents

- THIS YEAR LEFT OUT

12
Coping with Intractable Problems

Contents

- THIS YEAR LEFT OUT

END